# Constraints as Control

Ali Sinan Köksal Viktor Kuncak Philippe Suter *

School of Computer and Communication Sciences (I&C) - Swiss Federal Institute of Technology (EPFL), Switzerland
{firstname.lastname}@epfl.ch

## Abstract

We present an extension of Scala that supports constraint programming over bounded and unbounded domains. The resulting language, Kaplan, provides the benefits of constraint programming while preserving the existing features of Scala. Kaplan integrates constraint and imperative programming by using constraints as an advanced control structure; the developers use the monadic 'for' construct to iterate over the solutions of constraints or branch on the existence of a solution. The constructs we introduce have simple semantics that can be understood as explicit enumeration of values, but are implemented more efficiently using symbolic reasoning.

Kaplan programs can manipulate constraints at run-time, with the combined benefits of type-safe syntax trees and first-class functions. The language of constraints is a functional subset of Scala, supporting arbitrary recursive function definitions over algebraic data types, sets, maps, and integers.

Our implementation runs on a platform combining a constraint solver with a standard virtual machine. For constraint solving we use an algorithm that handles recursive function definitions through fair function unrolling and builds upon the state-of-the art SMT solver Z3. We evaluate Kaplan on examples ranging from enumeration of data structures to execution of declarative specifications. We found Kaplan promising because it is expressive, supporting a range of problem domains, while enabling full-speed execution of programs that do not rely on constraint programming.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Design, Languages

***Keywords*** Constraint Programming, Satisfiability Modulo Theories, Executable Specifications, Scala, Embedded Domain-Specific Languages, Non-determinism

## 1. Introduction

Modern mainstream programming languages incorporate advances in memory safety, type systems, meta-programming and modularity. However, the sequential control in widely used systems remains largely unchanged compared to some of the earliest imperative and functional language designs. Although this simplicity has advantages in terms of predictable compilation, it prevents languages from approaching the abstraction level used in software design. In design we often encounter a conjunction of multiple orthogonal requirements that we need to meet simultaneously [29]. When conjoining multiple requirements, each of them needs to be a *partial* specification of possible executions, otherwise the conjunction would be contradictory. The desire to use partial constraints leads to non-deterministic specification constructs. Examples include guarded commands [18], wide-spectrum languages [6, 43], intermediate verification languages [15], Temporal Logic of Actions [36], Alloy [30], as well as models in proof assistants Coq [62] and Isabelle [52]. In contrast to specification languages, most mainstream executable languages have difficulty in implementing a conjunction of programs; their sequential commands tightly specify the set of possible behaviors.

The idea of Logic Programming [14, 34] is to allow programs that are non-deterministic and specify relations (predicates) between values. The original inspiration came from a restricted form of resolution as the execution mechanisms, which uses unification of logical variables ranging over finite trees. Functional logic programming extends logic programming with the narrowing technique and with benefits of functional programming [3]. Constraint Logic Programming, CLP [31] and later generations of Prolog [14] extend logic programming with the ability to perform constraint solving not only over trees but also over numerical domains. These paradigms hold great promise to raise the abstraction level of software. They are related to program synthesis research [26, 35, 39, 53, 58], which can be viewed as a compilation mechanism for declarative specifications.

We believe that key obstacles preventing broader use of non-deterministic declarative programming constructs include:

- the difficulty of solving declarative constraints and
- the difficulty of incorporating these constructs into existing languages and platforms.

We next briefly outline the approach that our system uses to meet these challenges.

**Efficient interoperability with existing platforms.** In this paper we present Kaplan, a system that supports Constraint Programming on top of the Scala language [50]. We address the difficulty of incorporating into existing platforms by choosing not to modify the semantics of the core Scala language, but instead use the flexibility of for-comprehensions in Scala. Because for-comprehensions present a syntax for monads in Scala, this aspect of our approach is related to using non-determinism monads in Haskell [21]. An important difference is that our starting language is not purely functional, but can have side effects.

Our solution is therefore to identify a functional Turing-complete sublanguage for constraints, and use it locally for declarative programming within the full Scala language (which has features of

* Philippe Suter was supported by the Swiss National Science Foundation Grant 200021_120433.



*POPL'12,* January 25–27, 2012, Philadelphia, PA, USA.

object-oriented, imperative and functional languages). The declarative control that we explore is based on describing iterations as a search process. The iteration ranges are specified as the set of solutions to constraints. Our constraints are declared using first-class functions and familiar combinators, such as &&. From a language point of view the integration is appealing because users hardly need to learn any new notation.

One approach to implementation of non-deterministic languages is to implement a virtual machine that supports backtracking, such as Warren's Abstract Machine for Prolog [1] or the Java Pathfinder model checker and its extensions [24]. Unfortunately, such an approach is costly in terms of both performance and engineering effort. We propose instead to encapsulate any need for backtracking into a for-comprehension that iterates over solution spaces. In addition, we allow constraint programming in code outside of loops. In such cases, logical variables and a constraint store help ensure that a program can find a solution without backtracking over the host imperative program, keeping backtracking within a specialized constraint solver.

**Efficient solving of declarative constraints.** The first implementations of declarative paradigms predate algorithmic advances of modern SAT solvers [57, 66]. The advances in SAT led to a paradigm shift in solving combinatorial problems, where it became often profitable to outsource constraint solving tasks to dedicated implementations, instead of relying on a programmer to hard-code a search strategy.

Fueled by the developments of SAT solvers, a more expressive technology emerged as the field of Satisfiability Modulo Theories (SMT), with a number of efficient implementations available today [8, 19, 44]. SMT techniques combine SAT solvers with decision procedures and their combination methods [17, 22, 46] that were and remain motivated by program verification tasks [32, 45].

Our aim is to leverage the remarkable progress in dedicated constraint solvers by deploying an SMT solver as a part of the run-time of a programming language. Whereas the developers of Prolog were influenced by the state-of-the-art theorem proving technology of their time (resolution for first-order logic), we aim to explore the potential of SMT. The fact that SMT solvers were developed to model programming language constructs makes them a particularly appropriate choice. However, the original motivation for SMT solvers was program verification, whereas we aim to use them as an execution mechanism for declarative constraints.

Many logical theories are natively supported by modern solvers, including algebraic data types, uninterpreted functions, linear arithmetic, and arrays. Nonetheless, we believe that constraint solving can reach its full potential only if users can define their own classes of constraints. We therefore choose to work with a solver for a rich logic in which users can define their own recursive functions and recursive data types [60]. We thus trade performance and decidability for greater expressive power and flexibility.

**Contributions.** This paper makes a concrete and implemented proposal for incorporating constraint programming into an underlying stateful language through several individual contributions:

- first-class constraints, which can be generated at run-time and which carry type and free-variable information; the constraints can use unrestricted operations on booleans (including negation), as well as built-in and user-defined recursive operations on integers, sets, maps, and trees;
- a programming model for constraint programming based on creating and iterating over a stream of solutions of a constraint using explicit control constructs; our programming model (unlike most solutions with backtrackable virtual machines) has no penalty for the code that does not use constraint solving;
- logical variables of numeric and symbolic types that postpone constraint solving steps, often substantially reducing the size of the search space;
- the use of fair function unrolling and SMT solving technology to provide expressive power, predictability, and efficiency of solving in many domains of interest;
- implementation of the system (Kaplan is publicly available from `http://lara.epfl.ch`);
- evaluation of the system on examples from several domains, such as executing declaratively specified data structure operations, software testing, and counterexample-driven construction of functions of a given template and a given specification.

**Paper outline.** The rest of this paper is organized as follows. The next section presents features of Kaplan through an extensive set of examples. Section 3 presents in more detail the Turing-complete language that we use to describe constraints. Section 4 gives the semantics of key Kaplan constructs. Section 5 outlines main implementation aspects of Kaplan. We present further evaluation and illustrate use cases of Kaplan in Section 6. We finally review the remaining related work and conclude.

## 2. Examples and Features

We present some of the features of Kaplan through examples. The simpler examples can be tried out directly in a Scala shell, provided that it is launched with the Kaplan plugin. Throughout the paper, we assume basic familiarity with Scala, but explain more advanced concepts and constructs as needed.

### 2.1 First-class Constraints

```
val c1: Constraint2[Int,Int] =
  ((x: Int, y: Int) ⇒ 2∗x + 3∗y == 10 && x ≥ 0 && y ≥ 0)
```

This first command declares a constraint with two free variables. Note that the representation of the constraint is a lambda term. The only difference between a declaration of a constraint and an anonymous function is the type of the expression. As an alternative to explicitly declaring the value to be of a constraint type, one can also append to the function literal a method call .c, so the following declaration is identical to the previous one:

```
val c1 =
  ((x: Int, y: Int) ⇒ 2∗x + 3∗y == 10 && x ≥ 0 && y ≥ 0).c
```

The type of c1 is in this case determined by type inference, and this second way is generally shorter. In Kaplan, constraints are in fact extensions (in the object-oriented sense) of functions, and can thus be evaluated, given some values for their argument variables:

```
scala> c1(2,1)
result: false
scala> c1(5,0)
result: true
```

As is common in lambda calculus, the names given to the bound variables play no role; the constraint $((x\colon \mathsf{Int}) \Rightarrow x \geq 0)$ is equivalent to the constraint $((y\colon \mathsf{Int}) \Rightarrow y \geq 0)$. One should think of constraints being defined over (typed) De Bruijn indices rather than named variables.

