Programming with Useful Quantifiers

Jim E. Newton
EPITA/LRE
Le Kremlin-Bicêtre, France
jnewton@lrde.epita.fr

Abstract

Many programming languages have functions which implement universal and existential quantifiers. Typically these are implemented as higher order functions which accept a Boolean predicate and some sort of iterable and return a Boolean value indicating whether a predicate is validated over the iterable or rather, whether a counterexample or witness was encountered. These quantifiers are elegant to use and allow application code to be written which closely resembles axiomatic algebraic specifications. A disadvantage is seen when unexpected results occur, *e.g.*, when attempting to debug user defined predicates. These quantifiers do not have easy ways of returning the witness or counterexample as such would typically violate the expected Boolean return value, and thus modify the program flow or in some cases cause compilation errors. Users are usually forced to refactor code to avoid the elegant quantifiers in order to debug their code. We present implementations of universal and existential quantifiers in Clojure, Common Lisp, Python, and Scala, which conceptually extend the Boolean type with decorations which are useful not only for debugging but for mathematical purposes.

CCS Concepts

• **Theory of computation → Data structures design and analysis**; *Type theory*.

Keywords

common lisp,clojure,scala,python,quantifiers

1 Overview

In this article we present *Heavy Booleans*, and their implementation in Clojure [8], Common Lisp [1, 12], Python [13], and Scala [9, 10]. These data structures are values which on the one hand implement Boolean (*true*/*false*) semantics, and on the other hand provide reflective behavior [6]; *i.e.*, giving access to the witness of an existential quantifier, $\exists a \in M, p(a)$, or counterexample of a universal quantifier [2], $\forall a \in M, p(a)$.

Heavy Booleans convey information about the counterexample when universal quantification fails. If $\forall a \in M, p(a)$ returns *false*, the user should be able to ask which counterexample value of M made p *false*. Dually, if $\exists a \in M, p(a)$ returns *true*, the user asks which value of M satisfied p.

ELS'25, Zurich

https://doi.org/10.5281/zenodo.14775258

When we designed the system, we had several goals in mind. The heavy Boolean values should (as much as possible) behave like Boolean values in Boolean contexts, but should contain reflective data for debugging or for providing mathematical semantics. Furthermore, and perhaps most importantly, these Heavy Boolean values should compose with each other through the Boolean operations of conjunction, disjunction, and negation, and through concentric use of multiple quantifiers of uniform or mixed types without loosing critical information.

These ambitious goals provide implementation challenges depending on the programming language, as different languages handle Boolean values differently in primitives such as `and`, `or`, `not`, `if`, *etc.*. In the Scala and Python languages, we have added behavior to the Heavy Boolean objects themselves so that they behave in a reasonable way with the built-in equivalents of `if`, `and`, and `or`. However, in the Clojure and Common Lisp languages, we are not able to augment the behavior of `false` and instead we have created heavy-Boolean-friendly variants of `if`, `and`, and `or`.

On the other hand, using the meta-programming capabilities of the lisp dialects (Common Lisp and Clojure) we were able to create a more expressive `exists` and `forall` than in the non-lisp languages. In the Scala and Python versions of `exists` and `forall`, the user is required to provide redundant information. This redundant information is obviated in the lisp dialects thanks to their meta-programming layers.

2 Contributions

Our contributions in this article are libraries for Clojure, Common Lisp, Python, and Scala implementing the Heavy Boolean protocols.

- Clojure: https://github.com/jimka2001/heavybool/clojure
- Common Lisp: https://github.com/jimka2001/heavybool/common-lisp
- Python: https://github.com/jimka2001/heavybool/python
- Scala: https://github.com/jimka2001/heavybool/scala

3 Quantifiers in Various Languages

All programming languages which we are concerned with have built-in predicate functions which implement universal (1) and existential (2) quantifiers which satisfy DeMorgan's law (3). This law allows any proposition stated in terms of a universal quantifier to be refactored into an equivalent proposition stated in terms of an existential quantifier and vice versa.

$$\textbf{Universal Quantifier:} \quad \forall a \in M, p(a) \tag{1}$$

$$\textbf{Existential Quantifier:} \quad \exists a \in M, p(a) \tag{2}$$

$$\textbf{DeMorgan's Law:} \quad \exists a \in M, p(a) \iff \neg \forall a \in M, \neg p(a) \tag{3}$$

In Sections 3.1 through 3.4, we look at the syntax of quantifiers and DeMorgan's law in four programming languages.

3.1 Clojure

In Clojure the existential quantifier, $\exists a \in M, p(a)$, is implemented by the built-in `some` function as on lines 3 and 2 of Listing 1. The universal quantifier, $\forall a \in M, p(a)$, is expressed by the built-in `every?` function as on lines 7 and 6 of Listing 1. Furthermore, by DeMorgan's law, Equation (3), lines 10, 11, and 12 are equivalent.

