Functional SMT solving: A new interface for programmers

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Siddharth Agarwal

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CERTIFICATE

It is certified that the work contained in the thesis titled **Functional SMT solving: A new interface for programmers**, by **Siddharth Agarwal**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare

Prof Amey Karkare

Department of Computer Science & Engineering

IIT Kanpur

June, 2012

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

ABSTRACT

Name of student: **Siddharth Agarwal** Roll no: **Y7027429**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Functional SMT solving: A new interface for programmers**

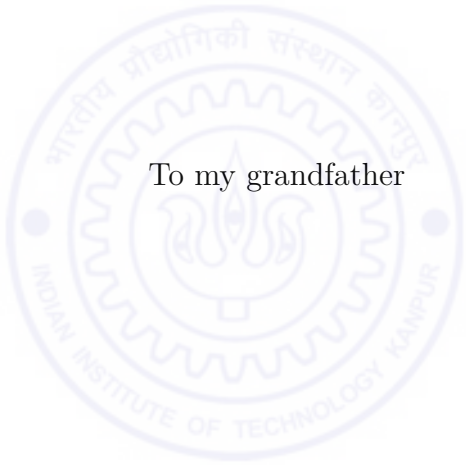
Name of Thesis Supervisor: **Prof Amey Karkare**

Month and year of thesis submission: **June, 2012**

Satisfiability Modulo Theories (SMT) solvers are powerful tools that can quickly solve complex constraints involving booleans, integers, first-order logic predicates, lists, and other data types. They have a vast number of potential applications, from constraint solving to program analysis and verification. However, they are so complex to use that their power is inaccessible to all but experts in the field.

We present an attempt to make using SMT solvers simpler by integrating the Z3 solver into a host language, Racket. Our system defines a programmer's interface in Racket that makes it easy to harness the power of Z3 to discover solutions to logical constraints. The interface, although in Racket, retains the structure and brevity of the SMT-LIB format. We demonstrate this using a range of examples, from simple constraint solving to verifying recursive functions, all in a few lines of code.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR



To my grandfather

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Acknowledgements

This project would have never have even been started had it not been for my thesis advisor Dr Amey Karkare's help and guidance. Right from the time we were looking for ideas to when we finally had a concrete proposal in hand, his support has been invaluable. He entertained all my crazy ideas, found which ones had merit, and helped me build on them. I wouldn't have been able to do this without him.

I'd like to thank my parents and the others in my life for being patient with me, especially towards the final stretch of thesis completion when I wasn't at my best socially.

The Racket community, at `#racket` on Freenode IRC, helped me understand some of the Racket syntax model's intricacies. That helped me avoid dead ends both early on and towards the end.

Special thanks to Leonardo de Moura at Microsoft Research for taking the time out to answer a few questions about Z3.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Contents

List of Tables

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

List of Figures

Chapter 1

Introduction: Satisfiability solvers and interfaces

The Boolean satisfiability or SAT problem asks: *Given a boolean formula with a set of variables in it, is there a way to assign each variable a value such that the formula becomes true?* The SAT problem is one of the cornerstones of computer science, with enormous theoretical and practical implications. Indeed, it was the very first problem to be proved *NP-complete* [1].

Yet, interest in efficiently solving so-called "natural" or "real-world" instances of SAT has remained. This is at least partly because a large number of practical problems are also NP-complete and can be *reduced* to SAT. Here *reduced* is as it is defined in [2]: roughly, can an instance of the problem be turned into a boolean formula decided by SAT that's at most polynomially larger?

1.1 Solving SAT: DPLL

Solving SAT is generally restricted to boolean formulas in conjunctive normal form (CNF), which consists of *clauses* of boolean *literals*[1] joined together with the OR operation ($\vee$), and these clauses joined together with the AND operation ($\wedge$). It is

[1]The word *literal* has two different technical meanings: in computer science, it represents a fixed value such as the number `1` or the string `"Hello"`. In mathematical logic, it means either an atom (variable: p) or its negation ($\neg p$). We use it here in this latter sense.

proved that the SAT problem for any boolean formula can be reduced to the SAT problem for formulas in CNF [2].

CNF formulas have useful properties, such as that they can be represented as a set of clauses, that at any point a clause once satisfied (shown to be true) can be dropped from the set, and that showing even one of the clauses to be false is enough to show that the formula cannot be satisfied.

The simplest way to solve SAT is with a basic backtracking algorithm (Figure 1.1). The algorithm's recursive nature means it takes exponential time in the worst case.

Figure 1.1 A naïve backtracking algorithm for SAT in Racket. `substitute` substitutes the given value for the atom in the boolean formula, and returns an updated formula or `#f`

```
; atoms is a list of variables to satisfy
(define (solve-sat atoms formula)
  (if (empty? atoms)
      #t   ; no more atoms to substitute means SAT
      (let*
       ([atom (car atoms)]    ; pick the first atom
        [rest-atoms (cdr atoms)]
        [subst-true (substitute formula atom #t)]
        [subst-false (substitute formula atom #f)])

        (or (and subst-true (solve-sat rest-atoms subst-true))
            (and subst-false (solve-sat rest-atoms subst-false))))))
```

There are two key insights that can be made here:

1. **Unit propagation, or the one-literal clause rule.** *If a clause contains just one literal, that literal has to be true.* For example, consider the set of clauses $\{a, a \vee b, \neg a \vee \neg c, b \vee c \vee d\}$. Since all the clauses need to be true, a must be true. Letting this happen, we see that $a \vee b$ is true so it is dropped, and $\neg a \vee \neg c$ simply becomes $\neg c$, which means we are left with the clauses $\{\neg c, b \vee c \vee d\}$.

   At this point unit propagation can be applied once again on c, setting it to false. We are finally left with just the one clause $\{b \vee d\}$.