Constraints can be queried for a single solution or for a stream of solutions by calling appropriate methods:

```
scala> c1.solve
result: (5,0)
scala> c1.findAll
result: non−empty iterator
```

The solve method computes a single solution to the constraint, while findAll returns an iterator over all solutions. The iterator computes and returns solutions on demand. (As a result of displaying the iterator in the console, a search for the first solution is triggered.) When the set of solutions is finite, one can compute it for instance as follows:

```
scala> c1.findAll.toList
result: List((5,0),(2,2))
```

The most general way to iterate over solutions is using a for-comprehension:

```
for(s ← c1.findAll) {
  println(s)
}
```

Constraints are first-class members of Kaplan, and can be created and manipulated as such. The following function, for instance, generates constraints describing an integer within bounds; if the second argument is true, the bounds are given by $[-m; m]$, else by $[0; m]$.

```
def bounded(m: Int, neg: Boolean = true) = {
  val basis: Constraint1[Int] = ((x: Int) ⇒ x ≤ m)
  basis && (if(neg)
    ((x: Int) ⇒ x ≥ −m)
  else
    ((x: Int) ⇒ x ≥ 0))
}
```

In this example, && is a call to a method defined as part of constraint classes, and which expects as argument a constraint of the same arity and with the same types of parameters as the receiver. This ensures that constraints can only be combined when the number and the types of their De Bruijn indices are compatible. Note that, thanks to type inference, we need not explicitly state that the two anonymous functions represent constraints. We can now use the function bounded to produce constraints:

```
scala> bounded(3, true).findAll.toList
result: List(0, −1, −2, −3, 1, 2, 3)
scala> bounded(3, false).findAll.toList
result: List(0, 1, 2, 3)
```

Another convenient constraint combinator is the product method, which combines two constraints into a new one whose solution set is the Cartesian product of the original two:

```
scala> (bounded(1, true) product bounded(1,false)).findAll.toList
result: List((−1,0),(0,0),(1,1),(−1,1),(1,0),(0,1))
```

Constructing constraints using combinators is a very concise way to solve general problems. For instance, consider the problem of solving a CNF SAT instance defined in a way similar to the standard DIMACS format, where the input

```
val p1 = Seq(Seq(1,−2,−3), Seq(2,3,4), Seq(−1,−4))
```

represents the problem

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

The following function defines a solver for such problems:

```
def satSolve(problem : Seq[Seq[Int]]) : Option[Map[Int,Boolean]] =
  problem.map(l ⇒ l.map(i ⇒ {
    val id = scala.math.abs(i)
    val isPos = i > 0
    ((m : Map[Int,Boolean]) ⇒ m(id) == isPos).c
  }).reduceLeft(_ || _)).reduceLeft(_ && _).find
}
```

Note that in this case we used the find method on the final constraint rather than solve. They provide the same functionality, but find returns its result in an Option[_] type, where None corresponds to an unsolvable constraint, while solve throws an exception if the constraint has no solution. Another observation is that we use here a constraint over a single variable of Map type to encode a constraint over an unknown number of boolean variables. We found this to be a convenient pattern when the more rigid syntax of anonymous functions with explicit variable naming does not apply. Because all variables are of the same type, the approach works in our type-safe framework. We can now solve SAT problems:

```
scala> satSolve(p1)
result: Some(Map(2 → true, 3 → false, 1 → false, 4 → false)))
scala> satSolve(Seq(Seq(1,2), Seq(−1), Seq(−2)))
result: None
```

### 2.2 Ordering Solutions

As illustrated by some of the previous examples, the findAll method generates solutions in no particular order. This corresponds to the intuition that it enumerates the unordered set of solutions to a constraint. Similarly, the semantics of calls to solve or find are simply that if a value is produced, then it is an element of that solution set. Two invocations of solve on the same constraint may or may not result in the same solution.

It is sometimes desirable, though, to enumerate solutions in a defined order. To this end, Kaplan constraints support two methods; minimizing and maximizing. These methods take as argument an objective function. Just like constraints, objective functions are represented using an anonymous function, in this case one that returns an integer. The minimizing and maximizing methods ensure that the De Bruijn indices' types match those of the constraint. From the user's point of view, these functions are typed as IntTerms.[1]

Consider the knapsack problem: given a maximum weight and a set of items, each with an associated value and weight, find a subset of the items for which the sum of values is maximal while the sum of weights is less or equal to the maximum. The following code produces and solves an instance of the problem, where the solution is represented as a map from the item indices to booleans indicating which should be picked.

```
def solveKnapsack(vals : List[Int], weights : List[Int], max : Int) = {
  def conditionalSumTerm(vs : List[Int]) = {
    vs.zipWithIndex.map(pair ⇒ {
      val (v,i) = pair
      ((m : Map[Int,Boolean]) ⇒ (if(m(i)) v else 0)).i
    }).reduceLeft(_ + _)
  }
  val valueTerm = conditionalSumTerm(vals)
  val weightTerm = conditionalSumTerm(weights)
  val answer = ((x : Int) ⇒ x ≤ max).compose0(weightTerm)
    .maximizing(valueTerm)
    .solve
}
```

We briefly explain the code. A solution to a knapsack instance is a map indicating a choice of which objects should be picked. The conditionalSumTerm function builds, from a list of integers, an integer term parameterized by a choice map and representing a sum of values. The map defines whether each element in the list participates or not to the sum. We use this function twice, to produce the two terms that, given a choice of items, encode the total value and the weight respectively. Observe that we build the final constraint using a function composition: we start the construction of the constraint as (x : Int) ⇒ x ≤ max and compose it with the weight term to produce the constraint that the weight should not exceed the maximum. Function composition is a general and type-safe way to build constraints from smaller terms, as demonstrated

[1] The relationship from an implementation point of view between constraints and terms is explained in more details in Section 5.

in this example. We can now find optimal choices for instances of the knapsack problem:

```
scala> val vals : List[Int] = List(4, 2, 2, 1, 10)
scala> val weights : List[Int] = List(12, 1, 2, 1, 4)
scala> val max : Int = 15
scala> solveKnapsack(vals, weights, max)
result: Map(0 → false, 1 → true, 2 → true, 3 → true, 4 → true)
```

### 2.3 User-defined Functions and Datatypes

An important feature of Kaplan is the ability to perform constraint solving in the presence of user-defined functions and data types. Consider the following functions which compute, for a $2 \times 2$ matrix $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ represented as a tuple$(a, b, c, d)$, its determinant and whether it is unimodular, respectively:

```
@spec def det(a: Int, b: Int, c: Int, d: Int): Int = a*d − b*c
@spec def unimodular(a: Int, b: Int, c: Int, d: Int): Boolean = {
  val dt = det(a, b, c, d)
  dt == 1 || dt == −1
}
```

The @spec annotation indicates that the user wishes to use the functions as part of constraints, and the Kaplan compiler enforces that such functions are written in the subset of Scala supported within constraints (see Section 3). We can now characterize unimodular matrices with small elements as

```
def boundedUnimodular(m: Int) = {
  val b = bounded(m, false)
  (b product b product b product b) && (unimodular _)
}
```

(The underscore after unimodular indicates to the Scala compiler that it should apply an $\eta$-conversion to produce an anonymous function, and ultimately a constraint, from the definition.) We can use these new definitions to generate unimodular matrices:

```
scala> boundedUnimodular(2).findAll.take(4).toList
result: List((0,1,1,0), (0,1,1,2), (0,1,1,1), (1,1,2,1))
```

Perhaps more interestingly, the @spec functions can be mutually recursive. As an example, consider the following declarative definition of prime numbers:

```
@spec def noneDivides(from : Int, j : Int) : Boolean = {
  from == j || (j % from != 0 && noneDivides(from+1, j))
}
@spec def isPrime(i : Int) : Boolean = (i ≥ 2 && noneDivides(2, i))
val primes = ((isPrime(_:Int)) minimizing ((x:Int) ⇒ x)).findAll
```

which we can subsequently enumerate:

```
scala> primes.take(10).toList
result: List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Kaplan users can also define their own recursive data types (also known as algebraic data types). The following code declares two such types for red-black trees:

```
@spec sealed abstract class Color
@spec case class Black() extends Color
@spec case class Red() extends Color
@spec sealed abstract class Tree
@spec case class Node(c : Color, l : Tree, v : Int, r : Tree) extends Tree
@spec case class Leaf() extends Tree
```

Algebraic data types are best manipulated using recursive functions and pattern-matching. Kaplan supports such function definitions:

```
@spec def size(tree : Tree) : Int = (tree match {
  case Leaf() ⇒ 0
  case Node(_, l, _, r) ⇒ 1 + size(l) + size(r)
}) ensuring(result ⇒ result ≥ 0)
```

This function also illustrates the use of *post-conditions*, written in Scala as anonymous functions in an **ensuring** clause [49]. In this case, the post-condition states that the result of size can never be negative. Such annotations can help the constraint solver discard parts of the search space. Kaplan uses the Leon verification system [60] to prove, at compile time, that the annotations are valid, so that it can then rely on them at run-time for constraint solving. We can similarly define recursive functions that compute whether a red-black tree contains sorted keys or has the right coloring properties. In the interest of space, we omit the complete definitions.