Listing 1: Clojure: Existential and Universal Quantifiers

```clojure
;; existential quantifiers
(some p M)
(some (fn [a] (p a)) M)

;; universal quantifiers
(every? p M)
(every? (fn [a] (p a)) M)

;; equivalent expressions: DeMorgan's Law
(some p M)
(not (every? (fn [a] (not (p a))) M))
(not (every? (complement p) M))
```

3.2 Common Lisp

In Common Lisp the existential quantifier, (2), is implemented by the built-in `some` function as on lines 3 and 2 in Listing 2. The universal quantifier, (1), is expressed by the built-in `every` function as on lines 7 and 6 in the same listing. Furthermore, DeMorgan's law, Equation (3), is verified in that the following expressions on lines 10, 11, and 12 are logically equivalent.

Listing 2: Common Lisp: Existential & Universal Quantifiers

```commonlisp
;; existential quantifiers
(some #'p M)
(some (lambda (a) (p a)) M)

;; universal quantifiers
(every #'p M)
(every (lambda (a) (p a)) M)

;; equivalent expressions: DeMorgan's Law
(some #'p M)
(not (every (lambda (a) (not (p a))) M))
(not (every (complement #'p) M))
```

3.3 Python

In Python, the built-in functions `all` and `any` implement the universal and existential quantifiers respectively. An example of the existential quantifier, (2), is shown on line 2 of Listing 3. An example of the universal quantifier, (1), is shown in the same listing on line 5. The `all` and `any` functions are related by DeMorgan's Law in that lines 8 and 9 are equivalent.

Listing 3: Python: Existential and Universal Quantifiers

```python
# existential quantifier
any(p(a) for a in M)

# universal quantifier
all(p(a) for a in M)

# equivalent expressions: DeMorgan's Law
all(p(a) for a in M)
not any(not p(a) for a in M)
```

3.4 Scala

In Scala the existential quantifier is implemented as a built-in method `exists` defined on the `Seq` class (and on other classes), while the universal quantifier is implemented as the built-in method `forall`. The existential quantifier, can be written as on lines 3 and 2 in Listing 4. The universal quantifier can be written as on lines 7 and 6. As in the three previous examples, DeMorgan's law, Equation (3), is expressed once again (lines 11 and 10).

Listing 4: Scala: Existential and Universal Quantifiers

```scala
// equivalent existential quantifiers
M.exists(p)
M.exists{(a) => p(a)}

// equivalent universal quantifiers
M.forall(p)
M.forall{(a) => p(a)}

// equivalent expressions: DeMorgan's Law
M.exists(p)
!M.forall{(a) => !p(a)}
```

4 Associativity Example

In Section 3 we examined the simple use of quantifiers. In each case the quantifier returns a Boolean value which lacks reflective capacity [6]. In this section we look at examples where an unadorned Boolean value does not suffice. In each case we first look at some standard techniques for working around this limitation, and in Section 6 we introduce the *Heavy Boolean* as a generic solution.

In this section we consider a pedagogical mathematical example from abstract algebra. A talented researcher, Helen Wheels[1], is writing some code to detect whether a given binary operation, $\circ$, is associative on a given finite set, M. The proposition to verify is stated succinctly, mathematically as

[1] https://www.youtube.com/watch?v=RSWQ0tkxtdk, in real life Hellen Wheels is a talented comedian who inspires others to excel beyond what some people see as a handicap.

$$\forall a \in M, \forall b \in M, \forall c \in M \ (a \circ b) \circ c = a \circ (b \circ c). \quad (4)$$

For example if $M = \{1, 2, 3\}$ and the operation is subtraction, although other solutions are possible, it would suffice that Helen find the pair $(1, 1, 2)$ because

$$-2 = (1 - 1) - 2 \neq 1 - (1 - 2) = 2\,.$$

4.1 Technique 1: Using Universal Quantifiers

Helen, being a formidable, polyglot programmer, implements Equation (4) in several programming languages as follows:

Listing 5: Clojure: Implementation of Eq (4)

```clojure
(every? (fn [x]
   (every? (fn [y]
      (every? (fn [z] (= (op (op x y) z)
                         (op x (op y z))))
              M))
      M))
   M)
```

Listing 6: Common Lisp: Implementation of Eq (4)

```lisp
(every #'(lambda (x)
  (every #'(lambda (y)
    (every #'(lambda (z)
               (equal (op (op x y) z)
                      (op x (op y z))))
           M))
  M)
```

Listing 7: Python: Implementation of Eq (4)

```python
all(op(op(x,y), z) == op(x, op(y,z))
    for x in M
    for y in M
    for z in M)
```

Listing 8: Scala: Implementation of Eq (4)

```scala
M.forall{(x) =>
  M.forall{(y) =>
    M.forall{(z) =>
      op(op(x,y),z) == op(x,op(y,z))}}}
```

Being an expert programmer, Helen always tests her code thoroughly. In this case she tests on a particular example and finds that it surprisingly returns `false`. Helen does not know if the operation really fails to be associative or whether her implementation has a bug. She knows that for some triple (x, y, z), the operation (as coded) is not associative. How can she determine which (x, y, z) triple serves as a counterexample so she can look more closely to understand the discrepancy?