2. **Pure literal elimination, or the affirmative-negative rule.** *If the occurrences of a given variable p are either all p (or all $\neg p$), then p can be assigned true (or false).* Consider the following set of clauses: $\{a \vee b, a \vee c, b \vee c \vee \neg d, \neg b \vee \neg c \vee d\}$. Since a always appears in its positive form, we can assign a true without needing to consider the case where a is false. Thus any clause that contains a is satisfied and can be dropped. We are left with $\{b \vee c \vee \neg d, \neg b \vee \neg c \vee d\}$.

The backtracking algorithm, plus these two insights applied at each step, form the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3, 4]. The DPLL algorithm, although over fifty years old, is tremendously successful and forms the core of many modern complete SAT solvers[2], including Chaff [6], GRASP [7] and MiniSat [8].

Of course, modern SAT solvers employ a large number of heuristic and optimisation tricks to speed things up. They are highly efficient for "real-world" problems and are used in a wide variety of computer science fields, from automated planning for robots or unmanned vehicles (via the *satplan* method [9]), to dependency resolution in Linux package managers such as Zypper [10]. They have also been heavily used in program analysis and verification, which is what brings us to the next section.

1.2 SMT and DPLL(T): SAT with a twist

Typically, program analysis tools that used SAT solvers would have to find a way to translate variables found in programs to boolean ones. For example, a 32-bit integer could be encoded as a set of 32 boolean variables[3].

It was soon realised that pushing this step into the SAT solver would help. Since

[2] A *complete* SAT solver is one where a definitive answer is returned, whether a formula is satisfiable or not. There are also *incomplete* SAT solvers which return an answer if the formula is satisfiable but do not return one if it is not. Examples include GSAT and WalkSat [5].

[3] It is also possible to represent integers as boolean variables via predicate abstraction [11], which is more efficient but potentially loses information.

users would still be asking whether formulas were satisfiable, except with variables from more complex domains or *theories*, this approach was dubbed Satisfiability Modulo Theories (SMT). Early SMT methods (e.g. [12, 13]) *eagerly* simply converted formulas to Boolean variables before handing them over to a SAT solver.

Alternatively, one could substitute placeholder boolean variables for theory variables, supply this *faux* boolean formula to a modified SAT solver, and write a separate program called a *T-solver* that interacts *on demand* with the SAT solver to deal with theory variables. The modified DPLL algorithm is called **DPLL(T)** [14], and this *lazy* approach turns out to be far more efficient than eager methods. (For a more detailed description of eager and lazy SMT, see [15, Section 3.2]).

State-of-the-art SMT solvers like Z3 [16], Yices [17] and CVC3 [18] are based on DPLL(T). They allow users to specify constraints over booleans, integers, real numbers, arrays, lists, trees, first-order predicates and other kinds of variables. They either come up with assignments that satisfy these constraints, or, if possible, a proof that the constraints aren't satisfiable. SMT solvers have been used to solve problems in planning and scheduling, program analysis [19], whitebox fuzz testing [20] and bounded model checking [21].

1.3 Using SMT solvers

After coming to know of the power SMT solvers have, the first question the intrepid programmer would ask is how to use one. Unfortunately, the standard way for programs to interact with SMT solvers is via powerful but relatively arcane C APIs that require the users to know the particular solver's internals. For example, Figure 1.2 on the next page lists a C program that asks Z3 whether the simple proposition $p \wedge \neg p$ is satisfiable.

Simultaneously, most SMT solvers also feature interaction via the standard input language SMT-LIB [22]. SMT-LIB is *significantly* easier to use in isolation. The same program in SMT-LIB would look something like Figure 1.3 on the facing page.

Figure 1.2 A C program to ask Z3 whether $p \wedge \neg p$ is satisfiable

```
Z3_config cfg = Z3_mk_config();
Z3_context ctx = Z3_mk_context(cfg);
Z3_del_config(cfg);
Z3_sort bool_sort = Z3_mk_bool_sort(ctx);

Z3_symbol symbol_p = Z3_mk_int_symbol(ctx, 0);

Z3_ast p = Z3_mk_const(ctx, symbol_p, bool_sort);
Z3_ast not_p = Z3_mk_not(ctx, p);

Z3_ast args[2] = {p, not_p};
Z3_ast conjecture = Z3_mk_and(ctx, 2, args);

Z3_assert_cnstr(ctx, conjecture);

Z3_lbool sat = Z3_check(ctx);

Z3_del_context(ctx);
return sat;
```

Figure 1.3 An SMT-LIB program to check whether $p \wedge \neg p$ is satisfiable

```
; Declare a variable we don't know the value of yet
(declare-fun p () Bool)
; Try to find a value satisfying a contradiction
(assert (and p (not p)))
(check-sat)
; Prints "unsat", meaning "unsatisfiable"
```

The complexity of the C interface keeps going up as we move to less trivial assertions. Figure 1.4 on the next page is a C program that asks Z3 to solve the two simultaneous equations $2x + 3y = 5, 4x + 5y = 7$. (The solution is $x = -2, y = 3$.)

The SMT-LIB interface is still remarkably brief, though (Figure 1.5 on page 8).

However, the SMT-LIB interfaces are generally hard to use directly from C programs and often not as full-featured[4] or extensible. Importantly, it is difficult to write programs that *interact* with the solver in some way, for example by adding assertions based on generated models. This makes it difficult to build new abstractions to enhance functionality.

Summary

In this chapter, we saw what SAT and SMT solvers are, how they work, and the power that they have. We also saw why few programmers use SMT solvers now, instead preferring to hand-code cumbersome search and backtracking algorithms.

[4] Z3, for instance, supports plugging in external theories via the C API, but not via the textual SMT-LIB interface.