```
@spec def orderedKeys(t : Tree) : Boolean = ...
@spec def validColoring(t : Tree) : Boolean = ...
@spec def validTree(t : Tree) = orderedKeys(t) && validColoring(t)
@spec def valsWithin(t : Tree, min : Int, max : Int) : Boolean = ...
```

We expect that what these functions compute is clear from their name. With them, we can for instance count the number of red-black trees with a given number of elements:

```
scala> (for(i ← (0 to 7)) yield ((t : Tree) ⇒ validTree(t) &&
    valsWithin(t, 0, i) && size(t) == i ).findAll.size).toList
result: List(1, 1, 2, 2, 4, 8, 16, 33)
```

More advanced applications of data structure enumeration are presented in Section 6, including testcase generation for sorted list and tree manipulations.

### 2.4 Timeouts

The language of constraints supported in Kaplan is very expressive. For that reason, one cannot expect that any constraint will be solvable. As an example, given the definition (valid in Kaplan)

```
@spec def pow(x : Int, y : Int) : Int =
  if(y == 0) 1 else x * pow(x, y − 1)
```

it would be unreasonable to expect the system to find a solution to the constraint

```
val fermat = ((x : Int, y : Int, z : Int, b : Int) ⇒ b > 2 &&
  pow(x,b) + pow(y,b) == pow(z,b)).c
```

The search algorithm at the core of Kaplan is a semi-decision procedure: if a solution exists, then that solution will be found eventually. If there are no solutions, the procedure can discover this fact and stop [59, 60], or loop forever. The solve, find and findAll methods all take a parameter describing the timeout strategy. That parameter is optional, so by default no timeout is used. Because it is also implicit, a timeout strategy can be defined for an entire scope by a single definition that the Scala compiler then automatically inserts at each call site:

```
implicit val timeoutStrategy = Timeout(1.0)
```

Given the above declarations, the following attempt to confirm Fermat's last theorem results in an exception after 1 second:

```
scala> fermat.find
result: TimeoutReachedException: No solution after 1.0 second(s)
  at .solve(<console>)
  at .<init>(<console>)
  ...
```

### 2.5 Logical Variables

All the examples so far have illustrated *eager* solution enumeration, where solving constraints immediately produces concrete values. While this by itself is a convenient facility, much of the power of constraint programming in general and of Kaplan in our case comes from the ability to produce logical variables, which represent the *promise* of a solution. Logical variables in turn can be used to control the execution flow of the program in novel ways. We

start by describing some of their basic properties. More advanced examples of programming with logical variables follow.

In Kaplan, logical variables are always produced as the result of *lazily* solving a constraint. This is done by calling lazySolve, lazyFind or lazyFindAll instead of solve, find or findAll respectively. Consider the constraint we defined earlier:

```
val c1 =
  ((x: Int, y: Int) ⇒ 2∗x + 3∗y == 10 && x ≥ 0 && y ≥ 0).c
```

We produce logical variables representing a solution as follows:

```
scala> val (x,y) = c1.lazySolve; println((x,y))
result: (L(?),L(?))
```

Notice that the result is not a pair of integers, as in the case of solve, but a pair of objects of type L[Int] representing the promise of integers. The question mark indicates that the value has not yet been fixed.

Logical variables in Kaplan have *singular* semantics [54], meaning that a given logical variable will always represent the same concrete value, even when it is copied or passed as an argument to a function. The identity of a logical variable is determined by the identity of the instance of the L[_] class. The value of a logical variable is fixed as soon as it is queried, which can be done with the .value method:

```
scala> x.value
result: 5
scala> println((x,y))
result: (L(5),L(?))
```

(Note that even though the y can at this point only hold the value 0, the solver has not necessarily detected that the solution is unique and thus still considers the value not to be fixed.) Fixing a value needs not always be done explicitly. The Kaplan library defines two implicit conversions between concrete and logical values:

```
implicit def concretize[T](l : L[T]) : T = l.value
implicit def lift[T](v : T) : L[T] = new FixedL[T](v)
```

The first one simply fixes the value whenever a function expects a concrete value and receives a logical variable. The second one converts a concrete value into a specialized representation of a logical variable. It is used to provide a common representation for concrete and logical values: because Kaplan is built as a thin layer over Scala and most code must execute as usual, we cannot afford to treat every value as logical. Instead, users decide which function can handle logical values by using L[_] types in their signature. Because concrete values can be lifted to logical status, such functions work indifferently with standard and logical variables. Logical variables created through this lifting mechanism are treated specially and do not add any complexity to the solving process.

### 2.6 Imperative Constraint Programming

Scala being a multi-paradigm language, users can alternate between different programming styles depending on their preferences and on the task at hand. The same is true for Kaplan; while eager solution enumeration is best used within functional style for-comprehensions, logical variables can be used rather naturally in an imperative style, thanks to novel control constructs. The Kaplan library defines the **assuming-otherwise** branching construct. It is similar in nature to if-then-else, except for an important difference: rather than strictly evaluating the branching condition, assuming-otherwise blocks check whether the condition is *feasible* and, if so, constrain the logical variables in the environment so that their values satisfy the branching constraint. If the condition contains no logical variable (or only logical variables that have already been fixed), then assuming-otherwise behaves exactly like if-then-else.

As an example, consider the classical puzzle that consists in finding distinct values for the letters representing digits in the following addition such that the sum is valid:

$$\begin{array}{cccccc} & & s & e & n & d \\ + & & m & o & r & e \\ \hline = & m & o & n & e & y \end{array}$$

We now present a solution in Kaplan written in an imperative style.

```
val anyInt = ((x : Int) ⇒ true).c
val letters @ Seq(s,e,n,d,m,o,r,y) = Seq.fill(8)(anyInt.lazySolve)
```

This second line uses pattern-matching syntax to achieve two things at once: 1) the eight letter variables are bound to eight independent (lazy) representations of a solution to the trivial anyInt constraint and 2) the variable letters is bound to the sequence of all letter variables. At this point, we have 8 unconstrained integer logical variables. We can imperatively add constraints:

```
for(l ← letters) asserting(l ≥ 0 && l ≤ 9)
```

If this loop terminates without any error, then at the end of it, the 8 variables each represent a number between 0 and 9. (Calling **asserting**(cond) is equivalent to **assuming**(cond) {} **otherwise** { **error** }, just like **assert**(cond) is (conceptually at least) equivalent to **if**(cond) {} **else** { **error** }. More details are given in Section 5.1.) We further constrain the letters representing most-significant digits:

```
asserting(s > 0 && m > 0)
```

We can now perform symbolic arithmetic using the existing and new logical variables. We define a new variable for the sum of each line and constrain it to the expected value:

```
val fstLine = anyInt.lazySolve
asserting(fstLine == 1000∗s + 100∗e + 10∗n + d)
val sndLine = anyInt.lazySolve
asserting(sndLine == 1000∗m + 100∗o + 10∗r + e)
val total = anyInt.lazySolve
asserting(total == 10000∗m + 1000∗o + 100∗n + 10∗e + y)
```

At this point, the value of the letters is still not fixed, however Kaplan ensures that all variables admit solutions satisfying the asserted constraints. Finally, we check for a solution to the puzzle using one last assuming-otherwise block:

```
scala> assuming(
  distinct(s,e,n,d,m,o,r,y) && fstLine + sndLine == total) {
    println("Solution: " + letters.map(_.value))
  } otherwise {
    println("The puzzle has no solution.")
  }
result: Solution: List(9, 5, 6, 7, 1, 0, 8, 2)
```

We mentioned that assuming-otherwise is conceptually close to if-then-else, and in fact equivalent in the absence of logical variables. One important difference is that the construct is asymmetrical; while

```
if(cond) thenExpr else elseExpr
```

is equivalent to

```
if(!cond) elseExpr else thenExpr
```

the same transformation cannot be applied to assuming-otherwise blocks; indeed, the semantics are that the control will attempt to satisfy the branching condition, so whether the positive or negative condition is tested has an impact on the rest of the execution. Section 5.1 discusses the implementation of assuming-otherwise blocks in terms of lazyFind.

$$\textsf{Int} \in \mathcal{T} \qquad \textsf{Boolean} \in \mathcal{T} \qquad \frac{T \in \mathcal{T}}{\textsf{Set}[T] \in \mathcal{T}}$$

$$\frac{T_1 \in \mathcal{T} \quad T_2 \in \mathcal{T}}{\textsf{Map}[T_1,T_2] \in \mathcal{T}} \qquad \frac{T_1 \in \mathcal{T} \quad T_2 \in \mathcal{T}}{T_1 \Rightarrow T_2 \in \mathcal{T}}$$

$$\frac{\textsf{sealed abstract class } C}{C \in \mathcal{T}}$$

$$\frac{\textsf{case class } C(\overline{n} : \overline{D}) \textsf{ extends } E \qquad \overline{D} \in \mathcal{T} \qquad E \in \mathcal{T}}{C \in \mathcal{T}}$$

**Figure 1.** Inductive definition of PureScala types, where Set and Map are the types defined in the package `scala.collection.immutable`.

## 3. Constraint Sublanguage

In this section, we describe the subset of Scala in which we can specify constraints in Kaplan, and which we call PureScala. PureScala is an extension of the language supported by the publicly available Leon verification system [60]. Since it is a subset of Scala, the language is executable and deterministic.

### 3.1 Language

**Data types.** Figure 1 presents an inductive definition of the data types supported in PureScala. An important feature is the ability to define (recursive) algebraic data types. In Scala, these types are defined using a hierarchy of special case classes. Algebraic data types are typically manipulated with pattern-matching, as shown in some of the examples in Section 2. A current limitation is that these user-defined types cannot take type parameters. The Map type represents finite, immutable maps, which can thus be viewed as partial functions. Because it supports unbounded data types and arbitrary recursive functions, the constraint language is itself Turing-complete. This can be viewed both as an advantage and an inconvenience: on one hand, this expressive power guarantees that just about any constraint is expressible, but on the other hand, standard incompleteness theorems predict that some constraints cannot be shown to have no solutions. In practice, we have found that constraints that come up in programming tasks such as the ones presented in this paper and in the context of functional software verification [60] (as opposed to theoretically hard ones) are handled well.

**Expressions and function definitions.** PureScala expressions can contain all standard arithmetic operators, map applications and updates, set operators and membership tests, function applications (of user-defined or anonymous functions), and constructor and selectors from user-defined data types. Expression can also contain **val**s to factor out common subexpressions. Indeed, a Scala block