4.2 Technique 2: Using Existential Quantifiers

In Section 4.1, Helen used the built-in universal quantifiers in each programming language. Thus *true* is returned when the operation is found to be associative. However, Helen's code detected a failure to be associative; *false* was returned. Helen attempts to apply DeMorgan's law (Eq (3)) to her code, inverting the logic so that it returns *true* when the operation is not associative. The advantage of a *true* return value is that Helen obtains information in addition to the Boolean *true*. For example a triple (x, y, z) would be *true*.

Helen exploits the fact that the triple (x, y, z) is *true* in the lisp dialects, Clojure and Common Lisp as shown in Listings 5 and 6. However, she finds that the technique (Listings 7 and 8) fails for Python and Scala.

Listing 9: Clojure Listing 5 with Inverted Logic

```clojure
(some (fn [x]
   (some (fn [y]
      (some (fn [z] (and (not (= (op x (op y z))
                                 (op (op x y) z)))
                         [x y z]))
            M))
      M))
   M)
```

Listing 10: Common Lisp Listing 6 with Inverted Logic

```lisp
(some #'(lambda (x)
  (some #'(lambda (y)
    (some #'(lambda (z)
              (and (not (equal (op x (op y z))
                               (op (op x y) z)))
                   (list x y z)))
          M))
  M)
```

Listing 11: Broken Python Listing 7 with Inverted Logic

```python
# returns True or False, not the counterexample
any(op(op(x,y), z) != op(x, op(y,z)) and [x, y, z]
    for x in M
    for y in M
    for z in M)

# equivalent but more idiomatic
any([x, y, z]
    for x in M
    for y in M
    for z in M
    if op(op(x,y), z) != op(x, op(y,z)))
```

Listing 12: Broken Scala Listing 8 with Inverted Logic

```scala
// This code does not compile
M.exists{(x) =>
  M.exists{(y) =>
    M.exists{(z) =>
      op(op(x,y),z) != op(x,op(y,z)) && (x,y,z)}}}`
```

The Python Listing 11 fails because, as documented, `any` returns `True` or `False` explicitly. In contrast to the lisp dialects, `any` unfortunately does not return the first *true* result it encounters.

The Scala Listing 12 fails to compile, because the function Helen has given to `M.exists...` (line 5) is expected to return a `Boolean`; whereas the tuple `(x,y,z)` is not a `Boolean`.

4.3 Technique 3: Sequence of Counterexamples

Helen, being dedicated and determined to find a solution, tries to refactor the code so that it returns an empty sequence if the operation is associative, otherwise a sequence of counterexamples if the operation fails to be associative.

This technique works for all the languages shown but with a notable disadvantage. Helen is generating exceedingly many counterexamples, when one would suffice to disprove associativity. The search for counterexamples is an n^3 search. Helen would like to terminate the search early once a counterexample is found.

Listing 13: Clojure: Sequence of Counterexamples

```
(for [x M
      y M
      z M
      :when (not (= (op (op x y) z)
                    (op x (op y z))))]
  [x y z])
```

Listing 14: Common Lisp: Sequence of Counterexamples

```
(loop :for x :in M
  :nconc (loop :for y :in M
           :nconc (loop :for z :in M
                    :unless (equal (op (op x y) z)
                                   (op x (op y z)))
                      :collect (list x y z))))
```

Listing 15: Python: Sequence of Counterexamples

```
[(x,y,z) for x in M
         for y in M
         for z in M
         if op(op(x,y), z) != op(x, op(y,z))]
```

Listing 16: Scala: Sequence of Counterexamples

```
for{x <- M
    y <- M
    z <- M
    if op(op(x,y),z) != op(x,op(y,z))
   } yield (x,y,z)
```

4.4 Technique 4: At Most One Counterexample

Not yet admitting defeat, Helen alters Listings 13 through 16 to terminate the loop on finding the first counterexample.

She again considers Listing 13. This expression, thanks to the `for` construct, returns a so-called *lazy-sequence* of all possible counterexamples. Helen tests whether the returned sequence is empty, and if not, calls `first` to obtain an actual counterexample. Unfortunately, Helen finds that so-called lazy-sequences in Clojure are greedily-lazy and sadly Clojure has continued to compute several additional counterexamples even though only the first one is consumed at the call site. Clojure experts do not consider this greedy-laziness to be a problem in practice—claiming that is it *usually* not a problem, which we interpret (via Demorgan's Law) to mean: sometimes *IS* a problem.

