Z3 - a Tutorial

Leonardo de Moura
Microsoft Research
leonardo@microsoft.com

Nikolaj Bjørner
Microsoft Research
nbjorner@microsoft.com

Abstract

This tutorial introduces the use of the state-of-the-art Satisfiability Modulo Theories Solver Z3. It integrates a host of theory solvers in an expressive and efficient combination. We here introduce the supported theories and their solvers using a collection of examples. Z3 is freely available from Microsoft Research.

Contents

1 Introduction

> Logic is the “Calculus of Computer Science”.
> Zohar Manna

Modern software analysis and model-based tools are increasingly complex and multi-faceted software systems. However, at their core is invariably a component using logical formulas for describing states and transformations between system states. In a nutshell, symbolic logic is *the calculus* of computation. The state-of-the art *Satisfiability Modulo Theories* (SMT) solver, Z3, from Microsoft Research, can be used to check the satisfiability of logical formulas over one or more theories. SMT solvers offer a compelling match for software tools, since several common software constructs map directly into supported theories.

This tutorial introduces the use of the state-of-the-art Satisfiability Modulo Theories Solver Z3 from Microsoft Research. The main objective of the tutorial is to introduce the reader on how to use Z3 effectively for logical modeling and solving. The tutorial provides some general background on logical modeling, but we have to defer a full introduction to first-order logic and decision procedures to excellent text-books [18, 4, 23].

Z3 is a low level tool. It is best used as a component in the context of other tools that require solving logical formulas. Consequently, Z3 exposes a number of API facilities to make it convenient for *tools* to map into Z3, but there are no stand-alone editors or user-centric facilities for interacting with Z3. The language syntax used in the front ends favor simplicity in contrast to linguistic convenience.

2 What is logic?

Logic is the art and science of effective reasoning. In logic, we investigate how to draw general and reliable conclusions from a collection of facts. *Formal logic* provides a precise characterization of well-formed expressions and valid deductions. It makes possible to calculate consequences at the symbolic level. Computer programs can be used to automate such symbolic calculations.

Logic studies the relationship between language, meaning and (proof) method. A logic consists of a language in which well-formed sentences are expressed; a semantic that distinguishes the valid sentences from the refutable ones; and a proof system for constructing arguments justifying valid sentences. Examples of logic include: propositional logic, first-order logic, higher-order logic and modal logics. In this tutorial, we will focus on propositional and first-order logic.

A language consists of logical symbols whose the interpretations are fixed, and non-logical ones whose the interpretation vary. These symbols are combined together to form well-formed formulas. In propositional logic (PL), the connectives $\vee$ (or), $\wedge$ (and), $\neg$ (not), $\Rightarrow$ (implies) have a fixed interpretation, whereas the constants *p*, *q*, *r* may be interpreted at will. We also say *p*, *q* and *r* are *propositional variables*. The set of well-formed PL formulas is described by the following grammar:

$$\begin{array}{rcl} \mathit{formula} & := & \mathit{constant} \\ & | & \mathit{true} \\ & | & \mathit{false} \\ & | & \mathit{formula} \ \vee \ \mathit{formula} \\ & | & \mathit{formula} \ \wedge \ \mathit{formula} \\ & | & \mathit{formula} \ \Rightarrow \ \mathit{formula} \\ & | & \neg\, \mathit{formula} \\ & | & (\mathit{formula}) \end{array}$$

The following expressions are well-formed PL formulas:

- $(p \vee q) \Rightarrow (q \vee p)$
- $(p \vee q) \Rightarrow r$
- $p \wedge ((\neg q) \wedge ((\neg p) \vee q))$

As a way of reducing the number of necessary parentheses, we assume the following precedence rules: $\neg$ has higher precedence than $\wedge$, $\wedge$ higher than $\vee$, and $\vee$ higher than $\Rightarrow$. Moreover, since $\vee$ and $\wedge$ are associative, we write $p \vee (q \vee r)$ as $p \vee q \vee r$. We say $\vee$ and $\wedge$ are *multiary* operators. Thus, using these rules, the examples above can be written as:

- $p \vee q \Rightarrow q \vee p$
- $p \vee q \Rightarrow r$
- $p \wedge \neg q \wedge (\neg p \vee q)$

We also say a formula $p \wedge q$ is a *conjunction*, and $p \vee q$ is a *disjunction*.

An *interpretation* M is a mapping from propositional variables to truth values $\{\mathit{true}, \mathit{false}\}$. Let F and G be arbitrary PL formulas. Then, the meaning of the connectives $\vee$ (or), $\wedge$ (and), $\neg$ (not), $\Rightarrow$

(implies) can be given using *truth-tables*.

F	G	$F \vee G$	$F \wedge G$	$F \Rightarrow G$	$\neg F$
false	*false*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*
true	*false*	*true*	*false*	*false*	*false*
true	*true*	*true*	*true*	*true*	*false*

A formula is *satisfiable* if it has an interpretation that makes it true. In this case, we say the interpretation is a *model* for the formula. A formula is *unsatisfiable* if it does not have any model. A formula is *valid* if it is true in any interpretation. A PL formula is valid if and only if its negation is unsatisfiable. For example,

- $p \vee q \Rightarrow q \vee p$ is valid,
- $p \vee q \Rightarrow q$ is satisfiable, and
- $p \wedge \neg q \wedge (\neg p \vee q)$ is unsatisfiable.

We say two formulas F and G are *equivalent* if and only if they evaluate to the same value (*true* or *false*) in every interpretation. Examples:

- $\neg\neg F$ is equivalent to F
- $\neg F \vee F$ is equivalent to *true*
- $\neg F \wedge F$ is equivalent to *false*
- $F \Rightarrow G$ is equivalent to $\neg F \vee G$
- $\neg(F \wedge G)$ is equivalent to $\neg F \vee \neg G$
- $\neg(F \vee G)$ is equivalent to $\neg F \wedge \neg G$
- $\neg F \Rightarrow G$ is equivalent to $\neg G \Rightarrow F$
- $F \vee (G \wedge H)$ is equivalent to $(F \vee G) \wedge (F \vee H)$
- $F \wedge (G \vee H)$ is equivalent to $(F \wedge G) \vee (F \wedge H)$

We say formulas F and G are *equisatisfiable* if and only if F is satisfiable if and only if G is. Most symbolic reasoning engines apply transformations that only preserve satisfiability.

A formula where negation is applied only to propositional variables is said to be in *negation normal form* (NNF). A *literal* is either a propositional variable or its negation. A formula that is a multiary conjunction of multiary disjunctions of literals is in *conjunctive normal form* (CNF). A formula that is a multiary disjunction of multiary conjunctions of literals is in *disjunctive normal form* (DNF). Most satisfiability checkers for propositional logic expect the input formula to be in CNF. Any propositional logic formula is equivalent to one in NNF, CNF and DNF. For example, the formula

$$(p \vee \neg q) \wedge (q \vee \neg(r \vee \neg p))$$

is not in NNF. It can be transformed into an equivalent one by applying the following equivalences:

- $\neg\neg F$ is equivalent to F

- $\neg(F \wedge G)$ is equivalent to $\neg F \vee \neg G$
- $\neg(F \vee G)$ is equivalent to $\neg F \wedge \neg G$
- $F \Rightarrow G$ is equivalent to $\neg F \vee G$

After applying these equivalences, we obtain the equivalent formula:

$$(p \vee \neg q) \wedge (q \vee (\neg r \wedge p))$$

After converting a formula into NNF, it can be converted into CNF or DNF by applying the *distributivity* rules:

- $F \vee (G \wedge H)$ is equivalent to $(F \vee G) \wedge (F \vee H)$
- $F \wedge (G \vee H)$ is equivalent to $(F \wedge G) \vee (F \wedge H)$

Thus, the formula

$$(p \vee \neg q) \wedge (q \vee (\neg r \wedge p))$$

is equivalent to the CNF formula

$$(p \vee \neg q) \wedge (q \vee \neg r) \wedge (q \vee p)$$

It is straightforward to check whether a formula in DNF is satisfiable or not. Unfortunately, it is too expensive, in general, to convert a formula into an equivalent one in DNF. The distributivity rule may produce an exponential blowup. For similar reasons, it is too expensive in general to convert a formula into an equivalent CNF one. However, there is a linear time translation to CNF that produces an equisatisfiable (not equivalent) formula. We use the notation $F[G]$ to denote a formula F that contains a sub-formula G. The basic idea consists in introducing new propositional variables that are "names" for nested sub-formulas. The distributivity rules are replaced by the following rules:

- $F[l_1 \vee l_2] \longrightarrow F[x] \wedge (\neg x \vee l_1 \vee l_2) \wedge (\neg l_1 \vee x) \wedge (\neg l_2 \vee x)$
- $F[l_1 \wedge l_2] \longrightarrow F[x] \wedge (\neg x \vee l_1) \wedge (\neg x \vee l_2) \wedge (\neg l_1 \vee \neg l_2 \vee x)$

In the rules above, x is a fresh variable, and l_1 and l_2 are literals. For example, given the formula

$$(p \wedge (q \vee r)) \vee t$$

Let x_1 be a "name" for $(q \vee r)$. The clauses $(\neg x_1 \vee q \vee r)$, $(\neg q \vee x_1)$, $(\neg r \vee x_1)$ are stating that x_1 is true if and only if $(q \vee r)$ is. Thus, we have

$$((p \wedge x_1) \vee t) \ \wedge \ (\neg x_1 \vee q \vee r) \ \wedge \ (\neg q \vee x_1) \ \wedge \ (\neg r \vee x_1)$$

Now, let x_2 be a "name" for $(p \wedge x_1)$. Then, we obtain the equisatisfiable CNF formula

$$(x_2 \vee t) \ \wedge \ (\neg x_1 \vee q \vee r) \ \wedge \ (\neg q \vee x_1) \ \wedge \ (\neg r \vee x_1) \ \wedge \ (\neg x_2 \vee p) \wedge (\neg x_2 \vee x_1) \wedge (\neg p \vee \neg x_1 \vee x_2)$$

If the formula input formula is in NNF, then simpler versions of these rules can be used

- $F[l_1 \vee l_2] \longrightarrow F[x] \wedge (\neg x \vee l_1 \vee l_2)$
- $F[l_1 \wedge l_2] \longrightarrow F[x] \wedge (\neg x \vee l_1) \wedge (\neg x \vee l_2)$

Since the input formula was in NNF in our previous example, by using the simpler rules, we obtain the equisatisfiable CNF formula

$$(x_2 \vee t) \ \wedge \ (\neg x_1 \vee q \vee r) \ \wedge \ (\neg x_2 \vee p) \wedge (\neg x_2 \vee x_1)$$

In practice, CNF translators use a mixture of distributivity and the rules above. The idea is to use distributivity whenever the formula size does not increase too much.

3 What is SMT?

The defining problem of *Satisfiability Modulo Theories* (SMT) is checking whether a given logical formula F is *satisfiable* in the context of some background theory which constraints the interpretation of the symbols used in F. We say a formula F is satisfiable, if there is an *interpretation* that makes F true. For example, the formula

$$a+b>3 \ \ \mathit{and} \ \ a<0 \ \ \mathit{and} \ \ b>0$$

is satisfiable in the context of the theory of arithmetic, because the interpretation

$$a \mapsto -1, \ \ b \mapsto 5$$

makes the formula true. We say a formula is *unsatisfiable*

It is worth emphasizing that *validity* is dual to the terminology *satisfiability*. Valid sentences are true under all structures. For example, the sentence $(\forall x\colon\ p(x)) \rightarrow p(a)$ is valid. Dually, sentences (the negation of valid sentences) are *unsatisfiable* if they are false under all structures. A *satisfiable* sentence is true in at least one structure. For example, the sentence $p(a) \vee p(b)$ is satisfiable, but it is not valid.

4 What is Z3?

Z3 is a state-of-the-art SMT solver from Microsoft Research. It integrates a host of theory solvers in an expressive and efficient combination. This tutorial introduces these theory solvers using a collection of examples. A short system description covering Z3 is available from [9].

4.1 Obtaining Z3

Z3 is freely available for academic research purposes from

`http://research.microsoft.com/projects/z3`.

Z3 is also re-distributed with a host of systems that use Z3. These include the Boogie/Spec# tools [2], HAVOC [6], Pex[1]. Z3 can be invoked from Isabelle, but does not necessarily require a download and instead relies on a stable internet connection.

4.2 Installing Z3

The default installation location for Z3 is the directory

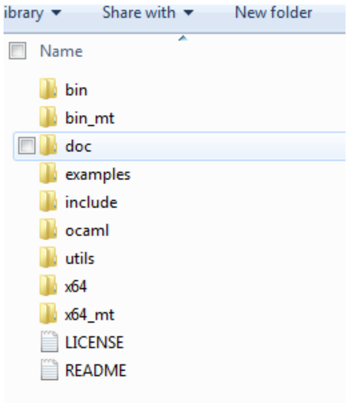
`C:/Program Files/Microsoft Research/Z3-<version-number>`

Most modern machines use 64 bit hardware and operating systems, the installation location is then:

`C:/Program Files/Microsoft Research (x86)/Z3-<version-number>`

In other words, Z3 gets installed as a 32-bit application. However, Z3 ships with both 32 and 64 bit binaries and assemblies. The distribution directory is of the form shown in Figure 1.

It comprises of several directories. The most important directory is `bin` it contains the command-line version of Z3, called `z3.exe`. It also contains C and managed DLLs, called `z3.dll` and `Microsoft.Z3.dll`, respectively. The binaries in this directory run on the Intel/AMD i386 and x64 platforms. A directory containing binaries compiled exclusively for x64 platforms is called `x64`. The directories `bin_mt` and `x64_mt` contain the parallel versions of Z3. These versions can spawn multiple copies of Z3 to cooperate solving a single problem. The `include` directory contains header files, the `examples` directory contains basic examples, the `ocaml` directory contains the OCaml interfaces, and the `utils` directory contains F# power-pack utilities (See Section 5.7).



ibrary ▾ Share with ▾ New folder

□ Name

bin
bin_mt
□ doc
examples
include
ocaml
utils
x64
x64_mt
LICENSE
README

Figure 1: Z3 Distribution directory

[1] `http://research.microsoft.com/pex`

4.3 What is Z3 not?

- A software system with user-friendly UI.
- A theorem prover for proofs by induction.
- A higher-order interactive theorem prover.
- A theorem prover for constructive logic.

