Constraints as Control

By: Sinan Koksal, Viktor Kuncak, Philippe Suter

Presented by: Matthew Rhea

Scala

- An object-oriented language
  - Every value is an object, and types of objects are described by classes and traits
- Also, a functional language
  - Every function is a value
- Statically typed
  - Enforces that abstractions are used, at compile-time, in a safe manner

Constraint Programming

- A programming paradigm in which users declaratively state constraints on possible solutions
  - As opposed to detailing the steps to take in execution, it specifies the properties of a solution to be found
- Users specify a method to solve the constraints

Kaplan

- An extension of Scala that supports Constraint Programming
- Integrates imperative programming by using constraints as a control structure
- As a functional subset of Scala, Kaplan supports recursive function definitions over algebraic data types, sets, maps, and integers

Motivation for Kaplan

- It is difficult to solve declarative constraints

- It is difficult to incorporate constraint constructs into existing languages and platforms

How to tackle these challenges?

- It is difficult to solve declarative constraints

==> At the time, solvers were making significant progress: Kaplan leverages this.

- It is difficult to incorporate constraint constructs into existing languages and platforms

==> Kaplan programs look just like Scala programs

How to tackle these challenges?

- Use already existing features of Scala
  - Exploit Scala for-comprehensions to describe iterations as a search process over solution spaces
- Allowing users to define their own classes of constraints enables more efficient solving of declarative constraints

Features: First-Class Constraints

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

- **A new constraint with two variables**

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

- **An alternative way to declare constraints**

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

- **Constraints are extensions of functions, so may be evaluated**

Features: First-Class Constraints

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

- **Querying the solver for a solution**

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

- **Finite set of solutions can be listed**

Features: First-Class Constraints

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

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

- **CNF SAT defined in Kaplan**

```
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
}
```

```
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
```

- **A solver for CNF SAT formulas**

Features: Ordering Solutions

```
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
}
```

```
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)
```

- **Kaplan supports enumerating two methods in a defined order: minimizing and maximizing that take as input an objective function**
- **Consider the Knapsack problem in which a solution is an instance of a map indicating a choice of which objects should be picked**
- ***conditionalSumTerm* builds an integer term parameterized by a choice map and representing a sum of values”**

Features: User-Defined Functions and Datatypes

```
@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
```

```
@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 = . . .
```

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

```
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)
```

- Users can define their own (recursive) data types
- @spec annotation indicates the user wants to use the function as part of constraints
- Here, we have two such types for red-black trees

Features: Timeouts

- Not all constraints will be solvable, so Kaplan supports timeout strategies

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

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

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

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

Features: Logical Variables

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

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

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

- Previous examples all illustrate *eager* solution enumeration (solving constraints immediately produced concrete values)
- Kaplan can instead produce a *promise* of a solution in the form of a logical variable (a result of *lazily* solving a constraint)

Features: Imperative Constraint Programming

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

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

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

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

```
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)
```

```
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)
```

- Kaplan defines **assuming-otherwise** branding as a new library construct
- Similar to if-then-else except that it checks whether the condition is "*feasible*" and, if so, constrain the logical variables in the environment so that their values satisfy the branching constraint

The Constraint Sublanguage: Data types

- Core sublanguage is PureScala, a subset of Scala
- Supports (recursive) algebraic data types
- Types are defined using a hierarchy of special *case* classes

The Constraint Sublanguage: Expressions and Function Definitions

- Expressions may contain all standard arithmetic operators, map applications and updates, set operators and membership tests, and function applications

- A PureScala function body is defined by a single expression whose free variables are the arguments of the function, and may optionally be annotated with a post-condition

The Constraint Sublanguage: Solver

- Kaplan invokes the Leon verification systems's core solving procedure at both compile-time and run-time
- At compile-time, it validates post-conditions and prove pattern-matching expressions are exhaustive
- At run-time, it finds solutions to constraints

Implementation

- Briefly: Kaplan has been implemented as both a run-time library and a compiler plugin (Both of which are implemented in Scala)

- For this presentation, we focus on the implementation of the core solving algorithms and its interactions with the SMT Solver

Implementation: Core Solving Algorithms

- Leon enables additional expressive power of recursive functions within constraints, hence Kaplan implements Z3 through this extension

Implementation: Solution Enumeration

- Uses *find* and *lazyFind* through the use of an iterator to implement *findAll* and *lazyFindAll*
- The iterator maintains constraints at all times
- Each time a new solution is required, iterator updates the constraint by adding to it the negation of the previous solution
- Made efficient by the incremental reasoning of Z3 to avoid solving the entire constraint each time

Implementation: Optimization Constraints

- Can be seen as a generalization of a binary search over a range of values an objective function can take

```
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.

Implementation: Ordered Enumeration

```
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.

- Composing the minimized term with a solution enumeration obtains an ordered enumeration

Discussion

-