Helen, determined more than ever, addresses the problem of greedy-lazy-sequences by employing so-called transducers in Listing 17. This refactoring further exacerbates the extent to which the original code and the debugging code are asymmetric.

Listing 17: Clojure: Return one Counterexample

```
(->> (for [x M
           y M
           z M]
       [x y z])
     (transduce (comp (filter (fn [[x y z]]
                                (not (= (op x (op y z))
                                        (op (op x y) z)))))
                      (take 1))
                conj []))
```

On to the next programming language, Helen refactors Listing 14 using `block`/`return-from` to avoid collecting. This Common Lisp code now returns `nil` if the operation is associative; otherwise it returns a triple, indicating the first counterexample encountered.

Listing 18: Common Lisp: Return one Counterexample

```
(block nil
  (loop
    :for x :in M
    :do (loop
          :for y :in M
          :do (loop :for z :in M
                :unless (equal (op (op x y) z)
                               (op x (op y z)))
                  :do (return-from nil
                        (list x y z))))))
```

For the next programming language, Python, Helen replaces the list comprehension from Listing 15 with a call to `next` Listing 19. The code now returns `None` if the operation is associative; otherwise it returns a triple which is the first counterexample encountered.

Listing 19: Python: Return one Counterexample

```
next(((x,y,z) for x in M
              for y in M
              for z in M
              if op(op(x,y), z) != op(x, op(y,z)),
     None)
```

Finally, Helen attacks the Scala Listing 16. She converts the input sequence `M`, to a lazy list with a call to `.to(LazyList)`. Now, the code in Listing 20 returns a possibly empty, lazy list. The lazy list is empty if the operation is associative, otherwise the call-site should invoke the method `.first` to obtain a counterexample.

Listing 20: Scala: Return one Counterexample

```scala
val lazyM = M.to(LazyList)
for{x <- lazyM
    y <- lazyM
    z <- lazyM
    if op(op(x,y),z) != op(x,op(y,z))
   } yield (x,y,y)
```

4.5 Summary of Standard Techniques

Each of the techniques which Helen has tried solves part of the problem, but there are significant disadvantages.

- In Technique 3 (Section 4.3) Helen is generating exceedingly many counterexamples, when one would suffice to disprove associativity.
- The code in Technique 4 has drastically diverged from Helen's first attempt in Listings 5 through 8.
- The code no longer resembles the elegant Equation 4 which Helen is attempting to verify.

5 Heavy Boolean Semantics

We briefly and tersely describe the semantics of so-called *Heavy Booleans* and in the following sections give examples of their implement in various programming languages.

Let $\mathcal{H}$ be the set called *Heavy Booleans*. Partition $\mathcal{H}$ into $\mathcal{H} = \mathcal{T} \cup \mathcal{F}$. Call $\mathcal{T}$ the *truthy* values and $\mathcal{F}$ the *falsey* values. Let $t \in \mathcal{T}$, $f \in \mathcal{F}$, and $b \in \mathcal{H}$. Let S be an arbitrary set, and $h :: s$ be a finite sequence of elements of S whose first element is h and remaining elements are the sequence s. Let [] denote the empty sequence. Let $p : S \to \mathcal{H}$ be a so-called *heavy predicate* function.

Let $\mathcal{W} \supset \{\emptyset\}$ be a set of so-called *reasons*, and $why : \mathcal{H} \to \mathcal{W}$ be a function for which $why : \mathcal{T} \to \mathcal{W}$ and $why : \mathcal{F} \to \mathcal{W}$ are both bijections. Call $why(t)$ a *witness* and $why(f)$ a *counterexample*. Let $\bot \in \mathcal{F}$ and $\top \in \mathcal{T}$, such that such that $why(\bot) = why(\top) = \emptyset$. The operator $\vdash$ *decorates* a Heavy Boolean with an *additional* reason.

The following axioms hold for unary operator $\neg : \mathcal{T} \to \mathcal{F}$ and $\neg : \mathcal{F} \to \mathcal{T}$, non-commutative operators $\vee : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ and $\wedge : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$, and quantification operators *exists* and *forall*.

$$why(\neg t) = why(t) \tag{5}$$
$$why(\neg f) = why(f) \tag{6}$$
$$f \vee b = b \tag{7}$$
$$t \vee b = t \tag{8}$$
$$f \wedge b = f \tag{9}$$
$$t \wedge b = b \tag{10}$$
$$exists(p, []) = \bot \tag{11}$$
$$forall(p, []) = \top \tag{12}$$
$$exists(p, h :: s) = (p(h) \vdash h) \vee exists(p, s) \tag{13}$$
$$forall(p, h :: s) = (p(h) \vdash h) \wedge forall(p, s) \tag{14}$$

We omit proofs of associativity, well-definedness, DeMorgan's Law, equalities such as: $\neg(\neg h) = h$, $\neg\bot = \top$, $\neg\top = \bot$, and more.