$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

$max = 8$

Solution

$t_{1,1} = 5,\ t_{1,2} = 7,\ t_{2,1} = 2,$
$t_{2,2} = 6,\ t_{3,1} = 0,\ t_{3,2} = 3$

Encoding

$$
\begin{array}{l}
(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge \\
(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge \\
(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge \\
((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge \\
((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge \\
((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge \\
((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge \\
((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge \\
((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))
\end{array}
$$

Figure 2: Encoding of job shop scheduling.

5 Satisfiability Modulo Theories - An Appetizer

We begin by introducing a motivating application and a simple instance of the application that we will use as a running example to illustrate Satisfiability Modulo Theories as well as using Z3's different interfaces.

5.1 A Scheduling Application

Consider the classical *job shop scheduling* decision problem. In this problem, there are n jobs, each composed of m tasks of varying duration that have to be performed consecutively on m machines. The start of a new task can be delayed as long as needed in order to wait for a machine to become available, but tasks cannot be interrupted once started. There are essentially two types of constraints in this problem:

- Precedence constraints between two tasks in the same job.
- Resource constraints specifying that no two different tasks requiring the same machine may execute at the same time.

Given a total time *max* and the duration of each task, the problem consists of deciding whether there is a schedule such that the end-time of every task is less than or equal to *max* time units. We use $d_{i,j}$ to denote the duration of the j-th task of job i. A schedule is specified by the start-time ($t_{i,j}$) for the j-th task of every job i. The job shop scheduling problem can be encoded in SMT using the theory of linear arithmetic. A precedence constraint between two consecutive tasks $t_{i,j}$ and $t_{i,j+1}$ is encoded using the inequality $t_{i,j+1} \geq t_{i,j} + d_{i,j}$. This inequality states that the start-time of task $j+1$ must be greater than or equal to the start-time of task j plus its duration. A resource constraint between two tasks from different jobs i and i' requiring the same machine j is encoded using the formula $(t_{i,j} \geq t_{i',j} + d_{i',j}) \vee (t_{i',j} \geq t_{i,j} + d_{i,j})$, which states that the two tasks do not overlap. The start-time of the first task of every job i must be greater than or equal to zero, thus we have $t_{i,1} \geq 0$. Finally, the end-time of the last task must be less than or equal to *max*, hence $t_{i,m} + d_{i,m} \leq max$. Figure 2 illustrates an instance of job scheduling problem, its encoding into an SMT formula, and a satisfying solution. The result is called an *SMT formula*; it combines logical connectives (conjunctions, disjunction, negation) with atomic formulas that are linear arithmetic inequalities.

$$\begin{array}{lcll}
z & - & t_{1,1} & \leq \; 0 \\
z & - & t_{2,1} & \leq \; 0 \\
z & - & t_{3,1} & \leq \; 0 \\
t_{3,2} & - & z & \leq \; 5 \\
t_{3,1} & - & t_{3,2} & \leq \; -2 \\
t_{2,1} & - & t_{3,1} & \leq \; -3 \\
t_{1,1} & - & t_{2,1} & \leq \; -2
\end{array}$$

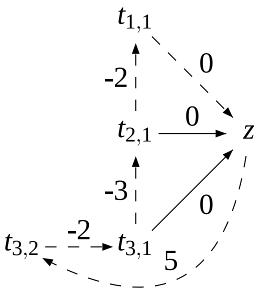


Figure 3: Difference arithmetic example

5.2 A Solver for Difference Arithmetic

The job shop scheduling decision problem can be solved by combining a SAT solver with a theory solver for *difference arithmetic*. Difference arithmetic is a fragment of linear arithmetic where predicates are restricted to be of the form $t - s \leq c$, where t and s are variables and c a numeric constant such as 1 or 3. Every atom in Figure 2 can be put into this form. For example, the atom $t_{3,1} \geq t_{2,1} + 3$ is equivalent to the atom $t_{2,1} - t_{3,1} \leq -3$. For atoms of the form $s \leq c$ and $s \geq c$, a special fresh variable z is used. We say z is the *zero variable*, and these atoms are represented in difference arithmetic as $s - z \leq c$ and $z - s \leq -c$ respectively. For example, the atom $t_{3,2} + 3 \leq 8$ is represented in difference arithmetic as $t_{3,2} - z \leq 5$. A set of difference arithmetic atoms can be checked very efficiently for satisfiability by searching for negative cycles in weighted directed graphs. In the graph representation, each variable corresponds to a node, and an inequality of the form $t - s \leq c$ corresponds to an edge from s to t with weight c. Figure 3 shows a subset of atoms (in difference arithmetic form) from our example in Figure 2, and the corresponding graph. The negative cycle, with weight -2, is shown by dashed lines. This cycle corresponds to the following schedule that cannot be completed in 8 time units:

$$\text{task 1/job 1} \rightarrow \text{task 1/job 2} \rightarrow \text{task 1/job 3} \rightarrow \text{task 2/job 3}$$

Recall that the scheduling problem from Figure 2 is feasible, but it requires assigning a different combination of atoms to true.

5.3 Scheduling in SMT-LIB v1

```
(benchmark
:status unknown
:logic QF_IDL
:extrafuns ((t11 Int) (t12 Int) (t21 Int) (t22 Int) (t31 Int) (t32 Int))
:assumption (and (>= t11 0) (>= t12 (+ t11 2)) (<= (+ t12 1) 8))
:assumption (and (>= t21 0) (>= t22 (+ t21 3)) (<= (+ t22 1) 8))
:assumption (and (>= t31 0) (>= t32 (+ t31 2)) (<= (+ t32 3) 8))
:assumption (or  (>= t11 (+ t21 3)) (>= t21 (+ t11 2)))
:assumption (or  (>= t11 (+ t31 2)) (>= t31 (+ t11 2)))
:assumption (or  (>= t21 (+ t31 2)) (>= t31 (+ t21 3)))
:assumption (or  (>= t12 (+ t22 1)) (>= t22 (+ t12 1)))
:assumption (or  (>= t12 (+ t32 3)) (>= t32 (+ t12 1)))
:assumption (or  (>= t22 (+ t32 3)) (>= t32 (+ t22 1)))
:formula true
)
```

5.4 Scheduling in SMT-LIB v2

SMT-LIB v2, henceforth called smt2, is an update on the SMT-LIB standard. It is based around a set of commands. Each command changes state or queries properties of the current state. You can enter smt2 files using the extension `smt2`, or you can enter smt2 commands from the prompt if you start Z3 using the options:

```
  z3.exe /m /smtc /si
```

These options tell Z3 to `/m` maintain state to produce models (this carries a very minor overhead, and it is disabled by default), `/smtc` parse in the smt2 format, and `/si` open an interactive input pipe.

The smt2 version of the scheduling example now takes the form:

```
(set-logic QF_IDL) ; optional in Z3
(declare-fun t11 () Int)
(declare-fun t12 () Int)
(declare-fun t21 () Int)
(declare-fun t22 () Int)
(declare-fun t31 () Int)
(declare-fun t32 () Int)
(assert (and (>= t11 0) (>= t12 (+ t11 2)) (<= (+ t12 1) 8)))
(assert (and (>= t21 0) (>= t22 (+ t21 3)) (<= (+ t22 1) 8)))
(assert (and (>= t31 0) (>= t32 (+ t31 2)) (<= (+ t32 3) 8)))
(assert (or  (>= t11 (+ t21 3)) (>= t21 (+ t11 2))))
(assert (or  (>= t11 (+ t31 2)) (>= t31 (+ t11 2))))
(assert (or  (>= t21 (+ t31 2)) (>= t31 (+ t21 3))))
(assert (or  (>= t12 (+ t22 1)) (>= t22 (+ t12 1))))
(assert (or  (>= t12 (+ t32 3)) (>= t32 (+ t12 1))))
(assert (or  (>= t22 (+ t32 3)) (>= t32 (+ t22 1))))
(check-sat)
; sat
(model) ; display the model
; ("model" "t11 -> 5
; t12 -> 7
; t21 -> 2
; t22 -> 5
; t31 -> 0
; t32 -> 2")
```

5.5 Scheduling using the C API

Let us write a self-contained program in C that uses the programmatic APIs to Z3 to solve the scheduling problem. For this purpose we create a file called `scheduling.cpp`, and assume that the file resides under the Z3 distribution in the `examples/c` directory. The file can be compiled by invoking the Microsoft C++ compiler `cl` from the command-line:

```
cl ..\..\include\ ..\..\bin\z3.lib scheduling.cpp
```

```
#include ”z3.h”
#include <iostream>

static Z3_ast mk_int(Z3_context ctx, int a) {
    return Z3_mk_int(ctx, a, Z3_mk_int_sort(ctx));
}

static Z3_ast mk_var(Z3_context ctx, Z3_string name) {
    Z3_symbol s = Z3_mk_string_symbol(ctx, name);
    return Z3_mk_const(ctx, s, Z3_mk_int_sort(ctx));
}

static Z3_ast mk_lo(Z3_context ctx, Z3_ast x) {
    return Z3_mk_ge(ctx, x, mk_int(ctx, 0));
}

static Z3_ast mk_mid(Z3_context ctx, Z3_ast x, Z3_ast y, int a) {
    Z3_ast args[2] = { x, mk_int(ctx, a) };
    return Z3_mk_ge(ctx, y, Z3_mk_add(ctx, 2, args));
}

static Z3_ast mk_hi(Z3_context ctx, Z3_ast y, int b) {
    Z3_ast args[2] = { y, mk_int(ctx, b) };
    return Z3_mk_le(ctx, Z3_mk_add(ctx, 2, args), mk_int(ctx, 8));
}

static Z3_ast mk_precedence(
    Z3_context ctx,
    Z3_ast x,
    Z3_ast y,
    int    a,
    int    b
    )
{
    Z3_ast args[3] = { mk_lo(ctx, x), mk_mid(ctx, x, y, a), mk_hi(ctx, y, b) };
    return Z3_mk_and(ctx, 3, args);
}

static Z3_ast mk_resource(
    Z3_context ctx,
    Z3_ast x,
    Z3_ast y,
    int a,
    int b
    )
{
    Z3_ast args1[2] = { y, mk_int(ctx, a) };
    Z3_ast ineq1 = Z3_mk_ge(ctx, x, Z3_mk_add(ctx, 2, args1));
    Z3_ast args2[2] = { x, mk_int(ctx, b) };
    Z3_ast ineq2 = Z3_mk_ge(ctx, y, Z3_mk_add(ctx, 2, args2));
    Z3_ast args3[2] = { ineq1, ineq2 };
    return Z3_mk_or(ctx, 2, args3);
}

int main() {
    Z3_config cfg = Z3_mk_config();
    Z3_set_param_value(cfg, ”MODEL”, ”true”);
    Z3_context ctx = Z3_mk_context(cfg);

    Z3_ast t11 = mk_var(ctx, ”t11”);
    Z3_ast t12 = mk_var(ctx, ”t12”);
    Z3_ast t21 = mk_var(ctx, ”t21”);
    Z3_ast t22 = mk_var(ctx, ”t22”);
    Z3_ast t31 = mk_var(ctx, ”t31”);
    Z3_ast t32 = mk_var(ctx, ”t32”);

    Z3_assert_cnstr(ctx, mk_precedence(ctx, t11, t12, 2, 1));
    Z3_assert_cnstr(ctx, mk_precedence(ctx, t21, t22, 3, 1));
    Z3_assert_cnstr(ctx, mk_precedence(ctx, t31, t32, 2, 3));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t11, t21, 3, 2));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t11, t31, 2, 2));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t21, t31, 2, 3));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t12, t22, 2, 3));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t12, t32, 3, 1));
    Z3_assert_cnstr(ctx, mk_resource(ctx, t22, t32, 3, 1));
    Z3_model m = 0;
    Z3_bool r = Z3_check_and_get_model(ctx, &m);
    if (m) {
        printf(”%s\n”, Z3_model_to_string(ctx, m));
        Z3_del_model(ctx, m);
    }

    Z3_del_context(ctx);
    Z3_del_config(cfg);
}
```

5.6 Scheduling in C#

We will here take advantage of some features that are specific to the .NET API. This API encapsulates Z3 contexts and terms into objects so that it can use the object-oriented conventions from C# and other .NET languages. It also, very conveniently, includes operator overloading for common operations, including addition (+), and comparison (<= and >=), and logical disjunction (|).

```
using Microsoft.Z3;

class Program {
  Context ctx;

  Term mk_int(int a) { return ctx.MkIntNumeral(a); }

  Term mk_var(string name) { return ctx.MkConst(name, ctx.MkIntSort()); }

  Term mk_lo(Term x) { return x >= mk_int(0); }

  Term mk_mid(Term x, Term y, int a) { return y >= (x + mk_int(a)); }

  Term mk_hi(Term y, int b) { return (y + mk_int(b)) <= mk_int(8); }

  Term mk_precedence(Term x, Term y, int a, int b) {
    return ctx.MkAnd(new Term[]{ mk_lo(x), mk_mid(x,y,a), mk_hi(y,b) });
  }

  Term mk_resource(Term x, Term y, int a, int b) {
    return (x >= (y + mk_int(a))) | (y >= (x + mk_int(b)));
  }

  void encode() {
    using(Config cfg = new Config()) {
      cfg.SetParamValue(”MODEL”,”true”);
      using(Context ctx = new Context(cfg)) {
        this.ctx = ctx;

        Term t11 = mk_var(”t11”);
        Term t12 = mk_var(”t12”);
        Term t21 = mk_var(”t21”);
        Term t22 = mk_var(”t22”);
        Term t31 = mk_var(”t31”);
        Term t32 = mk_var(”t32”);
        ctx.AssertCnstr(mk_precedence(t11, t12, 2, 1));
        ctx.AssertCnstr(mk_precedence(t21, t22, 3, 1));
        ctx.AssertCnstr(mk_precedence(t31, t32, 2, 3));
        ctx.AssertCnstr(mk_resource(t11, t21, 3, 2));
        ctx.AssertCnstr(mk_resource(t11, t31, 2, 2));
        ctx.AssertCnstr(mk_resource(t21, t31, 2, 3));
        ctx.AssertCnstr(mk_resource(t12, t22, 2, 3));
        ctx.AssertCnstr(mk_resource(t12, t32, 3, 1));
        ctx.AssertCnstr(mk_resource(t22, t32, 3, 1));
        Model m = null;
        LBool r = ctx.CheckAndGetModel(out m);
        if (m != null) {
          m.Display(System.Console.Out);
          m.Dispose();
        }
      }
    }
  }

   static void Main() {
      Program p = new Program();
      p.encode();
   }
};
```

