Scala to the Power of Z3: Integrating SMT and Programming

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter*

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract. We describe a system that integrates the SMT solver Z3 with the Scala programming language. The system supports the use of the SMT solver for checking satisfiability, unsatisfiability, as well as solution enumeration. The embedding of formula trees into Scala uses the host type system of Scala to prevent the construction of certain ill-typed constraints. The solution enumeration feature integrates into the iteration constructions of Scala and supports writing non-deterministic programs. Using Z3's mechanism of theory extensions, our system also helps users construct custom constraint solvers where the interpretation of predicates and functions is given as Scala code. The resulting system preserves the productivity advantages of Scala while simplifying tasks such as combinatorial search.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers have in the past few years become very powerful tools. Their efficient search heuristics have made them applicable to a wide variety of problems. However, they are still primarily used by *expert* users that have substantial understanding of constraint solvers, their languages and interfaces. Our aim is to make SMT solving accessible to a wider audience by integrating it into a familiar programming language.

This paper presents Scala$^{\mathsf{Z3}}$, a library to bring the power of the SMT solver Z3 [3] to users of the Scala programming language [4]. We identify two types of clients for our system:

– general programmers, who are not necessarily familiar with SMT, but who may want to use constraint solving as a library;
– SMT power users, who can use it in a way similar to how they would in C, yet will still benefit from a concise language with a strong type system.

2 Implicit Computation Using Z3

Our system enables programmers to state the properties that the values should satisfy instead of how to compute them. In that sense, it supports a form of implicit computation. We illustrate this approach through several examples.

* Alphabetical author order. Philippe Suter was supported by the Swiss NSF Grant 200021_120433.

N. Bjørner and V. Sofronie-Stokkermans (Eds.): CADE 2011, LNAI 6803, pp. 400–406, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Mixing searching with solving. Consider the following satisfiability problem: *Find three integers x, y, z such that $x > 0$, $y > x$, $2x+3y \leq 40$, $x \cdot z = 3y^2$, and y is prime?* We know of no decidable logic in which this problem can be naturally expressed. As an alternative to applying a decision procedure, we can search for a solution. Using the system we present in this paper, we can concisely program the search in Scala as follows:

```
val results = for(
  (x,y) ← findAll((x: Val[Int], y: Val[Int]) ⇒ x > 0 && y > x && x ∗ 2 + y ∗ 3 ≤ 40);
    if(isPrime(y));
      z ← findAll((z: Val[Int]) ⇒ z ∗ x === 3 ∗ y ∗ y))
        yield (x, y, z)
```

This for-comprehension constructs an iterator of integer triples. The iterator ranges over all solutions (in general, it can be infinite, here there are 8 solutions). The for-comprehension interleaves invocations of the SMT solver Z3 —the calls to findAll— and applications of Scala functions —here isPrime, whose definition we omit. Because findAll works by lazily generating a stream, Z3 is only invoked as more values are requested. For instance, if we only wish to check whether a solution exists, we can test results.isEmpty and only one solution will be computed. Similarly, when y is not prime, the inner constraint is not dispatched to the solver. Note that this constraint is, despite its appearance, in linear arithmetic, since x and y are known at the time of its construction. Note that the only constructs that a Scala programmer needs to learn to use the above example is the findAll function, and the Val[_] type constructor. The remaining constructs are a standard part of Scala [4].

N-Queens puzzle. We consider now the problem of solving the N-Queens puzzle: *In how many ways can N queens be placed on an $N \times N$ checkerboard such that they do not attack each other?* The following program encodes the problem using integer arithmetic and invokes the solver to count the number of solutions:

```
val z3 = new Z3Context("MODEL" → true)
val N = 8
val cols = (0 until N) map { _ ⇒ IntVar() } // column vars
val diffCnstr = Distinct(cols : _∗) // all queens on distinct cols
val boundsCnstr = for (c ← cols) yield (c ≥ 0 && c < N) // cols are within bounds
val diagonalsCnstr = // no two queens on same diagonal
  for (i ← 0 until N; j ← 0 until i) yield
    ((cols(i) − cols(j) !== i − j) && (cols(i) − cols(j) !== j − i))

z3.assertCnstr(diffCnstr)
boundsCnstr map (z3.assertCnstr(_))
diagonalsCnstr map (z3.assertCnstr(_))
println(z3.checkAndGetAllModels.size) // prints 92
```

