Programming Z3

Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger
Microsoft Research

Abstract. This tutorial provides a programmer's introduction to the Satisfiability Modulo Theories Solver Z3. It describes how to use Z3 through scripts, provided in the Python scripting language, and it describes several of the algorithms underlying the decision procedures within Z3. It aims to broadly cover almost all available features of Z3 and the essence of the underlying algorithms.

Contents

1. Introduction

Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is an efficient SMT solver with specialized algorithms for solving background theories. SMT solving enjoys a synergetic relationship with software analysis, verification and symbolic execution tools. This is in many respects thanks to the emphasis on supporting domains commonly found in programs and specifications. There are several scenarios where part of a query posed by these tools can be cast in terms of formulas in a supported logic. It is then useful for the tool writer to have an idea of what are available supported logics, and have an idea of how formulas are solved. But interacting with SMT solvers is not always limited to posing a query as a single formula. It may require a sequence of interactions to obtain a usable answer and the need emerges for the tool writer for having an idea of what methods and knobs are available. In summary, this tutorial aims to answer the following types of questions through examples and a touch of theory:

- What are the available features in Z3, and what are they designed to be used for?

- What are the underlying algorithms used in Z3?

- How can I program applications on top of Z3?

Figure 1 shows an overall systems diagram of Z3, as of version 4.8. The top left summarizes the interfaces to Z3. One can interact with Z3 over SMT-LIB2 scripts supplied as a text file or pipe to Z3, or using API calls from a high-level programming language that are proxies for calls over a C-based API. We focus on using the Python front-end as a means of interfacing with Z3, and start out describing the abstract syntax of terms and formulas accepted by Z3 in Section 2. Formulas draw from symbols whose meaning are defined by a set of *Theories*, Section 3. *Solvers*, Sections 4, 5 and 6, provide services for deciding satisfiability of formula. *Tactics*, Section 7, provide means for pre-processing simplification and creating sub-goals. Z3 also provides some services that are not purely satisfiability queries. *Optimization*, Section 8, services allow users to solve satisfiability modulo objective functions to maximize or minimize values. There are also specialized procedures for enumerating *consequences* (backbone literals) described in Section 4.6.6.
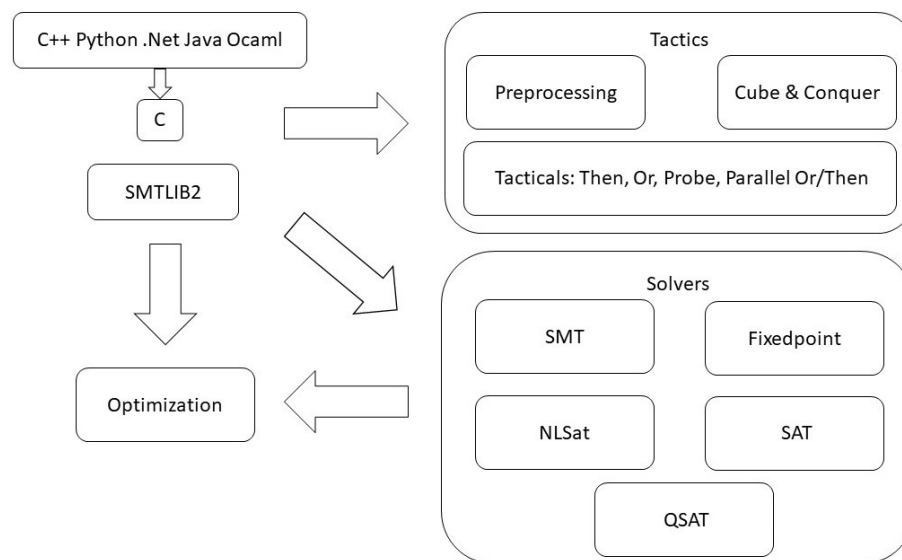
Figure 1. Overall system architecture of Z3

1.1. Resources

The main point of reference for Z3 is the GitHub repository

https://github.com/z3prover/z3

There is an interactive tutorial using the SMT-LIB2 front-end on

https://rise4fun.com/Z3/tutorial/guide

Examples from this tutorial that are executable can be found on

https://github.com/Z3Prover/doc/tree/master/programmingz3/code

There are several systems that program with Z3. They use a variety of front-ends, some use OCaml, others C++, and others use the SMT-LIB2 text interfaces. A few instances that use the Python front-end include

- Dennis Yurichev assembled a significant number of case studies drawn from puzzles and code analysis and presents many of the examples using the Python front-end https://yurichev.com/writings/SAT_SMT_by_example.pdf.

- The Ivy system is written in Python and uses Z3

  https://github.com/Microsoft/ivy.

- The binary analysis kit Angr system is written in Python and uses Z3

  https://docs.angr.io/.

- There was an online interactive Python tutorial. It is discontinued as it ended up being a target for hacks. A snapshot of the web pages, including the non-interactive examples can be found at

  http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py.

1.2. Sources

The material in this tutorial is assembled from several sources. Some of the running examples originate from slides that have circulated in the SAT and SMT community. The first SAT example is shamelessly lifted from Armin Biere's SAT tutorials and other examples appear in slides by Natarajan Shankar.

2. Logical Interfaces to Z3

Z3 takes as input simple-sorted formulas that may contain symbols with pre-defined meanings defined by a *theory*. This section provides an introduction to logical formulas that can be used as input to Z3.

As a basis, propositional formulas are built from atomic variables and logical connectives. An example propositional logical formula accepted by Z3 is:

```python
from z3 import *
Tie, Shirt = Bools('Tie Shirt')
s = Solver()
s.add(Or(Tie, Shirt),
      Or(Not(Tie), Shirt),
      Or(Not(Tie), Not(Shirt)))
print(s.check())
print(s.model())
```

The example introduces two Boolean variables `Tie` and `Shirt`. It then creates a `Solver` object and adds three assertions.

$$(Tie \lor Shirt) \land (\neg Tie \lor Shirt) \land (\neg Tie \lor \neg Shirt)$$

The call to `s.check()` produces a verdict `sat`; there is a satisfying assignment for the formulas. A satisfying model, where `Tie` is false and `Shirt` is true, can be extracted using `s.model()`. For convenience the Python front-end to Z3 contains some shorthand functions. The function `solve` sets up a solver, adds assertions, checks satisfiability, and prints a model if one is available.

Propositional logic is an important, but smaller subset of formulas handled by Z3. It can reason about formulas that combine symbols from several theories, such as the theories for arrays and arithmetic:

```python
Z = IntSort()
```

```
f = Function('f', Z, Z)
x, y, z = Ints('x y z')
A = Array('A', Z, Z)
fml = Implies(x + 2 == y, f(Store(A, x, 3)[y - 2]) == f(y - x + 1))
solve(Not(fml))
```

The formula `fml` is valid. It is true for all values of integers *x, y, z*, array *A*, and no matter what the graph of the function *f* is. Note that we are using `array[index]` as shorthand for `Select(array, index)`. We can manually verify the validity of the formula using the following argument: The integer constants `x` and `y` are created using the function `Ints` that creates a list of integer constants. Under the assumption that `x + 2 = y`, the right side of the implication simplifies to

```
f(Store(A, x, 3)[x]) == f(3)
```

as we have replaced occurrences of y by x - 2. There are no restrictions on what f is, so the equality with f on both sides will only follow if the arguments to f are the same. Thus, we are left to establish

```
Store(A, x, 3)[x] == 3
```

The left side is a term in the theory of arrays, which captures applicative maps. `Store` updates the array A at position x with the value 3. Then `...[x]` retrieves the contents of the array at index x, which in this case is 3. Dually, the *negation* of `fml` is unsatisfiable and the call to Z3 produces `unsat`.

Formulas accepted by Z3 generally follow the formats described in the SMT-LIB2 standard [4]. This standard (currently at version 2.6) defines a textual language for first-order multi-sorted logic and a set of *logics* that are defined by a selection of background theories. For example, the logic of *quantifier-free linear integer arithmetic*, known in SMT-LIB2 as `QF_LIA`, is a fragment of first-order logic, where formulas are *quantifier free*, variables range over integers, interpreted constants are integers, the allowed functions are +, -, integer multiplication, division, remainder, modulus with a constant, and the allowed relations are, besides equality that is part of every theory, also *<, <=, >=, >*. As an example, we provide an SMT-LIB and a Python variant of the same arbitrary formula:

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
(check-sat)
```

Python version:

```
solve((x % 4) + 3 * (y / 2) > x - y)
```

It is also possible to extract an SMT-LIB2 representation of a solver state.

```
from z3 import *
x, y = Ints('x y')
s = Solver()
s.add((x % 4) + 3 * (y / 2) > x - y)
print(s.sexpr())
```

produces the output

```
(declare-fun y () Int)
(declare-fun x () Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
```

2.1. Sorts

Generally, SMT-LIB2 formulas use a finite set of simple sorts. It includes the built-in sort `Bool`, and supported theories define their own sorts, noteworthy `Int`, `Real`, bit-vectors `(_ BitVec n)` for every positive bit-width *n*, arrays `(Array Index Elem)` for every sort *Index* and *Elem*, `String` and sequences `(Seq S)` for every sort *S*. It is also possible to declare new sorts. Their domains may never be empty. Thus, the formula

```
S = DeclareSort('S')
s = Const('s', S)
solve(ForAll(s, s != s))
```

is unsatisfiable.

2.2. Signatures

Formulas may include a mixture of interpreted and free functions and constants. For example, the integer constants 0 and 28 are interpreted, while constants *x*, *y* used in the previous example are free. Constants are treated as nullary functions. Functions that take arguments can be declared, such as `f = Function('f', Z, Z)` creates the function declaration that takes one integer argument and its range is an integer. Functions with Boolean range can be used to create formulas.

2.3. Terms and Formulas

Formulas that are used in assertions or added to solvers are terms of Boolean sort. Otherwise, terms of Boolean and non-Boolean sort may be mixed in any combination where sorts match up. For example

```
B = BoolSort()
f = Function('f', B, Z)
g = Function('g', Z, B)
a = Bool('a')
solve(g(1+f(a)))
```

could produce a solution of the form

```
[a = False, f = [else -> 0], g = [else -> True]]
```

The model assigns a to False, the graph of `f` maps all arguments to `0`, and the graph of `g` maps all values to `True`. Standard built-in logical connectives are `And`, `Or`, `Not`, `Implies`, `Xor`. Bi-implication is a special case of equality, so from Python, when saying a `==` b for Boolean *a* and *b* it is treated as a logical formula for the bi-implication of *a* and *b*.

A set of utilities are available to traverse expressions once they are created. Every function application has a function *declaration* and a set of *arguments* accessed as children.

```
x = Int('x')
y = Int('y')
n = x + y >= 3
print("num args: ", n.num_args())
print("children: ", n.children())
print("1st child:", n.arg(0))
print("2nd child:", n.arg(1))
print("operator: ", n.decl())
print("op name:  ", n.decl().name())
```

2.4. Quantifiers and Lambda binding

Universal and existential quantifiers bind variables to the scope of the quantified formula. For example

```
solve([y == x + 1, ForAll([y], Implies(y <= 0, x < y))])
```

has no solution because no matter what value we assigned to x, there is a value for y that is non-positive and smaller than that value. The bound occurrence of y is unrelated to the free occurrence where y is restricted to be x + 1. The equality constraint y == x + 1 should also not be mistaken for an assignment to y. It is *not* the case that bound occurrences of y are a synonym for x + 1. Notice that the slightly different formula

```
solve([y == x + 1, ForAll([y], Implies(y <= 0, x > y))])
```

has a solution where *x* is 1 and the free occurrence of *y* is 2.

Z3 supports also λ-binding with rudimentary reasoning support based on a model-constructing instantiation engine. λs may be convenient when expressing properties of arrays and Z3 uses array sorts for representing the sorts of lambda expressions. Thus, the result of memset is an array from integers to integers, that produces the value y in the range from lo to hi and otherwise behaves as m outside the range. Z3 reasons about quantifier free formulas that contains memset by instantiating the body of the λ.

```
m, m1 = Array('m', Z, Z), Array('m1', Z, Z)
def memset(lo, hi, y, m):
    return Lambda([x], If(And(lo <= x, x <= hi), y, Select(m, x)))
solve([m1 == memset(1, 700, z, m), Select(m1, 6) != z])
```

Lambda binding is convenient for creating closures. Recall that meaining of Lambda([x,y], e), where e is an expression with free occurrences of x and y is as a function that takes two arguments and substitutes their values for x and y in e. Z3 uses Lambda lifting, in conjunction with Reynold's defunctionalization, to reduce reasoning about closures to universally quantified definitions. Z3 treats arrays as general function spaces. All first-order definable functions may be arrays. Some second-order theorems can be established by synthesizing λ terms by instantiation. Thus,

```
Q = Array('Q', Z, B)
prove(Implies(ForAll(Q, Implies(Select(Q, x), Select(Q, y))),
              x == y))
```

is provable. Z3 synthesizes an instantiation corresponding to Lambda(z, z == x) for *Q*.

3. Theories

We will here summarize the main theories supported in Z3. In a few cases we will give a brief taste of decision procedures used for these theories. Readers who wish to gain a more in-depth understanding of how these decision procedures are implemented may follow some of the citations.

3.1. EUF: Equality and Uninterpreted Functions

The logic of *equality and uninterpreted function*, EUF, is a basic ingredient for first-order predicate logic. Before there are theories, there are constants, functions and predicate symbols, and the built-in relation of equality. In the following example, *f* is a unary function, *x* a constant. The first invocation of solve is feasible with a model where *x* is interpreted as an element in *S* and *f* is an identify function. The second invocation of *solve* is infeasible; there are no models where *f* maps *x* to anything but itself given the two previous equalities.