5.7 Scheduling using F# quotations

The Z3 distribution comes with power utilities for the F# programming language. A prolific feature of F# is the availability of *quotations*. Quotations have their origins in LISP: you can quote a piece of code and treat it as data. You can also quote code in F#, and access the abstract syntax tree for it. This feature is used for encoding formulas as F# expressions. It makes for quite legible syntax. The scheduling constraints using the quotation support from the Z3 distribution can be formulated as follows:

```
open Microsoft.Z3
open Microsoft.Z3.Quotations

do Solver.prove <@ Logic.declare
      (fun t11 t12 t21 t22 t31 t32 ->
         not
            ((t11 >= 0I) && (t12 >= t11 + 2I) && (t12 + 1I <= 8I) &&
             (t21 >= 0I) && (t22 >= t21 + 3I) && (t32 + 1I <= 8I) &&
             (t31 >= 0I) && (t32 >= t31 + 2I) && (t32 + 3I <= 8I) &&
             (t11 >= t21 + 3I || t21 >= t11 + 2I) &&
             (t11 >= t31 + 2I || t31 >= t11 + 2I) &&
             (t21 >= t31 + 2I || t31 >= t21 + 3I) &&
             (t12 >= t22 + 1I || t22 >= t12 + 1I) &&
             (t12 >= t32 + 3I || t32 >= t12 + 1I) &&
             (t22 >= t32 + 3I || t32 >= t22 + 1I)
            )
       )
    @>
```

Let us explain some of the features used in the example:

- `Solver.prove` consumes an expression of type `Expr<bool>`, a quoted expression of type `bool`. It checks for validity of the formula that results from compiling the expression. Since, we are interested in *satisfiability* of the scheduling constraints we check for validity of their negation.

- `Logic.declare` is a function that takes an arbitrary curried lambda expression and creates fresh constants for the variables that are bound by the lambda expression. In this case, it creates constants for `t11 t12 t21 t22 t31 t32`. The F# type inference will infer that these variables have type `BigInteger`. Z3 represents these as plain integers.
- The notation *big integer* literals in F# is to suffix numbers with an `I`, for example `1I` and `8I`. For "normal" integers, such as `1` and `8`, Z3's quotation compiler uses fixed-size bit-vectors.

5.8 Scheduling in other formats

It is furthermore possible to formulate scheduling constraints using the binary API to *OCaml*, the *Simplify* format that is a legal input format to Z3, and the native low-level format for Z3. The OCaml API follows the C API closely, using the same naming conventions. We don't recommend using these two text APIs for interactive use of Z3. The Simplify format allows tools that have already taken a dependency on Simplify to use Z3, and the native low-level text format allows dumping interactions from the binary APIs to a text file for reproducing potential problems.

6 Configuring Z3

Z3 exposes more than 200 different parameters that allow configuring Z3's search engine to use different heuristics and algorithms. You can list the configuration options from the command-line by calling:

```
z3.exe /ini?
```

6.1 Auto Configuration

AUTO_CONFIG is an option for Z3 that is specific to SMT-LIB benchmarks. SMT-LIB benchmarks are annotated with a *logic*, which indicates the set of theories and symbols that are relevant to the formula. By default, Z3 uses AUTO_CONFIG=true to automatically customize options based on the logic annotation and other structure information of the benchmark.

6.2 Displaying Configuration

DISPLAY_CONFIG=true allows you to retrieve the configuration settings used by Z3 after it completes. This is useful for knowing what AUTO_CONFIG decided to set.

6.3 Updating Configuration

Parameters are configured prior to running Z3 or prior to creating a logical context (where assertions are pushed). A few parameters can also be changed once Z3 is running, or once a Z3 logical context has been created over the API. Z3_update_param_value is the C-interface function for updating parameter values. The .NET method is called UpdateParamValue, and from the smt2 format you can update parameter values using

```
(set-option set-param "<parameter-name>" "<parameter-value>")
```

7 Propositional Solving

Propositional logic is a special case of predicate logic. In propositional logic, formulas are built from *Boolean variables*, called *atoms*, and composed using logical connectives such as conjunction, disjunction and negation. The satisfiability problem for propositional logic is famously known as an NP-complete problem [7], and therefore in principle computationally intractable. Yet, recent advances in efficient propositional logic algorithms have moved the boundaries for what is intractable when it comes to practical applications [24].

Most successful SAT solvers are based on an approach called *systematic search*. The search space is a tree with each vertex representing a Boolean variable and the out edges representing the two choices (i.e., *true* and *false*) for this variable. For a formula containing n Boolean variables, there are 2^n leaves in this tree. Each path from the root to a leaf corresponds to a truth assignment. A *model* is a truth assignment that makes the formula *true*. We also say the model satisfies the formula. Most search based SAT solvers are based on the DPLL approach [8]. The DPLL algorithm tries to build a model using three main operations: decide, propagate and backtrack. The algorithm benefits from a restricted representation of formulas in conjunctive normal form (CNF). CNF formulas are restricted to be conjunctions of *clauses*, each clause is, in turn, a disjunction of *literals*. A literal is an atom or the negation of an atom. For example, the formula $\neg p \wedge (p \vee q)$, is in CNF. The operation decide heuristically chooses an unassigned atom and assigns it to *true* or *false*. This operation is also called *branching* or *case-splitting*. The operation propagate deduces the consequences of a partial truth assignment using deduction rules. The most widely used deduction rule is the *unit-clause rule*, which states that if a clause has all but one literal assigned to *false* and the remaining literal l is unassigned, then the only way for this clause to evaluate to true is to assign l to *true*. Let C be the clause $p \vee \neg q \vee \neg r$, and M the partial truth assignment $\{p \mapsto \mathit{false}, r \mapsto \mathit{true}\}$, then the only way for C to evaluate to *true* is by assigning q to *false*. Given a partial truth assignment M and a clause C in the CNF formula such that all literals of C are assigned to *false* in M, then there is no way to extend M to a complete model M' that satisfies the given formula. We say this is a *conflict*, and C is a *conflicting clause*. A conflict indicates that some of the earlier decisions cannot lead to a truth assignment that satisfies the given formula, and the DPLL procedure must *backtrack* and try a different branch value. If a *conflict* is detected and there are no decisions to backtrack, then the formula is unsatisfiable, that is, it does not have a model. Many significant improvements of this basic procedure have been proposed over the years. The main improvements are: *lemma learning*, *non-chronological backtracking*, efficient *indexing techniques* for applying the unit-clause rule and *preprocessing* techniques [24].

7.1 A Propositional Example

The example on the right encodes satisfiability checking of the formula

$$(p_1 \rightarrow p_2) \wedge (p_1 \rightarrow p_3) \wedge (p_1 \rightarrow p_4) \wedge \neg p_2$$

```
(declare-preds ((p1) (p2) (p3) (p4) (p5)))
(assert (=> p1 p2))
(assert (=> p1 p3))
(assert (=> p1 p4))
(assert (not p2))
(check-sat)
; sat
(model)
; ("model" "p1 -> false
; p2 -> false")
(assert p1)
(check-sat)
; unsat
```

The values of predicates p3 and p4 are not included in the model. They are *dont-cares* and irrelevant to the satisfiability. Z3 uses *relevancy* propagation to discover this dependency. You can disable relevancy propagation by using the configuration RELEVANCY=0 from the command-line. In this case Z3 returns a model which includes an assignment to the other predicates:

```
("model" "p1 -> false
p2 -> false
p3 -> false
p4 -> false")
```

The example on the right also establishes that p1 must be false in all satisfying models, because the formula got unsatisfiable when asserting p1.

8 Relations, Functions and Constants

The basic building blocks of SMT formulas are constants, functions and relations. Constants are just functions that take no arguments. Relations are just functions that return a value of Boolean type. Functions can take arguments of Boolean type as well, so you can nest functions and relations arbitrarily. So everything is really just a function.

We here recall a few facts about functions.

8.1 All functions are total

Unlike programming languages, where functions have side-effects, can throw exceptions, or never return, functions in classical first-order logic are all *total*. That is, they are defined on all input values. This includes functions, such as division. Division by 0 is still defined, yet it is not specified what it means. Any interpretation for division by 0 is admissible. So for example, *x* divided by *x* can be 1 for all *x*, and 0 divided by *x* can be 0 for all *x*. But of course not both at the same time.

Z3 answers `sat` on both checks.

```
  (push)
  (assert (= 1 (div 0 0)))
  (check-sat)
; sat
  (pop)

  (push)
  (assert (= 0 (div 0 0)))
  (check-sat)
; sat
  (pop)
```

8.2 Uninterpreted functions and constants

Function and constant symbols in pure first-order logic are *uninterpreted*, or *free*, which means that no a priori interpretation is attached. This is in contrast to functions belonging to the signature of theories, such as arithmetic where the function + has a fixed standard interpretation (it adds two numbers). Uninterpreted functions and constants are maximally flexible; they allow any interpretation that is consistent with the constraints over the function or constant.

To illustrate uninterpreted functions and constants let us introduce an (uninterpreted) sort `A`, and the constants `x`, `y` ranging over `A`. Finally let `f` be an uninterpreted function that takes one argument of sort `A` and results in a value of sort `A`. The example illustrates how one can force an interpretation where `f` applied twice to `x` results in `x` again, but `f` applied one to `x` is different form `x`.

```
(declare-sort A)
(declare-funs ((x A) (y A)))
(declare-fun f (A) A)
(assert (= (f (f x)) x))
(assert (= (f x) y))
(assert (not (= x y)))
(check-sat)
; sat
(model)
; ("model" "x -> val!0
; y -> val!1
; f -> {
;   val!0 -> val!1
;   val!1 -> val!0
;   else -> val!0
; }")
```

The resulting model introduces abstract values for the elements in `A`, because the sort `A` is uninterpreted. The function graph for `f` in the model toggles between the two values for `x` and `y`, which are different. All other potential values in `A` map to the interpretation of `x`.

8.3 Recursive functions

Z3 does not provide any special support for recursive functions. You can axiomatize the graph of a recursive function by using first-order axioms, but one should be aware of that Z3 assigns standard first-order semantics with the equations and does not assign a least fixed-point solution as is standard with programming languages.

Let us consider the Fibonnachi function. We can axiomatize it using equations.

```
(declare-fun fib (Int) Int)
(assert (= 1 (fib 0)))
(assert (= 1 (fib 1)))
(assert (forall (x Int) (=> (x >= 2) (= (fib x) (+ (fib (- x 1) (- x 2)))))))
```

What is `(fib (~ 1))`? It is not undefined, it is just not specified by these equations.

The fact that axiomatizing a recursive function as a set of equations does not necessarily capture the semantics of functions can sometimes be confusing. Consider for example, the predicate `IsNat`. It returns `true` on the value 0, and if it is true on `x` then it is true on `x + 1`. If we ask Z3 whether -1 satisfies `IsNat`, then Z3 answers `unknown` (because the input has quantifiers, and even worse it integrates arithmetic, and Z3 is not a decision procedure in this case). It provides a model where the numbers 0 to 100 satisfy `IsNat`, but also -1 satisfies `IsNat`. While the model does not correctly provide a graph for `IsNat` on values greater than 100, there is no contradiction with having `IsNat` hold on -1. The point is that *any* fixed-point that satisfies the equations in the axioms is a legal first-order interpretation for `IsNat`.

```
(declare-fun IsNat (Int) Bool)
(assert (IsNat 0))
(assert (forall (x Int)
     (iff (IsNat (+ x 1))
          (or (= x 0) (IsNat x)))))
(assert (IsNat (~ 1)))
(check-sat)
; unknown
(model)
;("model" "IsNat -> {
;  0 -> true
;  -1 -> true
;  1 -> true
;  2 -> true
;  ...
;  100 -> true
;  101 -> true
;  else -> false
;}")
```

9 Arithmetic

Z3 contains decision procedures for linear arithmetic over the integers and real numbers. It furthermore contains some facilities for partial support of non-linear arithmetic using a module for Gröbner basis completion. Additional material on the main arithmetic decision procedure used in Z3 is available in [15].

9.1 Real linear arithmetic

The terminology and notation of SMT-LIB will be convenient for describing the support for linear arithmetic.

Linear arithmetic terms of type Real are formed using the functions +, -, ˜ (unary minus), * where all but one argument is a numeric constant, and / where the second argument is a numeric constant. You can compare terms using =, <, <=, >=, >.