```
{ val x1 = e2
  ...
  val xn = en
  e }
```

is to be understood as **let** x1 = e2 **in** ... **in let** xn = en **in** e. Expressions can contain pattern-matching on user-defined data types. Any sub-pattern can be bound to a variable and used accordingly on the right-hand side of patterns. Furthermore, patterns can contain arbitrary guards. At compile-time, Kaplan relies on Leon to prove that pattern-matching expressions are *exhaustive*, and thus rules out any possibility of a run-time match error in @spec functions. This exhaustiveness check goes beyond the capabilities of the standard Scala compiler, in that it takes into account the path conditions leading to the match expression.

A PureScala function body is defined by a single expression whose free variables are the arguments of the function. Functions can optionally be annotated with a post-condition, which can in some cases help the run-time constraint solver. These post-conditions are proved at compile-time, just like the pattern-matching expressions. Kaplan can thus be used purely as a verification system, and therefore strictly subsumes Leon.

### 3.2 Solver

Kaplan invokes Leon's core solving procedure both at compile-time, to validate post-conditions and prove that pattern-matching expressions are exhaustive, and at run-time, to find solutions to constraints. The procedure is based on a refinement loop that expands function definitions in a fair way, to guarantee that no valid solution is ever ignored. One can think of this search procedure as a form of bounded model-checking for functional programs. We have found that in practice it is fast in finding counter-examples. The details of the procedure are presented in [60].

Conceptually, Leon is the only solver used in the implementation of Kaplan. Because it is built as a layer on top of Z3, though, we will sometimes refer to Z3 directly when we discuss implementation details in Section 5. In particular, in the absence of user-defined recursive functions, Leon behaves exactly as Z3.

## 4. Semantics

In this section, we present an overview of the semantic aspects of Kaplan through operational semantic rules, shown in Figure 2. All features of Kaplan can be implemented in terms of the two constructs find and lazyFind, as well as conversions between concrete values and logical variables (denoted by $l$). Section 5 describes how to use the host language (Scala) to reduce other constructs (including solve, findAll, lazyFindAll) to this core.

A state consists of a triple $\textsf{expr} | \langle \mu, \kappa \rangle$, where expr is the expression under evaluation, $\mu$ encodes the part of the state that is proper to Scala, and $\kappa$ is a *constraint store*. A constraint store is conceptually a formula whose free variables correspond to all logical variables used since the beginning of the computation.

The HOST rule captures the intuition that in the absence of invocations to the constraint solver, Kaplan behaves exactly like Scala: we assume the existence of a transition relation $\rightarrow_H$ describing the execution of normal Scala code, and which is lifted to Kaplan through the HOST rule. As one would expect, applying such transitions leaves the constraint store unchanged.

The rules S-SAT and S-UNSAT describe the possible results of an invocation of find, the simplest form of constraint solving. We use $(\lambda \overline{x}.\phi(\overline{x}))$ to denote a constraint that ranges over the variables $\overline{x}$ and that does not refer to logical variables. By $\mathcal{M}$ we denote a map from variables to constants. The condition $\mathcal{M} \models \phi$ denotes that $\mathcal{M}$ is a valid model of $\phi$, i.e., a mapping of variables $\overline{x}$ of $\phi$ to constants such that $\phi[\overline{x} \mapsto \mathcal{M}(\overline{x})]$ holds. Examining the rules for find, S-SAT and S-UNSAT, we note that find has no impact and no dependency on the constraint store, which is consistent with its eager semantics.

The rules L-SAT and L-UNSAT for lazyFind are analogous, but with crucial differences: 1) logical variables can be present in the constraints to solve, which we therefore denote $(\lambda \overline{x}.\phi(\overline{x}, \overline{l}))$, and 2) the L-SAT rule does not produce concrete values, but rather fresh logical variables.

The LIFT rule applies whenever a constant value needs to be used in place of a logical variable. Conceptually, it adds to the constraint store a new logical variable whose value is immediately constrained to be the constant. Finally, the VALUE rule specifies how concrete values can be extracted from a satisfying assignment

$$\textsc{Host}\ \frac{t_1 \mid \mu_1 \ \rightarrow_H \ t_2 \mid \mu_2}{t_1 \mid \langle \mu_1, \kappa \rangle \ \rightarrow \ t_2 \mid \langle \mu_2, \kappa \rangle}$$

$$\textsc{S-Sat}\ \frac{\mathcal{M} \models \phi}{(\lambda \overline{x}.\phi(\overline{x})).\mathsf{find} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{Some}(\mathcal{M}(\overline{x})) \mid \langle \mu, \kappa \rangle} \qquad \textsc{S-Unsat}\ \frac{\neg \exists \mathcal{M} \,.\, \mathcal{M} \models \phi}{(\lambda \overline{x}.\phi(\overline{x})).\mathsf{find} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{None} \mid \langle \mu, \kappa \rangle}$$

$$\textsc{L-Sat}\ \frac{\mathcal{M} \models \kappa \wedge \phi[\overline{x} \mapsto \overline{l}_f] \qquad \overline{l}_f \text{ fresh in } \kappa}{(\lambda \overline{x}.\phi(\overline{x}, \overline{l})).\mathsf{lazyFind} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{Some}(\overline{l}_f) \mid \langle \mu, \kappa \wedge \phi[\overline{x} \mapsto \overline{l}_f] \rangle}$$

$$\textsc{L-Unsat}\ \frac{\neg \exists \mathcal{M} \,.\, \mathcal{M} \models \kappa \wedge \phi[\overline{x} \mapsto \overline{l}_f] \qquad \overline{l}_f \text{ fresh in } \kappa}{(\lambda \overline{x}.\phi(\overline{x}, \overline{l})).\mathsf{lazyFind} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{None} \mid \langle \mu, \kappa \rangle}$$

$$\textsc{Value}\ \frac{\mathcal{M} \models \kappa}{l.\mathsf{value} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathcal{M}(l) \mid \langle \mu, \kappa \wedge (l = \mathcal{M}(l)) \rangle} \qquad \textsc{Lift}\ \frac{c \text{ is a constant} \qquad l_f \text{ fresh in } \kappa}{c.\mathsf{lift} \mid \langle \mu, \kappa \rangle \ \rightarrow \ l_f \mid \langle \mu, \kappa \wedge (l_f = c) \rangle}$$

**Figure 2.** Small-step semantics of Kaplan-specific constructs.

(model) for the constraint store. Observe that when the rule applies, the value of the logical variable is then fixed for the rest of the program execution by the added equality to $\kappa$. This ensures singular semantics [54]: in any execution trace, all applications of the VALUE rule for a given logical variable $l$ will produce the same value; a different value would contradict the premise that $\mathcal{M}$ is a valid model for the store.

We now show that execution never gets stuck in one of the Kaplan-specific rules. Because we are examining the behavior of our constructs taking a constraint solver as a parameter, we deliberately ignore termination properties of the constraint solver and assume that the outcome of the constraint solver call, denoted $\models$, becomes immediately available to test the applicability of a rule.

**Theorem.** *Suppose the constraint store is satisfiable in the initial state and that the $\rightarrow_H$ relation is total. Then the transition relation given by Figure 2 is also total.*

*Proof.* The case of HOST is clear from the hypothesis that $\rightarrow_H$ is total. For evaluations of find, we observe that the rules S-SAT and S-UNSAT have complementary premises, and therefore one of them necessarily applies. Similarly for lazyFind, L-SAT and L-UNSAT are complementary, regardless of the value of $\kappa$. LIFT has no precondition, so it remains to show that VALUE cannot get stuck, i.e. that it cannot be the case that there is no model $\mathcal{M}$ for the constraint store $\kappa$.

Using the assumption that in the initial state the constraint store is satisfiable, it is sufficient to show that no transition will make it unsatisfiable. Observe that only three rules affect the constraint store; VALUE, LIFT, and L-SAT. The addition of the equality $(l_f = c)$ in LIFT clearly does not affect satisfiability, because $l_f$ is fresh. Similarly, the addition of $(l = \mathcal{M}(l))$ in VALUE preserves the model $\mathcal{M}$ by definition. Finally, L-SAT is guarded by a satisfiability check on precisely the next state of the store, so the model obtained in the premise is a valid model for the next state. ■

The proof suggests an implementation strategy: preserve at all times, alongside the constraint store, a satisfying assignment. When L-SAT is applied, cache the model $\mathcal{M}$ obtained from the satisfiability check. When LIFT applies, augment the cached model with the appropriate value for $l_f$. Finally, to apply VALUE, use $\mathcal{M}$ from the cache.

It is not hard to see that this strategy is a valid refinement of the presented rules, and it is, in fact, the strategy we have implemented in Kaplan, as explained in the next section.

## 5. Implementation

We implemented our extension to Scala as a combination of a run-time library and a compiler plugin, both implemented in Scala. In this section, we present the implementation aspect of these parts, along with the interaction with the underlying SMT solver Z3.

### 5.1 Run-Time Library

**First-class constraints.** In Kaplan, first-class constraints are implemented as a hierarchy of Term classes, as shown in Figure 3. The base Term class represents a lambda expression and is parameterized by its argument types and return type. A constraint is simply a Term instance where the return type is instantiated as boolean. We define subclasses of Term for each arity, generating them automatically, as it is the case for the tuple and function definitions in the Scala library. The base class defines methods common to terms of all arities, such as the solve, find and findAll methods for querying constraints. We ensure that these methods are only applicable when the return type is boolean by constraining it using an implicit parameter asBool. We use the same technique to guarantee that term instances are combined in a fully type safe way. We use the c method to trigger an implicit conversion from lambda expressions to Term instances.

Each term subclass extends the corresponding function class in Scala, and uses the original Scala code for the lambda expression to define function application. These classes define optimization methods (minimizing and maximizing) to obtain optimization constraints. The implementation of the optimization procedures is discussed later in this section.

The creation of terms from Scala lambda expressions relies on compile-time transformations. The transformations are implemented in a plugin for the official Scala compiler. We describe how the compile-time transformations work in Section 5.2.

**Logical variables.** In Kaplan, logical variables are instances of the L class, which is parameterized by the type of the symbolic value that it encapsulates. We define LIterator classes that extend the Iterator trait of Scala and that enumerate tuples of L values. We discuss the implementation of logical variables in Section 5.3.

**The assuming-otherwise construct.** We can define the assuming-otherwise construct naturally at the library level in Kaplan: it boils down to checking the satisfiability of a constraint of arity 0, and can therefore be implemented in terms of lazyFind. Figure 4 shows the code for this construct in Kaplan. The assuming block creates an Assuming instance. We rely on the implicit conversion mechanism