6 Heavy Boolean Values

Helen, being a resourceful programmer, stumbles upon the Heavy Boolean library at https://github.com/jimka2001/heavybool. She discovers that it provides a technique allowing her to verify certain quantified propositions or find counterexamples. She will be able verify Equation (4), all the while assuring that the code express the intent and remain symmetric with the mathematical formalism.

In this section we look at the implementation of Heavy Booleans in each of our target programming languages. A *Heavy Boolean* is an object which *represents true* or *false* in a Boolean context, but has meta data. The primary purpose of a Heavy Boolean is to be generated by universal and existential quantifiers. Because of the way Boolean values are treated in the various programming languages the design of these Heavy Boolean objects is subtly different in each case.

6.1 Clojure

Listing 21: Clojure Quantifier Syntax

```clojure
(+forall [a M]
  (+forall [b M]
    (+forall [c M]
      (= (op (op a b) c)
         (op a (op b c))))))

;; equivalently by macro expansion
(+forall [a M
          b M
          c M]
  (= (op (op a b) c)
     (op a (op b c))))
```

In Clojure, Helen represents Eq (4) as in Listing 21.

If `op` is the `+` (addition) function, then this universal quantifier is satisfied it returns a Heavy Boolean indicating *true*. However, if `op` is the `-` (subtraction) function, then the universal quantifier returns a value indicating *false*, but which also indicates the counterexample: $a = 0$, $b = 0$, and $c = 1$. Notice that since `+forall` and `+exists` are macros, the macro has access to the variable names, `a`, `b`, and `c`, and hence is able to incorporate them into the return value. As will be seen in Sections 6.3 and 6.4, this subtlety is missing from the Python and Scala implementations of Heavy Boolean as those implementations do not include any meta-programming.

Listing 22: Clojure Quantifier Use in Boolean Context

```clojure
(+if (+or (+forall [a M]
            (> a 10))
          (+exists [a M
                    b M]
            (< (* a b) 100)))
  :yes
  :no)
```

In both Clojure and in Common Lisp (Section 6.2) the language dictates which values are *false* and that all other values are *true*. If Helen attempts to use a `heavy-bool` in a Boolean context, it will

be considered *true*. For this reason, we provide wrappers around Boolean operators `+if`, `+or`, `+and`, `+not` and a few others. An example is shown in Listing 22.

Note that lines 17 and 9 of Listing 23 are exactly equivalent as the latter is a macro expansion of the former. Arguably line 9 better emphasizes how Heavy Booleans compose.

Listing 23: Clojure Quantifier Sample Output

```
(def M [0 1 2 3 4])

(+forall [a M
          b M
          c M]
  (= (+ (+ a b) c)
     (+ a (+ b c))))   => [true ()]

(+forall [a M
          b M
          c M]
  (= (- (- a b) c)
     (- a (- b c))))  => [false ({:witness 0, :var a}
                                 {:witness 0, :var b}
                                 {:witness 1, :var c})]

(+forall [a M]
  (+forall [b M]
    (+forall [c M]
      (= (- (- a b) c)
         (- a (- b c))))))) => [false
                                ({:witness 0, :var a}
                                 {:witness 0, :var b}
                                 {:witness 1, :var c})]
```

6.2 Common Lisp

The Heavy Boolean implementation in Common Lisp, Listing 24, resembles that in Clojure, except that Heavy Boolean objects are represented by instances of the CLOS [7] class `heavy-bool` and in particular its two direct subclasses, `heavy-true` and `heavy-false`.

Listing 24: Common Lisp Quantifier Sample Output

```
(setf  M '(0 1 2 3 4))

(+forall (a M
          b M
          c M)
  (= (+ (+ a b) c)
     (+ a (+ b c)))) => #<HEAVY-TRUE T>

(+forall (a M
          b M
          c M)
  (= (- (- a b) c)
     (- a (- b c)))) => #<HEAVY-FALSE F
                                ((:WITNESS 0 :VAR A)
                                 (:WITNESS 0 :VAR B)
                                 (:WITNESS 1 :VAR C))>
```

6.3 Python

In Python we have implemented Heavy Booleans with the class named `HeavyBool` with two subclasses `HeavyTrue` and `HeavyFalse`. The Python language has an interesting feature not available to the lisp dialects discussed above. We define the `__bool__` method on `HeavyBool` so that instances of `HeavyTrue` and `HeavyFalse` behave like the Boolean *true* and *false* respectively in Boolean contexts, such as with `if`, `and`, `or`, *etc.*.