Figure 1.4 A C program to ask Z3 to solve two simultaneous linear equations

```
Z3_config cfg = Z3_mk_config();
Z3_set_param_value(cfg, "MODEL", "true");
Z3_context ctx = Z3_mk_context(cfg);
Z3_del_config(cfg);

Z3_sort int_sort = Z3_mk_int_sort(ctx);

Z3_symbol symbol_x = Z3_mk_int_symbol(ctx, 0);
Z3_symbol symbol_y = Z3_mk_int_symbol(ctx, 1);

Z3_ast x = Z3_mk_const(ctx, symbol_x, int_sort);
Z3_ast y = Z3_mk_const(ctx, symbol_x, int_sort);
Z3_ast num2 = Z3_mk_int(ctx, 2, int_sort);
Z3_ast num3 = Z3_mk_int(ctx, 3, int_sort);
Z3_ast num4 = Z3_mk_int(ctx, 4, int_sort);
Z3_ast num5 = Z3_mk_int(ctx, 5, int_sort);
Z3_ast num7 = Z3_mk_int(ctx, 7, int_sort);

Z3_ast eq1 = Z3_mk_eq(ctx, Z3_mk_add(ctx, Z3_mk_mul(ctx, num2, x),
                                          Z3_mk_mul(ctx, num3, y)),
                           num5);
Z3_ast eq2 = Z3_mk_eq(ctx, Z3_mk_add(ctx, Z3_mk_mul(ctx, num4, x),
                                          Z3_mk_mul(ctx, num5, y)),
                           num7);

Z3_assert_cnstr(ctx, eq1);
Z3_assert_cnstr(ctx, eq2);

Z3_model m;
Z3_lbool sat = Z3_check_and_get_model(ctx, &m);

// Z3_L_TRUE means satisfied
if (sat == Z3_L_TRUE) {
  Z3_ast xsolved, ysolved;
  // Omitting some error checking
  Z3_eval(ctx, m, x, &xsolved);
  Z3_eval(ctx, m, x, &ysolved);
  int xval, yval;
  Z3_get_numeral_int(ctx, xsolved, &xval);
  Z3_get_numeral_int(ctx, xsolved, &yval);
  printf("x = %d, y = %d", xval, yval);
}
else {
  printf("No solution to equations");
}
```

Figure 1.5 SMT-LIB code to solve two simultaneous linear equations

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ (* 2 x) (* 3 y)) 5))
(assert (= (+ (* 4 x) (* 5 y)) 7))
(check-sat)
(eval x)
(eval y)
```

Chapter 2

A new interface: `z3.rkt`

In Chapter 1, we demonstrated how cumbersome SMT solvers are to use. Indeed, we faced the same issues while exploring novel methods to verify, debug and test functional programs. It felt like the C interface was hamstringing us, and the SMT-LIB interface was good for basic explorations but not anything more complicated.

We decided to attempt to solve this: our goal was to implement an SMT-LIB-like interface in a way that allowed for the same power as the C interface while appearing naturally integrated into a host language. Since SMT-LIB is *s-expression*-based, for the host language a Lisp dialect was a natural choice. We chose Microsoft Research's Z3 [16] as our SMT solver, and Racket [23], a popular dialect of Scheme [24, 25], for our implementation. We call our implementation **`z3.rkt`**. Racket has extensive facilities for implementing new languages [26], not just for the interface to the solver, but also for the resulting tools that the solver would make possible.

Using this system, the program to check whether a contradiction is satisfiable (Figure 2.1 on the next page) becomes almost as brief as the SMT-LIB version. The program to solve two simultaneous linear equations (Figure 2.2 on the following page) is similarly brief.

Figure 2.1 Using `z3.rkt` to determine whether $p \wedge \neg p$ is satisfiable

```
(smt:with-context
 (smt:new-context)
 (smt:declare-fun p () Bool)
 (smt:assert (and/s p (not/s p)))
 (smt:check-sat))
```

Figure 2.2 Solving simultaneous linear equations with `z3.rkt`

```
(smt:with-context
 (smt:new-context)
 (smt:declare-fun x () Int)
 (smt:declare-fun y () Int)
 (smt:assert (=/s (+/s (*/s 2 x) (*/s 3 y)) 5))
 (smt:assert (=/s (+/s (*/s 4 x) (*/s 5 y)) 7))
 (smt:check-sat)
 (values (smt:eval x) (smt:eval y)))
```

2.1 Interactive SMT solving: two examples

To demonstrate the value in integrating a language with an SMT solver, we turn our attention to a pair of classic logical puzzles.

2.1.1 Sudoku

A Sudoku puzzle asks the player to complete a partially pre-filled 9×9 grid with the numbers 1 through 9 such that no row, column, or 3×3 box has two instances of a number. This is a classic constraint satisfaction problem, and any constraint solver can handle it with ease.

Figure 2.3 on the next page lists a Racket program using `z3.rkt` to solve Sudoku.

Here we omit a couple of function definitions: `add-sudoku-grid-rules` asserts the standard Sudoku grid rules, and `add-grid` reads a partially filled grid in a particular format and creates assertions based on it. We note that the function `(select/s arr x)` retrieves the value at `x` from the array `arr`, and that this can be used to add constraints on the array (for instance, `(smt:assert (=/s (select/s`

Figure 2.3 Racket code using `z3.rkt` to solve Sudoku