```
S = DeclareSort('S')
f = Function('f', S, S)
x = Const('x', S)
solve(f(f(x)) == x, f(f(f(x))) == x)
```

```
solve(f(f(x)) == x, f(f(f(x))) == x, f(x) != x)
```

Decision procedures for quantifier-free EUF formulas are usually based on *union-find* [59] to maintain equivalence classes of terms that are equated. Pictorially, a sequence of equality assertions $a = b, b = c, b = s$ produce one equivalence class that captures the transitivity of equality.

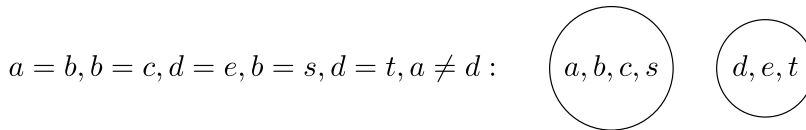$$a = b, b = c, d = e, b = s, d = t: \quad \boxed{a, b, c, s} \quad \boxed{d, e, t}$$
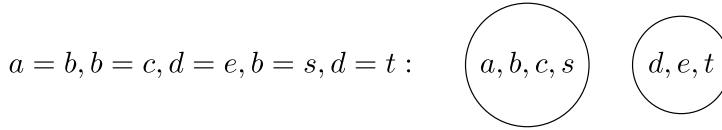
It is possible to check for satisfiability of disequalities by checking whether the equivalence classes associated with two disequal terms are the same or not. Thus, adding $a \neq d$ does not produce a contradiction, and it can be checked by comparing a's class representative with d's representative.

$$a = b, b = c, d = e, b = s, d = t, a \neq d: \quad \boxed{a, b, c, s} \quad \boxed{d, e, t}$$

On the other hand, when asserting $c \neq s$, we can deduce a conflict as the two terms asserted to be disequal belong to the same class. Class membership with union-find data-structures is amortized nearly constant time.

$$a = b, b = c, d = e, b = s, d = t, c \neq s: \quad \boxed{a, b, c, s} \quad \boxed{d, e, t}$$
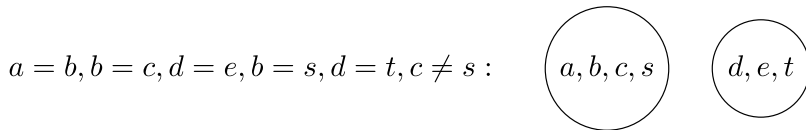
Union-find alone is insufficient when function symbols are used, as with the following example,

$$a = b, \ b = c, \ d = e, \ b = s, \ d = t, f(a, g(d)) \neq f(b, g(e))$$

In this case decision procedures require reasoning with the congruence rule

$$x_1 = y_1 \cdots x_n = y_n \ \Rightarrow \ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

As a preparation for solving our example, let us introduce constants that can be used as shorthands for sub-terms. Thus, introduce constants v_1, v_2, v_3, v_4 as representatives for the four compound sub-terms.

$$a = b, b = c, d = e, b = s, d = t, v_3 \neq v_4$$
$$v_1 := g(e), v_2 := g(d), v_3 := f(a, v_2), v_4 := f(b, v_1)$$

Having only the equality information available we obtain the equivalence classes:

$$\boxed{a, b, c, s} \quad \boxed{d, e, t} \quad \boxed{v_1} \quad \boxed{v_2} \quad \boxed{v_3} \quad \boxed{v_4}$$
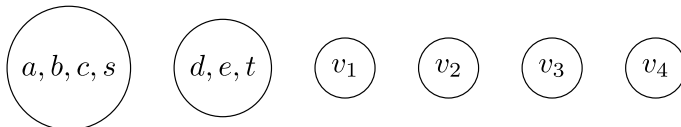
Working bottom-up, the congruence rule dictates that the classes for v_1 and v_2 should be merged. Thus,

$$e = d \ \Rightarrow \ g(e) = g(d)$$

implies the following coarser set of equivalences.
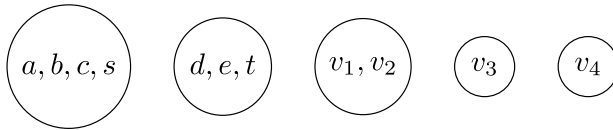
At this point, the congruence rule can be applied a second time,

$$a = b, v_2 = v_1 \Rightarrow f(a, v_2) = f(b, v_1)$$

producing the equivalence classes
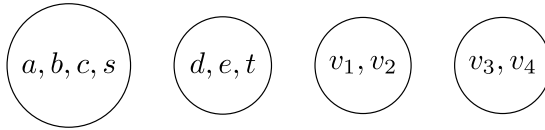
The classes for v_3 and v_4 are now merged. As our original formula required these to be distinct, congruence closure reasoning determines that the formula is unsatisfiable.

3.1.1. Congruence Closure

We have implicitly used the notion of *congruence closure* [27] to check satisfiability of equalities. Let us more formally define this notion. Let $\mathcal{T}$ be a set of terms and $\mathcal{E}$ set of equalities over $\mathcal{T}$. A *congruence closure* over $\mathcal{T}$ modulo $\mathcal{E}$ is the finest partition cc of $\mathcal{T}$, such that:

- if $(s = t) \in \mathcal{E}$, then s, t are in the same partition in cc.

- for $s := f(s_1, \ldots, s_k), t := f(t_1, \ldots, t_k) \in \mathcal{T}$,

    - if s_i, t_i are in same partition of cc for each $i = 1, \ldots k$,
    - then s, t are in the same partition under cc.

Definition 1.
$cc : \mathcal{T} \to 2^{\mathcal{T}}$, maps term to its equivalence class.

3.1.2. EUF Models

A satisfiable version of the running example is:

$$a = b, b = c, d = e, b = s, d = t, f(a, g(d)) \neq f(g(e), b)$$

It induces the following definitions and equalities: $a = b, b = c, d = e, b = s, d = t, v_3 \neq v_4$ $v_1 := g(e), v_2 := g(d), v_3 := f(a, v_2), v_4 := f(v_1, b)$ and we can associate a distinct value with each equivalence class.
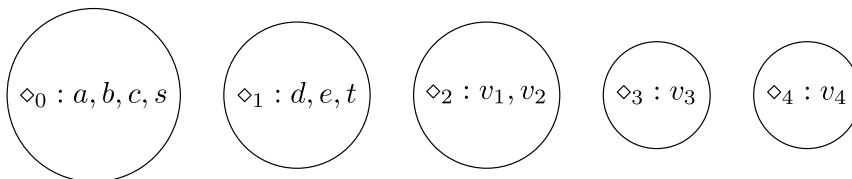
When presenting the formula to Z3 as

```
S = DeclareSort('S')
a, b, c, d, e, s, t = Consts('a b c d e s t', S)
f = Function('f', S, S, S)
g = Function('g', S, S)
solve([a == b, b == c, d == e, b == s,
       d == t, f(a, g(d)) != f(g(e), b)])
```

it produces a model, that may look as follows:

```
[s = S!val!0, b = S!val!0, a = S!val!0,
 c = S!val!0, d = S!val!1, e = S!val!1, t = S!val!1,
 f = [(S!val!2, S!val!0) -> S!val!4, else -> S!val!3],
 g = [else -> S!val!2]]
```

In the model the value `S!val!0` is a fresh constant that is distinct from `S!val!1`. The graph for `f` maps the arguments `(S!val!2, S!val!0)` to `S!val!4`. All other arguments are mapped by the `else` clause to `S!val!3`. The `else` clause is used as the default interpretation of arguments that are not listed in the interpretation. The interpretation of S is a finite set

```
{S!val!0, S!val!1, S!val!2, S!val!3, S!val!4}.
```

3.2. Arithmetic

Arithmetical constraints are nearly ubiquitous in software models. Even though mainstream software operates with finite precision arithmetic, that is modeled precisely using bit-vectors, arithmetic over unbounded integers can often be used in a sound way to model software. Furthermore, arithmetic over the reals has been used for diverse areas such as models of cyber-physical systems or for axiomatic economics.

3.2.1. Solving LRA: Linear Real Arithmetic

We provide an outline of Z3's main procedure for solving formulas over linear real arithmetic [28]. It maintains a (dual) Simplex tableau that encodes equalities of the form $Ax = 0$. Feasibility of the equalities depends on bounds, $lo_j \leq x_j \leq hi_j$, currently associated with the variables. For the following formula

```
x, y = Reals('x y')
solve([x >= 0, Or(x + y <= 2, x + 2*y >= 6),
               Or(x + y >= 2, x + 2*y > 4)])
```

Z3 introduces auxiliary variables s_1, s_2 and represents the formula as

$s_1 \equiv x + y, \ s_2 \equiv x + 2y,$
$x \geq 0, \ (s_1 \leq 2 \ \lor \ s_2 \geq 6), \ (s_1 \geq 2 \ \lor \ s_2 > 4)$

Only bounds (e.g., $s_1 \leq 2$) are asserted during search.

The first two equalities form the tableau. Thus, the definitions $s_1 \equiv x + y, s_2 \equiv x + 2y$ produce the equalities

$$s_1 = x + y, \ s_2 = x + 2y$$

They are equivalent to the normal form:

$$s_1 - x - y = 0, \ s_2 - x - 2y = 0$$

where s_1, s_2 are basic (dependent) and x, y are non-basic. In dual Simplex tableaux, values of a non-basic variable x_j can be chosen between lo_j and hi_j. The value of a basic variable is a function of non-basic variable values. It is the unique value that satisfies the unique row where the basic variable occurs. Pivoting swaps basic and non-basic variables and is used to get values of basic variables within bounds. For example, assume we start with a set of initial values $x = y = s_1 = s_2 = 0$ and bounds $x \geq 0, s_1 \leq 2, s_1 \geq 2$. Then s_1 has to be 2 and it is made non-basic. Instead y becomes basic:

$$y + x - s_1 = 0, \ s_2 + x - 2s_1 = 0$$

The new tableau updates the assignment of variables to $x = 0, s_1 = 2, s_2 = 4, y = 2$. The resulting assignment is a model for the original formula.

3.2.2. Solving Arithmetical Fragments

The solvers available to reason about arithmetical constraints are wildly different depending on what fragments of arithmetic is used. We summarize the main fragments, available decision procedures, and examples in Table 1 where x, y range over reals and a, b range over integers.

Logic	Description	Solver	Example
LRA	Linear Real Arithmetic	Dual Simplex [28]	$x + \frac{1}{2}y \leq 3$
LIA	Linear Integer Arithmetic	Cuts + Branch	$a + 3b \leq 3$
LIRA	Mixed Real/Integer	[7, 12, 14, 26, 28]	$x + a \geq 4$
IDL	Integer Difference Logic	Floyd-Warshall	$a - b \leq 4$
RDL	Real Difference Logic	Bellman-Ford	$x - y \leq 4$
UTVPI	Unit two-variable / inequality	Bellman-Ford	$x + y \leq 4$
NRA	Polynomial Real Arithmetic	Model based CAD [42]	$x^2 + y^2 < 1$
NIA	Non-linear Integer Arithmetic	CAD + Branch [41] Linearization [15]	$a^2 = 2$

Table 1. Arithmetic Theories

There are many more fragments of arithmetic that benefit from specialized solvers. We later discuss some of the fragments where integer variables are restricted to the values $\{0, 1\}$ when describing Pseudo-Boolean constraints. Other fragments that are *not* currently handled in Z3 in any special way include fragments listed in Table 2.

Description	Example
Horn Linear Real Arithmetic At most one variable is positive	$3y + z - \frac{1}{2}x \leq 1$
Two-variable per inequality [16]	$3x + 2y \geq 1$
Min-Horn [18]	$x \geq \min(2y + 1, z)$
Bi-linear arithmetic	$3xx' + 2yy' \geq 2$
Transcendental functions	$e^{-x} \geq y$
Modular linear arithmetic	$a + 3b + 2 \equiv 0 \mod 5$

Table 2. Fragments of Arithmetic

A user of Z3 may appreciate that a domain can be modeled using a fragment of the theory of arithmetic that is already supported, or belongs to a class where no special support is available. On a practical side, it is worth noting that Z3 uses infinite precision arithmetic by default. Thus, integers and rationals are represented without rounding. The benefit is that the representation ensures soundness of the results, but operations by decision procedures may end up producing large numerals taking most of the execution time. Thus, users who produce linear arithmetic constraints with large coefficients or long decimal expansions may face performance barriers.

3.3. Arrays

The declaration

```
A = Array('A', IntSort(), IntSort())
```

introduces a constant `A` of the array sort mapping integers to integers. We can solve constraints over arrays, such as

```
solve(A[x] == x, Store(A, x, y) == A)
```

which produces a solution where x necessarily equals y.

Z3 treats arrays as function spaces, thus a function `f(x, y)` can be converted to an array using a λ

```
Lambda([x, y], f(x, y))
```

If `f` has sort $A \times B \to C$, then `Lambda([x, y], f(x, y))` has sort `Array(A, B, C)`. A set of built-in functions are available for arrays. We summarize them together with their representation using `Lambda` bindings.

```
a[i]              # select array 'a' at index 'i'
                  # Select(a, i)

Store(a, i, v)    # update array 'a' with value 'v' at index 'i'
                  # = Lambda(j, If(i == j, v, a[j]))

K(D, v)           # constant Array(D, R), where R is sort of 'v'.
                  # = Lambda(j, v)

Map(f, a)         # map function 'f' on values of 'a'
                  # = Lambda(j, f(a[j]))

Ext(a, b)         # Extensionality
                  # Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b)
```

3.3.1. Deciding Arrays by Reduction to EUF

Formulas using the combinators `Store`, `K`, `Map`, `Ext` are checked for satisfiability by *expanding* the respective λ definitions on sub-terms. We illustrate how occurrences of `Store` produce constraints over EUF. In the following, assume we are given a solver `s` with ground assertions using arrays.

For each occurrence in `s` of `Store(a, i, v)` and `b[j]`, add the following assertions:

- `s.add(Store(a, i, v)[j] == If(i == j, v, a[j]))`
- `s.add(Store(a, i, v)[i] == v)`

The theory of arrays is *extensional*. That is, two arrays are equal if they behave the same on all selected indices. When Z3 produces models for quantifier free formulas in the theory of extensional arrays it ensures that two arrays are equal in a model whenever they behave the same on all indices. Extensionality is enforced on array terms a, b in `s` by instantiating the axiom of extensionality.

- `s.add(Implies(ForAll(i, a[i] == b[i]), a == b))`

Since the universal quantifier occurs in a negative polarity we can introduce a Skolem function `Ext` that depends on `a` and `b` and represent the extensionality requirement as:

- `s.add(Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b))`

We can convince ourselves that asserting these additional constraints force models of a solver `s` to satisfy the array axioms. Suppose we are given a model M satisfying all the additional asserted equalities. These equalities enforce the axioms for `Store` on all indices that occur in `s`. They also enforce extensionality between arrays: Two arrays are equal if and only if they evaluate to the same value on all indices in `s`.

3.4. Bit-Vectors

Let us play with some bit-fiddling. The resource

lists a substantial repertoire of bit-vector operations that can be used as alternatives to potentially more expensive operations. Note that modern compilers already contain a vast set of optimizations that automatically perform these conversions and Z3 can be used to check and synthesize such optimizations [46]. For example, to test whether a bit-vector is a power of two we can use a combination of bit-wise operations and subtraction:

```
def is_power_of_two(x):
    return And(x != 0, 0 == (x & (x - 1)))
x = BitVec('x', 4)
prove(is_power_of_two(x) == Or([x == 2**i for i in range(4)]))
```

The absolute value of a variable can be obtained using addition and xor with a sign bit.