An unfortunate (but documented) feature of the `any` function is that it explicitly returns `True` if any of the generated element is *true* and `False` otherwise. Likewise, `all` returns `False` on encountering a *false* element, otherwise returns `True`. *I.e.*, when Helen calls `any` or `all`, she cannot detect which element was the culprit of early termination. To solve this problem, we have implemented `anyM` and `allM`. The `anyM` function iterates over its input, expecting to find instances of `HeavyBool`, and returns the first instance of `HeavyTrue` it encounters, otherwise returns an undecorated instance of `HeavyFalse`. Analogously, `allM` returns the first `HeavyFalse` instance encountered, otherwise returns an undecorated instance of `HeavyTrue`.

Using `allM`, Helen implements something similar to Listings 23 and 24. The `anyM` functions works analogously.

Listing 25: Python: Existential Quantifier

```
allM(HeavyBool(((a - b) - c) == (a - (b - c)),
               {"a":a, "b":b, "c":c})
     for a in M
     for b in M
     for c in M)  => False[[{'a': 0,
                             'b': 0,
                             'c': 1}]]

# Alternative to annotate only counterexamples
forallM(M, lambda a:
  forallM(M, lambda b:
    # Cannot put arbitrary code here.
    # May only use single expression.
    # Cannot declare local variables.
    forallM(M, lambda c:
       HeavyBool(((a - b) - c) == (a - (b - c))
                ).annotateFalse({"a":a, "b":b, "c":c})))))

=> False[[{'a': 0, 'b': 0, 'c': 1},
          {'witness': 1},
          {'witness': 0},
          {'witness': 0}]]

# python quantifiers use in Boolean context
if allM(HeavyBool(a > 10)
        for a in M) or \
   anyM(HeavyBool(a*b < 100)
        for a in M
        for b in M):
   print("yes")
else:
   print("no")
```

When contrasting Listing 25 line 2 with Listings 23 and 24, we see that Helen must construct the *reason* argument herself

`{"a":a, "b":b, "c":c}` whereas the macro-based lisp implementation of `+forall` and `+exists` auto-construct it.

While the code in Listing 25 lines 1 through 5 resembles corresponding code using the built-in `all` (*e.g.*, Listing 11), each iteration wastefully constructs a `HeavyBool` with a populated `reason` argument. The code in Listing 25 line 17, as maladroit as it is, shows how to annotate the `HeavyBool` only in the counterexample case.

Even if some aspects of the Python implementation of Heavy Booleans are awkward, some parts do seem elegant: *e.g.*, the fact that `HeavyBool` behaves like built-in `bool` with respect to many built-in operators—illustrated in Listing 25 lines 24 through 32. We have not implemented heavy-Boolean-specific versions of `if`, `and`, *etc.*. as was the case in Listing 22.

The `forallM` (and `existsM`) implementation restricts Helen to the egregiously limited `lambda` syntax, prohibiting most coding patterns including prohibition of variable declarations (See Listing 25). This syntax is not imposed on the lisp programmer, as the macros `+exists` and `+forall` allow arbitrary code in the given code body.

Listing 26: Scala Quantifier Syntax

```
1  val M = 0 to 4
2
3  forallM("a",M){(a) =>
4    forallM("b",M){(b) =>
5      forallM("c",M){(c) =>
6        (a - b) - c == a - (b - c)}}}
7
8  => false[(witness->0, tag->a);
9           (witness->0, tag->b);
10          (witness->1, tag->c)]
11
12 // used in Boolean context
13 if (forallM("a",M){(a) =>
14       forallM("b",M){(b) =>
15         forallM("c",M){(c) =>
16           (a + b) + c == a + (b + c)}}})
17   "yes"
18 else
19   "no"
20
21 => "yes"
22
23 (forallM("a", M){(a) => a < 10}
24   && existsM("a", M){(a)=>
25        existsM("b", M){(b) =>
26          a*b < 100}})
27
28 => true[(witness->0, tag->a); (witness->0, tag->b)]
```

6.4 Scala

In Scala we implemented Heavy Booleans as an Algebraic Data Type (ADT) [11] named `HeavyBool`. An ADT in Scala is a class for which the compiler can assume all subclasses are fixed and known at compile time. The two subclasses `HeavyTrue` and `HeavyFalse` implement Heavy Boolean *true* and *false*. The `HeavyBool` class provides methods `forallM` and `existsM` which can be used as shown in Listing 26.

As is seen in Listing 26, the `forallM` method (and also `existsM`) takes a string argument where Helen must specify the variable name. This redundancy is because we are not using any Scala meta-programming [3] to auto-extract the variable name from the code. Meta-programming in Scala is exceedingly complicated, far beyond our expertise, and makes application code incompatible between major Scala compiler releases.