The second example on the right uses constraints of a special form: there are at most two variables per inequality, and they appear on different sides of the inequality. This fragment of arithmetic is called *difference arithmetic* or most often called *difference logic*. You can instrument Z3 to use specialized solvers for difference arithmetic using the option QF_RDL and QF_IDL (for quantifier free integer/real difference logic). The introductory example from Section 5 used difference arithmetic.

```
(declare-funs ((x Real) (y Real) (z Real)))
(push)
(assert (> (+ x y) (* 2.0 z)))
(assert (< (/ z 2.3) x))
(check-sat)
; sat
(model)
; ("model" "x -> 0
; y -> -18/5
; z -> -23/10")
(pop)
(assert (> x 2.0))
(assert (>= y x))
(assert (< y 1.3))
(check-sat)
; unsat
```

Z3 also accepts formulas over the reals that are non-linear. In this case, Z3 is not a decision procedure even for quantifier-free formulas. Nevertheless, it can handle several special cases of non-linear constraints over the reals by using simplification using Gröbner bases.

9.2 Integer linear arithmetic

In the terminology of SMT-LIB, terms over integer linear arithmetic are formed using the functions +, -, ˜ (unary minus), * where one argument is a numeric constant, and div, mod, rem where the second argument is a numeric constant different from 0. You can compare terms using =, <, <=, >=, >.

The binary APIs expose corresponding functions. In the terminology of the C-API these are: Z3_mk_add (+), Z3_mk_mul (*), Z3_mk_sub (-), Z3_mk_unary_minus (˜) Z3_mk_div (div and /), Z3_mk_mod (mod), Z3_mk_rem (rem), Z3_mk_lt (<), Z3_mk_le (<=), Z3_mk_gt (>), Z3_mk_ge (>=).

```
(declare-funs ((x Int) (y Int) (z Int)))
(push)
(assert (> (+ x y) (* 2 z)))
(assert (< (div z 3) x))
(check-sat)
; sat
(model)
; ("model" "x -> 1
; y -> 0
; z -> 0")
(pop)
(assert (and (> x 2) (>= y x) (< y 1))
(check-sat)
; unsat
```

9.3 Mixed linear arithmetic

You can create formulas over mixed integer and linear arithmetic by means of the conversion functions, that appear in the smt2 standard:

```
   (declare-fun to_real (Int)  Real)
   (declare-fun to_int  (Real) Int)
   (declare-fun is_int  (Real) Bool)
```

For example,

```
   (= 4.0 (to_real 4))
   (= 4 (to_int 4.5))
   (iff (is_int x) (= x (to_real (to_int x))))
```

9.4 Non-linear arithmetic

Consider the following equalities

```
  (declare-funs ((x Int) (y Int) (z Int)))
  (assert (= (* x x) (+ x 2)))
  (assert (= (* x y) x))
  (assert (= (* (- y 1) z) 1))
  (check-sat)
; unsat
```

Z3 determines that the equalities are unsatisfiable. To accomplish this, Z3 relies on a Gröbner basis completion of the equalities. The completion deduces from the first equation that x is different from 0, and therefore y must be 1 in the second equation. This contradicts the last equation. You can control the use of non-linear arithmetic in Z3 using the configuration options starting with NL_ARITH_*.

9.5 Quantifier Elimination for Linear Arithmetic

Quantified linear arithmetic formulas admit quantifier elimination. Z3 includes quantifier elimination procedures for linear arithmetic over the reals and integers (but not yet mixed linear arithmetic). For example,

```
(set-option set-param "ELIM_QUANTIFIERS" "true")
(simplify (forall (x Int) (exists (y Int) (> y (+ x 2)))))
; true
(simplify (forall (x Int) (> 0 (+ x 2))))
; false
(simplify
   (exists (l Int)
    (forall (x Int)
     (implies (>= x l)
      (exists (u Int) (v Int)
       (and (>= u 0) (>= v 0) (= x (+ (* 3 u) (* 5 v)))))))))
; true
```

The last problem asks to establish that there is a lower bound l, such that every x above l can be composed as a linear positive combination of 3 and 5. More background on the quantifier-elimination procedure used in Z3 is available in [3].

10 Data-types

Algebraic data-types, known from programming languages such as ML, offer a convenient way for specifying common data-structures. Records and tuples are special cases of algebraic data-types, and so are scalars (enumeration types). But algebraic data-types are more general. They can be used to specify finite lists, trees and other recursive structures.

10.1 Records

A record is specified as a data-type with a single constructor and as many arguments as record elements. The number of arguments to a record are always the same. The type system does not allow to extend records and there is no record sub-typing.

The following example illustrates that two records are equal only if all the arguments are equal. It introduces the type `int-pair`, with constructor `mk-pair` and two arguments that can be accessed using the selector functions `first` and `second`.

```
(declare-datatypes ((int-pair (mk-pair (first Int) (second Int)))))
(declare-funs ((p1 int-pair) (p2 int-pair)))
(push)
(assert (= p1 p2))
(assert (not (= (first p1) (first p2))))
(check-sat)
;unsat
(pop)
```

Just for the record, the same example, when entered in a self-contained C program looks as follows

```
#include ”z3.h”

int main() {
    Z3_config cfg = Z3_mk_config();
    Z3_context ctx = Z3_mk_context(cfg);

    Z3_symbol mk_pair = Z3_mk_string_symbol(ctx, ”mk−pair”);
    Z3_symbol field_names[2] = { Z3_mk_string_symbol(ctx, ”first”),
                                 Z3_mk_string_symbol(ctx, ”second”) };
    Z3_sort field_sorts[2] = { Z3_mk_int_sort(ctx), Z3_mk_int_sort(ctx) };
    Z3_func_decl mk_tuple;
    Z3_func_decl field_selects[2] = { 0, 0 };

    Z3_sort int_pair = Z3_mk_tuple_sort(ctx, mk_pair,
                                        2,
                                        field_names,
                                        field_sorts,
                                        &mk_tuple,
                                        field_selects);
    Z3_ast p1 = Z3_mk_const(ctx, Z3_mk_string_symbol(ctx, ”p1”), int_pair);
    Z3_ast p2 = Z3_mk_const(ctx, Z3_mk_string_symbol(ctx, ”p2”), int_pair);
    Z3_func_decl first = field_selects[0];

    Z3_assert_cnstr(ctx, Z3_mk_eq(ctx, p1, p2));
    Z3_assert_cnstr(ctx, Z3_mk_not(ctx,
                              Z3_mk_eq(ctx,
                                       Z3_mk_app(ctx, first, 1, &p1),
                                       Z3_mk_app(ctx, first, 1, &p2))));

    Z3_bool r = Z3_check(ctx);
    if (r == Z3_L_FALSE) {
        printf(”it␣is␣unsatisfiable␣as␣expected\n”);
    }
    Z3_del_context(ctx);
    Z3_del_config(cfg);
}
```

10.2 Scalars (enumeration types)

A scalar sort is a finite domain sort. The elements of the finite domain are enumerated as distinct constants. For example, the sort `S` is a scalar type with three values `A`, `B` and `C`. It is possible for three variables of sort `S` to be distinct, but not for four variables.

```
(declare-datatypes ((S (A) (B) (C))))
(declare-funs ((x S) (y S) (z S) (u S)))
(assert (distinct x y z))
(check-sat)
;sat
(assert (distinct x y z u))
(check-sat)
;unsat
```

The binary API contains a shorthand for declaring scalar sorts. It is called `Z3_mk_enumeration_sort`.

10.3 Recursive data-types

A recursive data-type declaration includes itself directly (or indirectly) as a component. A standard example of a recursive data-type is the one of lists. An integer list can be specified in Z3's smt2 front-end as:

```
 (declare-datatypes ((list (nil) (cons (hd Int) (tl list)))))
```

Recursive data-types are also uniquely determined by their arguments.

```
  (declare-funs  ((l1 list) (l2 list)))
  (push)
  (assert (not (= l1 nil)))
  (assert (not (= l2 nil)))
  (assert (= (hd l1) (hd l2)))
  (assert (= (tl l1) (tl l2)))
  (assert (not (= l1 l2)))
  (check-sat)
  ; unsat
  (pop)
```

Notice that we also assert that `l1` and `l2` are not `nil`. This is because the interpretation of `hd` and `tl` is under-specified on `nil`. So then head (`hd`) and tail (`tl`) would not be able to distinguish `nil` from `(cons (hd nil) (tl nil))`.

10.4 Mutually recursive data-types

You can also specify mutually recursive data-types for Z3. We list one example below.

```
(declare-datatypes ((opt (None) (Some (list olist)))
                    (olist (Nil) (Cons (ohd Int) (otl opt)))))
```

You cannot nest recursive data-type definitions inside other types, such as arrays. So the following declaration is not accepted by Z3:

```
(declare-datatypes
  ((Unsupported (mk-rec-array (hd (Array Int Unsupported))))))
```

10.5 You will not get Z3 to prove Inductive facts

The ground decision procedures for recursive data-types don't lift to establishing inductive facts. Z3 does not contain methods for producing proofs by induction. In particular, consider the following example where the predicate p is true on all natural numbers, which can be proved by induction over Nat. Z3 enters a matching loop as it attempts instantiating the universally quantified implication.

```
(declare-datatypes ((Nat zero (succ (pred Nat)))))
(declare-preds ((p Nat)))
(assert (p zero))
(assert (forall (?x Nat) (implies (p (pred ?x)) (p ?x))))
(assert (not (forall (?x Nat) (p ?x))))
(check-sat)
```

11 Bit-vectors

Modern CPUs and main-stream programming languages use arithmetic over fixed-size bit-vectors. The theory of bit-vectors allows modeling the precise semantics of unsigned and of signed two-complements arithmetic. There are a large number of supported functions and relations over bit-vectors. They are summarized on Z3's on-line documentation `http://research.microsoft.com/projects/z3` of the binary APIs and they are summarized on the SMT-LIB web-site `http://www.smtlib.org`. We will not try to give a comprehensive overview here, but touch on some of the main features.

In contrast to programming languages, such as C, C++, C#, Java, there is no distinction between signed and unsigned bit-vectors as numbers. Instead, the theory of bit-vectors provides special signed versions of arithmetical operations where it makes a difference whether the bit-vector is treated as signed or unsigned.

11.1 Basic Bit-vector Arithmetic

```
(declare-funs ((x BitVec[32]) (y BitVec[32]) (z Int)))
(define x (bvadd x y))  ; addition
(define x (bvsub x y))  ; subtraction
(define x (bvneg x))    ; unary minus
(define x (bvmul x y))  ; multiplication
(define x (bvurem x y)) ; unsigned remainder
(define x (bvsmod x y)) ; signed modulo
(define x (bvshl x y))  ; shift left
(define x (bvlshr x y)) ; unsigned (logical) shift right
(define x (bvashr x y)) ; signed (arithmetical) shift right
```

Let us illustrate a simple property of bit-wise arithmetic. There is a fast way to check that fixed size numbers are powers of two. It turns out that a bit-vector `x` is a power of two or zero if and only if `x + (x - 1)` is zero. We check this for four bits below.

```
  (define-fun is-power-of-two ((x BitVec[4])) Bool
               (= bv0[4] (bvand x (bvsub x bv1[4]))))
  (declare-funs ((a BitVec[4])))
  (push)
  (assert
     (not (iff (is-power-of-two a)
               (or (= a bv0[4]) (= a bv1[4])
                   (= a bv3[4]) (= a bv4[4]) (= a bv8[4])))))
  (check-sat)
; sat
  (model)
; ("model" "a -> bv2[4]")
  (pop)
```

Ups! There was a typo, we should have written 2 instead of 3.

```
  (assert
     (not (iff (is-power-of-two a)
               (or (= a bv0[4]) (= a bv1[4])
                    (= a bv2[4]) (= a bv4[4]) (= a bv8[4])))))
  (check-sat)
; unsat
```

Better!

11.2 Bit-wise Operations

```
(define x (bvand x y))  ; bit-wise and
(define x (bvor x y))   ; bit-wise or
(define x (bvnot x))    ; bit-wise not
(define x (bvnand x y)) ; bit-wise nand
(define x (bvnor x y))  ; bit-wise nor
(define x (bvxnor x y)) ; bit-wise xnor
```

We can prove a bit-wise version of de-Morgan's law:

```
(declare-funs ((x BitVec[64]) (y BitVec[64])))
(assert (not (= (bvand (bvnot x) (bvnot y)) (bvnot (bvor x y)))))
(check-sat)
; unsat
```

11.3 Predicates over Bit-vectors

11.3.1 Comparison

```
(define a (bvule x y))  ; unsigned less or equal
(define a (bvult x y))  ; unsigned less than
(define a (bvuge x y))  ; unsigned greater or equal
(define a (bvugt x y))  ; unsigned greater than
(define a (bvsle x y))  ; signed less or equal
(define a (bvslt x y))  ; signed less than
(define a (bvsge x y))  ; signed greater or equal
(define a (bvsgt x y))  ; signed greater than
```

Signed comparison, such as bvsle, takes the sign bit of bit-vectors into account for comparison, while unsigned comparison treats the bit-vector as unsigned (treats the bit-vector as a natural number).

```
  (declare-funs ((a BitVec[4]) (b BitVec[4])))
  (assert (not (iff (bvule a b) (bvsle a b))))
  (check-sat)
; sat
  (model)
; ("model" "a -> bv9[4]
; b -> bv0[4]")
  (eval (bv2int[Int] a))
; 9
  (eval (bv2int[Int] b))
; 0
  (define-fun bv2signed ((x BitVec[4])) Int
      (let ((xI (bv2int[Int] x)))
      (ite (bvsge x bv0[4]) xI (- xI 16))))
  (eval (bv2signed a))
; (- 0 7)
  (eval (bv2signed b))
; 0
```

11.3.2 Overflow Checks

Z3 exposes special predicates to check for the absence of unsigned multiplication overflows and check for the absence of signed multiplication overflows and underflows. The predicates take two bit-vectors of the same length and return true if no overflow or underflow occur.

```
  (define a (bvumul_noovfl x y))
  (define a (bvsmul_noovfl x y))
  (define a (bvsmul_noudfl x y))
```

11.3.3 Bit-wise operations

```
 (define a (bvredor x))  ; or-reduction
 (define a (bvredand x)) ; and-reduction
```

11.4 Conversions between Bit-vectors and Integers

Z3 exposes conversion functions between bit-vectors and integers.

```
(define b (int2bv[32] z))  ; Convert an integer to a 32-bit bit-vector
(define c (bv2int[Int] x)) ; Convert an (unsigned) bit-vector to an integer
```

It is possible, but expensive, to integrate the theory of integers with bit-vectors. It is therefore turned off by default in Z3. You can enable it by setting the configuration parameter `BV_ENABLE_INT2BV_PROPAGATION` to `true`.