In this example, we use Scala^{Z3} with the same degree of control we would have with the native interface: we build the context explicitly, push constraints, etc. We start by declaring a list of Z3 constants; the i-th constant representing the

(integer) column value of the queen that will be placed on row i. We then specify the constraints, stating that each queen is on a different column, row and diagonal. Finally we assert these three constraints in the current context, and invoke the solver to retrieve the stream of all solutions that satisfy the constructed formulas. Most variables in this program are of the type Tree yet type inference allows us to keep this transparent. Thanks to operator overloading, the meaning of the constraints is clear from the code.[1]

Calendar computation. Implicit computations are useful not only as a form of constraint solving, but also in cases where writing code that matches a precise specification may be hard; in such cases we can sometimes replace explicit code by an implicit definition. Our next example shows how we can use Scala^{Z3} to compute date differences while accounting for leap years.[2] The following program takes as input a number of days totalDays and computes the year and the day in the year that correspond to totalDays since January 1st, 1980.

```
val totalDays = 10593
val originYear = 1980

val (year, day) = choose((year: Val[Int], day: Val[Int]) ⇒ {
  def leapYearsUntil(y : Tree[IntSort]) = (y − 1) / 4 − (y − 1) / 100 + (y − 1) / 400

  totalDays === (year − originYear) ∗ 365
    + leapYearsUntil(year) − leapYearsUntil(originYear) + day &&
  day > 0 && day ≤ 366
})

println(year + ", " + day) // prints 2008, 366
```

Note that we defined a helper method leapDaysUntil which produces a tree expressing the number of leap years between year 1 and y. This is possible because this auxiliary definition doesn't affect the type of the predicate used in the call to choose. We can then use this method in our predicate to express conveniently the number of total days between January 1st, 1980 and the day specified by year and day.

3 Design and Implementation

Scala^{Z3} is implemented as a Scala library that connects to Z3's C interface through the Java Native Interface [2], and consists of just over 5,000 lines of a combination of Scala, Java, and C code. Although it is possible to use Scala^{Z3} as a simple Scala view of the C or OCaml interface of Z3, there are several features that enable more productive combinations of the two systems.

[1] We use the operators `===` and `!==` to construct ASTs because `==` and `!=` can only return booleans in Scala.

[2] A piece of code that incorrectly performed this computation is famously responsible for a bug that caused thousands of portable media devices to freeze in 2008.

```
abstract class Tree[+A >: Bottom <: Top]

sealed trait Top
trait IntSort extends Top
trait BoolSort extends Top
trait BVSort extends Top
trait ... extends Top
trait SetSort extends Top
trait Bottom extends IntSort with BoolSort with BVSort with ... with SetSort
```

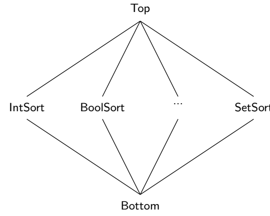

Fig. 1. Soft typing system for the domain specific language

Domain specific language. There are two possible representations of Z3 abstract syntax trees (ASTs) in Scala^{Z3}. The most basic is a Z3AST class that encapsulates a C pointer to the internal representation. This is used by all functions that are direct mappings to the C interface. The other representation encodes Z3 ASTs into typed Scala syntax trees. These trees can be combined using operators with which programmers are familiar, such as &&, +, as well as numerical constants, for instance. The examples throughout this paper are all written using this domain specific language (DSL), which is enabled by adding the import statement **import** z3.scala.dsl._.