Instances of `HeavyBool` may be used in Boolean contexts such as with `if`, `and`, `or`, *etc.*. as seen in Listing 26 lines 13 through 19.

7 Extended Use Case

The example discussed in Section 3 may not be convincing to programmers who are more concerned with concrete program development. We provide here a second example and solution which illustrates how the Clojure implementation of `heavy-bool` solves an annoying and perhaps serious limitation in unit testing.

Helen Wheels is now attempting to write a unit test using the standard, well-loved `clojure.test` testing framework. She begins by writing the test shown in Listing 27. The test follows the principle of PBT (property based testing) [4, 5] which rather than testing specific examples, instead tests expected invariants of functions based on randomly generated or exhaustively generated input data.

Listing 27: Test Case Using Standard API

```
1 (deftest t-plus-associative
2   (doseq [p1 polynomials
3           p2 polynomials
4           p3 polynomials]
5     (is (sut/==
6           (sut/+ (sut/+ p1 p2) p3)
7           (sut/+ p1 (sut/+ p2 p3)))
8         (format "non-associative: p1=%s\np2=%sp3=%s"
9                 p1 p2 p3))))
```

In Clojure, the `sut/` prefix is used to indicate symbols from the *System Under Test*. The unit test tries to verify that the function `sut/+` is associative. If a counterexample is found, the side-effecting call to `clojure.test/is` (line 5) registers the counter example, but the `doseq` loop continues. Thus, if n is the length of `polynomials`, then worst case the loop will report n^3 many violations. If running interactively, the IDE freezes while trying to construct a formatted string annotating thousands of almost identical counterexamples.

7.1 1st Vain Attempted Fix, `:while`

In the first attempt to fix this problem, Helen evokes the `:while` modifier as shown on line 5 in Listing 28.

This attempt works somewhat, but `:while` does not exactly stop the iteration. Instead, the `:while` modifier causes the inner most loop `p3` to exit, evoking the successive iteration of `p2`. The computation of success values of the expression continues for some

```
(deftest t-example
  (doseq [a (range 10)
          b (range 10)
          :while (is (< (+ a b) 15)
                     (format "a=%s b=%s" a b))]))
```

```
Test Summary
homework.polynomial-test 8 ms
  t-example 8 ms

Tested 1 namespaces in 8 ms
Ran 94 assertions, in 1 test functions
4 failures
cider-test-fail-fast: t


Results

homework.polynomial-test
4 non-passing tests:

Fail in t-example
a=6 b=9
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=7 b=8
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=8 b=7
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=9 b=6
expected: (< (+ a b) 15)
  actual: (not (< 15 15))
```

Figure 1: Example Test Failure

time after the first `false` value of `(is ...)`. The end effect is that we'll have n^2 rather than n^3 many failures.

Listing 28: 1st Vain Attempt to Fix Problem in Listing 27

```
(deftest t-plus-associative
  (doseq [p1 polynomials
          p2 polynomials
          p3 polynomials
          :while (is (sut/==
                      (sut/+ (sut/+ p1 p2) p3)
                      (sut/+ p1 (sut/+ p2 p3)))
                     (format
                       "non-associative: p1=%s\np2=%sp3=%s"
                       p1 p2 p3))]))
```

Figure 1 is a simpler example which shows that the `(doseq ... :while ....)` loop does not abort on the first failure.

7.2 2nd Vain Attempted Fix, `every?`

Helen wants the test to simply fail on the first violation, so she rewrites the code as shown in Listing 29, using the built-in universal quantifier, `every?`.

Listing 29: 2nd Vain Attempt to Fix Problem in Listing 27

```
1  (deftest t-plus-associative-b
2    (is (every? (fn [p1]
3          (every? (fn [p2]
4            (every? (fn [p3]
5              (sut/== (sut/+ (sut/+ p1 p2)  p3)
6                       (sut/+ p1  (sut/+ p2 p3))))
7              polynomials))
8            polynomials))
9          polynomials)
10       ;; regrettably, p1, p2, and p3 are out of scope
11       "WHAT TEXT TO PUT HERE?"))))
```

By pulling the `clojure.test/is` outside the loop, the test fails early, when it discovers one failure. However, Helen cannot construct the second argument of 'is' (line 11) which should indicate the values of `p1`, `p2`, and `p3` constituting the counterexample.

7.3 3rd Vain Attempted Fix, `some`

Helen notices that part of the problem with Listing 29 was that although the execution stops as soon as a counterexample was found, the iterations variables were no longer in scope. She decides to apply DeMorgan's law as was done in Listing 9, and refactor the test, writing Listing 30[2].