```
v = BitVec('v',32)
mask = v >> 31
prove(If(v > 0, v, -v) == (v + mask) ^ mask)
```

Notice that the Python variable mask corresponds to the expression v >> 31, the right arithmetic (signed) shift of v. Notice also, that in classical first-order logic, all operations are *total*. In particular, for bit-vector arithmetic -v is fully specified, in contrast to, say C, which specifies that -v is *undefined* when v is a signed integer with the value -2^{31}.

3.4.1. Solving bit-vectors

Z3 mostly uses a bit-blasting approach to deciding bit-vectors. By bit-blasting we refer to a reduction of bit-vector constraints to propositional logic by treating each bit in a bit-vector as a propositional variable. Let us illustrate how bit-vector addition is compiled to a set of clauses. The expression v + w, where v and w are bit-vectors is represented by a vector out of output bits. The relation between v, w and out is provided by clauses the encode a ripple-carry adder seen in Figure 2. The encoding uses an auxiliary vector of *carry bits* that are internal to the adder.
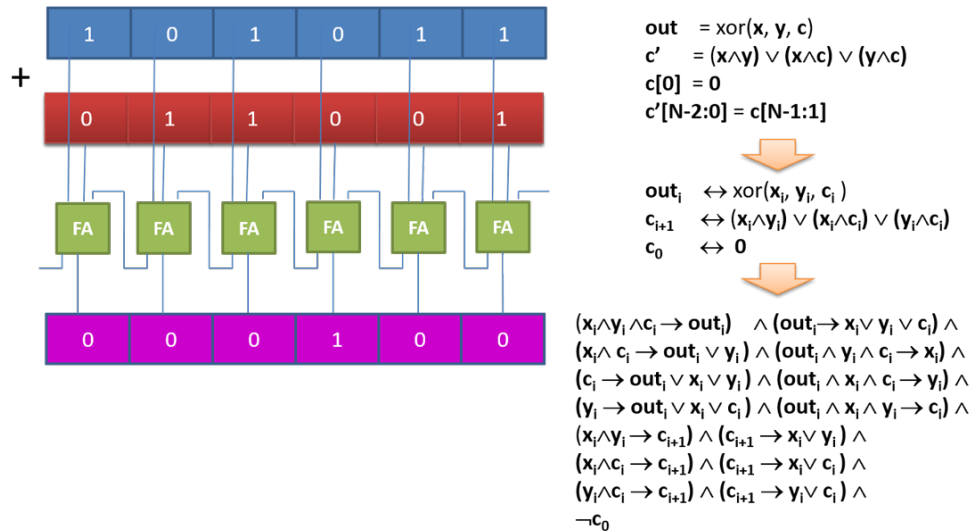

Figure 2. Bit-vector addition circuit

3.4.2. Floating Point Arithmetic

Floating points are bit-vectors with an interpretation specified by the IEEE floating point

standard.

```
x = FP('x', FPSort(3, 4))
print(10 + x)
```

It declares a floating point number x with 3 bits in the exponent and 4 for the significand. The result of adding 10 to x is $1.25*(2**3) + x$. We see that 10 is represented as a floating point number with exponent 3, that is the bit-vector 011. The significand is 1010.

3.5. Algebraic Datatypes

The theory of first-order algebraic data-types captures the theory of finite trees. It is characterized by the properties that:

- All trees are finite (occurs check).

- All trees are generated from the constructors (no junk).

- Two trees are equal if and only if they are constructed exactly the same way (no confusion).

A basic example of a binary tree data-type is given in Figure 3.

```
Tree = Datatype('Tree')
Tree.declare('Empty')
Tree.declare('Node', ('left', Tree), ('data', Z), ('right', Tree))
Tree = Tree.create()
t = Const('t', Tree)
solve(t != Tree.Empty)
```

Figure 3. Binary tree datatypes

It may produce the solution

```
[t = Node(Empty, 0, Empty)]
```

Similarly, one can prove that a tree cannot be a part of itself.

```
prove(t != Tree.Node(t, 0, t))
```

3.6. Sequences and Strings

The theory of strings and sequences extend on the theory of the free monoid with a few additional functions that are useful for strings and sequences. A length operation is built-in for strings and sequences, and there are operations for converting strings to natural numbers and back.

If the lengths of a prefix and suffix of a string add up to the length of the string, the string itself must be the concatenation of the prefix and suffix:

```
s, t, u = Strings('s t u')
prove(Implies(And(PrefixOf(s, t), SuffixOf(u, t),
                  Length(t) == Length(s) + Length(u)),
          t == Concat(s, u)))
```

One can concatenate single elements to a sequence as units:

```
s, t = Consts('s t', SeqSort(IntSort()))
solve(Concat(s, Unit(IntVal(2))) == Concat(Unit(IntVal(1)), t))
prove(Concat(s, Unit(IntVal(2))) != Concat(Unit(IntVal(1)), s))
```

There are two solvers available in Z3 for strings. They can be exchanged by setting the parameter

- `s.set("smt.string.solver","seq")` with contributions by Thai Trinh, or
- `s.set("smt.string.solver","z3str3")` by Murphy Berzish.

3.7. Special Relations

In some cases it is possible to use first-order axioms to capture all required properties of relations. For example, to say that `R` is a partial order it suffices to assert the axioms:

```
s = Solver()
A = DeclareSort()
R = Function('R', A, A, B)
x, y, z = Consts('x y z', A)
s.Add(ForAll([x], R(x, x)))
s.Add(ForAll([x,y], Implies(And(R(x, y), R(y, x)), x == y)))
s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(y, z)), R(x, z))))
# and other assertions using R
```

The catch is that reasoning about `R` often requires instantiating the full transitive closure of the relation. This is impractical when the number of instances can lead to a quadratic overhead compared to the size of the initial constraints. For example, when asserting `s.add(R(a1,a2),R(a2,a3),...,R(a999,a1000))` we obtain half a million instances. Instead of axiomatizing a relation to be a partial ordering, Z3 allows to declare a relation that as a built-in property satisfies the partial ordering axioms. Thus, the definition

```
R = PartialOrder(A, 0)
```

creates a binary relation `R` over `A` that satisfies the partial order axioms. The second argument is an unsigned integer used to distinguish the partial order `R` from other partial order relations over the same signature. Thus `PartialOrder(A, 2)` creates a partial ordering relation that is different from `R` because it uses a different index. As usual, other properties of the relation have to be added to the solver `s`. The built-in decision procedure for partial orders avoids the quadratic instantiation from the transitive closure axiom. It reasons about graph reachability to check consistency of assertions involving `R`.

Other binary relations that have special relations support are Total Linear orders:

```
s.Add(ForAll([x], R(x, x)))
s.Add(ForAll([x,y], Implies(And(R(x, y), R(y, x)), x == y)))
s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(y, z)), R(x, z))))
s.Add(ForAll([x,y], Or(R(x, y), R(y, x))))

# use instead:

R = TotalLinearOrder(A, 0)
```

Tree-like orderings:

```
s.Add(ForAll([x], R(x, x)))
s.Add(ForAll([x,y], Implies(And(R(x, y), R(y, x)), x == y)))
s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(y, z)), R(x, z))))
s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(x, z)), Or(R(y, z), R(z, y)))))

# use instead:

R = TreeOrder(A, 0)
```

Piecewise Linear orders:

```
s.Add(ForAll([x], R(x, x)))
s.Add(ForAll([x,y], Implies(And(R(x, y), R(y, x)), x == y)))
s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(y, z)), R(x, z))))
```

```
    s.Add(ForAll([x,y,z], Implies(And(R(x, y), R(x, z)), Or(R(y, z), R(z, y)))))
    s.Add(ForAll([x,y,z], Implies(And(R(y, x), R(z, x)), Or(R(y, z), R(z, y)))))

# use instead:
    R = PiecewiseLinearOrder(A, 0)
```

Additional examples of special relations constraints are [available online](#).

3.8. Transitive Closure

The transitive closure of a relation is a property that cannot be fully axiomatized using first-order formalisms. Quantifier-free formulas using the transitive closure of relations remain decidable, however, using a finite model construction. Given a relation binary `R`, the *transitive closure* of R is another relation `TC_R` that relates two elements by if there is a non-empty path that connect them through R. The *transitive-reflexive closure* extends transitive closure by also relating two elements if they are equal. To create a transitive closure or transitive reflexive closure of R.

```
    R = Function('R', A, A, B)
    TC_R = TransitiveClosure(R)
    TRC_R = TransitiveReflexiveClosure(R)
    s = Solver()
    a, b, c = Consts('a b c', A)
    s.add(R(a, b))
    s.add(R(b, c))
    s.add(Not(TC_R(a, c)))
    print(s.check())    # produces unsat
```

4. Interfacing with Solvers

Solvers maintain a set of formulas and supports satisfiability checking, and scope management: Formulas that are added under one scope can be retracted when the scope is popped. In this section we describe the interface to solvers. Section 5 provides a set of use cases and section 6 describes the underlying solver implementations available in Z3.

4.1. Incrementality

Solvers can be used to check satisfiability of assertions in an incremental way. An initial set of assertions can be checked for satisiability followed by additional assertions and checks. Assertions can be retracted using scopes that are pushed and popped. Under the hood, Z3 uses a one-shot solver during the first check. If further calls are made into the solver, the default behavior is to switch to an *incremental* solver. The incremental solver uses the SMT core, see 6.1, by default. For use-cases that don't require all features by the SMT core, it may be beneficiary to use specialized solvers, such as solvers for finite domains (bit-vectors, enumeration types, bounded integers, and Booleans) as specified using the `QF_FD` logic.

4.2. Scopes

The operations `push` and `pop` create, respectively revert, local scopes. Assertions that are added within a `push` are retracted on a matching `pop`. Thus, the following session results in the verdicts `sat`, `unsat`, and `sat`.

```
p, q, r = Bools('p q r')
s = Solver()
s.add(Implies(p, q))
s.add(Not(q))
print(s.check())
s.push()
s.add(p)
print(s.check())
s.pop()
```

```
print(s.check())
```

4.3. Assumptions

Alternative to scopes, it is possible to check satisfiability under the assumption of a set of literals. Thus, the session

```
s.add(Implies(p, q))
s.add(Not(q))
print(s.check(p))
```

also produces the verdict unsat as the conjunction of $p \rightarrow q$, $\neg q$, p is unsat. The method assert_and_track(q, p) has the same effect of adding Implies(p, q), and it adds p as an implicit assumption. Our running example becomes

```
p, q = Bools('p q')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, p)
print(s.check())
```

4.4. Cores

We can extract a subset of assumptions used to derive unsatisfiability. Such subsets of assumptions are known as *unsatisfiable cores*, or simply as a *core*. In the following example, the unsatisfiable core has the single element p. The unrelated assumption v does not appear in the core.

```
p, q, r, v = Bools('p q r v')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, p)
s.assert_and_track(r, v)
print(s.check())
print(s.unsat_core())
```

Note that we invoke s.check() prior to extracting a core. Cores are only available after the last call to s.check() produced unsat.

By default solvers do not return minimal cores. A core is *minimal* if there is no proper subset that is also a core. The default behavior can be changed when the solver corresponds to either the SMT Core or SAT Core (if the underlying solver is created from a sequence of pre-processing tactics, core minimization is not guaranteed to take effect). To force core minimization users can rely on setting the following parameters:

```
def set_core_minimize(s):
    s.set("sat.core.minimize","true")  # For Bit-vector theories
    s.set("smt.core.minimize","true")  # For general SMT
```

4.5. Models

When s.check() returns sat Z3 can provide a model that assigns values to the free constants and functions in the assertions. The current model is accessed using s.model() and it offers access to an interpretation of the active assertions in s. Consider the example:

```
f = Function('f', Z, Z)
x, y = Ints('x y')
s.add(f(x) > y, f(f(y)) == y)
print(s.check())
print(s.model())
```

A possible model for s is:

```
[y = 0, x = 2, f = [0 -> 3, 3 -> 0, else -> 1]]
```

You can access models. They have a set of entries. Each entry maps a constant or function declaration (constants are treated as nullary functions) to an interpretation. It maps constants to a constant expression and it maps functions to a *function interpretation*. The stub

```
m = s.model()
for d in m:
    print(d, m[d])
```

iterates over the assignments in a model and produces the output

```
y 0
x 2
f [0 -> 3, 3 -> 0, else -> 1]
```

Function interpretations comprise a set of entries that specify how the function behaves on selected argument combinations, and a *else_value* that covers arguments not listed in the entries.

```
num_entries = m[f].num_entries()
for i in range(num_entries):
    print(m[f].entry(i))
print("else", m[f].else_value())
```

It produces the output

```
[0, 3]
[3, 0]
else 1
```

The easiest way to access a model is to use the `eval` method that lets you evaluate arbitrary expressions over a model. It reduces expressions to a constant that is consistent with the way the model interprets the constants and functions. For our model from above

```
print(m.eval(x), m.eval(f(3)), m.eval(f(4)))
```

produces the output 2, 0, 1.

4.6. Other Methods

4.6.1. Statistics

You can gain a sneak peak at what the solver did by extracting statistics. The call

```
print(s.statistics())
```

displays values of internal counters maintained by the decision procedures. They are mostly valuable when coupled with a detailed understanding of how the decision procedures work, but may be used as an introductory view into the characteristics of a search.

4.6.2. Proofs

Proof objects, that follow a natural deduction style, are available from the Solver interface [20]. You have to enable proof production at top level in order to retrieve proofs.

