Lobot User's Guide

Ben Selfridge, Galois Inc.

Contents

1 Introduction

1.1 What is Lobot?

Lobot is a language for describing data types (ints, structs, etc.) with constraints attached to them. It is also a tool for enumerating the values of those types, and checking that the enumerated values satisfy certain properties.

Lobot uses an SMT solver to enumerate and check constrained types (called *kinds*), and it has very similar expressive power to constraint programming languages (like MiniZinc). However, Lobot does more than model a problem; it can directly interact with real-world applications and incorporate them into its reasoning engine. In other words, we can call a command line tool with Lobot and use it as part of a constraint in a kind we are defining. This allows us to talk about properties of external software and actually verify that various properties hold about that software.

Before we dive into the core concepts of Lobot, let's look at a couple of examples.

1.2 Example 1: pairs that sum to 100

Suppose I wanted to define a data type that represents all pairs of positive integers that sum to 100. I could do that in Lobot by first defining the concept of a positive integer:

```
-- Positive integers.
posint : kind of int where 1 <= self
```

Then, I can create pairs of positive integers by defining a kind of `struct`:

```
-- Unique pairs of positive integers
unique_posint_pair : kind of struct
  with x : posint
       y : posint
  where x <= y
```

Notice the `x <= y` constraint, which prevents us from getting the same pair, just in a different order. Now, we can define one more data type that further constrains `unique_posint_pair`:

```
-- Pairs that sum to 100
posint_sum_100 : kind of unique_posint_pair
  where x + y = 100
```

If I type the above definitions into a file, `posint_sum_100.lobot`, and then fire up the Lobot tool via

```
$ lobot posint_sum_100.lobot
```

I get the following output:

```
All checks pass. (File had no checks)
```

This means that the file was syntactically valid and well-typed, which is great! It also doesn't give us any information about the types we defined. We can count the number of instances of one of our `kind`s in the following way:

```
$ lobot -c posint_sum_100 posint_sum_100.lobot
Generating instances...
Found 50 valid instances, generated 0 invalid instances
```

Lobot determined that there are 50 instances of the `posint_sum_100` kind. If we want to enumerate them, we can use the `-e` option:

```
$ lobot -e posint_sum_100 posint_sum_100.lobot
Generating instances...
```

```
Instance 1:
  posint_sum_100 with {x = 50, y = 50}
Press enter to see the next instance
```

Each time we hit enter, we see a new instance.

What if we want to verify that some property always holds of our `posint_sum_100` kind? For instance, we might like to ensure that for any instance `p` of `posint_sum_100`, `p.x <= 50`. We can encode that as a `check` command in the Lobot source file:

```
-- Check that all instances satisfy p.x <= 50
check1 : check
  on p : posint_sum_100
  that p.x <= 50
```

We run this check via the `-r` option:

```
$ lobot -r check1 posint_sum_100.lobot
Generating instances...
Check 'check1' holds. (Generated 0 instances)
```

Lobot determined that the check holds for all instances of the `posint_sum_100` kind. What if we change the check condition `p.x <= 50` to `p.x < 50`?

```
$ lobot -r check1 posint_sum_100.lobot
Generating instances...
Check 'check1' failed with counterexample:
  p = struct with {x = 50, y = 50}
```

Lobot fails, and finds a counterexample for us. Great!

1.3 Example 2: The real world

Lobot is designed to directly interact with applications as a core language feature. For instance, suppose I have a shell command called `add1` that works like this:

```
$ add1 5
6
$ add1 -1000000
-999999
```

Context clues lead us to the conclusion that the `add1` command is probably supposed to add 1 to its input. In Lobot, we can actually import functions from the command-line environment and invoke them when defining new kinds. However, we do need to provide small wrapper scripts for Lobot to be able to call them; Lobot passes arguments to these commands by translating them to JSON and passing them to stdin, and then getting the JSON-encoded results from stdout. Below is an example implementation of `add1` in python that is compatible with Lobot's API:

```python
#!/usr/bin/python3

import json
import sys

json_str = ""

for line in sys.stdin:
  json_str += line

# Incoming JSON data has this format:
# [ { "variant": "int", "value": <value> } ]
json_data = json.loads(json_str)

i = json_data[0]['value']
json_data[0]['value'] = i + 1

print(json.dumps(json_data[0]))
```

After adding this to our `$PATH`, we can call it on the command line:

```
$ echo "[{\"variant\": \"int\", \"value\": 4}]" | add1
{"variant": "int", "value": 5}
```

This is, admittedly, not as clean to interact with directly as the original `add1` function. However, we shouldn't have to directly interact with it; that's Lobot's job!