Listing 30: 2nd Vain Attempt to Fix Problem in Listing 27

```
1  (deftest t-minus-associative
2    (is (empty?
3          (some (fn [[x y z]]
4                  (when (not (= (op x (op y z))
5                                (op (op x y) z)))
6                    [x y z]))
7                (for [p1 polynomials
8                      p2 polynomials
9                      p3 polynomials]
10               [p1 p2 p3])))))
```

This refactoring leads to a successful detection of violation of associativity by displaying a counterexample.

```
expected: (empty?
           (some
            (fn [[x y z]]
           (when (not (= (op x (op y z)) (op (op x y) z)))
                 [x y z]))
            (for [p1 M p2 M p3 M] [p1 p2 p3])))
  actual: (not (empty? [{0 1} {} {0 1}]))
```

Even though a counterexample is successfully discovered, the cost is that Helen refactored the code so that diverged from the mathematical expression (Eq 4) she was attempting to verify.

[2]Thanks to Ed Bowler aka https://github.com/l0st3d for providing this clever solution

7.4 Proposed Fix, `heavy-bool`

Using the `heavy-bool` library, Helen writes the test shown in Listing 31.

Listing 31: Proposed Fix for Problem in Listing 27

```
(deftest t-plus-associative-b
  (is (hb/+bool
       (hb/+forall [p1 polynomials
                    p2 polynomials
                    p3 polynomials]
          (hb/+heavy-bool (sut/==
                           (sut/+ (sut/+ p1 p1)
                                  p3)
                           (sut/+ p1
                                  (sut/+ p2 p3))))))))
```

If this test fails, we'll see a message such as the following. Cryptic, but all the information is there.

The `:var` and `:witness` tags of the error message indicates that when `p1 = {0 1}` and `p2 = {}` and `p3 = {0 1}` the associativity fails. The `:associative false` key/value pair indicates that it is the associativity check which is failing.

```
Fail in t-plus-associative-b
hb: plus associativity

expected: (hb/+bool
           (hb/+forall
            [p1 polynomials
             p2 polynomials
             p3 polynomials]
           (hb/+heavy-bool
            (sut/==
             (sut/+ (sut/+ p1 p1) p3)
             (sut/+ p1 (sut/+ p2 p3))))))
  actual: (not
           (hb/+bool
            [false
             ({:var p1, :witness {0 1}}
              {:var p2, :witness {}}
              {:var p3, :witness {0 1}})]))
```

8 Conclusion

The existential and universal quantifiers offered by the Heavy Boolean implementations provide useful alternatives to the built-in quantifiers of the languages we have investigated: Clojure, Common Lisp, Python, and Scala. In the lisp dialects the quantifiers `+exists` and `+forall` provide expressions which closely match the mathematical notation, express the intent, and abort an otherwise polynomial complexity search once a witness or counterexample is found. In addition, the quantifiers provided in lisp, provide additional elegance in that they provide useful information such as variable names allowing reflection for debugging or constructing error messages. A limitation of the lisp dialects is that they do not support overloading *false* values, thus the heavy-boolean objects cannot be used in the ordinary Boolean contexts such as `if`, `cond`, `and`, `or`, `not`, *etc.*. Because of this limitation, the packages provide *wrapped* versions such as `+if`, `+and`, *etc.*.

The Python and Scala languages do allow extending Boolean values, including *false*, which allows Heavy Boolean objects to be used in place of built in Boolean primitives such as `if`, `and`, *etc.*. However, the lack of meta programming means the user must provide redundant information such as variable names.

As explained in Section 6.3 the Python implementation imposes strict limitation on user expressiveness which is not an inconvenience experienced by the lisp programmer.

References

[1] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[2] Merrie Bergmann, James Moore, and Jack Nelson. *The Logic Book, Sixth Edition.* McGraw-Hill, 2015. ISBN 978-0-07-803841-9.

[3] Eugene Burmako. Scala macros: let our powers combine! on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320641. doi: 10.1145/2489837.2489840. URL https://doi.org/10.1145/2489837.2489840.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

[5] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351266. URL https://doi.org/10.1145/357766.351266.

[6] J. Malenfant F.-N. Demers. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95, Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995. URL http://fparreiras/papers/reflectionlogicfuncoocomparative.pdf.

[7] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991. ISSN 0001-0782.

[8] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[9] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[11] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. The essence of generalized algebraic data types. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632866. URL https://doi.org/10.1145/3632866.

[12] Guy L Steele Jr, Scott E Fahlman, Richard P Gabriel, David A Moon, Daniel L Weinreb, Daniel G Bobrow, Linda G DeMichiel, Sonya E Keene, Gregor Kiczales, Crispin Perdue, et al. *Common LISP: The Language (2nd Ed.).* Digital Press, Bedford, Massachusetts, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[13] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual.* Network Theory Ltd., 2011. ISBN 1906966141, 9781906966140.