```
s.set("produce-proofs", True)
s.add(φ)
assert unsat == s.check()
print(s.proof())
```

The granularity of proof objects is on a best-effort basis. Proofs for the SMT Core, are relatively fined-grained, while proofs for procedures that perform quantifier elimination, for instance QSAT

described in Section 6.4, are exposed as big opaque steps.

4.6.3. Retrieving Solver State

You can retrieve the current set of assertions in a solver using `s.assertions()`, the set of unit literals using `s.units()` and literals that are non-units using `s.non_units()`. The solver state can be printed to SMT-LIB2 format using `s.sexpr()`.

4.6.4. Cloning Solver State

The method `s.translate(ctx)` clones the solver state into a new solver based on the context that is passed in. It is useful for creating separate non-interferring states of a solver.

4.6.5. Loading formulas

The methods `s.from_file` and `s.from_string` adds constraints to a solver state from a file or string. Files are by default assumed to be in the SMT2 format. If a file name ends with `dimacs` they are assumed to be in the DIMACS propositional format.

4.6.6. Consequences

Product configuration systems use constraints to describe the space of all legal configurations. As parameters get fixed, fewer and fewer configuration options are available. For instance, once the model of a car has been fixed, some options for wheel sizes become unavailable. It is furthermore possible that only one option is available for some configurations, once some parameters are fixed. Z3 can be used to answer queries of the form: Given a configuration space of values V, when fixing values $V_0 \subseteq V$, what is the largest subset $V_0 \subseteq V_1 \subseteq V$ of values that become fixed? Furthermore, for some value v_1 that is fixed, provide an explanation, in terms of the values that were fixed in V_0, for why v_1 got fixed. The functionality is available through the `consequences` method.

```
a, b, c, d = Bools('a b c d')

s = Solver()
s.add(Implies(a, b), Implies(c, d))    # background formula
print(s.consequences([a, c],           # assumptions
                      [b, c, d]))       # what is implied?
```

produces the result:

```
(sat, [Implies(c, c), Implies(a, b), Implies(c, d)])
```

In terms for SAT terminology, consequence finding produces the set of all *backbone* literals. It is useful for finding fixed parameters [37] in product configuration settings.

Z3 relies on a procedure that integrates tightly with the CDCL, *Conflict Driven Clause Learning* [58], algorithm, and it contains two implementations of the procedure, one in the SAT core, another in the SMT core. Section 6.1.1 expands on CDCL and integrations with theories.

4.6.7. Cubes

You can ask Z3 to suggest a case split or a sequence of case splits through the cubing method. It can be used for partitioning the search space into sub-problems that can be solved in parallel, or alternatively simplify the problem for CDCL engines.

```
s = SolverFor("QF_FD")
s.add(F)
s.set("sat.restart.max", 100)
def cube_and_conquer(s):
```

```
for cube in s.cube():
    if len(cube) == 0:
        return unknown
    if is_true(cube[0]):
        return sat
    is_sat = s.check(cube):
    if is_sat == unknown:
        s1 = s.translate(s.ctx)
        s1.add(cube)
        is_sat = cube_and_conquer(s1)
    if is_sat != unsat:
        return is_sat
return unsat
```

Figure 4. Basic cube and conquer

When the underlying solver is based on the SAT Core, see Section 6.2, it uses a lookahead solver to select cubes [31]. By default, the cuber produces two branches, corresponding to a case split on a single literal. The SAT Core based cuber can be configured to produce cubes that represent several branches. An empty cube indicates a failure, such as the solver does not support cubing (only the SMT and SAT cores support cubing, and generic solvers based on tactics do not), or a timeout or resource bound was encountered during cubing. A cube comprising of the Boolean constant `true` indicates that the state of the solver is satisfiable. Finally, it is possible for the `s.cube()` method to return an empty set of cubes. This happens when the state of s is unsatisfiable. Each branch is represented as a conjunction of literals. The cut-off for branches is configured using

- `sat.lookahead.cube.cutoff`

We summarize some of the configuration parameters that depend on the value of `cutoff` in Table 3.

sat.lookahead		used when cube.cutoff is	Description
cube.depth	1	depth	A fixed maximal size of cubes is returned
cube.freevars	0.8	freevars	the depth of cubes is governed by the ratio of non-unit literals in a branch compared to non-unit variables in the root
cube.fraction	0.4	adaptive_freevars adaptive_psat	adaptive fraction to create lookahead cubes
cube.psat. clause_base	2	psat	Base of exponent used for clause weight

Table 3. Lookahead parameters

Heuristics used to control which literal is selected in cubes can be configured using the parameter:

- `sat.lookahead.reward`

5. Using Solvers

We now describe a collection of algorithms. They are developed on top of the interfaces described in the previous section.

5.1. Blocking evaluations

Models can be used to refine the state of a solver. For example, we may wish to invoke the solver in a loop where new calls to the solver blocks solutions that evaluate the constants to the exact same assignment.

```
def block_model(s):
    m = s.model()
    s.add(Or([ f() != m[f] for f in m.decls() if f.arity() == 0]))
```

5.2. Maximizing Satisfying Assignments

Another use of models is to use them as a guide to a notion of optimal model. A *maximal satisfying solution*, in short *mss*, for a set of formulas ps is a subset of ps that is consistent with respect to the solver state s and cannot be extended to a bigger subset of ps without becoming inconsistent relative to s. We provide a procedure, from [48], for finding a maximal satisfying subset in Figure 5. It extends a set mss greedily by adding as many satisfied predicates from ps in each round as possible. If it finds some predicate p that it cannot add, it notes that it is a backbone with respect to the current mss. As a friendly hint, it includes the negation of p when querying the solver in future rounds.

```
def tt(s, f):
    return is_true(s.model().eval(f))

def get_mss(s, ps):
    if sat != s.check():
       return []
    mss = { q for q in ps if tt(s, q) }
    return get_mss(s, mss, ps)

def get_mss(s, mss, ps):
    ps = ps - mss
    backbones = set([])
    while len(ps) > 0:
       p = ps.pop()
       if sat == s.check(mss | backbones | { p }):
          mss = mss | { p } | { q for q in ps if tt(s, q) }
          ps  = ps - mss
       else:
          backbones = backbones | { Not(p) }
    return mss
```

Figure 5. An algorithm for computing maximal satisfying subsets

Exercise 5a:
Suppose ps is a list corresponding to digits in a binary number and ps is ordered by most significant digit down. The goal is to find an mss with the largest value as a binary number. Modify get_mss to produce such a number.

5.3. All Cores and Correction Sets

The Marco procedure [45] combines models and cores in a process that enumerates all unsatisfiable cores and all maximal satisfying subsets of a set of formulas ps with respect to solver s. It maintains a map solver that tells us which subsets of ps are not yet known to be a superset of a core or a subset of an mss.

```
def ff(s, p):
    return is_false(s.model().eval(p))
```

```
def marco(s, ps):
    map = Solver()
    set_core_minimize(s)
    while map.check() == sat:
        seed = {p for p in ps if not ff(map, p)}
        if s.check(seed) == sat:
            mss = get_mss(s, seed, ps)
            map.add(Or(ps - mss))
            yield "MSS", mss
        else:
            mus = s.unsat_core()
            map.add(Not(And(mus)))
            yield "MUS", mus
```

Figure 6. The [MARCO](#) algorithm for computing cores and maximal satisfying assignments

Efficiently enumerating cores and correction sets is an active area of research. Many significant improvements have been developed over the above basic implementation [1–3, 48, 51, 55].

5.4. Bounded Model Checking

Figure 7 illustrates a bounded model checking procedure [5] that takes a transition system as input and checks if a goal is reachable. Transition systems are described as

$$\langle Init, Trans, Goal, \mathcal{V}, \mathcal{Y} \rangle$$

where $Init$ is a predicate over $\mathcal{V}$, that describes the initial states, $Trans$ is a transition relation over $\mathcal{V} \times \mathcal{Y} \times \mathcal{V}'$. The set of reachable states is the set inductively defined as valuations s of $\mathcal{V}$, such that either $s \models Init$ or there is a reachable s_0 and values v for $\mathcal{Y}$, such that $s_0, v, s \models Trans$. A goal is reachable if there is some reachable state where $s \models Goal$.

In Python we provide the initial condition as init, using variables xs, the transition trans that uses variables xs, xns, fvs, and goal using variables xs. Bounded model checking unfolds the transition relation trans until it can establish that the goal is reachable. Bounded model checking diverges if goal is unreachable. The function substitute(e, subst) takes an expression e and a list of pairs subst of the form [(x1, y1), (x2, y2),..] and replaces variables x1, x2,.. by y1, y2,.. in e.

```
index = 0
def fresh(s):
    global index
    index += 1
    return Const("!f%d" % index, s)

def zipp(xs, ys):
    return [p for p in zip(xs, ys)]

def bmc(init, trans, goal, fvs, xs, xns):
    s = Solver()
    s.add(init)
    count = 0
    while True:
        print("iteration ", count)
        count += 1
        p = fresh(BoolSort())
        s.add(Implies(p, goal))
        if sat == s.check(p):
            print (s.model())
            return
        s.add(trans)
        ys = [fresh(x.sort()) for x in xs]
        nfvs = [fresh(x.sort()) for x in fvs]
```

```
            trans = substitute(trans,
                               zipp(xns + xs + fvs, ys + xns + nfvs))
            goal = substitute(goal, zipp(xs, xns))
            xs, xns, fvs = xns, ys, nfvs
```

Figure 7. [Bounded Model Checking](#) of a transition system

Example 1.

Let us check whether there is some k, such that $\underbrace{3 + 3 + \ldots + 3}_{k} = 10$ when numbers are represented using 4 bits. The corresponding transition system uses a state variable x0 which is named x1 in the next state. Initially x0 == 0 and in each step the variable is incremented by 3. The goal state is x0 == 10.

```
x0, x1 = Consts('x0 x1', BitVecSort(4))
bmc(x0 == 0, x1 == x0 + 3, x0 == 10, [], [x0], [x1])
```

Bounded model checking is good for establishing reachability, but does not produce certificates for non-reachability (or safety). The IC3 [10] algorithm is complete for both reachability and non-reachability. You can find a simplistic implementation of IC3 using the Python API online

https://github.com/Z3Prover/z3/blob/master/examples/python/mini_ic3.py

5.5. Propositional Interpolation

It is possible to compute interpolants using models and cores [13]. A procedure that computes an interpolant I for formulas A, B, where $A \wedge B$ is unsatisfiable proceeds by initializing $I = true$ and saturating a state $\lceil A, B, I \rceil$ with respect to the rules:

$$\lceil A,\ B,\ I \rceil \qquad \implies \qquad \lceil A,\ B,\ I \wedge \neg L \rceil \qquad \text{if } B \vdash \neg L,\ A \wedge I \nvdash \neg L$$

$$I \qquad\qquad\qquad \text{if } A \vdash \neg I$$

The partial interpolant I produced by pogo satisfies $B \vdash I$. It terminates when $A \vdash \neg I$. The condition $A \wedge I \nvdash \neg L$ ensures that the algorithm makes progress and suggests using an implicant $L' \supseteq L$ of $A \wedge I$ in each iteration. Such an implicant can be obtained from a model for $A \wedge I$.

```
def mk_lit(m, x):
    if is_true(m.eval(x)):
        return x
    else:
        return Not(x)

def pogo(A, B, xs):
    while sat == A.check():
        m = A.model()
        L = [mk_lit(m, x) for x in xs]
        if unsat == B.check(L):
            notL = Not(And(B.unsat_core()))
            yield notL
            A.add(notL)
        else:
            print("expecting unsat")
            break
```

Figure 8. [Propositional interpolation](#)

Example 2.

The (reverse) interpolant between $A : x_1 = a_1 \neq a_2 \neq x_2$ and $B : x_1 = b_1 \neq b_2 = x_2$ using

vocabulary x_1, x_2 is $x_1 \neq x_2$. It is implied by B and inconsistent with A.

```
A = SolverFor("QF_FD")
B = SolverFor("QF_FD")
a1, a2, b1, b2, x1, x2 = Bools('a1 a2 b1 b2 x1 x2')
A.add(a1 == x1, a2 != a1, a2 != x2)
B.add(b1 == x1, b2 != b1, b2 == x2)
print(list(pogo(A, B, [x1, x2])))
```

5.6. Monadic Decomposition

Suppose we are given a formula $\varphi[x, y]$ using variables x and y. When is it possible to rewrite it as a Boolean combination of formulas $\psi_1(x), \ldots, \psi_k(x)$ and $\theta_1(y), \ldots, \theta_n(y)$? We say that the formulas ψ_j and θ_j are *monadic*; they only depend on one variable. An application of monadic decomposition is to convert *extended* symbolic finite transducers into *regular* symbolic finite transducers. The regular versions are amenable to analysis and optimization. A procedure for monadic decomposition was developed in [60], and we here recall the Python prototype.

```
from z3 import *
def nu_ab(R, x, y, a, b):
    x_ = [ Const("x_%d" %i, x[i].sort()) for i in range(len(x))]
    y_ = [ Const("y_%d" %i, y[i].sort()) for i in range(len(y))]
    return Or(Exists(y_, R(x+y_) != R(a+y_)), Exists(x_, R(x_+y) != R(x_+b)))

def isUnsat(fml):
    s = Solver(); s.add(fml); return unsat == s.check()

def lastSat(s, m, fmls):
    if len(fmls) == 0: return m
    s.push(); s.add(fmls[0])
    if s.check() == sat: m = lastSat(s, s.model(), fmls[1:])
    s.pop(); return m

def mondec(R, variables):
    print(variables)
    phi = R(variables);
    if len(variables)==1: return phi
    l = int(len(variables)/2)
    x, y = variables[0:l], variables[l:]
    def dec(nu, pi):
        if isUnsat(And(pi, phi)):
            return BoolVal(False)
        if isUnsat(And(pi, Not(phi))):
            return BoolVal(True)
        fmls = [BoolVal(True), phi, pi]
        #try to extend nu
        m = lastSat(nu, None, fmls)
        #nu must be consistent
        assert(m != None)
        a = [ m.evaluate(z, True) for z in x ]
        b = [ m.evaluate(z, True) for z in y ]
        psi_ab = And(R(a+y), R(x+b))
        phi_a = mondec(lambda z: R(a+z), y)
        phi_b = mondec(lambda z: R(z+b), x)
        nu.push()
        #exclude: x~a and y~b
        nu.add(nu_ab(R, x, y, a, b))
        t = dec(nu, And(pi, psi_ab))
        f = dec(nu, And(pi, Not(psi_ab)))
        nu.pop()
        return If(And(phi_a, phi_b), t, f)
    #nu is initially true
    return dec(Solver(), BoolVal(True))
```

Figure 9. Monadic Decomposition

Example 3.

A formula that has a monadic decomposition is the bit-vector assertion for x, y being bit-vectors of bit-width $2k$.

$$y > 0 \land (y\&(y-1)) = 0 \land (x\&(y\%((1 << k) - 1))) \neq 0$$

We can compute the monadic decomposition