12 Arrays

As part of formulating a programme of a mathematical theory of computation McCarthy [25] proposed a *basic* theory of arrays as characterized by the select-store axioms

$$\forall A, i, v \; . \; store(A,i,v)[i] \simeq v$$

and

$$\forall A, i, j, v \; . \; i \simeq j \lor store(A,i,v)[j] \simeq A[j].$$

We wrote $A[i]$ instead of the SMT-LIB syntax `(select` A i`)`. Z3 contains a decision procedure for the basic theory of arrays. By default, Z3 assumes that arrays are extensional over select. In other words, Z3 also enforces that if two arrays agree on all reads, then the arrays are equal:

$$\forall A, B \; . \; (\forall i \; . \; A[i] \simeq B[i]) \;\rightarrow\; A \simeq B \; .$$

It also contains various extensions for operations on arrays that remain decidable and amenable to efficient saturation procedures (here efficient means, with an NP-complete satisfiability complexity). We describe these extensions in the following using a collection of examples. Additional background on these extensions is available in [10].

12.1 Select and Store

Let us first check a basic property of arrays. Suppose a_1 is an array of integers, then the constraint

$$a_1[x] \simeq x \land store(a_1, x, y) \simeq a_1$$

is satisfiable for an array that contains an index x that maps to x, and when $x = y$ (because the first equality forced the range of x to be x). We can check this constraint.

```
  (define-sorts ((A (Array Int Int))))
  (declare-funs ((x Int) (y Int) (z Int)))
  (declare-funs ((a1 A) (a2 A) (a3 A)))
  (push)
  (assert (= (select a1 x) x))
  (assert (= (store a1 x y) a1))
  (check-sat)
; sat
  (get-info model)
; (("model" "x -> 0
; a1 -> (store (const 1) 0 0)
; y -> 0"))
```

On the other hand, the constraints become unsatisfiable when asserting $x \not\simeq y$.

```
  (assert (not (= x y)))
  (check-sat)
; unsat
  (pop)
```

12.2 Constant Arrays

The array that maps all indices to some fixed value can be specified in Z3 using the `const[`*ArrayType*`]` construct. It takes one value from the range type of the array and creates an array. The *ArrayType* parameter helps Z3's type checker to infer the correct type for the constant array. The constant arrays satisfy the axiom

$$\forall v,\ i\ .\ (\mathit{const}[A]v)[i] \simeq v\ .$$

So if we try to select an arbitrary index from the constant array, we get the fixed value back.

```
  (define all1_array (const[A] 1))
  (simplify (select all1_array x))
; 1
```

12.3 Array models

Models provide interpretations of the variables (constants) and functions that appear in the satisfiable formula. An interpretation for arrays consists of a finite number of key-value pairs together with a default value such that everything that does is not mentioned in the finite set of key-value pairs maps to the default value.

Schematically, interpretations for arrays are written in the form

```
(store (store .. (const[A] <val-0>) <key-1> <val-1>) .. <key-n> <val-n>)))
```

The term `(const[A] <val-0>)` is an array that maps all indices to `<val-0>`. The finite set of key-val pairs `<key-1> <val-1>, .., <key-n> <val-n>` represent the keys where the interpretation different values for the specified keys.

12.4 Mapping Functions on Arrays

In the following, we will simulate basic Boolean algebra (set theory) using the array theory extensions in Z3. Z3 provides a parametrized *map* function on arrays. It allows applying arbitrary functions to the range of arrays. The map functions satisfy the axioms

$$\begin{array}{l}
\forall A, i\ .\ \mathit{map}[f](A)[i] \simeq f(A[i]) \\
\forall A, B, i\ .\ \mathit{map}[f](A,B)[i] \simeq f(A[i],B[i]) \\
\forall A, B, C, i\ .\ \mathit{map}[f](A,B,C)[i] \simeq f(A[i],B[i],C[i]) \\
\ldots
\end{array}$$

for the cases where f is unary, binary, ternary, or generally n-ary. The advantage of using the map function is of course that the term $\mathit{map}[f](A)$ can be used without accessing the resulting array at the index i.

In the SMT-LIB syntax, the function parameter to *map* should be an uninterpreted function symbol. So if we want to map `and`, `or` and `not` on Boolean arrays, we will need to define auxiliary functions with the same interpretation as these logical connectives.

```
  (define-sorts ((IntSet (Array Int Bool))))
  (declare-funs ((and_fn Bool Bool Bool)
                 (or_fn Bool Bool Bool)
                 (not_fn Bool Bool)))
  (declare-funs ((a IntSet) (b IntSet) (c IntSet)))
  (assert (forall (x Bool) (y Bool)
                  (iff (and_fn x y) (and x y))))
  (assert (forall (x Bool) (y Bool)
                  (iff (or_fn x y)  (or x y))))
  (assert (forall (x Bool)
                  (iff (not_fn x) (not x))))
```

Let us check that

$$a \cap b \;=\; \overline{\overline{b} \cup \overline{a}}$$

```
  (push)
  (assert
   (not
    (= (map[and_fn] a b)
       (map[not_fn] (map[or_fn] (map[not_fn] b) (map[not_fn] a)
    )))))
  (check-sat)
; unsat
  (pop)
```

For convenience, Z3 exposes shorthands for set operations, so the same example can be written in a much more readable way:

```
  (push)
  (assert
   (not
    (= (intersect a b)
       (complement (union (complement b) (complement a))))))
  (check-sat)
; unsat
  (pop)
```

We can also check facts about set membership. The `select` function simulates set membership.

$$x \in (a \cap b) \;\rightarrow\; x \in a$$

```
  (push)
  (assert (select (map[and_fn] a b) x))
  (assert (not (select a x)))
  (check-sat)
; unsat
  (pop)
```

It is of course *not* the case that

$$x \in (a \cup b) \;\rightarrow\; x \in a$$

```
  (push)
  (assert (select (map[or_fn] a b) x))
  (assert (not (select a x)))
  (check-sat)
; unknown
```

```
  (get-info model)
; (("model" "a -> (const false)
; x -> 0
; b -> (store (const false) 0 true)
; or_fn -> {
;   false false -> false
;   false true -> true
;   else -> false
; }"))
```

while it is the case that

$$x \in (a \cup b) \quad \rightarrow \quad x \in a \vee x \in b$$

```
  (assert (and (not (select b x))))
  (check-sat)
  (pop)
```

12.5 Default array values

The function `default[`*ArrayType*`]` takes as argument an array *a* and returns a value from the range. Z3 ensures that in the default value used in models for the array *a* are equal to (`default[A]` *a*).

The following example illustrates constraints that force `a1` to be different from the array that maps all keys to 1, yet, the default value is constrained to be 1. In other words, `a1` must map some key to a value different from 1. Z3 selects the key arbitrary to be 0, and the value to be 2 in the model.

```
  (push)
  (assert (= (default[A] a1) 1))
  (assert (not (= a1 (const[A] 1))))
  (check-sat)
  (model) ; short for (get-info model)
; ("model" "a1 -> (store (const 1) 0 2)")
```

More than one array can have the same default value.

```
  (assert (= (default[A] a2) 1))
  (assert (not (= a1 a2)))
  (check-sat)
  (model)
; ("model" "a1 -> (store (store (const 1) 0 2) 3 4)
; a2 -> (store (const 1) 3 5)")
  (pop)
```

12.6 Bags as Arrays

We can use the parametrized map function together with the default accessor to encode finite sets and finite bags. Finite bags can be modeled similarly to sets. A bag is here an array that maps elements to their multiplicity. Main bag operations include *union*, obtained by adding multiplicity, *intersection*, by taking the minimum multiplicity, and a dual *join* operation that takes the maximum multiplicity. The bag operations

```
(declare-sort A)
(define-sorts ((ABag (Array A Int))))
(declare-funs ((bag_min Int Int Int)
               (bag_or Int Int Int)
               (bag_max Int Int Int)))
(declare-funs ((a ABag) (b ABag) (c ABag)))
(assert (forall (x Int) (y Int)
                (= (bag_min x y) (ite (< x y) x y))))
(assert (forall (x Int) (y Int)
                (= (bag_max x y) (ite (< x y) y x))))
(assert (forall (x Int) (y Int)
                (= (bag_or x y)  (+ x y))))
```

We can then use `default` to enforce that finite bags map everything but a finite set of integers to 0.

```
(assert (= (default[ABag] a) 0))
(assert (= (default[ABag] b) 0))
(assert (= (default[ABag] c) 0))
```

12.7 Summary of Array operations

Let us summarize the array operations available in Z3 (using smt2 syntax). We use `A` as a name for `(Array I V)`, `A1` for `(Array I V1)`, `A2` for `(Array I V2)` and `SetI` as a name for `(Array I Bool)`.

Usage	Signature	Description
`(select i)`	`A I V`	selects contents at index `i`
`(store a i v)`	`A I V A`	produces array where contents of index `i` is updated to `v`
`(const[A] v)`	`V A`	produces the constant array. All indices map to `v`
`(default[A] a)`	`A V`	selects an default value for the array. It complements `const`
`(map[f] a b ..)`	`A1 A2 .. A`	maps function `f` on the range of `a b ..`
`(union a b)`	`SetI SetI SetI`	creates the union of two arrays as sets
`(intersect a b)`	`SetI SetI SetI`	creates the intersection of two sets
`(difference a b)`	`SetI SetI SetI`	creates the intersection of two sets
`(complement a)`	`SetI SetI`	creates the complement of a set

13 Quantifiers

Apart from linear quantifier-elimination introduced in Section 9.5 and in connection with recursive function axioms 8.3, all formulas have so far been *quantifier-free*. Z3 is a *decision procedure* for the combination of the previous quantifier-free theories. That is, it can answer whether a quantifier-free formula, modulo the theories referenced by the formula, is satisfiable or whether it is unsatisfiable. Z3 also accepts and can work with formulas that use quantifiers. It is no longer a decision procedure for such formulas in general (and for good reasons, as there can be no decision procedure for first-order logic).

Nevertheless, Z3 is often able to handle formulas involving quantifiers. It uses two main approaches to handle quantifiers. The most prolific approach is using *pattern-based* quantifier instantiation (Section 13.2). This approach allows instantiating quantified formulas with ground terms that appear in the current search context based on *pattern annotations* on quantifiers. The second approach is based on *saturation theorem proving* using a superposition calculus which is a modern method for applying resolution style rules with equalities. Section 13.4 introduces this component. The pattern-based instantiation method is quite effective, even though it is inherently incomplete. The saturation based approach is complete for pure first-order formulas, but does not scale as nicely and is harder to predict.

Besides the two main quantifier engines, Z3 also contains a model-based quantifier instantiation component (Section 13.5) that uses a model construction to find good terms to instantiate quantifiers with; and Z3 also handles the array property fragment [5], described in Section 13.6.

13.1 Modeling with Quantifiers

Suppose we want to model an object oriented type system with single inheritance. We would need a predicate for sub-typing. Sub-typing should be a partial order, and respect single inheritance. For some built-in types, such as for `List`, sub-typing should be monotone. Figure 4 axiomatizes the sub-typing relationship using first order quantifiers.

$$\begin{array}{l}
(\forall x\colon \mathit{sub}(x,x)) \\
(\forall x,y,z\colon \mathit{sub}(x,y) \wedge \mathit{sub}(y,z) \rightarrow \mathit{sub}(x,z)) \\
(\forall x,y\colon \mathit{sub}(x,y) \wedge \mathit{sub}(y,x) \rightarrow x = y) \\
(\forall x,y,z\colon \mathit{sub}(x,y) \wedge \mathit{sub}(x,z) \rightarrow \mathit{sub}(y,z) \vee \mathit{sub}(z,y)) \\
(\forall x,y\colon \mathit{sub}(x,y) \rightarrow \mathit{sub}(\mathit{List}(x),\mathit{List}(y)))
\end{array}$$

Figure 4: Axioms for *sub*

The axioms are rewritten as smt2 assertions in Figure 5. In the following, we describe how these axioms can be further tailored to work in the context of the SMT solver Z3.