```
(define (solve-sudoku grid)
  (smt:with-context
   (smt:new-context)
   ; Declare a scalar datatype (finite domain type) with 9 entries
   (smt:declare-datatypes ()
     ((Sudoku S1 S2 S3 S4 S5 S6 S7 S8 S9)))
   ; Represent the grid as an array from integers to this type
   (smt:declare-fun sudoku-grid () (Array Int Sudoku))
   ; Assert the standard grid rules (row, column, box)
   (add-sudoku-grid-rules)
   ; Add pre-filled entries
   (add-grid grid)
   (define sat (smt:check-sat))
   ; 'sat means we found a solution, 'unsat means we didn't
   (if (eq? sat 'sat)
       ; Retrieve the values from the model
       (for/list ([x (in-range 0 81)])
         (smt:eval (select/s sudoku-grid x)))
       #f)))
```

`arr x) y)))`). We also note that if a set of constraints is satisfiable, Z3 can generate a *model* showing this; values can be extracted out of this model using the `smt:eval` command.

However, simply finding a solution isn't enough for a good Sudoku solver: it must also verify that there aren't any other solutions. The usual way to do that for a constraint solver is by retrieving a generated model, adding assertions such that this model cannot be generated again, and then asking the solver whether the system of assertions is still satisfiable. If it is, a second solution exists and the puzzle is considered invalid.

In such situations, the interactivity offered by `z3.rkt` becomes useful: it lets the programmer add dynamically discovered constraints on the fly. The last part of the solution might then become something like Figure 2.4 on the following page.

This part can even be abstracted out into a function that returns a lazily-generated sequence of satisfying assignments for any given set of constraints.

Figure 2.4 Ensuring that a Sudoku grid has exactly one solution

```
...
(if (eq? sat 'sat)
    ; Make sure no other solution exists
    (let ([result-grid
      (for/list ([x (in-range 0 81)])
        (smt:eval (select/s sudoku-grid x)))])
      ; Assert that we want a brand new solution by
      ; asserting (not <current solution>)
      (smt:assert
       (not/s (apply and/s
                (for/list ([(x i) (in-indexed result-grid)])
                  (=/s (select/s sudoku-grid i) x)))))
      (if (eq? (smt:check-sat) 'sat)
          #f ; Multiple solutions
          result-grid))
    #f)))
```

2.1.2 Number Mind

The deductive game Bulls and Cows, commercialised as Master Mind [27], is popular all around the world. The rules may vary slightly, but their essence stays the same: Two players play the game. One player (we'll call her Alice) thinks of a 4-digit number, and the other (Bob) tries to find it. Bob guesses a number, and Alice tells him how many digits he has correct and in the correct position (*bulls*) and how many he has correct but in the wrong position (*cows*). Through repeated guessing Bob tries to arrive at the answer.

The game is deceptively simple: while even the standard 4-digit variant is challenging for humans, the general problem for n digits is NP-complete [28]. As such, it becomes an interesting problem for constraint solvers.

For simplicity, we tackle a variant of the game: Number Mind [29], where Bob only tells Alice how many digits are correct and in the correct place (bulls). The user is Alice and the computer Bob, which means that the game is *interactive*. An API to solve Number Mind would have

(a) a way to tell the computer how many digits the number has

(b) a way for the computer to guess a number

(c) a way for the user to tell the computer how many digits it got correct in the last guess.

The constraint solver would have an important role in not just (a) and (c) but also (b), since we would like the computer to make "reasonable" guesses and not just wild ones. We do this by never guessing a number that would be impossible because of the answers already given.

Our system makes all three tasks simple. Figure 2.5 on the next page defines three functions, each corresponding to one of the tasks above.

As a demonstration of `z3.rkt`, we have written a small web application around the code. The web application is available at

`http://numbermind.less-broken.com`

The source is also available:

`https://github.com/sid0/numbermind`

2.2 Design and implementation

`z3.rkt` is currently implemented as a few hundred lines of Racket code that interface with the Z3 engine via the provided library. Since the system is still a work in progress, some of these details might change in the future.

The Z3 wrapper.

We use Racket's foreign interface [30] to map the Z3 library's C functions into Racket. The programmer interface communicates with Z3 by calling the Racket functions defined by the wrapper. While it is possible to use the Z3 wrapper directly, we highly recommend using the programmer interface instead.

Figure 2.5 Solving Number Mind using `z3.rkt`

```
; (a) Create variables for each digit
(define (make-variables num-digits)
  (define vars (smt:make-fun/list num-digits () Int))
  ; Every variable is between 0 and 9
  (for ([var vars]) (smt:assert (and/s (>=/s var 0) (<=/s var 9))))
  vars)

; (b) Guess a number. Returns the guess as a list of digits,
; or #f meaning no number can satisfy all the constraints.
(define (get-new-guess vars)
  (define sat (smt:check-sat))
  (if (eq? sat 'sat)
      ; Get a guess from the SMT solver
      (map smt:eval vars)
      #f))

; (c) How many digits the computer got correct. If a digit is
; correct then we assign it the value 1, otherwise 0. We sum up
; the values and assert that that's equal to the number of correct
; digits.
(define (add-guess vars guess correct-digits)
  (define correct-lhs
    (apply +/s
           (for/list ([x guess]
                      [var vars])
             (ite/s (=/s var x)
                    1      ; Correct guess
                    0))))  ; Wrong guess
  (smt:assert (=/s correct-lhs correct-digits)))
```

Built-in functions.

Z3 comes with a number of built-in functions that operate on booleans, numbers, and more complex values. We expose these functions directly but add a `/s` suffix to their usual names in the SMT-LIB standard, because most SMT-LIB names are already defined as functions by Racket and we want to avoid colliding with them.

The core commands.

This is a small set of Racket macros and functions layered on top of the Z3 wrapper. The aim here is to hide the complexities of the C wrapper (as discussed in Section 1.3) and stay as close to SMT-LIB version 2 commands [22] as possible. We prefix commands with `smt:` to avoid collisions with Racket functions.

2.2.1 Derived abstractions

Since the full power of Racket is available to us, we can define abstractions that allow users to simplify their code. For example, SMT-LIB allows users to define macros via the `define-fun` command, as demonstrated by Figure 2.6.

Figure 2.6 An SMT-LIB macro, defined with `define-fun`