```
def test_mondec(k):
    R = lambda v:And(v[1] > 0, (v[1] & (v[1] - 1)) == 0,
                     (v[0] & (v[1] % ((1 << k) - 1))) != 0)
    bvs = BitVecSort(2*k)                          #use 2k-bit bitvectors
    x, y = Consts('x y', bvs)
    res = mondec(R, [x, y])
    assert(isUnsat(res != R([x, y])))             #check correctness
    print("mondec1(", R([x, y]), ") =", res)
test_mondec(2)
```

6. Solver Implementations

There are five main solvers embedded in Z3. The SMT Solver is a general purpose solver that covers a wide range of supported theories. It is supplemented with specialized solvers for SAT formulas, polynomial arithmetic, Horn clauses and quantified formulas over theories that admit quantifier-elimination.

6.1. SMT Core

The SMT Solver is a general purpose solver that covers a wide range of supported theories. It is built around a CDCL(T) architecture where theory solvers interact with a SAT + EUF blackboard. Theory solvers, on the right in Figure 10, communicate with a core that exchanges equalities between variables and assignments to atomic predicates. The core is responsible for case splitting, which is handled by a CDCL SAT solver, and for letting each theory learn constraints and equalities that are relevant in the current branch.
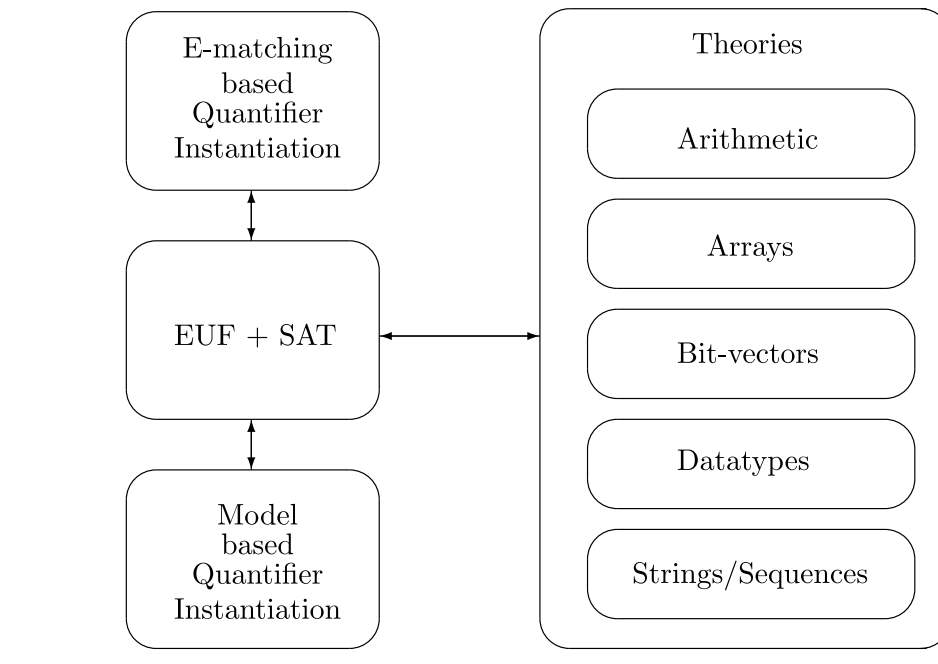
Figure 10. Architecture of Z3's SMT Core solver.

To force using the SMT solver a user can create a *simple solver* using the function `SimpleSolver`.

The SMT solver integrates two strategies for quantifier instantiation. By default, both strategies are enabled. To disable them, one has to disable automatic configuration mode and then disable the instantiation strategy:

```
s.set("smt.auto_config", False)    # disable automatic SMT core
                                   # configuration
s.set("smt.mbqi", False)           # disable model based
                                   # quantifier instantiation
s.set("smt.ematching", False)      # disable ematching based
                                   # quantifier instantiation
```

6.1.1. CDCL(T): SAT + Theories

The architecture of mainstream SMT solvers, including Z3's SMT core, uses a SAT solver to enumerate combinations of truth assignments to atoms. The truth assignments satisfy a propositional abstraction of the formula. Theory solvers are used to check if assignment admit a model modulo the theories. The resulting architecture is known as DPLL(T) [54], but we refer to this as CDCL(T) because it really relies on SAT solvers that incorporate *Conflict Driven Clause Learning* [58], which goes beyond the algorithm associated with DPLL [19]. Importantly, CDCL supplies facilities for learning new clauses during search. The learned clauses block future case splits from exploring the same failed branches. Take the following example

```
s.add(x >= 0, y == x + 1, Or(y > 2, y < 1))
```

by introducing the names:

```
p1, p2, p3, p4 = Bools('p1 p2 p3 p4')
#              = x >= 0, y == x + 1, y > 2, y < 1
```

we obtain a propositional formula

```
And(p1, p2, Or(p3, p4))
```

It is satisfiable and a possible truth assignment is

```
p1, p2, p3, p4 = True, True, False, True
```

It requires satisfiability of the following conjunction:

```
x >= 0, y == x + 1, Not(y > 2), y < 1
```

It is already the case that

```
x >= 0, y == x + 1, y < 1
```

is `unsat`. To avoid this assignment we require also satisfying the blocking clause

```
Or(Not(p1), Not(p2), Not(p4))
```

The new truth assignment

```
p1, p2, p3, p4 = True, True, True, False
```

produces

```
x >= 0, y == x + 1, y > 2, Not(y < 1)
```

which is satisfiable. The example illustrates the steps used in a CDCL(T) integration where the Theory Solver processes the final result of a SAT Solver. We can simulate this procedure using

Z3's API. Figure 11 shows a CDCL(T) solver that leverages a propositional solver `prop` to check a propositional abstraction and a theory solver `theory` whose role is to check conjunctions of literals produced by `prop`. Figure 12 lists auxiliary routines required to create the abstraction.

```
def simple_cdclT(clauses):
    prop   = Solver()
    theory = Solver()
    abs = {}
    prop.add(abstract_clauses(abs, clauses))
    theory.add([p == abs[p] for p in abs])
    while True:
        is_sat = prop.check()
        if sat == is_sat:
            m = prop.model()
            lits = [mk_lit(m, abs[p]) for p in abs]
            if unsat == theory.check(lits):
                prop.add(Not(And(theory.unsat_core())))
            else:
                print(theory.model())
                return
        else:
            print(is_sat)
            return
```

Figure 11. Simple CDCL(T)

```
index = 0
def abstract_atom(abs, atom):
    global index
    if atom in abs:
        return abs[atom]
    p = Bool("p%d" % index)
    index += 1
    abs[atom] = p
    return p

def abstract_lit(abs, lit):
    if is_not(lit):
        return Not(abstract_atom(abs, lit.arg(0)))
    return abstract_atom(abs, lit)

def abstract_clause(abs, clause):
    return Or([abstract_lit(abs, lit) for lit in clause])

def abstract_clauses(abs, clauses):
    return [abstract_clause(abs, clause) for clause in clauses]

def mk_lit(m, p):
    if is_true(m.eval(p)):
        return p
    else:
        return Not(p)
```

Figure 12. Auxiliary routines for lazy CDCL(T)

We call it a *simple* CDCL(T) solver as it does not expose important features to drive performance. Importantly, efficient CDCL(T) solvers integrate *theory propagation* that let theories interact with the SAT solver to propagate assignments to atoms. Instead of adding blocking clauses by the time the SAT solver is done the theory solver interacts tightly with the SAT solver during back-jumping.

Exercise 6a:

Dual propagation and implicants: The propositional assignment produced by `prop` is not necessarily minimal. It may assign truth assignments to literals that are irrelevant to truth of the set of clauses. To extract a smaller assignment, one trick is to encode the *negation* of the clauses in a separate *dual* solver. A truth assignment for the *primal* solver is an unsatisfiable core for the *dual* solver. The exercise is to augment `simple_cdclT` with a dual solver to reduce assignments sent to the theory solver.

6.1.2. Theories + Theories

In practice we need to solve a combination of theories. The formulas we used in the initial example

```
x + 2 == y, f(Store(A, x, 3)[y - 2]) != f(y - x + 1)
```

integrate several theory solvers. For modularity, it is desirable to maintain separate solvers per theory. To achieve this objective the main questions that an integration needs to address are:

- Determine when the union of two theories $T_1 \cup T_2$ is consistent.
- Given solvers for T_1 and T_2, how can we build a solver for $T_1 \cup T_2$.

We can address this objective when there is an *effective* theory T_0 over the shared signature of T_1, T_2, that when embedable into T_1, T_2 implies $T_1 \cup T_2$ is consistent. Sufficient conditions for this setting were identified by Nelson and Oppen [52]:

Theorem 1.
The union of two consistent, disjoint, stably infinite theories is consistent.

Let us define the ingredients of this theorem.

Disjoint Theories
Two theories are disjoint if they do not share function/constant and predicate symbols. $=$ is the only exception. For example,

- The theories of arithmetic and arrays are disjoint.
  - Arithmetic symbols: `0`, `-1`, `1`, `-2`, `2`, +, -, *, >, <, ==, >=.
  - Array symbols: `Select`, `Store`

The process of *purification* can be used as a formal tool to bring formulas into signature-disjoint form. It introduces fresh symbols for shared sub-terms. A purified version of our running example is:

```
Functions:   f(v1) != f(v2)
Arrays:      v1 == v3[v4], v3 == Store(x, y, v5)
Arithmetic: x + 2 == y, v2 == y - x + 1, v4 == y - 2, v5 == 2
```

In reality, purification is a no-op: the fresh variables correspond directly to nodes in the abstract syntax trees for expressions.

Stably Infinite Theories
A theory is stably infinite if every satisfiable quantifier-free formula is satisfiable in an infinite model.

- EUF and arithmetic are stably infinite.

- Bit-vectors are not.

Nelson-Oppen combination.

Let T_1 and T_2 be consistent, stably infinite theories over disjoint (countable) signatures. Assume satisfiability of conjunction of literals can be decided in $O(T_1(n))$ and $O(T_2(n))$ time respectively. Then

1. The combined theory T is consistent and stably infinite.

2. Satisfiability of quantifier free conjunction of literals can be decided in $O(2^{n^2} \times (T_1(n) + T_2(n)))$.

3. If T_1 and T_2 are *convex*, then so is T and satisfiability in T can be decided in $O(n^3 \times (T_1(n) + T_2(n)))$.

Convexity.

A theory T is *convex* if for every finite sets S of literals, and every disjunction $a_1 = b_1 \vee \ldots \vee a_n = b_n$:

$S \models a_1 = b_1 \vee \ldots \vee a_n = b_n$ *iff* $S \models a_i = b_i$ *for some* $1 \leq i \leq n$.

Many theories are convex and therefore admit efficient theory combinations

- Linear Real Arithmetic is convex.

- Horn equational theories are convex.

  - Horn equations are formulas of the form $a_1 \neq b_1 \vee \ldots a_n \neq b_n \vee a = b$.

Finally note that every convex theory with non trivial models is stably infinite.

But, far from every theory is convex. Notably,

- Integer arithmetic

  - $1 \leq a \leq 2, b = 1, c = 2$ implies $a = b \vee a = c$.

- Real non-linear arithmetic

  - $a^2 = 1, b = 1, c = -1$ implies $a = b \vee a = c$.

- The theory of arrays

  - `Store(a, i, v)[j] == v` implies `i == j` or `a[j] == v`.

A Reduction Approach to Theory Combination [21, 43].

Theory Combination in Z3 is essentially by reduction to a set of core theories comprising of Arithmetic, EUF and SAT. Bit-vectors and finite domains translate to propositional SAT. Other theories are *reduced* to core theories. We provided an example of this reduction in Section 3.3.

6.1.3. E-matching based quantifier instantiation

E-matching [24] based quantifier instantiation uses ground terms to find candidate instantiations of quantifiers. Take the example

```
a, b, c, x = Ints('a b c x')
f = Function('f', Z, Z)
g = Function('g', Z, Z, Z)
prove(Implies(And(ForAll(x, f(g(x, c)) == a), b == c, g(c, b) == c),
              f(b) == a))
```

The smallest sub-term that properly contains x is `g(x, c)`. This *pattern* contains all the bound

variables of the universal quantifier. Under the ground equality `b == c` and instantiation of `x` by `c`, it equals `g(c, b)`. This triggers an instantiation by the following tautology

```
Implies(ForAll(x, f(g(x, c)) == a), f(g(c, c)) == a))
```

Chasing the equalities `f(g(c, c)) == a`, `g(c, b) == c`, `b == c` we derive `f(b) == a`, which proves the implication.

The example illustrated that E-matching takes as starting point a *pattern* term p, that captures the variables bound by a quantifier. It derives a substitution θ, such that $p\theta$ equals some *useful* term t, modulo some *useful* equalities. A useful source of *useful* terms are the current ground terms $\mathcal{T}$ maintained during search, and the current asserted equalities during search may be used as the *useful* equalities. The *congruence closure* structure cc introduced in Section 3.1.1 contains relevant information to track ground equalities. For each ground term it represents an equivalence class of terms that are congruent in the current context. Now, given a pattern p we can compute a set of substitutions modulo the current congruence closure by invoking

$$\bigcup_{t \in \mathcal{T}} match(p, t, \emptyset)$$

where E-matching is defined by recursion on the pattern p:

$$
\begin{aligned}
match(x,\ t,\ S) \quad &= \quad \{\,\theta[x \mapsto t] \ \mid\ \theta \in S,\ x \notin \theta\,\} \\
&\quad \cup \{\,\theta \ \mid\ \theta \in S,\ x \in \theta,\ \theta(x) \in cc(t)\,\} \\
match(c,\ t,\ S) \quad &= \quad \emptyset \qquad \text{if } c \notin cc(t) \\
match(c,\ t,\ S) \quad &= \quad S \qquad \text{if } c \in cc(t) \\
match(f(\vec{p}),\ t,\ S) \quad &= \quad \bigcup_{f(\vec{t})\,\in cc(t)}\ match(\vec{p}_n,\ \vec{t}_n,\ \ldots,\ match(\vec{p}_1,\ \vec{t}_1,\ S))
\end{aligned}
$$

It is not always possible to capture all quantified variables in a single pattern. For this purpose E-matching is applied to a sequence of patterns, known as a *multi-pattern*, that collectively contains all bound variables.

The secret sauce to efficiency is to find instantiations

- with as little overhead as possible,
- across large sets of terms, and
- incrementally.

Z3 uses code-trees [56] to address scale bottlenecks for search involving thousands of patterns and terms.

6.1.4. Model-Based Quantifier Instantiation

E-matching provides a highly syntactic restriction on instantiations. An alternative to E-matching is based on using a current model of the quantifier-free part of the search state. It is used to evaluate the universal quantifiers that have to be satisfied in order for the current model to extend to a full model of the conjunction of all asserted constraints. We call this method *Model-Based Quantifier Instantiation* [9, 25, 29, 61]. Take the following example:

```
from z3 import *
Z = IntSort()
f = Function('f', Z, Z)
g = Function('g', Z, Z)
a, n, x = Ints('a n x')
solve(ForAll(x, Implies(And(0 <= x, x <= n), f(x + a) == g(x))),
      a > 10, f(a) >= 2, g(3) <= -10)
```

It may produce a model of the form

```
[a = 11,
 n = 0,
 f = [else -> 2],
 g = [3 -> -10, else -> f(Var(0) + 11)]]
```

The interpretation of g maps 3 to -10, and all other values x are mapped to however $f(11 + x)$ is interpreted (which happens to be the constant 2).

The method that allowed finding this satisfying assignment is based on a model evaluation loop. At a high level it can be described as the following procedure, which checks satisfiability of

$$\psi \wedge \forall x \; . \; \varphi[x]$$

where ψ is quantifier free and for sake of illustration we have a single quantified formula with quantifier free body φ. The Model-Based Quantifier Instantiation, MBQI, procedure is described in Figure 13:

```
s.add(ψ)
while True:
    if unsat == s.check():
        return unsat
    M = s.model()
    checker = Solver()
    checker.add(¬φ^M[x])
    if unsat == checker.check():
        return sat
    M = checker.model()
    find t, such that x ∉ t, t^M = x^M .
    s.add(φ[t])
```

Figure 13. Model-Based Quantifier Instantiation algorithm. Notice that this proto-algorithm code is not directly executable.

We use the notation t^M to say that t is partially evaluated using interpretation M, for example:

- Let $M := [y \mapsto 3, f(x) \mapsto \textit{if } x = 1 \textit{ then } 3 \textit{ else } 5]$, and

- $t := y + f(y) + f(z)$, then

- $t^M = 3 + 5 + \textit{if } z = 1 \textit{ then } 3 \textit{ else } 5$

For our example formula assume we have a model of the quantifier-free constraints as follows

```
[a = 11, n = 0, f = [else -> 2], g = [else -> -10]]
```

The negated body of the quantifier, instantiated to the model is

```
And(0 <= x, x <= 0, [else -> 2](x + 11) != [else -> -10](x))
```

It is satisfied with the instantiation x = 0, which is congruent to n under the current model. We therefore instantiate the quantifier with x = n and add the constraint

```
Implies(And(0 <= n, n <= n), f(n + a) == g(n))
```

But notice a syntactic property of the quantifier body. It can be read as a definition for the graph of g over the range 0 <= x, x <= n. This format is an instance of *guarded definitions* [35]. Hence, we record this reading when creating the next model for g. In the next round, a, n, and f are instantiated as before, and g(3) evaluates to -10 as before, but elsewhere follows the graph of f(x + a), and thus the model for g is given by [3 -> -10, else -> f(11 + Var(0))].

Model-Based Quantifier Instantiation is quite powerful when search space for instantiation terms is finite. It covers many decidable logical fragments, including EPR (Effectively Propositional Reasoning), UFBV (uninterpreted functions and bit-vectors), the Array property fragment [11] and extensions [29]. We will here only give a taste with an example from UFBV [61]:

```
Char = BitVecSort(8)
f  = Function('f', Char, Char)
f1 = Function('f1', Char, Char)
a, x = Consts('a x', Char)
solve(UGE(a, 0), f1 (a + 1) == 0,
      ForAll(x, Or(x == a + 1, f1(x) == f(x))))
```

The following model is a possible solution:

```
[a = 0, f = [else -> 1], f1 = [1 -> 0, else -> f(Var(0))]]
```

UFBV is the quantified logic of uninterpreted functions of bit-vectors. All sorts and variables have to be over bit-vectors, and standard bit-vector operations are allowed. It follows that the problem is finite domain and therefore decidable. It isn't easy, however. The quantifier-free fragment is not only NP hard, it is NEXPTIME hard; it can be encoded into EPR [57]. The quantified fragment is another complexity jump. Related to UFBV, decision procedures for quantified bit-vector formulas were developed by John and Chakraborty in [38–40], and by Niemetz et al in [53].

Recall taht EPR is a fragment of first-order logic where formulas have the quantifier prefix $\exists \vec{x} \forall \vec{y}$, thus a block of existential quantified variables followed by a block of universally quantified variables. The formula inside the quantifier prefix is a Boolean combination of equalities, disequalities between bound variables and free constants as well as predicate symbols applied to bound variables or free constants. Noteworthy, EPR formulas do not contain functions. It is easy to see that EPR is decidable by first replacing the existentially quantified variables by fresh constants and then instantiate the universally quantified variables by all combinations of the free constant. If the resulting ground formula is satisfiable, we obtain a finite model of the quantified formula by bounding the size of the universe by the free constants. The formula $\exists x \forall y.(p(x, y) \lor q(a, y) \lor y = a)$, where a is a free constant, is in EPR.

6.2. SAT Core

The SAT Core is an optimized self-contained SAT solver that solves propositional formulas. It takes advantage of the fact that it operates over propositional theories and performs advanced in-processing steps. The SAT solver also acts as a blackboard for select Boolean predicates that express cardinality and arithmetical (pseudo-Boolean) constraints over literals.

Generally, theories that are finite domain, are solved using the SAT solver. Z3 identifies quantifier-free finite domain theories using a designated logic QF_FD. It supports propositional logic, bit-vector theories, pseudo-Boolean constraints, and enumeration data-types. For example, the following scenario introduces an enumeration type for color, and bit-vectors u, v. It requires that at least 2 out of three predicates u + v <= 3, v <= 20, u <= 10 are satisfied.
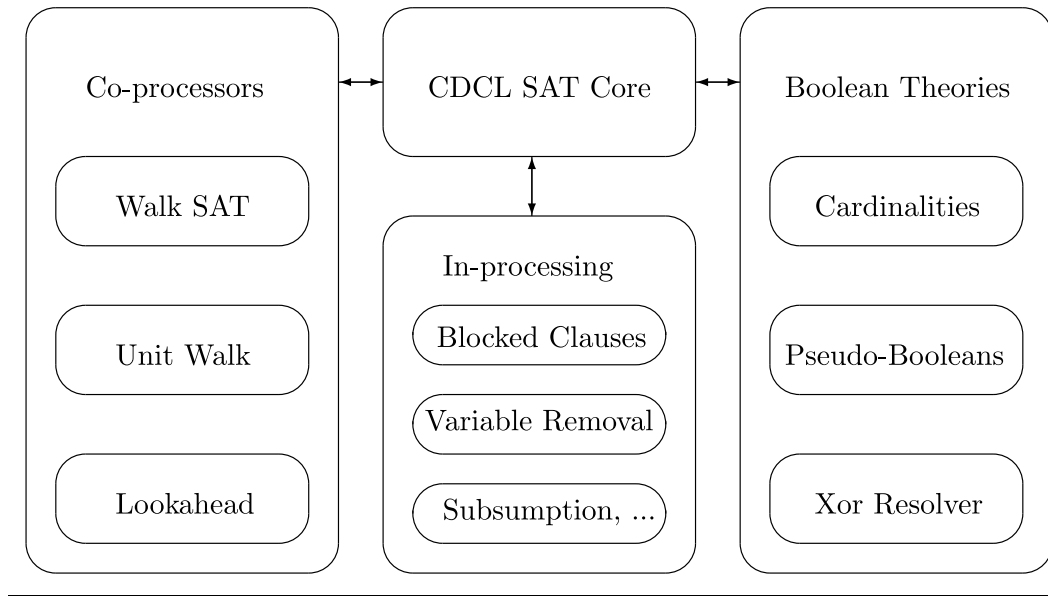
```
from z3 import *
s = SolverFor("QF_FD")
Color, (red, green, blue) = EnumSort('Color', ['red','green','blue'])
clr = Const('clr', Color)
u, v = BitVecs('u v', 32)
s.add(u >= v,
      If(v > u + 1, clr != red, clr != green),
      clr == green,
      AtLeast(u + v <= 3, v <= 20, u <= 10, 2))
print(s.check())
print(s.model())
```

is satisfiable, and a possible model is:

```
[v = 4, u = 2147483647, clr = green]
```

Figure 14 shows the overall architecture of Z3's SAT solver.

Figure 14. Architecture of Z3's SAT Solver

There are four main components. Central to the SAT solver is an engine that performs case splits, lemma learning and backtracking search. It is the main CDCL engine and is structured similar to mainstream CDCL solvers. It can draw on auxiliary functionality.

6.2.1. In-processing

In-processing provides a means for the SAT solver to simplify the current set of clauses using *global* inferences. In-processing is performed on a periodic basis. It integrates several of the techniques that have been developed in the SAT solving literature in the past decade, known as Blocked Clause Elimination, Asymmetric Literal Addition, Asymmetric Covered Clause Elimination, Subsumption, Asymmetric Branching [32].

6.2.2. Co-processing

A set of co-processors are available to support alternative means of search. The SAT Core solver can also be a co-processor of itself.

- `s.set("sat.local_search_threads", 3)` spawns 3 concurrent threads that use walk-sat to find a satisfying assignment while the main CDCL solver attempts to find either a satisfying assignment or produce an empty clause.
- `s.set("sat.threads", 3)` spawns 2 concurrent threads, in additional to the main thread, to find a proof of the empty clause or a satisfying assignment. The threads share learned unit literals and learned clauses.
- `s.set("sat.unit_walk_threads", 1)` spawns 1 concurrent thread that uses a local search heuristic that integrates unit propagation.
- `s.set("sat.lookahead_simplify", True)` enables the lookahead solver as a simplifier during in-processing. It enables slightly more powerful techniques for learning new units and binary clauses.

The lookahead solver is used to find case splits through the Cube features, described in Section 4.6.7.

6.2.3. Boolean Theories

Three classes of Boolean functions are supported using specialized Boolean theory handlers. They are optional, as many problems can already be solved using the SAT core where the functions have been clausified. The cardinality and Pseudo-Boolean theory handlers are suitable for constraints where the encoding into clauses causes a significant overhead. The Xor solver is unlikely to be worth using, but is available for evaluation.

Cardinality Constraints

Cardinality constraints are linear inequalities of the form

$$\sum_{i=1}^{n} F_i \geq k, \ \sum_{i=1}^{n} F_i \leq k$$

where F_i are formulas and k is a constant between 1 and n. They say that at least k of the F_i;s have to hold, and at most k of the F_i's hold, respectively. Cardinality constraints do not have to appear at top-level in formulas. They can be nested in arbitrary sub-formulas and they can contain arbitrary formulas. For instance,

```
p, q, r, u = Bools('p q r u')
solve(AtMost(p, q, r, 1), u,
      Implies(u, AtLeast(And(p, r), Or(p, q), r, 2)))
```

has no solution.

The cardinality solver is enabled by setting the parameter

- `s.set("sat.cardinality.solver", True)`

If the parameter is false, cardinality constraints are compiled to clauses. A few alternative encoding methods are made available, and they can be controlled using the parameter `sat.cardinality.encoding`.

Pseudo-Boolean Constraints

Pseudo-Boolean constraints generalize cardinality constraints by allowing coefficients in the linear inequalities. They are of the form

$$\sum_{i=1}^{n} a_i F_i \geq k, \ \sum_{i=1}^{n} a_i F_i \leq k$$

where a_i are positive natural numbers. A value of a_i above k is legal, but can be safely truncated to k without changing the meaning of the formulas.

The constraints

$$p + 2q + 2r \leq 2 \wedge p + 2u + 3r \geq 4 \wedge u$$

can be written as

```
solve(PbLe([(p,1),(q,2),(r,2)], 3),
      PbGe([(p,1),(u,2),(r,3)], 4),
      u)
```

and have a solution

```
[q = False, u = True, r = True]
```

The pseudo-Boolean solver is enabled by setting the parameter

- `s.set("sat.pb.solver", "solver")`

Other available options for compiling Pseudo-Boolean constraints are `circuit`, `sorting`, and `totalizer`. They compile Pseudo-Booleans into clauses.

6.3. Horn Clause Solver

The Horn Solver contains specialized solvers for *Constrained Horn Clauses* [8, 30, 33, 34, 47]. As a default it uses the SPACER Horn clause solver by Arie Gurfinkel to solve Horn clauses over arithmetic [44]. A Constrained Horn Clause is a disjunction of literals over a set of uninterpreted predicates and interpreted functions and interpreted predicates (such as arithmetical operations + and relations <=). The uninterpreted predicates, may occur negatively without restrictions, but only occur positively in at most one place.

The solver also contains a Datalog engine that can be used to solve Datalog queries (with stratified negation) over finite domains and "header spaces" that are large finite domains, but can be encoded succinctly using ternary bit-vectors. The `Fixedpoint` context contains facilities for building Horn clauses, and generally a set of stratified Datalog rules, and for querying the resulting set of rules and facts. Additional information on the Fixedpoint engine can be found on https://rise4fun.com/z3/tutorial/fixedpoints.

We provide a very simple illustration of Horn clause usage here. McCarthy's 91 function illustrates nested recursion in a couple of lines, but otherwise makes no sense: It computes a function that can be described directly as

```
If(x > 101, 91, x - 10).
```

We will pretend this is a partial and interesting specification and prove this automatically using Horn clauses.

```
def mc(x):
    if x > 100:
        return x - 10
    else:
        return mc(mc(x + 11))

def contract(x):
    assert(x > 101 or mc(x) == 91)
    assert(x < 101 or mc(x) == x - 10)
```

Rewriting the functional program into logical form can be achieved by introducing a binary relation between the input and output of `mc`, and then representing the functional program as a logic program, that is, a set of Horn clauses. The assertions are also Constrained Horn Clauses: they contain the uninterpreted predicate `mc` negatively, but have no positive occurrences of `mc`.

```
s = SolverFor("HORN")
mc = Function('mc', Z, Z, B)
x, y, z = Ints('x y z')
s.add(ForAll(x, Implies(x > 100, mc(x, x - 10))))
s.add(ForAll([x, y, z],
            Implies(And(x <= 100, mc(x + 11, y), mc(y, z)),
                    mc(x, z))))
s.add(ForAll([x, y], Implies(And(x <= 101, mc(x, y)), y == 91)))
s.add(ForAll([x, y], Implies(And(x >= 101, mc(x, y)), x == y + 10)))
print(s.check())
```

Z3 finds a solution for `mc` that is a sufficient invariant to establish the assertions.

We get a better view of the invariant for `mc` by evaluating it on symbolic inputs `x` and `y`.