13.2 Patterns

The Stanford Pascal verifier and the subsequent Simplify theorem prover [14] pioneered the use of pattern-based quantifier instantiation. The basic idea behind pattern-based quantifier instantiation is in a sense straight-forward: Annotate a quantified formula using a *pattern* that contains all the bound variables. So a pattern is a term (that does not contain binding operations, such as quantifiers) that contains variables bound by a quantifier. Then instantiate the quantifier whenever a term that matches the pattern is created during search. This is a conceptually easy starting point, but there are several subtleties that are important.

```
(declare-sort Type)
(declare-fun subtype (Type Type) Bool)
(delcare-fun List (Type) Type)
(assert (forall (x Type) (subtype x x)))
(assert (forall (x Type) (y Type) (z type)
          (=> (and (subtype x y) (subtype y z))
              (subtype x z))))
(assert (forall (x Type) (y Type)
          (=> (and (subtype x y) (subtype y x))
              (= x y))))
(assert (forall (x Type) (y Type) (z type)
          (=> (and (subtype x y) (subtype x z))
              (or (subtype y z) (subtype z y)))))
(assert (forall (x Type) (y Type)
          (=> (subtype x y)
              (subtype (List x) (List y)))))
```

Figure 5: Axioms for *sub*

For example, if we annotate the last axiom from Figure 5 with the following pattern (and Z3's automatic pattern inference algorithm might very well do so):

```
(assert (forall (x Type) (y Type)
         (=> (subtype x y) (subtype (List x) (List y)))
         :pat { (subtype x y) }))
```

the axiom gets instantiated whenever there is some ground term of the form `(subtype s t)`. The instantiation causes a fresh ground term `(subtype (List s) (List t))`, which enables a new instantiation. This undesirable situation is called a *matching loop*. It could be tempting to use the alternative pattern annotation

```
(assert (forall (x Type) (y Type)
         (=> (subtype x y) (subtype (List x) (List y)))
         :pat { (subtype (List x) (List y)) }))
```

but this annotation does not admit instantiate all relevant instances of the axioms. Take the following example of assertions that are unsatisfiable in the context of the axioms:

```
  (declare-funs ((a Type) (b Type) (c Type) (d Type)))
  (assert (and (subtype a (List b)) (subtype b c) (subtype (List c) d)))
  (assert (not (subtype a d)))
```

Unfortunately, the missing link for the transitive closure axiom `(subtype (List b) (List c))` does not get instantiated using this pattern annotation. You will be better of splitting the pattern into two patterns. One that binds `x` and another that binds `y`. This is called a *multi-pattern*.

```
(assert (forall (x Type) (y Type)
         (=> (subtype x y) (subtype (List x) (List y)))
         :pat { (List x) (List y) }))
```

Before elaborating on the subtleties, we should address an important first question. What defines the terms that are created during search? In the context of most SMT solvers, and of the Simplify theorem prover, terms exist as part of the input formula, they are of course also created by instantiating quantifiers, but terms are also implicitly created when equalities are asserted. The last point means that terms are considered up to congruence and pattern matching takes place modulo ground equalities. We call the matching problem *E-matching*. For example, if we have the following equalities:

```
(declare-funs ((f Int Int) (g Int Int) (a Int) (b Int) (c Int)))
(assert (= (g (g b)) b))
(assert (= (f b) c))
(forall (x Int) (= (f (g (g x))) x) :pat { (f (g (g x))) })
```

then the terms `c`, `(f b)`, `(f (g (g b)))`, `(f (g (g (g (g b)))))`, etc. are equal modulo the equalities. The pattern (f (g (g x))) can be matched and `x` bound to `b` (and the equality `(= b c)` is deduced).

While E-matching is an NP-complete problem, the main sources of overhead in larger verification problems comes from matching thousands of patterns in the context of an evolving set of terms and equalities. Z3 integrates an efficient E-matching engine [11] using term indexing techniques.

13.2.1 An operational context of pattern-based instantiation

The context where E-matching is used is inherently operational. We will therefore here review the basic setting of the DPLL(T) search and how quantifiers are instantiated. We will call DPLL(T) with quantifiers DPLL(QT) (obviously pronounced cute) as it exemplifies a plain combination of quantifier instantiation with propositional search.

In DPLL(QT), all maximal sub-formulas that use a quantifier are replaced by a fresh propositional variable. So we rewrite the formula $\varphi[\forall x.\psi]$ where the sub-formula $\forall x.\psi$ occurs instead of as $\varphi[p_{\forall x.\psi}]$.

In the context of Z3, we can furthermore assume an optimization: that the sub-formulas that are rewritten in this way are all positive sub-formulas with universal quantifiers, because we can replace existential quantifiers by constants without changing the satisfiability of the formula (in other words, we can *Skolemize* the formula prior to creating the propositional abstraction).

Suppose that DPLL(QT) sets $p_{\forall x.\psi}$ to *true*, then any model M for $\varphi[p_{\forall x.\psi}]$ extends to a model of $\varphi[\forall x.\psi]$ if it satisfies $\psi[t/x]$ for every ground term t. In other words, we can add the axioms

$$p_{\forall x.\psi} \;\rightarrow\; \psi[t/x]$$

for any ground term t to enforce this property. Since we assume that p occurs with positive polarity, we don't need to bother if DPLL(QT) sets $p_{\forall x.\psi}$ to *false*.

13.2.2 Pattern and multi-pattern annotations

So, which ground terms should be used for these axioms? This is where E-matching is used. To identify the terms to substitute for x, E-matching relies on a pattern annotated with the quantified formula. Reasonable patterns associated with the McCarthy array axioms are provided in curly braces after the binding:

$$\forall A,i,v.\{write(A,i,v)\}\ .\ read(write(A,i,v),i) = v \tag{1}$$

$$\forall A,i,v.\{read(write(A,i,v),j)\}\ .\ i = j\ \vee\ read(write(A,i,v),j) = read(A,j) \tag{2}$$

These two patterns are *reasonable*, but they turn out to not be complete (not even for non-extensional arrays). The following *multi-pattern*, it consists of two terms that must be matched (in order to bind all the quantified variables), completes the picture for the McCarthy array theory (without extensionality).

$$\forall A,i,v.\{write(A,i,v), read(A,j)\}\ .\ i = j\ \vee\ read(write(A,i,v),j) = read(A,j) \tag{3}$$

This axiom easily leads to an explosion in the number of instances because it can combine a cross-product of array accesses to A with a set of independent updates to it. Equation (2) is less harmful, even though it could create $n \cdot m$ terms, when there are n nested writes, accessed by m independent reads. We illustrate the "harmful" example below.

```
(declare-fun A () (Array Int Int))
(declare-funs ((i_1 Int) ... (i_n Int)))
(assert
 (let (x (store A i_1 1))
  ..
  (let (x (store x i_n n))
      (= 0 (+ (select x 1) .. (select x m))))))
```

13.2.3 Injective functions

Another prime example where a multi-pattern causes harm is in the context of axiomatizing injectivity. The function `f` is injective if:

```
 (declare-sorts (A B))
 (declare-fun f (A) B)
 (assert (forall (x A) (y A) (=> (= (f x) (f y)) (= x y))))
```

The straight-forward pattern annotation of this axiom is:

```
 (assert (forall (x A) (y A)
                 (=> (= (f x) (f y)) (= x y))
          :pat { (f x) (f y) }))
```

Thus, the axiom is instantiated for every pair of occurrences of `f`. A simple trick allows formulating injectivity of `f` in such a way that only a linear number of instantiations is required. The trick is to realize that `f` is injective if and only if it has a partial inverse.

```
 (declare-fun f-inv (B) A)
 (assert (forall (x A) (= x (f-inv (f x))) :pat { (f x) }))
```

13.2.4 No-patterns

The annotation `:nopat` can be used to instrument Z3 not to use a certain sub-term as a pattern. The pattern inference engine may otherwise choose arbitrary sub-terms as patterns to direct quantifier instantiation.

For example,

```
 (declare-fun g (B) A)
 (declare-fun f (A) B)
 (assert (forall (x A) (= x (g (f x))) :nopat { (f x) }))
```

causes Z3 to use the pattern `(g (f x))` instead of `(f x)`, which is annotated as a `:nopat`.

13.2.5 Programming with Triggers

The following section uses examples from the paper [26]. The paper is based on several practical experiences with using pattern-based quantifier-instantiation in the context of the Verifying C Compiler project. The examples are quite illustrative.

Consider the axiom

$$(A \setminus B) \setminus C \;=\; A \setminus (B \cup C)$$

if we for a moment disregard the encoding of sets into arrays (as described in Section 12.4), we may choose to axiomatize the equality as a rewrite from left to right, by specifying the left hand side as a pattern.

```
(declare-sort Set)
(declare-funs ((sub Set Set Set) (cup Set Set Set)))
(assert (forall (A Set) (B Set) (C Set)
              (= (sub (sub A B) C) (sub A (cup B C)))
         :pat { (sub (sub A B) C) }
        ))
```

We can now check the theorem on nested terms. Three nested occurrences of `sub` requires four quantifier instantiations.

```
(declare-funs ((a1 Set) (a2 Set) (a3 Set) (a4 Set) (a5 Set) (a6 Set)))
(push)
(assert (not (= (sub (sub (sub a1 a2) a3) a4)
                (sub a1 (cup (cup a2 a3) a4)))))
(check-sat)
; unsat
(get-info statistics)
; ..
; num. qa. inst:      4
; ..
(pop)
```

Five nested occurrences of `sub` require 12 quantifier instantiations (if you run this in extension of the previous check Z3 indicates 16 instantiations, but this figure is the cumulative number of instances).

```
(push)
(assert (not (= (sub (sub (sub (sub a1 a2) a3) a4) a5)
                (sub a1 (cup (cup (cup a2 a3) a4) a5)))))
(check-sat)
; unsat
(get-info statistics)
; num. qa. inst:      12
(pop)
```

Six, seven and eight nestings result in 38, 98 and 344 quantifier instantiations, respectively. The number of quantifier instantiations grows exponentially with the number of nested `sub`s. One can avoid the exponential number of instantiations by using the following trick, which uses a special version of `sub` to control quantifier instantiation.

```
(declare-fun subM (Set Set) Set)

(assert (forall (A Set) (B Set)
              (= (sub A B) (subM A B))
          :pat { (sub A B) }))

(assert (forall (A Set) (B Set) (C Set)
              (= (sub (subM A B) C) (subM A (cup B C)))
          :pat { (sub (subM A B) C) }
        ))
```

It carefully shifts a marker such that only a quadratic number of instantiations is required. The number of instantiations grow from 7, 11, 16, 22 to 29 for the same set of nestings.

13.3 The Axiom Profiler

As should be evident, programming with triggers is both an art and a craft. The *Z3 Axiom Profiler* is a tool that can be used to profile quantifier instantiations, and especially track how quantifier instantiations trigger other instantiations. The Z3 Axiom Profiler is available from

`http://vcc.codeplex.com/Wiki/View.aspx?title=Z3%20Axiom%20Profiler`

13.4 Saturation

While pattern-based quantifier instantiation is quite effective and has advantages with respect to how quantifier instantiation can be controlled, it also comes with some basic limitations. One limitation is that patterns require variables to be in the scope of a function symbol. Equality is excluded. There are no ways to annotate the quantifier with patterns for Z3:

```
(declare-sorts (Person))
(declare-funs ((Adam Person) (Eve Person) (p Person)))
(assert (forall (x Person) (or (= x Adam) (= x Eve))))
(assert (not (or (= p Adam) (= p Eve))))
(check-sat)
; unknown
; (error
;  "WARNING: failed to find a pattern for quantifier (quantifier id: k!2)")
```

Saturation-based theorem proving, using superposition, overcomes several of the limitations of pattern-based quantifier instantiation. On the other hand, the overhead of search can be more unpredictable and is not controllable in the same way. You can enable saturation by setting the configuration option `SATURATE` to `true`. With this option, Z3 produces the expected result `unsat`. The background of the saturation algorithms used in Z3 are elaborated on in cite [13].

13.5 Model-based Quantifier Instantiation

Z3 can use some information from the current partial model for the ground goal to instantiate quantifiers. You can enable model-based quantifier instantiation in Z3 using the configuration option `QI_MODEL_CHECKER=1` (which instantiates quantifiers by values from the current model if these values contradict the quantifiers), or `QI_MODEL_CHECKER=2`, which instantiates quantifiers by instances that either evaluate to `false` or are not specified in the current model.

13.6 The Array Property Fragment

The option `ARRAY_PROPERTY=true` enables the *array property* fragment. This works in the context of quantified formulas that use the array functions `store` and `select`. The array property fragment is described in [5].

14 Simplification

The main use of Z3 is to check whether formulas are satisfiable and optionally obtain a model, or unsatisfiable and optionally obtain a proof. The core of satisfiability checking uses and benefits from pre-processing simplification, but does not require it, other than for eliminating functions that can be replaced by others. For example, `x - y` is replaced by `x + -1*y`, such that Z3's core does not need to handle subtraction as a special case. Thus, simplification is used as a convenience, but not as a necessity. This contrasts BDD (binary decision diagram) packages, that besides check propositional formulas for satisfiability also normalize propositional formulas into a unique normal form. Two propositional formulas are equivalent if and only if their binary decision diagrams are equal.

Since several applications may benefit from simplification, Z3 exposes its expression simplifier. Using the simplifier in context of Z3 can have an advantage if the same formula is used in several different scopes. Then it is an advantage to operate with the pre-simplified formula in contrast to potentially re-simplify the same formula multiple times.

14.1 Invoking the Simplifier

Simplification can be invoked from the `smt2` command-line, or it can be invoked over the binary APIs. The C function for calling the simplifier is called `Z3_simplify` and the .NET method is called `Simplify`. They both return a simplified expression.

The expression is simplified in the context of the formulas that may have been asserted to the current context.

14.2 Configuring Simplification

You can control the strength of the simplifier using a few configuration options.

14.2.1 ELIM_QUANTIFIERS

Z3's simplifier can also be used to exercise the quantifier elimination routines. This was illustrated in Section 9.5.

14.2.2 CONTEXT_SIMPLIFIER

This setting can be used to simplify sub-formulas to *true* or *false*. For example it can be used to simplify $p \wedge (p \vee q)$ to p by realizing that the second nested occurrence of p is subsumed by the top-level p. It uses syntactic matching to simplify sub-formulas, it does not invoke any decision procedures.