Now that `add1` is on our path, we can use it in Lobot. In a new file named `add1.lobot`, we write

```
abstract add1 : int -> int
```

This is a *function declaration*. We have declared to Lobot that there exists a function called `add1` mapping integers to integers. Now, we are free to use this function as part of a new kind definition:

```
add1_0 : kind of int where self = add1(0)
```

Just like any other kind, we can enumerate instances one-by-one, or we can count the number of instances. Let's enumerate `add1_0`:

```
Instance 1:
  1
Press enter to see the next instance

Enumerated all 1 valid instances, generated 1 invalid instances
```

Lobot has determined that there is exactly one instance of `add1_0`, which is not terribly impressive, since every function has exactly one value when you call it.

Let's see what happens if we try the inverse problem of finding the solution to `add1(x) = 0`:

```
add1_is_0 : kind of int where add1(self) = 0
```

Let's enumerate this kind:

```
$ lobot -e add1_is_0 add1.lobot
Generating instances of 'add1_is_0'...
Hit instance limit of 100!
Found 0 valid instances, generated 100 invalid instances
Press enter to continue enumerating up to 100 more instances.
```

Looks like we generated 100 invalid instances, but were not able to find a solution. We could keep pressing enter to just continue the search, but instead, let's use the `-v` option to try and figure out what's going on:

```
$ lobot -e add1_is_0 -v add1.lobot
Generated an invalid instance:
  1
The constraints that failed were:
  add1(self) = 0
Learned the values of the following function calls:
  add1(1) = 2
Press enter to see the next instance

Generated an invalid instance:
  3
The constraints that failed were:
  add1(self) = 0
Learned the values of the following function calls:
  add1(3) = 4
Press enter to see the next instance

Generated an invalid instance:
  5
The constraints that failed were:
  add1(self) = 0
Learned the values of the following function calls:
  add1(5) = 6
Press enter to see the next instance
```

Here, we see that Lobot is attempting to find a solution to `add1(x) = 0` by enumerating positive values of `x` until it finds one (for some reason, it is only checking the odd values). However, this strategy will never yield a valid instance, because the solution to this equation is `-1`, which is negative. We can help Lobot out by constraining the search space a bit:

```
add1_is_0 : kind of int
```

```
  where add1(self) = 0
        -10 <= self
        self <= 10
```

Now, when we attempt to enumerate values of this kind, we have more success:

```
$ lobot -e add1_is_0 add1.lobot
Instance 1:
  -1
Press enter to see the next instance

Enumerated all 1 valid instances, generated 20 invalid instances
```

Not only has Lobot found the solution, `-1`, it has determined that it is the only solution in the range `-10 <= x <= 10` via exhaustively checking every value in that range. Since `add1` could have been any function, and Lobot knows nothing about its behavior, we couldn't reasonably expect it to do anything more clever than an exhaustive search.

In general, Lobot knows nothing about real-world functions until it actually evaluates calls to it for particular values. Therefore, we need to be careful when enumerating instances of kinds that are constrained by such functions.

2 The Lobot language

2.1 Types and kinds

The base types of Lobot are:

- booleans
- integers
- enums
- enumsets
- structs
- abstract types

A *constraint* is an expression that can be evaluated over a value of a particular type. For instance, if `a : bool` and `b : bool`, then `a | b` is the constraint that either `a = true` or `b = true`.

A *kind* is a type plus a list of constraints. For instance, we can define the kind of integers between 0 and 5, or 5 and 10:

```
int_0_5  : kind of int where 0 <= self, self <= 5
int_5_10 : kind of int where 5 <= self, self <= 10
```

We can then use values of this kind in other kinds:

```
two_ints : kind of struct
  with x : int_0_5
```

```
       y : int_5_10
```

Equivalently, we could also specify this via explicit constraints:

```
two_ints : kind of struct
  with x : int
       y : int
  where x : int_0_5
        y : int_5_10
```

Both of the above definitions are equivalent to:

```
two_ints : kind of struct
  with x : int
       y : int
  where 0 <= x
        x <= 5
        5 <= y
        y <= 10
```

We can also combine multiple kinds into a kind that represents all the constraints of the “parent” kind:

```
int_5 : kind of int_0_5 int_5_10
```

2.1.1 Type synonyms

Before we discuss these types individually, it will be helpful to introduce *type synonyms*. We can declare a new type synonym:

```
type <name> = <type>
```

where `<name>` is the name of the type synonym, and `<type>` is a base type. Type names must start with a lowercase letter. For instance, we could alias `int` with `ident`:

```
type ident = int
```