```
print(s.model().eval(mc(x, y)))
```

produces the invariant

```
And(Or(Not(y >= 92), Not(x + -1*y <= 9)),
    Not(x + -1*y >= 11),
    Not(y <= 90))
```

6.4. QSAT

The QSAT Solver is a decision procedure for satisfiability of select theories that admit quantifier elimination. It can be used to check satisfiability of quantified formulas over Linear Integer (Figure 15), Linear Real (Figure 16), Non-linear (polynomial) Real arithmetic (Figure 17), Booleans, and Algebraic Data-types (Figure 18). It is described in [6]. It is invoked whenever a solver is created for one of the supported quantified logics, or a solver is created from the qsat tactic.

```
s = SolverFor("LIA") # Quantified Linear Integer Arithmetic
x, y, u, v = Ints('x y u v')
a = 5
b = 7
s.add(ForAll(u, Implies(u >= v,
                        Exists([x, y], And(x >= 0, y >= 0, u == a*x + b*y)))))
print(s.check())
print(s.model())
```

Figure 15. Given a supply of 5 and 7 cent stamps. Is there a lower bound, after which all denominations of stamps can be produced? Thus, find v, such that every u larger or equal to v can be written as a non-negative combination of 5 and 7.

```
s = SolverFor("LRA") # Quantified Linear Real Arithmetic
x, y, z = Reals('x y z')
s.add(x < y, ForAll(z, Or(z <= x, y <= z)))
print(s.check())
```

Figure 16. The set of reals is dense

```
s = SolverFor("NRA") # Quantified Non-linear Real Arithmetic
x, y, z = Reals('x y z')
s.add(x < y)
s.add(y * y < x)
s.add(ForAll(z, z * x != y))
print(s.check())
```

Figure 17. Quantified non-linear real polynomial arithmetic

```
from z3 import *
s = Tactic('qsat').solver()
Nat = Datatype('Nat')
Nat.declare('Z')
Nat.declare('S', ('pred', Nat))
Nat = Nat.create()
Z = Nat.Z
S = Nat.S
def move(x, y):
    return Or(x == S(y), x == S(S(y)))
def win(x, n):
    if n == 0:
        return False
    y = FreshConst(Nat)
    return Exists(y, And(move(x, y), Not(win(y, n - 1))))

s.add(win(S(S(S(S(Z)))), 4))
print(s.check())
```

Figure 18. Checking for winning positions in a game of successors

Figure 18 encodes a simple game introduced in [17]. There is no SMT-LIB2 logic for quantified algebraic data-types so we directly instantiate the solver that performs QSAT through a tactic. Section 7 provides a brief introduction to tactics in Z3.

The solver builds on an abstraction refinement loop, originally developed for quantifier elimination in [49]. The goal of the procedure is, given a quantifier-free f, find a quantifier free G, such that $G \equiv \exists \vec{v} \, . \, F$. It assumes a tool, *project*, that eliminates $\vec{v}$ from a conjunction M into a satisfiable strengthening. That is, project$(\vec{v}, M) \Rightarrow \exists \vec{v} \, . \, M$. The procedure, uses the steps:

- **Initialize**: $G \leftarrow \bot$

- **Repeatedly**: find conjunctions M that imply $F \wedge \neg G$

- **Update**: $G \leftarrow G \vee \text{project}(\vec{v}, M)$.

An algorithm that realizes this approach is formulated in Figure 19.

```
def qe(∃⃗v . F):
    e, a = Solver(), Solver()
    e.add(F)
    a.add(¬F)
    G = False
    while sat == e.check():
        M₀ = e.model()
        M₁ = [ lit for lit in literals(F) if is_true(M₀.eval(lit)) ]
        # assume F is in negation normal form
        assert unsat == a.check(M₁)
        M₂ = a.unsat_core()
        π = project(M₂, ⃗v)
        G = G ∨ π
        e.add(¬π)
    return G
```

Figure 19. Quantifier elimination by core extraction and projection. Notice that this proto-algorithm code is not directly executable

QESTO [36] generalizes this procedure to nested QBF (Quantified Boolean Formulas), and the implementation in Z3 generalizes QESTO to SMT. The approach is based on playing a quantifier game. Let us illustrate the game for Boolean formulas. Assume we are given:

$$G \;=\; \forall u_1, \, u_2 \, \exists e_1, \, e_2 \;\; . \;\; F$$

$$F \;=\; (u_1 \, \wedge \, u_2 \, \to \, e_1) \, \wedge \, (u_1 \, \wedge \, \neg u_2 \, \to \, e_2) \, \wedge \, (e_1 \, \wedge \, e_2 \, \to \, \neg u_1)$$

Then the game proceeds as follows:

- $\forall$: starts. $u_1, u_2, \overline{e}_1, \overline{e}_2 \models \neg F$.
- $\exists$: strikes back. $u_1, u_2, e_1, \overline{e}_2 \models F$.
- $\forall$: has to backtrack. It doesn't matter what u_1 is assigned to. It is already the case that $u_2, e_1, \overline{e}_2 \models F$.
- $\forall$: learns $\neg u_2$.
- $\forall$: $\overline{u}_2, u_1, \overline{e}_1, \overline{e}_2 \models \neg F$.
- $\exists$: counters - $\overline{u}_2, u_1, \overline{e}_1, e_2 \models F$.
- $\forall$: has lost!. It is already the case that $\overline{u}_2, \overline{e}_1, e_2 \models F$.

To summarize the approach:

- There are two players

    - $\forall$ - tries to satisfy $\neg F$
    - $\exists$ - tries to satisfy F

- Players control their variables. For example, take $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots F$ at round 2:

    - value of x_1 is already fixed,
    - $\forall$ fixes value of x_2,
    - $\forall$ fixes value of x_4, but can change again at round 4,
    - $\forall$ can guess values of x_3 to satisfy $\neg F$.

- Some player loses at round $i + 2$:

    - Create succinct *no-good* to strengthen F resp. $\neg F$ depending on who lost.
    - Backjump to round i (or below).

The main ingredients to the approach is thus *projection* and *strategies*.

- Projections are added to *learn* from mistakes. Thus, a player avoids repeating same losing moves.

- Strategies *prune* moves from the opponent.

We will here just illustrate an example of projection. Z3 uses *model based projection* [22, 44] to find a satisfiable quantifier-free formula that implies the existentially quantified formula that encodes the losing state.

Example 4.
Suppose we would want to compute a quantifier-free formula that implies $\exists x \; . \; (2y \leq x \wedge y - z \leq x \wedge x \leq z)$. Note that the formula is equivalent to a quantifier free formula:

$$\exists x \; . \; (2y \leq x \wedge y - z \leq x \wedge x \leq z) \equiv (y - z \leq 2y \leq z) \vee (2y \leq y - z \leq z)$$

but the size of the equivalent formula is quadratic in the size of the original formula. Suppose we have a satisfying assignment for the formula inside of the existential quantifier. Say $M = [x \mapsto 3, y \mapsto 1, z \mapsto 6]$. Then $2y^M = 2$ and $(y - z)^M = -5$, and therefore $2y > y - z$ under M. The greatest lower bound for x is therefore $2y$ and we can select this branch as our choice for elimination of x. The result of projection is then $y - z \leq 2y \leq z$.

6.5. NLSat

The solver created when invoking `SolverFor('QF_NRA')` relies on a self-contained engine that is specialized for solving non-linear arithmetic formulas [42]. It is a decision procedure for quantifier-free formulas over the reals using polynomial arithmetic.

```
s = SolverFor("QF_NRA")
x, y = Reals('x y')
s.add(x**3 + x*y + 1 == 0, x*y > 1, x**2 < 1.1)
print(s.check())
```

The NLSat solver is automatically configured if the formula is syntactically in the `QF_NRA` fragment. So one can directly use it without specifying the specialized solver:

```
set_option(precision=30)
print "Solving, and displaying result with 30 decimal places"
solve(x**2 + y**2 == 3, x**3 == 2)
```

7. Tactics

In contrast to solvers that ultimately check the satisfiability of a set of assertions, tactics transform assertions to sets of assertions, in a way that a proof-tree is comprised of nodes representing goals, and children representing subgoals. Many useful pre-processing steps can be formulated as tactics. They take one goal and create a subgoal.

7.1. Tactic Basics

You can access the set of tactics

```
print(tactics())
```

and for additional information obtain a description of optional parameters:

```
for name in tactics():
    t = Tactic(name)
    print(name, t.help(), t.param_descrs())
```

We will here give a single example of a tactic application. It transforms a goal to a simplified subgoal obtained by eliminating a quantifier that is trivially reducible and by combining repeated formulas into one.

```
x, y = Reals('x y')
g  = Goal()
g.add(2 < x, Exists(y, And(y > 0, x == y + 2)))
print(g)

t1 = Tactic('qe-light')
t2 = Tactic('simplify')
t  = Then(t1, t2)
print(t(g))
```

Additional information on tactics is available from [23], https://rise4fun.com/Z3/tutorial/strategies and http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/strategies-examples.htm.

7.2. Solvers from Tactics

Given a tactic `t`, the method `t.solver()` extracts a solver object that applies the tactic to the current assertions and reports `sat` or `unsat` if it is able to reduce subgoals to a definite answer.

7.3. Tactics from Solvers

There is no method that corresponds to producing a tactic from a solver. Instead Z3 exposes a set of built-in tactics for the main solvers. These are accessed through the names `sat`, `smt`, `qsat` (and `nlqsat` for quantified non-linear real arithmetic, e.g., the logic `NRA`), `qffd` for `QF_FD` and `nlsat` for `QF_NRA`.

7.4. Parallel Z3

The parameter `set_param("parallel.enable", True)` enables Z3's parallel mode. Selected tactics, including `qfbv`, that uses the SAT solver for sub-goals the option, when enabled, will cause Z3 to use a cube-and-conquer approach to solve subgoals. The tactics `psat`, `psmt` and `pqffd` provide direct access to the parallel mode, but you have to make sure that

`"parallel.enable"` is true to force them to use parallel mode. You can control how the cube-and-conquer procedure spends time in simplification and cubing through other parameters under the `parallel` name-space.

The main option to toggle is `parallel.threads.max`. It caps the maximal number of threads. By default, the maximal number of threads used by the parallel solver is bound by the number of processes.

8. Optimization

Depending on applications, learning that a formula is satisfiable or not, may not be sufficient. Sometimes, it is useful to retrieve models that are *optimal* with respect to some objective function. Z3 supports a small repertoire of objective functions and invokes a specialized optimization module when objective functions are supplied. The main approach for specifying an optimization objective is through functions that specify whether to find solutions that *maximize* or *minimize* values of an arithmetical (in the case of Z3, the term has to a *linear* arithmetic term) or bit-vector term t. Thus, when specifying the objective $maximize(t)$ the solver is instructed to find solutions to the variables in t that maximizes the value of t. An alternative way to specify objectives is through *soft constraints*. These are assertions, optionally annotated with weights. The objective is to satisfy as many soft constraints as possible in a solution. When weights are used, the objective is to find a solution with the least penalty, given by the sum of weights, for unsatisfied constraints. From the Python API, one uses the `Optimize` context to specify optimization problems. The `Optimize` context relies on the built-in solvers for solving optimization queries. The architecture of the optimization context is provided in Figure 20.
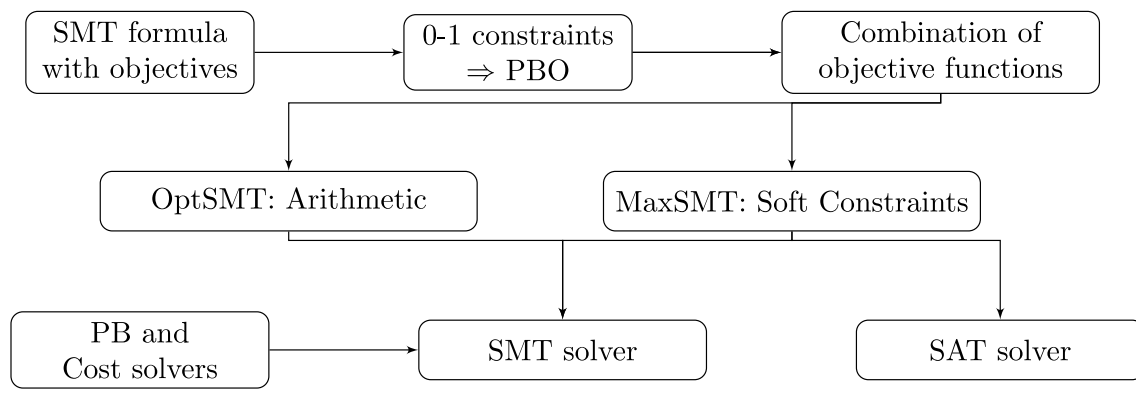

Figure 20. Optimization Engines in Z3

The `Optimize` context provides three main extensions to satisfiability checking:

```
o = Optimize()

x, y = Ints('x y')
o.maximize(x + 2*y)              # maximizes LIA objective

u, v = BitVecs('u v', 32)
o.minimize(u + v)               # minimizes BV objective

o.add_soft(x > 4, 4)            # soft constraint with
                                # optional weight
```

Using soft assertions is equivalent to posing an 0-1 optimization problem. Thus, the following formulations are equivalent and Z3 detects the second variant and turns it into a set of weighted soft assertions.