14.2.3 STRONG_CONTEXT_SIMPLIFIER

This setting can also be used to simplify sub-formulas to *true* or *false*. It uses decision procedures to check subsumption.

```
(declare-funs ((x Int) (y Int)))
(simplify (or (and (< -1 (+ y x)) (< x y)) (and (< -1 (+ x y)) (>= x y))))
; (or (not (or (<= (+ y x) (- 0 1)) (<= (+ y (* (- 0 1) x)) 0)))
; (not (or (<= (+ y x) (- 0 1)) (not (<= (+ y (* (- 0 1) x)) 0)))))

(set-option set-param "STRONG_CONTEXT_SIMPLIFIER" "true")
(simplify (or (and (< -1 (+ y x)) (< x y)) (and (< -1 (+ x y)) (>= x y))))
```

```
; (not (<= (+ y x) (- 0 1)))
```

15 Implied Equalities

The formula

$$\varphi :\ x \leq y+1 \wedge y \leq z-1 \wedge z \leq x$$

is satisfiable, but it constrains the interpretations for x, y, and z such that $x = y+1 = z$. Some applications would like to use such information. Z3 exposes APIs for learning implied equalities. From smt2, you can learn the implied equalities using a query of the form:

```
  (declare-funs ((x Int) (y Int) (z Int)))
  (assert (and (<= x (+ y 1)) (<= y (- z 1)) (<= z x)))
  (get-implied-equalities x z (+ y 1) y (- z 1))
; (18 18 18 19 19)
```

The `get-implied-equalities` function takes a list of terms as arguments. It produces a list of integers. Each integer identifies a partition, so that two terms in the same equivalence class receive the same partition identifier. In the example the terms `x`, `z` and `(+ y 1)` are equal, so are `y` and `(- z 1)`.

From the C-API you can query implied equalities using

```
Z3_context context;
unsigned num_terms;           // number of terms
Z3_ast terms[num_terms];      // array of terms
usigned class_ids[num_terms]; // output array of partition identifiers

Z3_bool is_sat;
is_sat  = Z3_get_implied_equalities(context, num_terms, terms, class_ids);
```

16 Unsatisfiable Cores

Z3 exposes a way to extract an unsatisfiable set of assertions, commonly called an *unsatisfiable core*. The easiest way to extract an unsatisfiable core is by using the smt2 interface. Figure 6 illustrates how to invoke Z3 to extract an unsatisfiable core. Extracting unsatisfiable cores imposes extra run-time overhead, so Z3 requires that you instruct it to enable core extraction using the command (set-option enable-cores).

The C API function Z3_check_assumptions takes as input a set of literals. The literals are assumed. It returns a proof object and a sub-list of the input literals that were used in the unsatisfiable core (if the assertions and assumptions together are unsatisfiable).

```
(set-option enable-cores)
(declare-funs ((f U U) (a U) (b U)))
(declare-funs ((c U) (d U) (e U)))
(declare-funs ((x U) (y U)))
(declare-preds ((p U U) (p1) (p2)))
(declare-preds ((p3) (p4) (p5)))
(assert (or p1 (= a b)))
(assert (or (not p1) (= a b)))
(assert (= x b))
(assert (= y a))
(assert (or false (= c d)))
(assert (and (= b c) (= d e)))
(assert (or p1 p2))
(assert (or p2 p3))
(assert (not (= (f a) (f e))))
(get-unsat-core)
```

Figure 6: Unsatisfiable core with smt2

17 Parallel Z3

The Z3 distribution comes with two specially designated binary directories `bin_mt` and `x64_mt`, for the 32-bit and a 64-bit version a multi-threaded (and parallel) Z3. By default, all versions of parallel Z3 behave like the sequential Z3.

To run a parallel Z3, run `z3.exe` in the directory `bin_mt` (or `x64_mt` for 64 bit machines) and supply the number of cores that you want to run, e.g.,

```
z3 <file.smt> PAR_NUM_THREADS=4
```

The different cores communicate with each-other by exchanging learned lemmas. Lemma sharing may be configured via two options:

- `CC_SHARING`: Sets the sharing mode (0=off, 1=dynamic, 2=static)
- `CC_SHARING_LIMIT_NEAR`: Sets the maximum lemma size for inter-core sharing.

E.g., to run four cores with static sharing of lemmas up to size 8, use

```
z3 <file.smt> PAR_NUM_THREADS=4 PAR_SHARING=2 PAR_SHARING_LIMIT_NEAR=8
```

17.1 Portfolio Setup

By default, Z3 will start up to eight different SAT-strategies. Users can configure their own portfolio on the command line; automatic configuration is turned off if this feature is used. For example, to run on three cores with identical configuration, but different initial restart intervals, use

```
z3 <file.smt> PAR_NUM_THREADS=3 RESTART_INITIAL=100{200,300}
```

where 100 is the default option and 200,300 configures the first two cores. I.e., using the above configuration, the first two cores on the machine will use 200 and 300 as their initial restart intervals, while the third core uses 100.

18 Proofs

Z3 can generate proof objects. Fine-grained proofs (enabled by setting PROOF_MODE=2) include detailed rewrite steps from the pre-processor, coarse-grained proofs (enabled by setting PROOF_MODE=1) summarize several rewriting steps into one.

Proofs objects are represented as terms and proof rules are pre-defined functions. The proof leaves are either asserted formulas, recognized by the function asserted applied to a Boolean formula, or hypotheses. The main proof rules are modus ponens mp, which applies to implications and equivalences, and unit resolution unit_resolution, which resolves literals away from a clause.

The following is a simple proof object generated by Z3.

```
  (declare-preds ((p) (q)))
  (assert (implies p q))
  (assert p)
  (assert (not q))
  (check-sat)
; unsat
  (get-proof "stdout")
; (let (?x27 (asserted p))
; (let ($x28 (not q))
; (let (?x30 (asserted $x28))
; (let ($x23 (not p))
; (let ($x24 (or $x23 q))
; (let (?x29 (mp (asserted (implies p q))
;                (rewrite (iff (implies p q) $x24)) $x24))
; (unit_resolution ?x29 ?x30 ?x27 false)))))))
```

A high-level overview of the proof objects in Z3 is presented in [12]. The Isabelle[2] and HOL4[3] theorem provers integrate Z3's proof objects to create tactics, such that Z3 can be run as an untrusted oracle.

[2] http://isabelle.in.tum.de/

[3] http://hol.svn.sourceforge.net/viewvc/hol/HOL/src/HolSmt/

19 External Theory Solvers

In Section 13.1, we described an example showing how quantifiers can be used to encode a theory (subtyping) not supported by Z3. The main disadvantage of this approach is that, in general, Z3 is not a decision procedure for formulas containing quantifiers. So, Z3 may not terminate or return *unknown* for satisfiable formulas. In this Section, we describe how to implement an external theory solver and connect it to Z3.

We describe the API using a simple example: a theory T that contains a sort S, a constant u, a binary function f, and a binary predicate p. The theory axioms are:

$$\begin{array}{l} \forall x\colon S.\ f(x,u) = x \\ \forall x\colon S.\ f(u,x) = x \\ \forall x\colon S.\ p(x,x) \end{array}$$

The constant u is the *unit* for f, and p is reflexive.

The first steps for creating a new theory are:

1. Define a new structure for storing theory specific data.
2. Register the new theory in the logical context, and bind it to the theory specific data-structure.
3. Define the theory sorts, constants, functions and predicates.

The implement the steps above, we use the following API functions:

```
Z3_theory Z3_mk_theory(
    Z3_context c,
    Z3_string th_name,
    Z3_theory_data data
    );

Z3_sort Z3_theory_mk_sort(
    Z3_theory t,
    Z3_symbol s
    );

Z3_ast Z3_theory_mk_constant(
    Z3_theory t,
    Z3_symbol n,
    Z3_sort s
    );

Z3_func_decl Z3_theory_mk_func_decl(
    Z3_theory t,
    Z3_symbol n,
    unsigned domain_size,
    Z3_sort const domain[],
    Z3_sort range
    );
```

```
typedef void Z3_theory_callback_fptr(Z3_theory t);

void Z3_set_delete_callback(
    Z3_theory t,
    Z3_theory_callback_fptr f
    );
```

The type Z3_theory_data is just an alias for void *. The function Z3_mk_theory creates a new theory. Internally, the theory is identified by the given name (th_name), and it contains a reference to the theory specific data-structure data. The functions Z3_theory_mk_sort, Z3_theory_mk_constant, Z3_theory_mk_func_decl are used to register theory sorts, constants, functions and predicates. Z3 does not distinguish between functions and predicates. A predicate is just a function that returns a Boolean. The function Z3_set_delete_callback(t, f) registers a callback f that is invoked before theory t is deleted by Z3 when the logical context containing t is deleted. You can use this to release memory and other resources that are allocated with the theory.

To implement our simple theory, we first define the following C structure:

```
typedef struct _SimpleTheoryData {
    Z3_sort      S;
    Z3_func_decl f;
    Z3_func_decl p;
    Z3_ast       u;
} SimpleTheoryData;
```

The following C function registers the simple theory in the given logical context ctx.

```
Z3_theory mk_simple_theory(Z3_context ctx) {
    Z3_sort f_domain[2];
    Z3_symbol s_name      = Z3_mk_string_symbol(ctx, "S");
    Z3_symbol f_name      = Z3_mk_string_symbol(ctx, "f");
    Z3_symbol p_name      = Z3_mk_string_symbol(ctx, "p");
    Z3_symbol u_name      = Z3_mk_string_symbol(ctx, "u");
    Z3_sort B             = Z3_mk_bool_sort(ctx);
    SimpleTheoryData * td = (SimpleTheoryData*)malloc(sizeof(SimpleTheoryData));
    Z3_theory Th          = Z3_mk_theory(ctx, "simple_th", td);
    td->S                 = Z3_theory_mk_sort(Th, s_name);
    f_domain[0] = td->S; f_domain[1] = td->S;
    td->f                 = Z3_theory_mk_func_decl(Th, f_name, 2, f_domain, td->S);
    td->p                 = Z3_theory_mk_func_decl(Th, p_name, 1, &td->S, B);
    td->u                 = Z3_theory_mk_constant(Th, u_name, td->S);

    //
    // At this point, we register the theory callback functions.
    // We describe the first callback in some detail below.
    //
    Z3_set_reduce_app_callback(Th, Th_reduce_app);
    Z3_set_new_app_callback(Th, Th_new_app);
    Z3_set_new_elem_callback(Th, Th_new_elem);
    Z3_set_init_search_callback(Th, Th_init_search);
    Z3_set_push_callback(Th, Th_push);
    Z3_set_pop_callback(Th, Th_pop);
    Z3_set_reset_callback(Th, Th_reset);
    Z3_set_restart_callback(Th, Th_restart);
    Z3_set_new_eq_callback(Th, Th_new_eq);
    Z3_set_new_diseq_callback(Th, Th_new_diseq);
    Z3_set_new_relevant_callback(Th, Th_new_relevant);
    Z3_set_new_assignment_callback(Th, Th_new_assignment);
    Z3_set_final_check_callback(Th, Th_final_check);

    return Th;
}
```

Our function `mk_simple_theory` is not finished yet. We still have to register callbacks for our new theory. Z3 communicates with an external theory implementation using callbacks: C function pointers. There are several callbacks that can be registered for each given external theory. When a theory is deleted by Z3, we should free the theory specific data-structure. We can accomplish that by defining the C function, and registering it as a *delete callback*.

```
void Th_delete(Z3_theory t) {
    SimpleTheoryData * td = (SimpleTheoryData *)Z3_theory_get_ext_data(t);
    printf("Delete\n");
    free(td);
}
```

In the callback above, the function `Z3_theory_get_ext_data` is used to retrieve the theory specific data-structure. We also add the following statement in the function `mk_simple_theory`.

```
    Z3_set_delete_callback(Th, Th_delete);
```

The next set of callbacks is used to implement theory specific simplifications in the Z3 simplifier. The Z3 simplifier is applied to any formula asserted into the logical context, any instance of a universal quantified formula, and axioms asserted by external theories. It can also be directly invoked using the function `Z3_simplify`.

```
typedef Z3_bool Z3_reduce_app_callback_fptr(
    Z3_theory, Z3_func_decl, unsigned, Z3_ast const [], Z3_ast *);

typedef Z3_bool Z3_reduce_eq_callback_fptr(
    Z3_theory t, Z3_ast a, Z3_ast b, Z3_ast * r);

typedef Z3_bool Z3_reduce_distinct_callback_fptr(
    Z3_theory, unsigned, Z3_ast const [], Z3_ast *);

void Z3_set_reduce_app_callback(
    Z3_theory t, Z3_reduce_app_callback_fptr f);

void Z3_set_reduce_eq_callback(
    Z3_theory t, Z3_reduce_eq_callback_fptr f);
```

```
void Z3_set_reduce_distinct_callback(
    Z3_theory t, Z3_reduce_distinct_callback_fptr f);
```

All simplification (reduction) callbacks return a Boolean value, it is `Z3_TRUE` if callback managed to simplify the input, and the result is stored in the last argument. If the callback returns `Z3_FALSE`, then the result is ignored, and Z3 uses its default procedure for handling the application. A *reduce application* callback is set using the function `Z3_set_reduce_app_callback(t, f)`, the callback `f` is invoked whenever the Z3 simplifier tries to simplify a term of the form $g(t_1,\ldots,t_n)$, where g is a theory function/predicate. The equality and distinct callback are invoked whenever the simplifier finds a term of the form $t_1 = t_2$ or $\mathit{distinct}(t_1,\ldots,t_n)$ respectively, where the sort of t_1 is a theory sort. In our example, we only need to define the *reduce application* callback. We define the following C function:

```
/*
   This callback whenever the Z3 simplifier is trying to create
   an expression d(args[0], ..., args[n-1]), and d is a theory symbol.
*/
Z3_bool Th_reduce_app(Z3_theory t,
                      Z3_func_decl d,
                      unsigned n,
                      Z3_ast const args[],
                      Z3_ast * result) {
    SimpleTheoryData * td = (SimpleTheoryData*)Z3_theory_get_ext_data(t);
    if (d == td->f) {
        if (args[0] == td->u) {
            *result = args[1];
            return Z3_TRUE;
        }
        else if (args[1] == td->u) {
            *result = args[0];
            return Z3_TRUE;
        }
    }
    else if (d == td->p) {
        if (args[0] == args[1]) {
            *result = Z3_mk_true(Z3_theory_get_context(t));
            return Z3_TRUE;
        }
    }
    return Z3_FALSE; // failed to simplify
}
```

In the callback above, the function `Z3_theory_get_context` is used to retrieve the logical context that *owns* the theory `t`. The implementation of simplification/reduction functions is optional. That is, we can implement a complete external theory solver without using these callbacks. Of course, in practice, it is useful to reduce the size of the input formula using as many simple/cheap simplifications as possible.