We can now use `ident` as an alternative name for `int`. However, it’s usually more useful for user-defined types:

```
type my_struct = struct
  with a : bool
       x : int
       e : {A, B, C}
```

2.1.2 Booleans

The built-in keyword `bool` signifies the type of boolean values. We can enumerate `bool` instances by declaring a synonym in lobot:

```
-- file: bool.lobot
type my_bool = bool
```

```
$ lobot -e my_bool bool.lobot
Instance 1:
  false
Press enter to see the next instance

Instance 2:
  true
Press enter to see the next instance

Enumerated all 2 valid instances, generated 0 invalid instances
```

If we are in an introductory logic course and are assigned the following homework problem:

```
Fill out the truth table of the proposition "P => ((Q | R) => (R => !P))".
```

We can use Lobot to cheat:

```
-- file: logic_homework1.lobot
problem1 : kind of struct
  with p : bool, q : bool, r : bool
  where p => ((q | r) => (r => not p))
```

```
$ lobot -e problem1 logic_homework1.lobot
Instance 1:
  problem1 with {p = false, q = false, r = false}
Press enter to see the next instance

Instance 2:
  problem1 with {p = false, q = false, r = true}
Press enter to see the next instance

Instance 3:
  problem1 with {p = false, q = true, r = true}
Press enter to see the next instance

Instance 4:
  problem1 with {p = false, q = true, r = false}
Press enter to see the next instance

Instance 5:
  problem1 with {p = true, q = false, r = false}
Press enter to see the next instance

Instance 6:
  problem1 with {p = true, q = true, r = false}
```

```
Press enter to see the next instance

Enumerated all 6 valid instances, generated 0 invalid instances
```

2.1.3 Integers

Similarly to `bool`, `int` signifies the type of integer values.

```
-- file: int.lobot
type my_int = int
```

We can attempt to count the integers (rather than enumerate them one-by-one) with the `-c` option:

```
lobot -c my_int int.lobot
Hit instance limit of 100!
Found 100 valid instances, generated 0 invalid instances
```

Lobot informs us that we have hit the built-in instance limit of `100`, resulting in `100` instances of `int`. We can increase this limit with `-l`:

```
lobot -c my_int -l 10000 int.lobot
Hit instance limit of 10000!
Found 10000 valid instances, generated 0 invalid instances
```

After a minute or two, Lobot finally determines that there are at least `10000` instances of `int`. So, in case you didn't realize it – there are more than ten thousand integers!

Lobot can factor integers:

```
factor_120 : kind of struct
  with p : int, q : int
  where p * q = 120
        p > 1
        q > 1
```

```
$ lobot -e factor_120 int.lobot
Instance 1:
  factor_120 with {p = 20, q = 6}
Press enter to see the next instance.
```

If we are in a high school algebra class, we can use Lobot to do our homework on linear systems of equations for us:

```
type pair = struct with x : int, y : int

problem1 : kind of pair
  where x - 7 * y = -11
        5 * x + 2 * y = -18
```

```
lobot -e problem1 algebra_homework1.lobot
Instance 1:
  problem1 with {x = -4, y = 1}
Press enter to see the next instance

Enumerated all 1 valid instances, generated 0 invalid instances
```

This of course assumes that the linear system has an integer solution (which they usually do when they arrive in the form of algebra homework).

2.1.4 Enumerations and enumeration sets

Enumerations are like `enum`s in the C programming language, or "sum types" in Haskell. They are finite, user-defined types that encode a variety of different choices.

```
type abc = {A, B, C}
```

```
$ lobot -e abc enum.lobot
Instance 1:
  A
Press enter to see the next instance

Instance 2:
  B
Press enter to see the next instance

Instance 3:
  C
Press enter to see the next instance

Enumerated all 3 valid instances, generated 0 invalid instances
```

Here, we introduce an enum type called `abc` with three constructors, `A`, `B`, and `C`. The constructors of an enum must start with capital letters. We can also use subsets of enumerations:

```
type abc_set = subset abc
```

```
$ lobot -e abc_set enum.lobot
Instance 1:
  {}
Press enter to see the next instance

Instance 2:
  {A}
Press enter to see the next instance

Instance 3:
  {A, B}
Press enter to see the next instance

Instance 4:
  {A, B, C}
Press enter to see the next instance

Instance 5:
  {B, C}
Press enter to see the next instance

Instance 6:
  {C}
Press enter to see the next instance

Instance 7:
  {A, C}
Press enter to see the next instance

Instance 8:
  {B}
Press enter to see the next instance

Enumerated all 8 valid instances, generated 0 invalid instances
```