```
(define-fun max ((a Int) (b Int)) Int
  (ite (> a b) a b)) ; ite is short for if-then-else
...
(assert (= (max 4 7) 7))
```

However, Z3's API exposes no such command. Our first attempt was to define a Racket function to do the same thing, as in Figure 2.7.

Figure 2.7 A first attempt at "macros" in `z3.rkt`

```
(define (smt-max a b)
  (ite/s (>/s a b) a b))
...
(smt:assert (=/s (smt-max 4 7) 7))
```

This works for smaller macros like `max`, but in our experience this sort of naïve substitution can result in final expressions for deeply nested functions becoming too large for Z3 to handle[1].

We note, however, that any macro can also be written as a universally quantified formula. For example, `max` can be rewritten as shown in Figure 2.8.

Figure 2.8 Macros as universally quantified formulas

```
(declare-fun max (Int Int) Int)
(assert (forall ((a Int) (b Int))
                (= (max a b)
                   (ite (> a b) a b))))
```

Indeed, Z3 has a *macro finder* component that identifies and eliminates universal quantifiers that are macros in disguise. We finally solved the problem by providing a Racket macro, `smt:define-fun`, that has the same syntax as the SMT-LIB command and that performs precisely this transformation.

The definition of `smt:define-fun` is listed in Figure 2.9 on the next page. We use Scheme's `syntax-rules` macro system [24, Section 4.3.2] to its fullest extent. `syntax-rules` accepts pairs of input and output patterns and goes with the output pattern for the first input that can be matched, somewhat like the `cond` construct found in many Lisps. We handle two separate cases: (a) we're defining a plain identifier, in which case we have no need for the `forall`, and (b) we're defining a macro as above, in which case we do. The `...` as part of the macro definition is a special form recognized by `syntax-rules`: wherever it sees them in the output pattern, it substitutes for them a list of whatever was present in the input pattern. An example of this substitution is listed in Figure 2.10 on the facing page.

[1] In theory, we could merge common parts of expressions to reduce the number of AST nodes generated. In our experiments, this proved to be effective, yet still significantly slower than the solution we finally adopted.

Figure 2.9 A routine for defining macros in `z3.rkt`

```
(define-syntax smt:define-fun
  (syntax-rules ()
    [(_ id () type body) ; Plain identifiers don't need a forall
     (begin
       (smt:declare-fun id () type)
       (smt:assert (=/s id body)))]
    [(_ id ((argname argtype) ...) type body)
     (begin
       (smt:declare-fun id (argtype ...) type)
       (smt:assert (forall/s ((argname argtype) ...)
                             (=/s (id argname ...) body))))]))
```

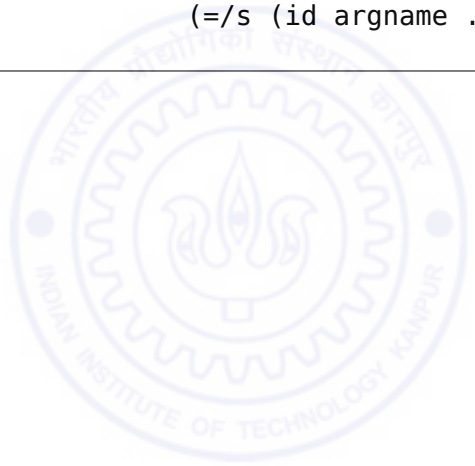


Figure 2.10 `smt:define-fun` in action

```
(smt:define-fun foo ((x Int) (y Bool)) Int
                    (+/s x (ite/s y 20 0)))
```

↓ expands to

```
(smt:declare-fun foo (Int Bool) Int)
(smt:assert (forall/s ((x Int) (y Bool))
                      (=/s (foo x y) (+/s x (ite/s y 20 0)))))
```

2.2.2 Porting existing SMT-LIB code

One of our explicit goals is to enable existing SMT-LIB version 2 code to be ported with a small number of systematic changes. Table 2.1 lists the minimal set of changes that needs to be made to port existing SMT-LIB code to `z3.rkt`. We expect many SMT-LIB programs to become shorter as authors use Racket features wherever appropriate.

Table 2.1: Differences between SMT-LIB and `z3.rkt`

SMT-LIB code	`z3.rkt` code
Options: `(set-option :foo true)`	Keyword arguments: `(smt:new-context #:foo #t)`
Logics: `(set-logic QF_UF)`	The `#:logic` keyword: `(smt:new-context #:logic "QF_UF")`
Commands: `declare-fun`, `assert`, …	Prefixed with `smt:`
Functions: `and`, `or`, `+`, `distinct` …	Suffixed with `/s`
Boolean literals: `true` and `false`	`#t` and `#f`

Summary

In this chapter, we introduced our attempt to make a simple and accessible SMT solver interface. We demonstrated its power with simple programs to solve logical puzzles, including a web application for an NP-complete logical game. We saw how the Racket language provides facilities that make the jobs of both the interface's designer and its users easier.

Chapter 3

Experiences and applications

We have successfully used `z3.rkt` in a number of applications, from constraint-solving puzzles as illustrated in Section 2.1, to verification and counterexample generation for functional programs. Along the way, we have been pleasantly surprised by how well the power of Z3 and the expressiveness of Racket combine. In this chapter we discuss the lessons we have learnt and a few ideas we have explored.

3.1 Handling quantified formulas

Z3 and other SMT solvers support lists and other recursive types. Z3 provides only basic support for lists: `insert` (cons), `head`, and `tail`. Further, Z3's macros are substitutions and do not support recursion. This makes it challenging to define functions that operate over the entirety of an arbitrary-length list.

Our first thought was to use a universal quantifier, as in Section 2.2.1. Figure 3.1 lists an example of a function that calculates the length of an integer list.

Figure 3.1 Calculating the length of a list with a quantified formula