```
a, b = Bools('a b')
o.add_soft(a, 3)
o.add_soft(b, 4)
```

is equivalent to

```
o.minimize(If(a, 0, 3) + If(b, 0, 4))
```

8.1. Multiple Objectives

It is possible to add multiple objectives. There are three ways to combine objective functions.

$\text{Box}(x, y)$	$v_x := \max\{x \mid \varphi(x, y)\}$
	$v_y := \max\{y \mid \varphi(x, y)\}$
$\text{Lex}(x, y)$	$v_x := \max\{x \mid \varphi(x, y)\}$
	$v_y := \max\{y \mid \varphi(v_x, y)\}$
$\text{Pareto}(x, y)$	$\left\{ (v_x, v_y) \mid \begin{array}{l} \varphi(v_x, v_y), \ \forall x, y. \\ \varphi(x, y) \to x \le v_x \lor y \le v_y \end{array} \right\}$

For instance, Pareto objectives can be specified as follows:

```
x, y = Ints('x y')
opt = Optimize()
opt.set(priority='pareto')
opt.add(x + y == 10, x >= 0, y >= 0)
mx = opt.maximize(x)
my = opt.maximize(y)
while opt.check() == sat:
    print (mx.value(), my.value())
```

8.2. MaxSAT

The conventional definition of MaxSAT is to minimize the number of violated *soft* assertions. There are several algorithms for MaxSAT, and developing new algorithms is a very active area of research. We will here describe MaxRes from [50]. It is also Z3's default solver for MaxSAT/MaxSMT problems. As an illustration assume we are given an *unweighted* (all soft constraints have weight 1) MaxSAT problem $F, F_1, \ldots, F_5$, where the first four soft constraints cannot be satisfied in conjunction with the hard constraint F. Thus, we have the case:

$$A : F, \underbrace{F_1, F_2, F_3, F_4}_{core}, F_5$$

The system is transformed to a weakened MaxSAT problem as follows:

$$A' : F, \ F_2 \lor F_1, F_3 \lor (F_2 \land F_1), F_4 \lor (F_3 \land (F_2 \land F_1)), F_5$$

The procedure is formalized in Figure 21. We claim that by solving A', we can find an optimal solution to A. For this purpose, consider the *cost* of a model with respect to a MaxSAT problem. The cost, written $cost(M, A)$ is the number of soft constraints in A that are *false* under M. More precisely,

Lemma 1.
For every model M of F, $cost(M, A) = 1 + cost(M, A')$

Proof. (*of lemma 1*) To be able to refer to the soft constraints in the transformed systems A' we will give names to the new soft constraints, such that F_1' is a name for $F_2 \lor F_1$, F_2' names $F_3 \lor (F_2 \land F_1)$, F_3' is the name for $F_4 \lor (F_3 \land (F_2 \land F_1))$ and F_4' is the new name of F_5.

Consider the soft constraints in the core. Since it is a core, at least one has to be false under M. Let j be the first index among where $M(F_j)$ is false. Then M evaluates all other soft constraints

the same, e.g., $\forall i < j : M(F_i') = M(F_i)$, and $\forall i > j : M(F_{i-1}') = M(F_i)$. $\square$

Thus, eventually, it is possible to satisfy all soft constraints (weakening could potentially create 0 soft constraints), and a solution to the weakened system is an optimal solution.

```python
def add_def(s, fml):
    name = Bool("%s" % fml)
    s.add(name == fml)
    return name

def relax_core(s, core, Fs):
    prefix = BoolVal(True)
    Fs -= { f for f in core }
    for i in range(len(core)-1):
        prefix = add_def(s, And(core[i], prefix))
        Fs |= { add_def(s, Or(prefix, core[i+1])) }

def maxsat(s, Fs):
    cost = 0
    Fs0 = Fs.copy()
    while unsat == s.check(Fs):
        cost += 1
        relax_core(s, s.unsat_core(), Fs)
    return cost, { f for f in Fs0 if tt(s, f) }
```

Figure 21. Core based MaxSAT using MaxRes

Weighted assertions can be handled by a reduction to unweighted MaxSAT. For example,

```python
a, b, c = Bools('a b c')
o = Optimize()
o.add(a == c)
o.add(Not(And(a, b)))
o.add_soft(a, 2)
o.add_soft(b, 3)
o.add_soft(c, 1)
print(o.check())
print(o.model())
```

Efficient implementations of MaxSAT flatten weights on demand. Given a core of soft constraints it is split into two parts: In one part all soft constraints have the same coefficient as the weight of the soft constraint with the minimal weight. The other part comprises of the remaining soft constraints. For our example, a, b is a core and the weight of a is 2, while the weight of b is 3. The weight of b can therefore be split into two parts, one where it has weight 2, and the other where it has weight 1. Applying the transformation for the the core we obtain the simpler MaxSAT problem:

```python
a, b, c = Bools('a b c')
o = Optimize()
o.add(a == c)
o.add(Not(And(a, b)))
o.add_soft(Or(a, b), 2)
o.add_soft(b, 1)
o.add_soft(c, 1)
print(o.check())
print(o.model())
```

9. Summary

This tutorial presented an overview of main functionality exposed by Z3. By presenting some of the underlying algorithms in an example driven way we have attempted to give a taste of the

underlying decision procedures and proof engines. By presenting examples of programming queries on top of Z3 we have attempted to provide an introduction to turning SMT solving into a service for logic queries that go beyond checking for satisfiability of a single formula. Tuning extended queries on top of the basic services provided by SAT and SMT solvers is a very active area of research with new application scenarios and new discoveries.

References

Mario Alviano. Model enumeration in propositional circumscription via unsatisfiable core analysis. *TPLP*, 17 (5-6): 708–725, 2017. 🔍

Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 70–86, 2015. doi:10.1007/978-3-319-21668-3_5. URL http://dx.doi.org/10.1007/978-3-319-21668-3_5. 🔍

Fahiem Bacchus and George Katsirelos. Finding a collection of muses incrementally. In *CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings*, volume 9676, pages 35–44. Springer, 2016. 🔍

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016. 🔍

Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009. ISBN 978-1-58603-929-5. doi:10.3233/978-1-58603-929-5-457. URL https://doi.org/10.3233/978-1-58603-929-5-457. 🔍

Nikolaj Bjørner and Mikolás Janota. Playing with alternating quantifier satisfaction. In *LPAR Short presentation papers*, 2015. 🔍

Nikolaj Bjørner and Lev Nachmanson. Theorem recycling for theorem proving. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2017. Proceedings of the 4th Vampire Workshop*, volume 53 of *EPiC Series in Computing*, pages 1–8. EasyChair, 2018. doi:10.29007/r58f. URL https://easychair.org/publications/paper/qGfG. 🔍

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015. doi:10.1007/978-3-319-23534-9_2. URL http://dx.doi.org/10.1007/978-3-319-23534-9_2. 🔍

Maria Paola Bonacina, Christopher Lynch, and Leonardo Mendonça de Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reasoning*, 47 (2): 161–189, 2011. 🔍

Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 173–180, 2007. doi:10.1109/FAMCAD.2007.15. URL http://dx.doi.org/10.1109/FAMCAD.2007.15. 🔍

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006. doi:10.1007/11609773_28. URL https://doi.org/10.1007/11609773_28. 🔍

Martin Bromberger and Christoph Weidenbach. New techniques for linear arithmetic: cubes and equalities. *Formal Methods in System Design*, 51 (3): 433–461, 2017. doi:10.1007/s10703-017-0278-7. URL https://doi.org/10.1007/s10703-017-0278-7. 🔍

Hana Chockler, Alexander Ivrii, and Arie Matsliah. Computing interpolants without proofs. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857, pages 72–85. Springer, 2012. doi:10.1007/978-3-642-39611-3_12. URL http://dx.doi.org/10.1007/978-3-642-39611-3_12. 🔍

Jürgen Christ and Jochen Hoenicke. Cutting the mix. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2015. ISBN 978-3-319-21667-6. doi:10.1007/978-3-319-21668-3_3. URL https://doi.org/10.1007/978-3-319-21668-3_3. 🔍

Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Experimenting

on solving nonlinear integer arithmetic with incremental linearization. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2018. ISBN 978-3-319-94143-1. doi:10.1007/978-3-319-94144-8_23. URL `https://doi.org/10.1007/978-3-319-94144-8_23`.

Edith Cohen and Nimrod Megiddo. Improved algorithms for linear inequalities with two variables per inequality. *SIAM J. Comput.*, 23 (6): 1313–1347, 1994. doi:10.1137/S0097539791256325. URL `https://doi.org/10.1137/S0097539791256325`.

Alain Colmerauer and Thi-Bich-Hanh Dao. Expressiveness of full first order constraints in the algebra of finite or infinite trees. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2000. ISBN 3-540-41053-8. doi:10.1007/3-540-45349-0_14. URL `https://doi.org/10.1007/3-540-45349-0_14`.

Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005. ISBN 3-540-27231-3. doi:10.1007/11513988_46. URL `https://doi.org/10.1007/11513988_46`.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 1962.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL `http://ceur-ws.org/Vol-418/paper10.pdf`.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 45–52, 2009. doi:10.1109/FMCAD.2009.5351142. URL `http://dx.doi.org/10.1109/FMCAD.2009.5351142`.

Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 1–12, 2013. doi:10.1007/978-3-642-35873-9_1. URL `https://doi.org/10.1007/978-3-642-35873-9_1`.

Leonardo Mendonça de Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2013. ISBN 978-3-642-36674-1. doi:10.1007/978-3-642-36675-8_2. URL `https://doi.org/10.1007/978-3-642-36675-8_2`.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *CADE*, pages 183–198, 2007.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In *IJCAR*, pages 400–411, 2010.

Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2009. ISBN 978-3-642-02657-7. doi:10.1007/978-3-642-02658-4_20. URL `https://doi.org/10.1007/978-3-642-02658-4_20`.

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27 (4): 758–771, 1980. doi:10.1145/322217.322228. URL `http://doi.acm.org/10.1145/322217.322228`.

B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, 2006.

Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416, 2012.

doi:10.1145/2254064.2254112. URL http://doi.acm.org/10.1145/2254064.2254112. 🔍

Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011. ISBN 978-3-642-34187-8. doi:10.1007/978-3-642-34188-5_8. URL https://doi.org/10.1007/978-3-642-34188-5_8. 🔍

Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53: 127–168, 2015. doi:10.1613/jair.4694. URL https://doi.org/10.1613/jair.4694. 🔍

Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 157–171, 2012. doi:10.1007/978-3-642-31612-8_13. URL http://dx.doi.org/10.1007/978-3-642-31612-8_13. 🔍

Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. µZ- an efficient engine for fixed points with constraints. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 457–462, 2011. doi:10.1007/978-3-642-22110-1_36. URL http://dx.doi.org/10.1007/978-3-642-22110-1_36. 🔍

Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2008. ISBN 978-3-540-78799-0. doi:10.1007/978-3-540-78800-3_19. URL https://doi.org/10.1007/978-3-540-78800-3_19. 🔍

Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 325–331, 2015. URL http://ijcai.org/Abstract/15/052. 🔍

Mikolás Janota, Inês Lynce, and Joao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28 (2): 161–177, 2015. doi:10.3233/AIC-140640. URL http://dx.doi.org/10.3233/AIC-140640. 🔍

Ajith K. John and Supratik Chakraborty. A quantifier elimination algorithm for linear modular equations and disequations. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 486–503, 2011. doi:10.1007/978-3-642-22110-1_39. URL https://doi.org/10.1007/978-3-642-22110-1_39. 🔍

Ajith K. John and Supratik Chakraborty. Extending quantifier elimination to linear inequalities on bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 78–92, 2013. doi:10.1007/978-3-642-36742-7_6. URL https://doi.org/10.1007/978-3-642-36742-7_6. 🔍

Ajith K. John and Supratik Chakraborty. A layered algorithm for quantifier elimination from linear modular constraints. *Formal Methods in System Design*, 49 (3): 272–323, 2016. doi:10.1007/s10703-016-0260-9. URL https://doi.org/10.1007/s10703-016-0260-9. 🔍

Dejan Jovanovic. Solving nonlinear integer arithmetic with MCSAT. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2017. ISBN 978-3-319-52233-3. doi:10.1007/978-3-319-52234-0_18. URL https://doi.org/10.1007/978-3-319-52234-0_18. 🔍

Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 339–354, 2012. doi:10.1007/978-3-642-31365-3_27. URL http://dx.doi.org/10.1007/978-3-642-31365-3_27. 🔍

Deepak Kapur and Calogero Zarba. A reduction approach to decision procedures. Technical report, University of New Mexico, 2006. URL https://www.cs.unm.edu/~kapur/mypapers/reduction.pdf. 🔍

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *CAV*, pages 17–34, 2014. 🔍

Mark H Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21 (2): 223–250, 2016. 🔍

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical verification of peephole optimizations with alive. *Commun. ACM*, 61 (2): 84–91, 2018. doi:10.1145/3166064. URL `http://doi.acm.org/10.1145/3166064`.

Kenneth L. McMillan. Lazy annotation revisited. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 243–259, 2014. doi:10.1007/978-3-319-08867-9_16. URL `http://dx.doi.org/10.1007/978-3-319-08867-9_16`.

Carlos Mencía, Alessandro Previti, and João Marques-Silva. Literal-based MCS extraction. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1973–1979, 2015. URL `http://ijcai.org/Abstract/15/280`.

David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *LPAR (Logic for Programming Artificial Intelligence and Reasoning)*, number 5330 in Lecture Notes in Computer Science, pages 243–257. Springer Verlag, 2008. ISBN 978-3-540-89439-1. doi:10.1007/978-3-540-89439-1_18.

Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In Carla E. Brodley and Peter Stone, editors, *AAAI 2014, July 27 -31, 2014, Quebec City, Quebec, Canada.*, pages 2717–2723. AAAI Press, 2014.

Nina Narodytska, Nikolaj Bjørner, Maria-Cristina Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 1353–1361. ijcai.org, 2018. ISBN 978-0-9992411-2-7. doi:10.24963/ijcai.2018/188. URL `https://doi.org/10.24963/ijcai.2018/188`.

Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1 (2): 245–257, 1979. doi:10.1145/357073.357079. URL `http://doi.acm.org/10.1145/357073.357079`.

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Solving quantified bit-vectors using invertibility conditions. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2018. doi:10.1007/978-3-319-96142-2_16. URL `https://doi.org/10.1007/978-3-319-96142-2_16`.

R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53 (6), 2006.

Alessandro Previti, Carlos Mencia, Matti Jarvisalo, and Joao Marques-Silva. Improving MCS enumeration via caching. In Serge Gaspers and Toby Walsh, editors, *SAT 2017 August 28 - September 1, 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 2017.

I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1853–1964. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

Martina Seidl, Florian Lonsing, and Armin Biere. qbf2epr: A tool for generating EPR formulas from QBF. In *Third Workshop on Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30 - July 1, 2012*, pages 139–148, 2012. URL `http://www.easychair.org/publications/paper/145184`.

João P. Marques Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48 (5): 506–521, 1999.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22 (2): 215–225, 1975. doi:10.1145/321879.321884. URL `http://doi.acm.org/10.1145/321879.321884`.

Margus Veanes, Nikolaj Bjørner, Lev Nachmanson, and Sergey Bereg. Monadic decomposition. *J. ACM*, 64 (2): 14:1–14:28, 2017. doi:10.1145/3040488. URL `http://doi.acm.org/10.1145/3040488`.

Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42 (1): 3–23, 2013.

Created with Madoko.net.