The full theory implementation contains several other callbacks that are set. The online documentation for Z3 contains the full source code for this example.

20 Some Applications

We have covered a number of ways to directly interact with Z3. The bigger picture, how can Z3 be used, is perhaps not obvious from the detached examples. This section, on the other hand, summarizes a set of applications of Z3 that might help inspire additional useful applications.

20.1 Dynamic Symbolic Execution

SMT solvers play a central role in *dynamic* symbolic execution, also called *smart white-box fuzzing*. There are a number of tools used in industry that are based on dynamic symbolic execution, including CUTE, Exe, DART, SAGE, Pex, and Yogi [20]. These tools collect explored program paths as formulas and use solvers to identify new test inputs that can steer execution into new branches. SMT solvers are a good fit for symbolic execution because the semantics of most program statements can be easily modeled using theories supported by the solvers. We will later introduce the various theories that are used, but here let us first focus on connecting feasibility constraints with a solver. To illustrate the basic idea of dynamic symbolic execution, consider the greatest common divisor program 20.1. It takes the inputs x and y and produces the greatest common divisor of x and y.

```
int GCD(int x, int y) {
  while (true) {
    int m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

Program 20.1: GCD Program

Program 20.2 represents the static single assignment unfolding corresponding to the case where the loop is exited in the second iteration. Assertions are used to enforce that the condition of the if-statement is not satisfied in the first iteration, and it is in the second. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations.

```
int GCD(int x_0, int y_0) {
  int m_0 = x_0 % y_0;
  assert (m_0 != 0);
  int x_1 = y_0;
  int y_1 = m_0;
  int m_1 = x_1 % y_1;
  assert (m_1 == 0);
}
```

$$\begin{array}{ll}
(m_0 = x_0 \;\%\; y_0) & \wedge \\
\neg(m_0 = 0) & \wedge \\
(x_1 = y_0) & \wedge \\
(y_1 = m_0) & \wedge \\
(m_1 = x_1 \;\%\; y_1) & \wedge \\
(m_1 = 0) &
\end{array}$$

Program 20.2: GCD Path Formula

The resulting path formula is satisfiable. One satisfying assignment that can be found using an SMT solver is of the form:

$$x_0 = 2,\ y_0 = 4,\ m_0 = 2,\ x_1 = 4,\ y_1 = 2,\ m_1 = 0\ .$$

It can be used as input to the original program. In the case of this example, the call `GCD(2,4)` causes the loop to be entered twice, as expected. Smart white-box fuzzing is actively used at Microsoft. It complements traditional black-box fuzzing, where the program being fuzzed is opaque, and fuzzing is performed by pertubing input vectors using random walks. It has been instrumental in uncovering several subtle security critical bugs that black-box methods have been unable to find.

20.2 Program Model Checking

Dynamic symbolic execution finds some input that can guide execution into bugs. This method alone does not produce any guarantee that programs are free of all of the errors being checked for. The goal of *program model checking* tools is to automatically check for freedom from selected categories of errors. The basic idea of program model-checking is to explore all possible executions using a finite and sufficiently small abstraction of the program state space. The tools BLAST [22], SDV [1] and SMV from Cadence [4], perform such program model checking. Both SDV and SMV are used as part of commercial tool offerings. We will use the program fragment in Program 20.3 as an example of finite state abstraction. It accesses requests using `GetNextRequest`. The call is protected by a lock. A question is whether it is possible to exit the loop without having a lock. The program has an infinite (or very large) number of states, since the value of `count` can grow arbitrarily.

```
do {
      lock();
      old_count = count;
      request = GetNextRequest();
      if (request != NULL) {
         unlock();
         ProcessRequest(request);
         count = count + 1;
      }
}
while (old_count != count);
unlock();
```

Program 20.3: Processing requests using locks

Yet, from the point of view of locking, the actual values of `count` and `old_count` are not really interesting. On the other hand, the *relationship* between these contains useful information. Program 20.4 shows a finite state abstraction of the same locking program. The Boolean variable `b` encodes the relation `count == old_count`. We use the symbol $*$ to represent a Boolean expression that can non-deterministically evaluate to *true* or *false*. We can now explore the finite number of branches of the abstract program to verify that the lock is always held when exiting the loop.

[4] `http://www.kenmcmil.com/`

```
do {
      lock();
      b = true;
      request = GetNextRequest();
      if (request != NULL) {
         unlock();
         ProcessRequest(request);
         if (b) b = false; else b = *;
      }
}
while (!b);
unlock();
```

Program 20.4: Processing requests using locks, abstracted

SMT solvers are used for constructing finite state abstractions like the one provided. There are to date several approaches for creating these abstractions. In one of these approaches each statement in the program is individually abstracted. For example, let us consider the statement `count = count + 1`. The abstraction of this statement is essentially a relation between the current and new values of the Boolean variable `b`. SMT solvers are used to compute this relation by proving theorems such as

```
count == old_count → count+1 != old_count
```

which is dual to checking unsatisfiability of the negation:

```
count == old_count ∧ count+1 == old_count
```

The theorem says that if the current value of `b` is *true*, then after executing the statement `count = count + 1` it will be *false*. Note that if `b` is *false*, then neither of the following conjectures is valid.

```
count != old_count → count+1 == old_count

count != old_count → count+1 != old_count
```

In both cases, an SMT solver will produce a *counter-example*, that is, a model for the negation of the conjecture. Therefore, when the current value of `b` is *false*, nothing can be said about its value after the execution of the statement. The result of these three proof attempts is then used to replace the statement `count = count + 1;` by `if (b) b = false; else b = *;`. A finite state model checker can now be used on the Boolean program. It will establish that `b` is always *true* when control reaches this statement, verifying that calls to `lock()` are balanced with calls to `unlock()` in the original program.

20.3 Static Program Analysis

Static program analysis tools work in a similar way as dynamic symbolic execution tools. They also check feasibility of program paths. On the other hand they never require executing programs and they can analyze software libraries and utilities independently of how they are used. One advantage of using modern SMT solvers in static program analysis is that SMT solvers nowadays accurately capture the semantics of most basic operations used by commonly used programming languages. We use the program in Figure 20.5 to illustrate the need for static program analysis to use bit-precise reasoning. The program searches for an index in a sorted array `arr` that contains a key.

```
int binary_search(
  int[] arr, int low, int high, int key) {
    assert (low > high || 0 <= low < high);
    while (low <= high) {
        // Find middle value
        int mid = (low + high) / 2;
        assert (0 <= mid < high);
        int val = arr[mid];
        // Refine range
        if (key == val) return mid;
        if (val > key) low = mid+1;
        else high = mid-1;
    }
    return -1;
}
```

Program 20.5: Binary search

The `assert` statement is a *pre-condition*, for the procedure. It restricts the input to fall within the bounds of the array `arr`. The program performs several operations involving arithmetic, so a theory and corresponding solver that understands arithmetic appears to be a good match. It is important, however, to take into account that languages, such as Java, C# and C/C++ all use fixed-width bit-vectors as the representation for values of type `int`. This means that the accurate theory for `int` is two-complements modular arithmetic. Assuming a bit-width of 32 bits, the maximal positive 32-bit integer is $2^{31}-1$ and the smallest negative 32-bit integer is -2^{31}. If both `low` and `high` are 2^{30}, `low + high` evaluates to 2^{31}, which is treated as the negative number -2^{31}. The presumed assertion $0 \leq \mathit{mid} < \mathit{high}$ therefore does not hold. Fortunately, several modern SMT solvers support the theory of *bit-vectors*, which accurately captures the semantics of modular arithmetic. The bug does not escape an analysis based on the theory of bit-vectors. Such an analysis would check that the array read `arr[mid]` is within bounds during the first iteration by checking the formula:

$$\begin{array}{ll} & \mathit{low} > \mathit{high} \vee 0 \leq \mathit{low} < \mathit{high} < \mathit{arr.length} \\ \wedge & \mathit{low} \leq \mathit{high} \\ \rightarrow & 0 \leq (\mathit{low} + \mathit{high})/2 < \mathit{arr.length} \end{array}$$

As we saw, the formula is not valid. The values $\texttt{low} = \texttt{high} = 2^{30}, \texttt{arr.length} = 2^{30} + 1$ provide a counter-example. The use of SMT solvers for bit-precise static analysis tools is an active area of current development and research. An integration with the solver Z3 [9] and the static analysis tool PREfix led to the automatic discovery of several overflow-related bugs in Microsoft's rather large code-base.

20.4 Program Verification

The ideal of verified software has been a long-running quest since Floyd and Hoare introduced program verification by assigning logical assertions to programs. *Extended static checking* uses the methods developed for program verification, but in the more limited context of checking absence of run-time errors. The SMT solver Simplify [14] was developed in the context of the extended static checking systems ESC/Modula 3 and ESC/Java [19]. This work has been the inspiration for several subsequent extended static program checkers, including Why [17] and Boogie [2]. These systems are actively used as bridges from several different front-ends to SMT solver backends. Boogie, for instance, is used as a backend for systems that verify code from languages, such as an extended version of C# (called Spec#), as well as low level systems code written in C. Current practice indicates that one person can drive these tools to verify selected extended static properties of large code bases with several hundreds of thousands of lines. A more ambitious project is the Verifying C-Compiler system [16], which targets functional correctness properties of Microsoft's Viridian Hyper-Visor. The Hyper-Visor is a relatively small (100K lines) operating system layer, yet correctness properties are challenging to formulate and establish. The entire verification effort is estimated to be around 60 man-years.

20.5 Modeling

SMT solvers present an interesting opportunity for high-level software modeling tools. In some contexts these tools use domains inspired from mathematics, such as algebraic data-types, arrays, sets and maps. These domains have also been the subject of long-running research in the context of SMT solvers. Let us first introduce the array domain. Software modeling areas include:

Model-based testing uses high-level models of software systems, including network protocols, to derive test oracles. SMT solvers have been used in this context for exploring the models using a symbolic execution and search. Model-based testing is used on a large scale at Microsoft in the context of the disclosure and documentation of Microsoft network protocols [21]. The model-based tools use SMT solvers for generating combinations of test inputs as well as symbolic exploration of models.

Model programs are behavioral specifications that can be described succinctly and at a high-level of abstraction. These descriptions are state machines that use abstract domains. SMT solvers are used to perform *bounded model-checking* of such descriptions. The main idea of bounded model-checking is to explore a bounded symbolic execution of a program or model. Thus, given a bound, such as 17, the transitions of the state machines are unrolled into a logical formula that describes all possible executions using 17 steps.

Model-based designs use high-level languages for describing software systems. Implementations are derived by refinements. Modeling languages present an advantage as they allow exploring a design space without committing all design decisions up front. SMT solvers play various roles in model-based designs. They are used for type-checking designs and they are useful in the search for different consistent choices in a design space.

20.6 Qex

Qex[5] implements automatic data generation methods for parametrized SQL queries. Data generation involves both parameter data, as well as, concrete table data generation. The data generation is driven by test conditions that represent various coverage criteria. Qex works directly on top of SQL queries and retains the semantics at the level of SQL. An advantage of working at the level of SQL, in contrast to the main workings of Pex and SAGE that rely on instructions resulting from a compiler, is that several high-level SQL constructs can be mapped almost directly into Z3.

[5] `http://research.microsoft.com/projects/qex`

20.7 VS3

Sumit Gulwani uses Z3 in several research projects related to program analysis and program synthesis[6]. In particular, the VS3 project uses Z3 to automatically discover inductive invariants for proving given safety properties of systems. The project also explores techniques for using SMT solvers to synthesize systems in the first place given enough specifications. Other projects involving Z3 aim to determine the precise asymptotic run-time bounds of programs. For example, he has analyzed the .NET base class library routines and extracted asymptotic bounds (of the form $O(n)$, $O(n\log(n))$ etc.), for the vast majority of routines.

20.8 Program Termination

A nice blog on of one of the core algorithms used for finding ranking functions in programs uses Z3 and F# is accessible from`http://www.foment.net/byron/fsharp.shtml`.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.

[3] Nikolaj Bjørner. Linear Quantifier-Elimination as an Abstract Decision Procedure. In *IJCAR*, 2010.

[4] Aaron R. Bradley and Zoha Manna. *The calculus of computation*. Springer, 2007. BRA a7 2007:1 1.Ex.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[6] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 302–314. ACM, 2009.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

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

[9] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 08*, volume 4963 of *LNCS*. Springer, 2008.

[10] L. de Moura and N. Bjørner. Efficient, Generalized Array Decision Procedures. In *FMCAD*. IEEE, 2009.

[11] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT Solvers. In *CADE'07*. Springer-Verlag, 2007.

[12] Leonardo de Moura and Nikolaj Bjørner. Proofs and Refutations, and Z3 . In *IWIL*, 2008.

[13] Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. 4th IJCAR*, volume 5195, pages 475–490, 2008.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

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

[16] E. Cohen and M. Dahlweid and M. Hillebrand and D. Leinenbach and M. Moskal and T. Santen and W. Schulte and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOL*, 2009.

[17] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.

[18] Melvin Fitting. *First-order logic and automated theorem proving*. Springer, 1996. FIT m 1996:1 1.Ex.

[6] `http://research.microsoft.com/~{}sumitg/pubs/vs3.html`

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.

[20] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.

[21] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. L. Wurden. Model-Based Quality Assurance of Windows Protocol Documentation. In *ICST*, pages 502–506. IEEE Computer Society, 2008.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[23] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008. KRO d2 2008:1 1.Ex.

[24] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[25] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[26] Michał Moskal. Programming with Triggers. In *SMT'09*, 2009.