We can create more constrained kinds of subsets:

```
a_implies_c : kind of abc_set
  where A in self => C in self

doesnt_have_c : kind of abc_set
  where not (C in self)

both : kind of a_implies_c doesnt_have_c
```

```
$ lobot -e both enum.lobot
Instance 1:
  {}
Press enter to see the next instance

Instance 2:
  {B}
Press enter to see the next instance

Enumerated all 2 valid instances, generated 0 invalid instances
```

2.1.5 Structs

In Lobot, *structs* are the basic way we form compound types and package multiple values into a single data structure. We have seen several examples of structs already. Here's another one:

```
perfect_square : kind of struct
  with x : int
       rt_x : int
  where x = rt_x * rt_x

even_perfect_square : check
  on s : perfect_square
  where s.x % 2 = 0
  that s.x % 4 = 0
```

2.1.6 Abstract types

We can declare brand new types in Lobot, without specifying the values they take:

```
abstract type t

type two_ts = struct
  with t1 : t
       t2 : t
```

However, if we try to generate instances of `two_ts`, something interesting happens:

```
$ lobot -e two_ts abstract.lobot
Cannot generate instances of abstract type.
```

Abstract types cannot be enumerated by Lobot, because we know nothing about their value set. So, why are they part of Lobot? Because they can be used as return values and arguments to *abstract functions*.

2.2 Abstract functions

The main distinction of Lobot over other constraint solving languages is that it is designed to establish properties of real-world applications.

Suppose I have I have a command-line tool called `write_nlines` that takes a single integer argument `n`, and outputs a file called `tmp.txt` that contains `n` lines. I also have another tool that takes a filepath as an argument and counts the number of lines in the file. The `wc` command, with the `-l` option, is such a tool. I would like to verify that if I call `write_nlines` with an argument `n`, and then I call `wc -l` on the resulting file, I get `n` back.

How do we specify the type of `write_nlines`? It isn't really a "function" in the mathematical sense of the word, as it doesn't exactly return a value so much as it affects the world. In order to make it something Lobot can reason about, we need to find a way to package its affect on the world as a Lobot value. We can do this by modeling it as a function that returns a *filepath*, which will be the name of the file that gets written (`tmp.txt`). However, since a filepath is not a built-in Lobot type, our first task is to declare a new abstract type:

```
abstract type filepath
```

Now, we can declare the `write_nlines` function:

```
abstract write_nlines : int -> filepath
```

The `wc` command can take a variety of options, so we might as well model that:

```
-- C for characters, L for lines, W for words
wc_option = { C, L, W }

wc_wrapper : (wc_option, filepath) -> int
```

We can now state our desired property as a `check`:

```
write_nlines_check : check
  on i : int
  where 0 <= i, i <= 50
  that wc_wrapper(L, write_nlines(i)) = i
```

Now, in order to actually run this Lobot file, the `write_nlines` and `wc_wrapper` commands must be on our `PATH`, and they must conform to Lobot's function call API (described in a later section of this document).

Here is a python-based implementation of `write_nlines`:

```
#!/usr/bin/python3

import json
import sys

json_str = ""

for line in sys.stdin:
  json_str += line

json_data = json.loads(json_str)

num_lines = json_data[0]['value']

f = open("tmp.txt", "w")
for i in range(0, num_lines):
  f.write("Line " + str(i) + "\n")
```

```
json_output = {
  "variant": "filepath",
  "value": "tmp.txt"
}

print(json.dumps(json_output))
```

Notice that the input to `write_nlines` is provided via JSON-encoded stdin, and the output is provided via JSON-encoded stdout. The input is a JSON array with a single entry, which is an integer. It is decoded in the line:

```
num_lines = json_data[0]['value']
```

Lobot will evaluate a function call of `write_nlines(5)` by executing the following command in the background:

```
echo "[{\"variant\": \"int\", \"value\": 5}]" | write_nlines
```

The JSON data passed through stdin is a JSON-encoded array with a single element. That element is a JSON object (key/value mapping) with two entries: a `variant`, specifying the Lobot type of the data, and a `value`, giving the concrete value of the data.

The output of the above command is:

```
{"variant": "filepath", "value": "tmp.txt"}
```

This corresponds to the Lobot abstract type `filepath`, with the string value `"tmp.txt"`. Values of abstract types are encoded as JSON strings. Since such values are never directly handled within Lobot, Lobot does not need to care about the details of their representation. All that matters is that the various command line tools that are implementing the relevant functions all agree on that representation.