```
abstract class Term[T,R] { self ⇒
  def find(implicit isBool: R =:= Boolean): Option[T] = ...
  def solve(implicit isBool: R =:= Boolean): T =
    this.find.getOrElse(throw new UnsatException)
  def findAll(implicit isBool: R =:= Boolean) : Iterator[T] = ...
  def c(implicit isBool: R =:= Boolean): self.type = this
  ...
}

class Term0[R] extends Term[Unit,R] with Function0[R] {
  ...
}
class Term1[T1,R] extends Term[T1,R] with Function1[T1,R] {
  ...
}
class Term2[T1,T2,R] extends Term[(T1,T2),R]
    with Function2[T1,T2,R] {
  def ||(other: Term2[T1,T2,Boolean])
    (implicit isBool: R =:= Boolean): Term2[T1,T2,Boolean] = ...
  def &&(other: Term2[T1,T2,Boolean])
    (implicit isBool: R =:= Boolean): Term2[T1,T2,Boolean] = ...
  def unary_!(implicit isBool: R =:= Boolean):
    Term2[T1,T2,Boolean] = ...

  def compose0[A1](other: Term1[A1,T1]): Term2[A1,T2,R] = ...
  def compose1[A1](other: Term1[A1,T2]): Term2[T1,A1,R] = ...
  def compose0[A1,A2](other: Term2[A1,A2,T1]):
    Term3[A1,A2,T2,R] = ...
  def compose1[A1,A2](other: Term2[A1,A2,T2]):
    Term3[T1,A1,A2,R] = ...
  ...
  def product1[A1](other: Term1[A1,Boolean])
    (implicit isBool: R =:= Boolean):
      Term3[T1,T2,A1,Boolean] = ...
  def product2[A1,A2](other: Term2[A1,A2,Boolean])
    (implicit isBool: R =:= Boolean):
      Term4[T1,T2,A1,A2,Boolean] = ...

  def minimizing(objective: Term2[T1,T2,Int])
    (implicit asBool : R =:= Boolean):
      MinConstraint2[T1,T2] = ...
  ...
}
...
type Constraint[T] = Term[T,Boolean]
type Constraint0 = Term0[Boolean]
type Constraint1[T1] = Term1[T1,Boolean]
...
```

**Figure 3.** Term class hierarchy.

of Scala for implementing the construct: if the optional otherwise block is not defined, the type checking phase will insert a call to assuming2value, which will trigger a conversion from the Assuming instance to the value it encapsulates.

There exists a difference in the treatment of the optional second part of the built-in if-then-else construct and our library extension for assuming-otherwise: without an optional else block, an if(...) { ... } block is always type-checked as Unit. This is built in in the Scala compiler. We cannot achieve the same effect with assuming-otherwise without making deep changes to the compiler, so in our case, an assuming(...) { ... } block will throw an exception at runtime if the following three conditions occur: 1) the assuming test fails, 2) no otherwise block is defined, and 3) an expression of a type different from Unit is required.

### 5.2 Scala Compiler Plugin

The compile-time transformations of Kaplan programs include the following:

```
def assuming[A](cond: Constraint0)(block: ⇒ A): Assuming[A] = {
  val v: Option[A] = cond.lazyFind match {
    case Some(_) ⇒ Some(block)
    case None ⇒ None
  }
  new Assuming(v)
}

final class Assuming[A](val thenResult: Option[A]) {
  def otherwise(elseBlock: ⇒ A): A = thenResult match {
    case None ⇒ elseBlock
    case Some(tr) ⇒ tr
  }
}

implicit def assuming2value[A](a: Assuming[A]): A = {
  a.thenResult match {
    case Some(tr) ⇒ tr
    case None ⇒
      throw new Exception("otherwise block not defined")
  }
}
```

**Figure 4.** Implementation of assuming-otherwise in terms of lazyFind.

1. extracting user-defined specification functions and algebraic data types;
2. generating methods to allow conversion between values of these data types and their representation in our solver Leon;
3. transforming implicit calls to conversion methods in order to instantiate Term instances.

These transformations are implemented as a compiler plugin that constitutes a compiler phase that follows the type checking phase. Functions and classes that carry the @spec annotation are expected to be in the PureScala constraint sublanguage, presented in Section 3. Alternatively, developers can group these specification functions and data types in annotated Scala objects, instead of annotating each of them. These specifications are extracted during compilation to obtain a representation that we use in solving. In addition, method definitions are generated and inserted into the code in order to convert between these types and their representation.

We rely on the Scala compiler to signal to us the locations where a function literal needs to be lifted to a constraint literal. The type checking phase, which runs before ours, does so by surrounding such function literals by a call to an implicit function2term conversion function. These functions are defined in the Kaplan library but have no implementation: they simply serve as a guide to indicate to the type checker that the conversion is legal. (The effect of compiling code written for Kaplan without the Kaplan plugin is thus that all constraint manipulation operations result in a run-time NotImplemented exception.)

### 5.3 Implementation of the Core Solving Algorithms

Our implementation leverages the SMT solver Z3 through its extension, Leon [60]. In the following we refer simply to Z3, as the algorithms we cover in this section would remain the same if we used Z3 alone; what we gain by using Leon is the additional expressive power of recursive functions within constraints. We now describe the interactions with the solver that allow us to put into practice features such as enumeration, minimization, and logical variables.

**Solution enumeration.** find and lazyFind are used to implement findAll and lazyFindAll, respectively, through the use of an iterator.

```
def solveMinimizing(φ, t_m) {
  solve(φ) match {
    case ("SAT", m) ⇒
      model = m
      v = modelValue(m, t_m)
      pivot = v − 1
      lo = null
      hi = v + 1
      while (lo == null ∨ hi − lo > 2) {
        solve(φ ∧ t_m ≤ pivot) match {
          case ("SAT", m) ⇒
            model = m
            if (lo == null) {
              pivot = pivot ≥ 0 ? −1 : pivot × 2
              hi = pivot + 1
            } else {
              l_v = modelValue(m, t_m)
              pivot = l_v + (pivot + 1 − lv)/2
              hi = pivot + 1
            }
          case ("UNSAT", _) ⇒
            pivot = pivot + (hi − pivot)/2
            lo = pivot
        }
      }
      return ("SAT", model)
    case ("UNSAT", _) ⇒ return ("UNSAT", null)
  }
}
```

**Figure 5.** Pseudo-code of the solving algorithm with minimization. We invoke our base satisfiability procedure via calls to solve.

The iterator maintains a constraint at all times, starting with the original one. Each time a new solution is required, the iterator destructively updates the constraint by adding to it the negation of the previous solution, thus ensuring that all following solutions will be different. To make this process efficient, Kaplan relies on the incremental reasoning capabilities of Z3 (and thus Leon) to avoid solving the entire constraint each time. The implementation of lazyFindAll is conceptually identical, with the difference that it returns logical variables instead of concrete values. These logical variables are constrained in the store to be all-different. For an enumeration using lazyFindAll to terminate, Leon must therefore prove that the constraint has finitely many solutions.

**Optimization constraints.** Our procedure for optimizing a constraint with respect to an objective function can be seen as a generalization of binary search over the range of values that the objective can take. Let us consider the case of minimization (the maximization procedure is analogous). The pseudo-code for the algorithm can be seen in Figure 5.[2] It starts by attempting to find a satisfying assignment for the constraint. It then repeatedly looks for a model in which the objective is smaller than the last satisfying value, by exponentially increasing the difference until a lower bound is found. It further reduces the interval until the optimal value for the objective is found. The procedure maintains the invariants that: 1) $lo$ is always less than any satisfying assignment to $t_m$; and 2) there always exists a satisfying assignment to $t_m$ which is less than $hi$.

**Ordered enumeration.** Having defined the solving procedure that minimizes a given term, we can now compose it with solution enumeration to obtain ordered enumeration. We present in Figure 6,

[2] We use pseudo-code for clarity, but it is not hard to see that it can be implemented using only standard Scala along with find or lazyFind.