The representations are mutually compatible, and through the mechanism of *implicit conversions* [4, Chapter 21], the Scala compiler automatically inserts calls to conversion functions where needed. The DSL trees are *typed* using a soft-typing approach to prevent the construction of some ill-typed terms. Figure 1 shows the type system, which relies on Scala generic types and multiple inheritance. For instance, the $<$ operator defined on trees of integer sort has the following signature:

```
def <(that: Tree[_ <: IntSort]): Tree[BoolSort]
```

This declaration indicates that $<$ expects an operand of a type equal to (or a subtype of) Tree[IntSort] and returns a tree of type Tree[BoolSort]. Because Z3ASTs are by nature untyped and should be usable in combination with the DSL, they are converted to trees of type Tree[Bottom]. Because trees are covariant in their type parameter, such trees can be used in place of any type. In these cases, the library performs a runtime check to ensure that the types match, so as to avoid triggering an error in the Z3 native library.

High-level model navigation. One key feature of the system is the ability to evaluate the model of a Z3 constant as a Scala type. This is achieved by a generic method whose signature is:

```
def evalAs[T](ast: Z3AST)(implicit extr: (Z3Model, Z3AST) ⇒ Option[T]): Option[T]
```

It returns an Option type because the model may not define a value for the desired tree. The definition refers to an *implicit* parameter extr. Implicit parameters are parameters that can be omitted at the call-site, and that will be filled according to objects that are marked as implicit in the scope. Here, extr is the

function responsible for building a value of the proper Scala type from a Z3 model and constant. How this is done depends on the requested type. Scala^{Z3} thus defines such functions for base types, and Scala's mechanism for resolving implicits automatically inserts the right definitions according to the type T. Because implicit resolution is done at compile-time, invocations of evalAs with unsupported types result in a compile error. Experienced users can also extend this mechanism by writing their own extractors, for instance to automatically build algebraic data types such as lists or trees from models. Scala^{Z3} also provides methods to recover models of uninterpreted function symbols or arrays, and to wrap them in an object that can then be used as a Scala function.

The choose, find and findAll constructs. These three constructs are defined as part of the domain specific language. choose attempts to find one assignment to a constraint, and throws an exception if it could not, while find returns an Option type using None to describe failures. findAll enumerates all models, as in the introductory example. They all take a predicate that describes the constraint as an argument. The particularity of these functions is that all the interaction with Z3 is completely transparent. Similarly to evalAs, choose and findAll rely on implicit arguments to build Z3 trees of the right kind and to retrieve values from the models. Additionally, the type constructor Val[_] encapsulates more implicit conversion functions to build ASTs from it. The complete signature for (the one argument version of) findAll is as follows:

```
def findAll[T](predicate : Val[T] ⇒ Tree[BoolSort])
              (implicit cons : T ⇒ Tree[Bottom],
                        extr : (Z3Model, Z3AST) ⇒ T) : Iterator[T]
```

In the implementation, an iterator is constructed by maintaining the Z3 context and successively pushing the negation of the previous model as a new constraint when the next model is requested. Since iterators are standard in Scala, we can use all the usual higher-order constructs on the result, including for instance map, filter or for-comprehensions.

4 Theory Plugins

Z3 supports user-defined *theory plugins*; users can integrate their decision procedure into Z3's DPLL engine by specifying callbacks that are invoked by Z3 on events such as context pushes and pops, newly propagated equalities, etc. We now give two examples of how this mechanism can be exploited through Scala^{Z3}.

Theory plugin for sets with cardinality constraints. We implemented a decision procedure for Boolean Algebra with Presburger Arithmetic, a logic that supports sets with cardinality constraints,[3] as a full-fledged theory extension to Z3 using Scala^{Z3} [5]. The details of the implementation are too complex to be presented here, but we shortly illustrate here some of the programming language aspects that we believe simplified this development. Scala is also an object-oriented language, and following that paradigm, user-defined theory plugins are

[3] Z3 natively supports sets, but without the cardinality operator.