```
(declare-fun len ((List Int)) Int)
(assert (forall ((xs (List Int)))
  (ite (= xs nil)
       (= (len xs) 0)
       (= (len xs) (+ 1 (len (tail xs)))))))
```

There is a drawback with this approach: solving quantified assertions in Z3 requires model-based quantifier instantiation (MBQI) [31]. MBQI, while powerful, can also be very slow. In our experience, it is very easy to write a quantified formula that Z3's MBQI engine fails to solve in reasonable time[1].

So avoiding quantified formulas altogether seems like a good idea, but how do we do that? The easiest way is to unroll and bound the recursion to a desired depth [34]. One way to do this is to define macros `len-0`, `len-1`, `len-2`, …`len-N`, where each `len-k` returns the length of the list if it is less than or equal to `k`, and `k` otherwise (Figure 3.2).

Figure 3.2 A series of SMT-LIB macros to calculate the length of a list

```
(define-fun len-0 ((xs (List Int)))
  0)

(define-fun len-1 ((xs (List Int)))
  (ite (= xs nil)
       0
       (+ 1 (len-0 (tail xs)))))

(define-fun len-2 ((xs (List Int)))
  (ite (= xs nil)
       0
       (+ 1 (len-1 (tail xs)))))
...
```

Our system makes defining a series of macros like this very easy, as shown in Figure 3.3 on the facing page.

`(make-length 5)` returns an SMT function that works for lists of up to length 5, and returns 5 for anything bigger than that. Note how freely the Racket `if` and `let` forms are mixed into the SMT body. These constructs are evaluated at definition time, meaning that this definition reduces to the series of macros defined above up to length `n`.

[1] In general, it is hard to deal with quantified formulas containing even linear arithmetic, because there is no sound and complete decision procedure for them [32].

Z3 has another way to solve quantified assertions, called *E-matching* [33]. E-matching uses patterns based on ground terms to instantiate quantifiers. We have not yet explored this approach.

Figure 3.3 `z3.rkt` code to generate a bounded recursive function to calculate the length of a list

```
(define (make-length n)
  (smt:define-fun len ((xs (List Int))) Int
    (if (zero? n)
        0         ; len-0 always returns 0
        (ite/s (=/s xs nil/s)
               0
               (let ([sublen (make-length (sub1 n))])
                 (+/s 1 (sublen (tail/s xs)))))))
  len)
```

It is easy to define other bounded recursive functions along the same lines that reverse lists, concatenate them, filter them on a predicate and much more. For example, Figure 3.4 defines a function that creates bounded recursive functions that reverse a list (using an accumulator).

Figure 3.4 A bounded recursive function to reverse lists up to length `n`

```
(define (make-reverse n)
  ; We're using an accumulator so create an internal function
  (define (make-reverse-internal n)
    (smt:define-fun reversen ((xs (List Int)) (accum (List Int)))
                             (List Int)
      (if (zero? n)
          accum     ; reverse-0 always returns the accumulated list

          ; Recursive step: generate function for n-1
          (let ([subreverse (make-reverse-internal (sub1 n))])
            (ite/s (=/s xs nil/s)
                   accum
                   (subreverse (tail/s xs) (insert/s (head/s xs)
                                                     accum))))))
    reversen)

  (define reverse (make-reverse-internal n))
  (lambda (xs) (reverse xs nil/s)))
```

Using these building blocks we can now verify properties of recursive functions.

3.2 Verifying recursive functions

For this section we will work with a simple but non-trivial example: *quicksort*. A simple functional implementation of quicksort might look like Figure 3.5.

Figure 3.5 A functional quicksort implementation

```
(define (qsort lst)
  (if (null? lst)
      null
      (let*
       ([pivot (car lst)]
        [rest (cdr lst)]
        [left (qsort (filter (lambda (x) (<= x pivot)) rest))]
        [right (qsort (filter (lambda (x) (> x pivot)) rest))])
        (append left (cons pivot right)))))
```

This definition is correct, but what if the programmer mistakenly types in `<` instead of `<=`, or perhaps uses `>=` instead of `>`? We note that (a) for a correct implementation, the length of the output will always be the same as that of the input, and that (b) in either buggy case, the length of the output will be different whenever a pivot is repeated in the rest of the list. So comparing the two lengths is a good property to verify.

Using the method discussed in Section 3.1, we can write `make-qsort` that generates bounded recursive versions of `qsort`. We believe that both automatic and manual methods would be feasible (Figure 3.6 on the next page is a manual translation).

With `make-qsort` we can now verify the length property for all input lists up to a certain length `n`.

Proving a property is done by checking that its negation is unsatisfiable. A quicksort works *correctly*[2] for lists up to length `n` iff the above code returns `'unsat`. For quicksorts that are buggy (`'sat`), we can find a counterexample using `(smt:eval`

[2]Here correctness is in the context of the property we are considering, i.e. the length of the output.

Figure 3.6 A bounded recursive version of quicksort (cf Figure 3.5 on the preceding page). `make-le` and `make-gt` create functions to filter lists based on their relation to `pivot`, respectively `<=` and `>`. `make-append` creates functions to append lists

```
(define (make-qsort n lessop-fn greaterop-fn)
  (smt:define-fun qsort ((xs (List Int))) (List Int)
    (if (zero? n)
        nil/s

        ; From here on is the usual definition of quicksort.
        (ite/s (=/s xs nil/s)
               nil/s