```
def orderedEnum(φ, t_m) {
  solveMinimizing(φ) match {
    case ("SAT", m) ⇒
      v_m = modelValue(m, t_m)
      findAll(φ ∧ t_m = v_m) ++ orderedEnum(φ ∧ t_m > v_m, t_m)
    case ("UNSAT", _) ⇒ return Iterator.empty
  }
}
```

**Figure 6.** Pseudo-code of the ordered enumeration algorithm.

a recursive algorithm that will enumerate solutions to $\phi$, ordered by the value of $t_m$, which should be minimized.

In this pseudo-code, we use findAll to get an iterator of all values satisfying the given predicate, and $++$ to concatenate iterators.

**Handling logical variables.** We use a global context to keep track of the constraints associated with logical variables. Given a logical variable $l$ and the constraint $c_l$ associated with it, we create a guard (a boolean literal) $g_l$, denoting the *liveness* of the logical variable, i.e. whether its value has not been fixed yet. Throughout the execution, we maintain in the global context the set of guards that are still alive. Upon creation of the variable $l$, we add $g_l$ to the set of alive variables and we assert $c_l$, guarded by the disjunction of all the guards in the set $G$, defined as the set containing $g_l$ and the guards associated with all the logical variables that appear in $c_l$:

$$\bigvee_{g \in G} g \Rightarrow c_l$$

The reason for considering the guards associated with the other logical variables is to ensure the single value semantics of these. Consider the following simple example that illustrates the situation:

```
for (x ← ((x: Int) ⇒ x ≥ 0 && x < 4).lazyFindAll) {
  val y = ((y : Int) ⇒ y == x).lazyFind
  assuming (x ≥ 2) {
    ... // first block
  }
  assuming (y < 2) {
    ... // second block
  }
}
```

In each iteration of the outer for-comprehension, if the first block is entered, we do not want to enter the second one, as this would violate the single value semantics for the variable x. Since the constraint y == x will be enforced as long as either x or y has not been fixed, the conflicting situation is correctly avoided.

In the terminology of Section 4, $g_l$ denotes whether the VALUE rule has been used for $l$ (true means it has not), $c_l$ is the constraint on which L-SAT was applied to introduce $l$ into $\kappa$ (as part of $\bar{l}_f$) and the logical variables represented by the set of guards $G$ are those that contributed to the constraint (the $\bar{l}$ variables in L-SAT). Note that using individual guards for logical variables is not required by the semantics, but rather is an optimization that allows us to reduce the size of the constraint store when some values become known. When the value $v_l$ of the variable $l$ is set, we remove $g_l$ from the set of alive guards, and we assert the following:

$$\neg g_l \wedge l = v_l$$

When all of the literals guarding a constraint $c_l$ are removed from the alive set, the formula $\bigvee_{g \in G} g \Rightarrow c_l$ that was asserted becomes trivially true, and the constraint can be discarded by Z3. Another source of optimization is that we override the finalize method of the L instances such that, even if their value is never fixed, their guard is removed from the set of alive variables when they are being considered for garbage collection by the JVM. This

| example | time (s) |
|---|---|
| first examples | 0.16 |
| unimodular matrices | 0.76 |
| sat solving | 0.05 |
| knapsack | 0.18 |
| prime numbers | 0.80 |
| all red-black trees up to 7 nodes | 27.45 |
| send+more=money | 1.17 |

**Figure 7.** Evaluation results for the examples presented in Section 2.

is again not strictly speaking required by the semantics, but helps reducing the overhead of tracking all logical variables.

**Invocations of the solver.** We summarize this section by presenting a list of all the places where an invocation of the solver occurs:

- calls to find, as they search for a solution eagerly,
- calls to lazyFind, as they check whether a satisfying assignment exists for logical variables,
- calls to the hasNext method of the iterators returned by calls to findAll and lazyFindAll, which are then translated into calls to find and lazyFind respectively, and
- evaluation of the constraint of assuming blocks, as they indirectly invoke the solver by invoking lazyFind on the constraint.

Note that, as the proof in Section 4 suggested, calls to the value method of logical variables do not trigger a solver invocation.

## 6. Advanced Usage Scenarios and Evaluation

In this section, we present an experimental evaluation of our constraint programming system by considering a number of examples.

As a first overview of the performance of our implementation, we present the running times for the examples that were introduced in Section 2. The results of our evaluation can be seen in Figure 7. We observe that most problems are solved almost instantly, with the exception of the red-black tree enumeration. We discuss the difficulty of enumerating data structures satisfying an invariant in the subsequent examples.

### 6.1 Enumerating Data Structures for Testing

One use of the findAll construct consists in enumerating data structures that satisfy given invariants. This is a problem that has been studied previously, and was motivated by [11]. Subsequent work [24] presents a Java-based language with non-deterministic choice operators that can be used for enumerating linked data structures.

We describe our experience in using Kaplan to enumerate functional data structures to find input values that violate function contracts. We consider a functional specification of red-black trees. We enumerate solutions to function preconditions and check whether the postconditions hold. As in [11, 24], we enumerate the data structures while bounding the range of values than can be stored in nodes. A red-black tree is a binary search tree characterized by the following additional properties: 1) each node is either red or black; 2) all leaves are black; 3) both children of every red node are black; and 4) every simple path from the root to any leaf contains the same number of black nodes.

The first method we consider is balance, which defines one of the cases erroneously to duplicate one of the subtrees and forgetting another while rebalancing the tree. This results in violating the post-condition of the add method as the result tree does not have the expected content. In this case, an example violating the post-condition is found after enumerating twelve trees. Our test harness consists of enumerating trees that satisfy the precondition of add and calling the method in the body of the loop.

| list size | 20 | 40 | 60 | 80 | 100 |
|---|---|---|---|---|---|
| time (s) | 0.24 | 0.45 | 0.56 | 0.72 | 0.98 |

**Figure 8.** Evaluation results for declarative last method.

We then consider the case where the add method has a missing precondition, namely that the tree must be black-balanced. In this case, the precondition to a method that is called within add fails, and we find a bug-producing value using a similar harness after enumerating four trees, in 0.187 seconds. We argue that a random test-case generation approach would be insufficient in enumerating such data structures that satisfy complex invariants. While the results using constraint solving is not as fast as the specialized solving in UDITA [24], we should keep in mind that this is an experiment in using a general-purpose constraint solving engine. The generality of Kaplan is in contrast to previously proposed solutions for data structure enumeration, which rely on specialized techniques for linked heap data structures, or even techniques specialized to a particular data structure [7]. In the light of the generality of our technique and the overall difficulty of the problem, we consider these to be good results.

### 6.2 Executable Specifications

As another application of implicit computation, we now explore examples that consist of the "execution" of declarative specifications instead of explicit computation. Execution of specifications is the approach taken in Plan B [55], in which specifications are used as a fallback mechanism upon contract violations.

Consider, for instance, a function that computes the last element of a given list. We can define this function declaratively, by stating that the input list is equal to some list concatenated with the list that has only the element that we are looking for:

```
def last(list : List) : Int = {
  val (elem, _) = ((e: Int, zs: List) ⇒
                     concat(zs, Cons(e, Nil())) == list).solve
  elem }
```

As a more elaborate example, consider adding an element to (or removing an element from) a red-black tree. The explicit insertion and removal have to consider multiple cases in order to keep the invariants that red-black trees should satisfy, and are known to be tricky to implement. On the other hand, these methods can be stated succinctly in a declarative manner, using functions that check if a given tree is indeed a red-black tree, along with functions computing the content of the tree as a set:

```
def addDeclarative(x: Int, tree: Tree) : Tree =
  ((t: Tree) ⇒ isRedBlackTree(t) &&
    content(t) == content(tree) ++ Set(x)).solve
def removeDeclarative(x: Int, tree: Tree) : Tree =
  ((t: Tree) ⇒ isRedBlackTree(t) &&
    content(t) == content(tree) -- Set(x)).solve
```

The performance of the above methods is presented in Figure 9. We also show the results for the similar case of inserting into and removing from a sorted list. Note that we encounter essentially the same running times for the problem of declaratively sorting a list, by asking for a sorted list of the same content as a given list.

The above examples show that it is possible to replace the explicit computation for a method by its purely declarative specification, even for sophisticated contracts such as the ones on red-black trees. The declarative variants are notably slower than the imperative implementations, but it is likely preferable to rely on these executable specifications instead of simply crashing when the imperative version violates the contract.

| size | list add | list remove | RBT add | RBT remove |
|---|---|---|---|---|
| 0 | 0.07 | 0.02 | 0.03 | 0.02 |
| 1 | 0.08 | 0.02 | 0.10 | 0.05 |
| 2 | 0.12 | 0.05 | 0.14 | 0.09 |
| 3 | 0.16 | 0.10 | 0.55 | 0.41 |
| 4 | 0.24 | 0.18 | 0.66 | 0.76 |
| 5 | 0.39 | 0.38 | 1.07 | 0.91 |
| 6 | 0.55 | 0.45 | 1.51 | 1.56 |
| 7 | 0.97 | 0.67 | 9.32 | 13.09 |
| 8 | 1.48 | 1.09 | 11.13 | 18.80 |
| 9 | 2.27 | 1.80 | 24.49 | 25.79 |
| 10 | 3.32 | 2.22 | 11.51 | 20.55 |

**Figure 9.** Evaluation results for declarative add and remove, for red-black trees and for sorted lists. "size" is the size of the structure without the element. All times are in seconds.

As a final example, let us consider the implicit computation that gives the integer square root $\lfloor\sqrt{i}\rfloor$ of a positive integer $i$. This is concisely stated as following:

```
def sqrt(i : Int) : Int =
  ((res: Int) ⇒ res > 0 &&
                res * res ≤ i &&
                (res + 1) * (res + 1) > i).solve
```

Our implementation can handle numbers as large as hundreds of thousands, performing under 0.3 seconds in all cases.

### 6.3 Counter-example Guided Inductive Synthesis

Counter-example guided inductive synthesis (CEGIS) is a method for effectively solving $\exists\forall$ constraints using SMT solvers. Such constraints are particularly important for program synthesis, identified as the class of $\forall\exists$ synthesis problems in [53]. The counter-example guided approach for this problem was successfully applied to software synthesis in the algorithm for combinatorial sketching using SAT solvers [58]. This technique was later applied with an SMT solver as the satisfiability engine to synthesize loop-free bit-vector code fragments [26].

The technique starts by choosing some initial set of values for the universally quantified variables, then solving the constraint for the existentially quantified variables. If the values for the existentially quantified variables work for all values of the universally quantified variables, a solution has been found. Otherwise, there is a counterexample which is some valuation of the universally quantified variables. This counterexample is added to the set of values for universally quantified variables and the procedure is repeated until a solution is found.

This synthesis loop can be expressed in our system succinctly, using first-class constraints. As an example we consider the following: *Are there integers $a$ and $b$ such that, for every integer $x$, $a \cdot (x-1) < b \cdot x$ ?* In Kaplan we can describe the CEGIS approach as in Figure 10. When executed, the program finds correct values for $a$ and $b$ in two iterations of the loop. The output of the program is:

```
Initial x: 0
candidate parameters a = 1, b = 0
counterexample for x: 1
candidate parameters a = 1, b = 1
proved!
```

The example generalizes to arbitrary constraints (which, in this example, are cnstrGivenX and cnstrGivenParams), keeping the same simple structure. Note that we here explicitly constructed increasingly stronger first-class constraints; we can instead use logical variables and incrementally augment the constraint store.

```
var continue = true
val initialX = ((x: Int) ⇒ true).solve
def cnstrGivenX(x0: Int): Constraint2[Int,Int] =
  ((a: Int, b: Int) ⇒ a * (x0 − 1) < b * x0)
def cnstrGivenParams(a0: Int, b0: Int): Constraint1[Int] =
  ((x: Int) ⇒ a0 * (x − 1) < b0 * x)
var currentCnstr = cnstrGivenX(initialX)
while (continue) {
  currentCnstr.find match {
    case Some((a, b)) ⇒ {
      println("candidate parameters a = " + a + ", b = " + b)
      (! cnstrGivenParams(a, b)).find match {
        case None ⇒
          println("proved!")
          continue = false
        case Some(ce) ⇒
          println("counterexample for x: " + ce)
          currentCnstr = currentCnstr && cnstrGivenX(ce)
      }
    }
    case None ⇒
      println("cannot prove property!")
      continue = false
  }
}
```

**Figure 10.** Counter-example guided inductive synthesis in Kaplan

| list size | time in Kaplan (s) | time in Curry (s) |
|---|---|---|
| 10000 | $< 0.01$ | 0.21 |
| 100000 | $< 0.01$ | 2.41 |
| 1000000 | $< 0.01$ | 23.88 |

**Figure 11.** Evaluation results for functional last method.

### 6.4 Comparison to Other Systems

Providing a fair comparison of running times of Kaplan against competing systems is difficult because Kaplan covers many areas for which specialized tools have been developed. We do not expect to match performance of each of these specialized systems. At the same time, there is no single system that subsumes Kaplan. This section illustrates how other tools can solve some of the problems we presented in this paper and solved using Kaplan. The running times should not be taken as a definitive statement of the relative merits of the systems, but rather as a guide to understanding key differences of the underlying constraint solving techniques.

**Last element of the list.** We first compare the performance of Kaplan and Curry on computing the last element of a list, both in a declarative way as presented in Section 6.2, and as a tail-recursive functional method. In Curry, the declarative version can be expressed as:

```
last xs | concat ys [x] =:= xs
        = x where x,ys free
```

The performance of the Curry implementation surpasses the performance of Kaplan that we reported in Figure 8, running under 0.01 seconds in lists of size up to 100. However, in Kaplan we also have the freedom of writing the constraint directly as a functional program. In such style, Curry performs at about the same speed as for a constraint-based definition, so Kaplan outperforms Curry on longer lists. Figure 11 shows the evaluation results for this case. It is of course the Scala compiler and the Java Virtual Machine that take credit for the good performance of the functional implementation; to the credit of Kaplan is simply that it does not lose any of this underlying efficiency. This basic consequence of Kaplan design

is very important from a practical point of view. In principle, a static analysis could be used to recognize among logical constraints special classes that do not require constraint solving (consider, for example, mode analysis of Mercury [51]). However, this approach clearly requires significant compilation effort to merely recover the baseline performance of a functional or imperative code, whereas in Kaplan it follows by construction.

**Generating data structures with complex invariants.** We now report on our experience in comparing Kaplan with ScalaCheck [48] for generating red-black trees (results were similar for sorted lists). ScalaCheck is a tool for producing test cases using random test generation, similar to QuickCheck [13] for Haskell. We implemented basic generators for lists and trees using generator combinators in ScalaCheck. The problem with random generation is that, for many classes of properties, the probability of a random structure satisfying it may tend to zero as the structure size grows [10], making test cases vacuous for larger structures. To see this in practice, consider first our own performance: Figure 7 shows that Kaplan takes 27.45 seconds in total to generate *all* red-black trees of size $n$ containing elements 1 to $n$, for all $n$ from 0 to 7. We therefore defined a basic ScalaCheck generator that, with equal probability, generates an empty or a non-empty tree. We ran it for 29.1 seconds, generating 200'000 trees. Although 62% of all of them were red-black, that is because 104'544 were, in fact, empty. Of the rest, 18'191 had size one, 934 size two, and there were no red-black trees of larger size. In this experiment, we gave both Kaplan and ScalaCheck only the constraints, without any additional insight, which makes the comparison fair. Experienced users could write better ScalaCheck generators, but they could similarly write better Kaplan checkers. Like UDITA [24], Kaplan supports a full spectrum of intermediate approaches, where one puts more attention into either writing better generators, to increase the ratio of valid testcases, or into manually decomposing the implicit computation to reduce the search space.

**Constraint satisfaction problems.** Tools for constraint solving and optimization over finite domains have developed somewhat independently of SMT and the related technologies [4, 23, 47, 61]. Our anecdotal experience suggests that, for problems of moderate size over finite domains, the two technologies yield comparable results. A full experimental comparison between the state of the art of these two communities is beyond the scope of this paper. Note that we could incorporate into Kaplan solvers from either class, as long as they support our domains of interest (including, for example, algebraic data types).

## 7. Related Work

**Existing languages.** Functional logic programming [3] amalgamates the functional programming and logic programming paradigms into a single language. Functional logic languages, such as Curry [41], benefit from efficient demand-driven term reduction strategies proper to functional languages, as well as non-deterministic operations of logic languages, by using a technique called *narrowing*, a combination of term reduction and variable instantiation. Instantiation of unbound logic variables occur in constructive guessing steps, only to sustain computation when a reduction needs their values. The performance of non-deterministic computations depends on the evaluation strategy, which are formalized using definitional trees [2]. Applications using functional logic languages include programming of graphical and web user interfaces [27, 28] as well as providing high-level APIs for accessing and manipulating databases [12]. Our work can be seen as the practical experiment of building from existing components a system close in functionality to Curry: we proceeded by extending an existing language while preserving its execution model, rather than starting from scratch. As such, we lose the luxury of a total integration of paradigms; for instance, users of Kaplan need to specify which functions can handle logical variables. On the other hand, such a separation of features comes with benefits: we can readily use the full power of Z3 and the Leon verification system for constraint solving, and the execution efficiency of the Java virtual machine for regular code. Implementations of functional logic languages, on the other hand, must typically focus on efficiently executing either the logical or the functional components of the code.

The Oz language and the associated Mozart Programming System is another admirable combination of multiple paradigms [64], with applications in functional, concurrent, and logic programming. In particular, Oz supports a form of logical variables, and logic programming is enabled through unification. One limitation is that one cannot perform arithmetic operations with logical variables (which we have demonstrated in several of our examples), because unification only applies to constructor terms.

Some of the earlier efforts to integrate constraint and imperative programming resulted in the languages Kaleidoscope [37], Alma-0 [5] and Turtle [25]. This line of research put emphasis on designing novel ways to define constraints, ideally to resemble imperative-style programming. Kaleidoscope, for instance, promotes the integration of constraints with objects; programmers can define instance constraints that relate various members of an object, and constraints can be reassigned, just like mutable variables. Constraints are also associated to a *duration* (*once* or *always*), specifying intuitively the scope in which they can affect variables. Kaplan currently does not support constraints over mutable data types and could benefit from the integration of such features.

The integration of constraints as first-class members of the language is tighter in Kaplan than in any of the languages discussed above. For example, in Curry programmers can define constraints as functions that return the special type Success (which is *not* identical to Boolean). However, such functions (or rather predicates) in Curry can only be built from equality predicates and cannot be combined freely: while conjunction and disjunction are valid combinators, negation, for instance, is not. In Kaplan, on the other hand, any predicate expressed in PureScala can be used as a constraint, and terms of other types can also be manipulated and composed freely.

Another distinguishing feature of Kaplan is the native support for many theories (integers, sets, maps, data types) that comes from using Leon [60] and Z3 as the underlying constraint solver.

**Language design.** Monadic constraint programming [56] integrates constraint programming into purely functional languages by using monads to define solvers. The authors define monadic search trees, corresponding to a base search, that can be transformed by the use of *search transformers* in a composable fashion to increase performance. Our system differs from this work in its use of SMT solvers for search, and a more flexible way of mixing constraints with imperative programming.

Localized code synthesis techniques have been proposed to turn implicit declarations into explicit code [35]. The main limitation of that approach is the requirement that the theory should be decidable. We have shown that using a powerful, undecidable, logic for constraints can be beneficial. Ideally, compile-time techniques such as [35] should be combined with the run-time approach of constraint solving.

The work on uniform reduction to bit-vector arithmetic [40] (URBIVA) proposes a C-like language for specifying constraints. It uses symbolic execution to encode problems into bit-vector arithmetic and invokes one of the underlying bit-vector and SAT solvers. The system allows for the comparison of different solvers on examples involving solution enumeration and functional equivalence

checking. The use of symbolic values in conditional statements and array indexing is not permitted. Because it uses native enumeration capability of solvers such as CLASP, it can be faster than our system on some of the benchmarks. URBIVA does not support unbounded domains; we are aware of no techniques to enumerate solutions for unbounded domains more efficiently than in Kaplan.

**SMT as a programming platform.** The Dminor language [9] introduces the idea of using an SMT solver to check subtyping relations between refinement types; in Dminor, all types are defined as logical predicates, and subtyping thus consists of proving an implication between two such predicates. The authors show that an impressive number of common types (including for instance algebraic data types) can be encoded using this formalism. In this context, generating values satisfying a predicate is framed as the *type inhabitation* problem, and the authors introduce the expression elementof $T$ to that end. It is evaluated by invoking Z3 at run-time and is thus conceptually comparable to our find construct but without support for recursive function unfolding. We have previously found that recursive function unfolding works better as a mechanism for satisfiability checking than using quantified axiomatization of recursive functions [60]. In general, we believe that our examples are substantially more complex than the experiences with elementof in the context of Dminor.

The Scala$^{Z3}$ library [33] is our earlier effort to integrate invocations to Z3 into a programming language. Because it is implemented purely as a library, we were then not able to integrate user-defined recursive functions and data types into constraints, so the main application is to provide an embedded domain-specific language to access the constraint language of Z3 (but not to extend it). A similar approach has been taken by others to invoke the Yices SMT solver [19] from Haskell.[3]

**Applications of declarative programming.** One approach in using specifications for software reliability is data structure repair [16, 20], where the goal is to recover from corrupted data structures by transforming states that are erroneous with respect to integrity constraints into valid ones, performing local heuristic search. [65] uses method contracts instead of data structure integrity constraints to be able to support rich behavioral specifications. While the primary goal is to perform run-time recovery of data structures, recent work [38] extends the technique for debugging purposes, by abstracting concrete repair actions to program statements performing the same actions. Data structure repair differs from our system in that it can perform local search to modify existing states, while we do not currently do so. We do not expect that a general-purpose constraint solving such as ours can immediately compare with these dedicated techniques.

The idea to use specifications as a fall-back mechanism, as in one of our application examples, was adopted in [55]. Similarly to our setting, dynamic contract checking is applied and, upon violations, specifications can be *executed*. The technique ignores the erroneous state and computes output values for methods given concrete input values and the method contract. The implementation uses a relational logic similar to Alloy [30] for specifications, and deploys the Kodkod model finder [63]. A related tight integration between Java and the Kodkod engine is presented in [42]. In both cases, due to the finite bound on the search space, a satisfying answer may not always be found, which makes the techniques incomplete. We have shown that executing declarative specifications is possible in our setting. We expect that a constraint solver such as the one deployed in Leon will ultimately be better suited than an approach based on KodKod, due to the presence of unbounded or large data types such as integers and recursive structures.

[3] http://hackage.haskell.org/package/yices-easy

## 8. Conclusion

We presented Kaplan, an extension of the multi-paradigm Scala language that integrates constraint programming while preserving the existing functional and object-oriented features of Scala. The behavior and performance of Kaplan is identical to Scala in the absence of declarative constraint solving. Kaplan integrates constraints as first-class objects in Scala, and logical variables for solving constraints lazily. It allows for solving optimization problems, as well as enumerating solutions in a user-specified order. Kaplan is implemented as a combination of a compiler plugin and a runtime library, and allows users to express constraints in a powerful and expressive logic. Kaplan relies on a procedure for satisfiability modulo recursive functions [59, 60] to solve these constraints.

We evaluated our system by considering applications such as solution enumeration, execution of declarative specifications, test-case generation for bug-finding and counterexample guided inductive synthesis, as well as numerous smaller, yet expressive, snippets. We observed that our model for introducing non-determinism using for-comprehensions integrates well with Scala.

Based on our experience with the Z3 SMT solver and the Leon verification system in constraint programming, we found that a number of features, if natively supported by solvers, could directly bring benefits to constraint programming. These include 1) support for enumeration of theory models and 2) solving constraints while minimizing/maximizing a given term. Overall, we believe there is great potential in extending standard programming languages with constraint solving capabilities. Many interesting problems are left open, both in language design and in constraint solving; a system such as Kaplan that integrates state-of-the-art tools from both domains is likely to benefit from progress made in each of them.

## Acknowledgments

We thank Sergio Antoy for an interesting discussion on the status of the Curry programming language. We thank Nikolaj Bjørner and Leonardo de Moura for their help in using Z3, and Adriaan Moors for his help with the Scala compiler. We thank Swen Jacobs, Eva Darulová, and the anonymous reviewers for their feedback.

## References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[2] S. Antoy. Definitional trees. In *ALP*, pages 143–157, 1992.

[3] S. Antoy and M. Hanus. Functional logic programming. *CACM*, 53 (4):74–85, 2010.

[4] K. R. Apt and M. Wallace. *Constraint logic programming using Eclipse*. Cambridge University Press, 2007.

[5] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-O: An imperative language that supports declarative programming. *TOPLAS*, 20(5):1014–1066, 1998.

[6] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer-Verlag, 1998.

[7] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *Softw. Test., Verif. Reliab.*, 10 (3), 2000.

[8] C. Barrett and C. Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.

[9] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, pages 105–116, 2010.

[10] A. Blass, Y. Gurevich, and D. Kozen. A zero-one law for logic with a fixed-point operator. *Inf. Control*, 67, October 1986.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[12] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *PADL*, pages 316–332, 2008.

[13] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[14] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Last steps towards an ultimate PROLOG. In *IJCAI*, pages 947–948, 1981.

[15] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[16] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

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

[18] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[19] B. Dutertre and L. de Moura. The Yices SMT solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.

[20] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE*, pages 855–858, 2008.

[21] S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy nondeterministic programming. In *ICFP*, volume 44, pages 11–22, 2009.

[22] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[23] I. P. Gent, C. Jefferson, and I. Miguel. MINION: A fast, scalable, constraint solver. In *European Conference on Artificial Intelligence*, pages 98–102. IOS Press, 2006.

[24] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *ICSE (1)*, pages 225–234, 2010.

[25] M. Grabmüller and P. Hofstedt. Turtle: A constraint imperative programming language. In *Innovative Techniques and Applications of Artificial Intelligence*, 2003.

[26] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[27] M. Hanus. Type-oriented construction of web user interfaces. In *PPDP*, pages 27–38, 2006.

[28] M. Hanus and C. Kluß. Declarative programming of user interfaces. In *PADL*, pages 16–30, 2009.

[29] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4), October 1995.

[30] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[31] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, 1987.

[32] J. C. King. *A Program Verifier*. PhD thesis, CMU, 1970.

[33] A. S. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: Integrating SMT and programming (system description). In *CADE*, pages 400–406, 2011.

[34] R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[35] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[36] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wessley, 2002.

[37] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope: A constraint imperative programming language. In *Constraint Programming*, pages 313–329. Springer-Verlag, 1994.

[38] M. Z. Malik, J. H. Siddiqui, and S. Khurshid. Constraint-based program debugging using data structure repair. In *ICST*, pages 190–199, 2011.

[39] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357084.357090.

[40] F. Maric and P. Janicic. Urbiva: Uniform reduction to bit-vector arithmetic. In *IJCAR*, pages 346–352, 2010.

[41] E. Michael Hanus. Curry: An integrated functional logic language. `http://www.curry-language.org`, 2006. vers. 0.8.2.

[42] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.

[43] C. Morgan. *Programming from Specifications (2nd ed.)*. Prentice-Hall, Inc., 1994.

[44] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[45] G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[46] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322198.

[47] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming*, pages 529–543, 2007.

[48] R. Nilsson. Scalacheck user guide. `http://code.google.com/p/scalacheck/wiki/UserGuide`, 2011.

[49] M. Odersky. Contracts in Scala. In *International Conference on Runtime Verification*. Springer LNCS, 2010.

[50] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[51] D. Overton, Z. Somogyi, and P. J. Stuckey. Constraint-based mode analysis of Mercury. In *ACM SIGPLAN Workshop on Principles and practice of declarative programming (PPDI)*, 2002.

[52] L. C. Paulson, T. Nipkow, et al. Isabelle theorem prover - official website. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[53] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[54] A. Riesco and J. Rodríguez-Hortalá. Programming with singular and plural non-deterministic functions. In *PEPM*, pages 83–92, 2010.

[55] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[56] T. Schrijvers, P. J. Stuckey, and P. Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, 2009.

[57] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[58] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[59] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *ACM SIGPLAN POPL*, 2010.

[60] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symposium (SAS)*, 2011.

[61] G. Tack. *Constraint Propagation - Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.

[62] The Coq Development Team; INRIA LogiCal Project. The Coq proof assistant - official website. `http://coq.inria.fr`.

[63] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[64] P. Van Roy. Logic programming in Oz with Mozart. In *ICLP*, 1999.

[65] R. N. Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598, 2010.

[66] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.