created by subclassing a class Z3Theory defined as part of Scala^{Z3}. A definition such as the following is all that is needed to add a theory solver to Z3's DPLL engine:

```
class BAPATheory(val z3: Z3Context) extends Z3Theory(z3, "Sets with cardialities") {
  // User−defined sorts, constant values and functions:
  val setSort = mkTheorySort("setSort")
  val emptySet = mkTheoryValue("empty", setSort)
  // Declares a unary function from sets to integers:
  val cardinality = mkTheoryFuncDecl("card.", setSort, z3.mkIntSort)

  // This method is automatically called when a new term enters the logical context:
  override def newApp(ast: Z3AST) : Unit = ast.getKind match {
    case Z3AppAST(`cardinality`, arg) ⇒ processCard(arg)
    case _ ⇒ ...
}}
```

The interaction with Z3 is done by overriding the right methods, like newApp in the example above, which replace the callback functions used by the C interface. Theory plugins typically need to manipulate many abstract syntax trees communicated from Z3. To simplify such tasks, Scala^{Z3} defines *extractors*, which are functions that can be used in pattern-matching expressions [1]. The newApp method contains an example, where with a single line of code we test whether a Z3 tree corresponds to an application of the cardinality function and at the same time bind the variable args to its argument.

Procedural attachments. Scala^{Z3} provides special support for *procedural attachment* extensions. Procedural attachments are a special kind of theory plugins where the interpretation of ground terms is provided as executable functions. To illustrate their use, consider the code below, where we define two predicates and one function over strings:

```
val z3 = new Z3Context()
// Defines a new theory of strings with two predicates and one function symbol.
val strings = new ProceduralAttachment[String](z3) {
  val oddLength = predicate(s ⇒ s.length % 2 == 1)
  val isSubstr = predicate((s1,s2) ⇒ s2.contains(s1))
  val concat = function((s1,s2) ⇒ s1 + s2)
}
```

From this declaration, Scala^{Z3} constructs a Z3 theory plugin for a new sort representing strings and creates the proper predicate and function symbols. It also registers callbacks such that any ground term built over string constants is 1) translated back into Scala, 2) evaluated using the function definitions passed by the user, 3) converted back into Z3 trees. A usage example follows:

```
import strings._
val s1, s2 = variable
z3.assertCnstr(s1 === "hello" && (s2 === "world" || s2 === "moon")
  && oddLength(concat(s2, s1)) && isSubstr("low", concat(s1,s2)))
println(z3.check) // unsatisfiable
```

The import statement brings into scope not only the predicate and function symbols, which can then be used as part of the domain specific language, but also helper functions such as variable, which creates a Z3 tree for a variable representing a string, as well as an implicit conversion function which converts any string into a tree node representing its constant value. As a result, the constraints can be expressed very naturally. Using Z3's DPLL engine to assign truth values to literals, the system concludes that the constraints cannot be satisfied. Procedural attachment theories are in general not complete and may return *unknown* when some variables never become ground. They remain very useful extensions, though, for instance when all variables are known to range over a finite domain.

5 Conclusions

We have demonstrated that it is possible and fruitful to smoothly integrate a modern programming language and a powerful SMT solver. Our system enables users to dynamically construct constraints, while supporting the syntax of the underlying programming language. It enables combinatorial search that combines Z3's constraint solving with explicit tests and enumeration in the programming language, as well as the creation of custom theory solvers based on executable functions.

We have found a number of uses for Scala^{Z3} in our research group, including several program verification tools under development, as well as a new theory plugin for Z3 [5]. We have also recently received interest from other groups to use and contribute to Scala^{Z3}. Our implementation is freely available at:

`http://lara.epfl.ch/w/ScalaZ3`

We hope that the community will join the effort in enhancing the implementation further. The current version includes mappings for all Z3 operations (including manipulation of abstract data types, arrays and bitvectors, for instance), so expert users can already use it as a substitute for the C interface. Among the particularly desirable future extensions are: high-level support for further data types, parallel invocation of Z3 instances, and reconstruction of proof objects.

References

1. Emir, B., Vetta, A., Williams, J.: Matching objects with patterns. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 273–298. Springer, Heidelberg (2007)
2. Liang, S.: The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, London (1999)
3. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)
5. Suter, P., Steiger, R., Kuncak, V.: Sets with cardinality constraints in satisfiability modulo theories. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 403–418. Springer, Heidelberg (2011)