               (let*
                ([subqsort (make-qsort (sub1 n))]
                 [pivot (head/s xs)]
                 [rest (tail/s xs)]
                 [left
                  (subqsort ((make-le (sub1 n)) pivot rest))]
                 [right
                  (subqsort ((make-gt (sub1 n)) pivot rest))])

                 ((make-append (sub1 n)) left
                                         (insert/s pivot right))))))
  qsort)
```

Figure 3.7 Verifying length for quicksort

```
(smt:with-context
 (smt:new-context)
 (define qsort (make-qsort n))
 ; adding 1 to the maximum length is enough to show inequality
 (define len (make-length (add1 n)))
 (smt:declare-fun xs () (List Int))
 ; set a bound on the length
 (smt:assert (<=/s (len xs) n))
 ; prove the length property by asserting its negation
 (smt:assert (not/s (=/s (len xs) (len (qsort xs)))))
 (smt:check-sat))
```

`xs)` and `(smt:eval (qsort xs))`. For `n` = 4 on a buggy quicksort with filters `<=` and `>=`, Z3 returned us the counterexample with input `'(-3 -2 -1 -2)`, which as expected contains a repeated element.

Summary

In this chapter, we discussed our experiences using our interface for advanced applications. We described methods to overcome fundamental SMT limitations, and successfully verified or found bugs in non-trivial functional programs with the help of these methods.

In our approach, there is nothing specific to quicksort: this can easily be generalised to other functions that operate on lists and other data structures. The properties to prove can be more complex, such as whether a given sorting algorithm is stable. The only limits are computational constraints and the user's imagination.

Chapter 4

Related work

4.1 SMT integration

Integrating an SMT solver with a language enables programmers in that language to solve whatever logical constraints arise in a program, without needing to resort to hand-coding a backtracking algorithm or other cumbersome methods. The solutions thus obtained can be used in the rest of the program. Thus, it isn't surprising that several such projects exist, most of them available freely on the Internet. These projects differ mainly in the host language, the interface, and the constructs they support.

As most languages support some form of interaction with C functions, they can be said to be already integrated with Z3 (or other SMT solvers) through the C API. However, we do not consider this to be true integration because it doesn't simplify the job of the programmer and, as noted in Section 1.3, it requires her to deal with the internals of the solver.

The integration of Z3 with Scala [35] is one of the most complete implementations available right now. It provides support for adding new theories and procedural abstractions, and also takes advantage of Scala's type system to deal with some type related errors at compile time. The system has been used to solve several challenging problems within and outside the group that developed it. The main disadvantage of this system is that the syntax is quite different from SMT-LIB, and

is sometimes almost as verbose as using the C bindings.

Z3Py [36] is a new Python interface bundled with Z3 4.0: it has its own domain-specific language that is different from SMT-LIB; however, it is much more pleasant to use than the C interface and supports virtually all of Z3's features.

SMT Based Verification (SBV) [37] is a Haskell package that can be used to prove properties about bit-precise Haskell programs. Given a constraint in a Haskell program, SBV generates SMT-LIB code that can be run against either Yices or Z3. SBV supports bit-vectors, integers, reals, and arrays, but not lists or other recursive datatypes.

Yices-Painless [38] integrates Haskell with Yices via its C API. This project does not support arrays, tuples, lists and user defined data types yet. Further, the development of the tool seems to be stalled for some time now (last change to the repository was in January 2011).

The Z3 documentation page lists bindings to other languages like OCaml. These bindings correspond almost one-to-one with the C API, and thus they suffer from the same disadvantages.

4.2 Logic programming and constraint programming

Many of the problems SMT solvers can tackle can also be solved within the logic programming paradigm, where programs are written as first-order logic predicates. However, logic programming languages like Prolog typically have well-defined and transparent search strategies, preventing the sorts of automatic heuristics that allow SMT solvers to be fast. Instead, programmers need to manually bound the search space with goals and cuts in the appropriate places.

Racket supports logic programming via Racklog [39], which works in much the same way as Prolog.

Many languages, including most Prolog variants, have access to libraries that al-

low some form of constraint solving. Advanced toolkits include Gecode [40] for C++ and JaCoP [41] for Java and Scala. Typically, these are limited to problems traditionally associated with constraint programming: booleans, finite domains, integers and perhaps real numbers. However, they also have built-in support for *optimisation* problems, something that is lacking in SMT solvers but can be emulated with a binary search on the cost function.

A classic example deserves a mention here: SICP [42, Section 4.3] describes an `amb` macro for Scheme, which can choose for a variable one out of a set of values given *ambiguously*, so as to satisfy given constraints. `amb` is a simple, lightweight form of logic programming.

4.3 Bounded verification

Our work is inspired by the Leon verifier [34], which goes further and alternately considers underapproximations and overapproximations. Where in Section 3.1 we simply return a default value if we've reached the limit of our recursion, the Leon verifier alternately always satisfies or always rejects once it gets to that point. We chose to simplify our implementation to avoid being mired in mechanics, since that wasn't the main focus of this paper. In the future, we plan to extend our method with ideas from the Leon verifier.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 5

Conclusions