Here is a similar implementation of `wc_wrapper`, which actually wraps the `wc` command, while providing the correct API for Lobot:

```
#!/usr/bin/python3

import json
import sys
import os

json_str = ""

for line in sys.stdin:
  json_str += line

json_data = json.loads(json_str)
```

```
wc_config = json_data[0]['constructors'][json_data[0]['value']]
filepath = json_data[1]['value']

command = "wc "
if wc_config == "C":
  command += "-c"
elif wc_config == "L":
  command += "-l"
elif wc_config == "W":
  command += "-w"
command += " " + filepath

words = os.popen(command).read().split()
x = int(words[0])

json_output = {
  "variant": "int",
  "value": x
}

print(json.dumps(json_output))
```

This command takes two arguments (as specified in the Lobot file). If we create a file called `wc_wrapper_input.json` with the following contents:

```
[ { "variant": "enum",
    "constructors": ["C", "L", "W"],
    "value": 1
  },
  { "variant": "filepath",
    "value": "tmp.txt"
  }
]
```

then we can execute the `wc_wrapper` command directly like so:

```
$ cat wc_wrapper_input.json | wc_wrapper
{"variant": "int", "value": 5}
```

Once both these commands are on our PATH, we can verify the `check`:

```
$ lobot -r write_nlines_check fn.lobot
Check 'write_nlines_check' holds. (Generated 51 instances)
```

2.3 JSON API

In order to use a command line tool from Lobot, we typically need to *wrap* the function so that it conforms with Lobot's JSON-based function call API. This section provides a complete specification for how Lobot function declarations translate to this API; i.e. given an abstract function's type, we can determine exactly how the input arguments are encoded, and how the return value needs to be encoded for Lobot to be able to use it.

We've seen examples of wrapper scripts elsewhere in this guide. In this section, we define the JSON encoding of Lobot values, as well as the calling convention Lobot uses to pass values to/from shell commands.

2.3.1 Function calls

Consider `add1`:

```
abstract add1 : int -> int
```

When lobots evaluates the call `add1(5)`, it first converts the single argument `5` into the following JSON array:

```
[ { "variant" : "int", "value": 5 } ]
```

That is, an array with one object, the value `5`. Then it calls the `add1` command (which must be on the `PATH`) by passing this JSON byte string to the command over stdin. Finally, it collects the output of `add1` on stdout, which happens to be:

```
{ "variant": "int", "value": 6 }
```

In general, when evaluating a call of `f(x1, x2, ...)`, Lobot first converts the `xi` to JSON strings `j1, j2, ...`, packages them up into a JSON array:

```
[ j1, j2, ... ]
```

and calls the `f` shell command, passing the resulting string over stdin. The command then should return some JSON-encoded value `y` over stdout.

2.3.2 Booleans and integers

The boolean value `true` is encoded as

```
{ "variant": "bool", "value": "true" }
```

`false` is encoded similarly. An integer literal `<x>` is encoded as

```
{ "variant": "int", "value": <x> }
```

Notice that the integer `<x>` appears without quotes, as it is a numeric JSON value.

2.3.3 Enums and enumsets

Consider the type

```
type abc = {A, B, C}
```

The value `B` is encoded as

```
{
  "variant": "enum",
  "constructors": ["A", "B", "C"],
  "value": 1
}
```

Notice that all the information about the type is encoded; we don't simply encode `"enum"` and `"B"`, but we actually encode all the constructors as well so that the type is fully determined. The `value` field is an index into the array of `constructors`, so it because straightforward to recover the string representation.

The enumset `{B, C}`, with type `subset {A, B, C}`, would be encoded as

```
{
  "variant": "set",
  "constructors": ["A", "B", "C"],
  "values": [1, 2]
}
```

Here, instead of a single `value`, we have an array of `values`, each of which is a unique index into the array of `constructors`.

2.3.4 Structs

Consider a struct type defined as

```
type s = struct
  with f1 : tp1,
       f2 : tp2,
       ...
```

with fields `f1, ...` of types `tp1, ...`. Suppose we have a value of such a struct, with values `f1 = <x1>, f2 = <x2>, ...`, where each `<xi>` has encoding `<ji>`. This value is encoded as

```
{
  "variant": "struct",
  "fields": [ { "name": "f1", "value": <j1> },
              { "name": "f2", "value": <j2> },
              ...
            ]
}
```

2.3.5 Abstract types

Consider an abstract type:

```
abstract type foo
```

Concrete values of abstract types are represented “under the hood” as arbitrary strings of bytes (or, equivalently, characters) `<s>`. We encode this value in JSON like so:

```
{ "variant": "foo", "value": "<s>" }
```

In other words, we simply dump the underlying contents of the abstract value into a JSON string. The representation looks very similar to that of `int` and `bool`. Functions that create abstract values simply encode arbitrary data into the `<s>` string, and functions that consume abstract values decode arbitrary data fron the `value` key.