In this thesis, we have presented a new SMT interface called `z3.rkt`, which lets Racket programmers interact with an SMT solver programmatically. We have demonstrated through examples the simplicity and usefulness of such an interaction. The power of `z3.rkt` comes from the facilities provided by Racket to build abstractions on top of the SMT-solving capabilities of Z3. From the user's perspective, the integration is seamless and fully transparent.

Our implementation is open source and freely available at

`http://www.cse.iitk.ac.in/users/karkare/code/z3.rkt/`

5.1 Scope for further work

`z3.rkt`, like all large projects, is a work in progress. What has been implemented as of the writing of this thesis is a useful subset of Z3 functionality, but there are several gaps still to be filled:

- Supporting more Z3 constructs, including bit-vectors and external theories
- Deriving new abstractions guided by practical use cases
- Possibly integrating with other SMT solvers

In the long term, we hope the community will find this system useful and will contribute to the project to solve large practical problems.

References

[1] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *STOC*. 1971, pp. 151–158.

[2] Richard M. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. 1972, pp. 85–103.

[3] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *Journal of the ACM* 7.3 (1960), pp. 201–215.

[4] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[5] Bart Selman, Henry A. Kautz, and Bram Cohen. “Local Search Strategies for Satisfiability Testing”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. 1996, pp. 521–532.

[6] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. “Chaff: Engineering an Efficient SAT Solver”. In: *DAC*. 2001, pp. 530–535.

[7] João P. Marques Silva and Karem A. Sakallah. “GRASP - A New Search Algorithm for Satisfiability”. In: *ICAD*. 1996, pp. 220–227.

[8] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *SAT*. 2003, pp. 502–508.

[9] Henry Kautz and Bart Selman. “Planning as satisfiability”. In: *ECAI*. 1992, pp. 359–363.

[10] OpenSUSE Team. *Portal:Libzypp - OpenSUSE*. `http://en.opensuse.org/Portal:Libzypp`. June 2012 (last accessed).

[11] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. “Automatic predicate abstraction of C programs”. In: *PLDI*. 2001, pp. 203–213.

[12] Randal E. Bryant, Steven German, and Miroslav N. Velev. “Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic”. In: *ACM Transactions on Computational Logic* 2 (1999), pp. 37–53.

[13] Shuvendu K. Lahiri and Sanjit A. Seshia. “The UCLID Decision Procedure”. In: *CAV*. 2004, pp. 475–478.

[14] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "DPLL(T): Fast Decision Procedures". In: *CAV*. Vol. 3114. LNCS. 2004, pp. 175–188.

[15] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)". In: *Journal of the ACM* 53.6 (2006), pp. 937–977.

[16] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. LNCS. 2008.

[17] Bruno Dutertre and Leonardo de Moura. *The Yices SMT solver*. Tech. rep. 2006.

[18] Clark Barrett and Cesare Tinelli. "CVC3". In: *CAV*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. 2007, pp. 298–302.

[19] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. "Program analysis as constraint solving". In: *PLDI*. 2008, pp. 281–292.

[20] Patrice Godefroid, Michael Y. Levin, and David A Molnar. "Automated Whitebox Fuzz Testing". In: *NDSS*. 2008. URL: `http://www.trustst c.org/pubs/499.html`.

[21] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. "Bounded model checking of software using SMT solvers instead of SAT solvers". In: *STTT* 11.1 (2009), pp. 69–83. ISSN: 1433-2779.

[22] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Ed. by A. Gupta and D. Kroening. 2010.

[23] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. `http://racket-lang.org/tr1/`. PLT Inc., 2010.

[24] "Fifth Revised Report on the Algorithmic Language Scheme". In: *ACM SIGPLAN Notices* 33.9 (1998). Ed. by Richard Kelsey, William Clinger, and Jonathan Rees, pp. 26–76.

[25] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jocob Matthews. "Revised^6 Report on the Algorithmic Language Scheme". In: *Journal of Functional Programming* 19.S1 (2009), pp. 1–301.

[26] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. "Languages as libraries". In: *PLDI*. 2011, pp. 132–141.

[27] Donald E. Knuth. "The Computer as a Master Mind". In: *Journal of Recreational Mathematics* 9.1 (1976–77), pp. 1–6.

[28] Jeff Stuckman and Guo qiang Zhang. "Mastermind is NP-Complete". In: *INFOCOMP Journal of Computer Science* 5 (2006), pp. 25–28.

[29] Colin Hughes. *Problem 185 - Project Euler*. `http://projecteuler.net/problem=185`. May 2012 (last accessed).

[30] Eli Barzilay. *The Racket Foreign Interface*. `http://docs.racket-lang.org/foreign/`. March 2012 (last accessed).

[31] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories". In: *CAV*. Grenoble, France, 2009, pp. 306–320.

[32] Joseph Y. Halpern. "Presburger Arithmetic With Unary Predicates is Π^1_1 Complete". In: *Journal of Symbolic Logic* 56 (1991), pp. 56–2.

[33] Leonardo de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers". In: *CADE-21*. 2007, pp. 183–198.

[34] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. "Satisfiability Modulo Recursive Programs". In: *SAS*. 2011, pp. 298–315.

[35] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. "Scala to the Power of Z3: Integrating SMT and Programming". In: *ICAD*. Wroclaw, Poland, 2011, pp. 400–406.

[36] Microsoft Research. *Z3Py - Python interface for the Z3 Theorem Prover*. `http://rise4fun.com/z3py/`. May 2012 (last accessed).

[37] Levent Erkok. *SMT Based Verification: Symbolic Haskell theorem prover using SMT solving*. `http://hackage.haskell.org/package/sbv`. June 2012 (last accessed).

[38] Donald Stewart. *yices-painless: An embedded language for programming the Yices SMT solver*. `http://hackage.haskell.org/package/yices-painless-0.1.2`. March 2012 (last accessed).

[39] Dorai Sitaram. *Racklog: Prolog-Style Logic Programming*. `http://docs.racket-lang.org/racklog/`. May 2012 (last accessed).

[40] Gecode Team. *Gecode: Generic Constraint Development Environment*. `http://www.gecode.org`. May 2012 (last accessed).

[41] JaCoP Team. *JaCoP - Java Constraint Programming solver*. `http://jacop.osolpro.com/`. May 2012 (last accessed).

[42] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd. MIT Press, 1996.