Continuations

CS 105 Assignment

Due Tuesday, October 11, 2022 at 11:59PM

Contents

This assignment is all individual work. There is **no pair programming**.

Overview

Continuations are a key technology in event-driven user interfaces, such as are found in games and in many Web frameworks. For example, continuations are used in JavaScript as "callbacks." Continuations are also used to implement sophisticated control flow, including backtracking. This assignment introduces you to continuations through backtracking search. It also gives you additional experience with higher-order, polymorphic functions. The assignment builds on the previous two assignments, and it adds new ideas and techniques that are described in section~2.10 of *Build, Prove, and Compare*.

Setup

The executable µScheme interpreter is in `/comp/105/bin/uscheme`; if you are set up with `use comp105`, you should be able to run `uscheme` as a command. The interpreter accepts a `-q` ("quiet") option, which

turns off prompting. Your homework will be graded using `uscheme`. When using the interpreter interactively, you may find it helpful to use `ledit`, as in the command

```
ledit uscheme
```

In this assignment, you may find the `&trace` feature especially useful. It is described in the Scheme homework.

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `println`, `print`, `printu`, and `begin` from your vocabulary! If you break this rule for any exercise, you get No Credit for that exercise.** You may find it useful to use `begin` and `println` while debugging, but they must not appear in any code you submit. As a substitute for assignment, use `let` or `let*`.

Except as noted below, do not define helper functions at top level. Instead, use `let` or `letrec` to define helper functions. When you do use `let` to define inner helper functions, avoid passing as parameters values that are already available in the environment.

Your solutions must be valid μScheme; in particular, they must pass the following test:

```
/comp/105/bin/uscheme -q < myfilename > /dev/null
```

without any error messages or unit-test failures. If your file produces error messages, we won't test your solution and you will earn No Credit for functional correctness. (You can still earn credit for structure and organization). If your file includes failing unit tests, you might possibly get some credit for functional correctness, but we cannot guarantee it.

We will evaluate functional correctness by testing your code extensively. **Because this testing is automatic, each function must be named exactly as described in each question. Misnamed functions earn No Credit.**

Testing

Two of the programming problems deal with Boolean formulas, which include four forms of data (variable, `not`, `and`, and `or`). As usual, you'll want to test with all four forms of data. *In addition*, you'll want to test your Boolean formula solver (below) with each of the four forms *underneath* a `not` operator: "not variable," "not not", "not and," and "not or". This is your best chance to make sure that your tests exercise every line of code in your implementation, so you can avoid such issues as calling a function with the wrong number of arguments—which always leads to a failing grade.

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module. It is also OK to alternate between reading-comprehension questions and related homework questions.

Programming and Language Design

You will explore an alternative semantics for `val` (**46**), and you will solve four problems related to Boolean formulas: recognize lists (**L**), recognize formulas (**F**), evaluate formulas (**E**), and solve formulas (**S**). You will also submit test cases for the solver (**T**). Problems **L**, **F**, **E**, and **S** require algebraic laws.

Language-design problem

46. *Operational semantics and language design*. Do all parts of exercise~46 on page~196 of *Build, Prove, and Compare*. Be sure your answer to part (b) loads into `uscheme` and runs without error.

Related reading: Rules for evaluating definitions in section~2.11.4, especially the two rules for VAL.

Representing Boolean formulas

This homework involves Boolean formulas. We represent a formula either as a symbol or a record ("struct"), using the following record definitions:

```
(record not [arg])
(record or  [args])
(record and [args])
```

In the context of these definitions, a formula is one of the following:

- A symbol, which stands for a variable
- A record `(make-not` *f*`)`, where *f* is a formula
- A record `(make-or` *fs*`)`, where *fs* is a list of formulas
- A record `(make-and` *fs*`)`, where *fs* is a list of formulas

The forms of data appropriate to formulas are as follows:

- `x`, where `x` is a symbol
- `(make-not f)`
- `(make-or fs)`
- `(make-and fs)`

Programming problems

L. *Recognizing lists*. Define a higher-order function `list-of?`, which takes two arguments:

- The first argument, `A?`, is a predicate that can be applied to any value.
- The second argument, `v`, is an *arbitrary* µScheme value.

Calling `(list-of? A? v)` returns a Boolean that is `#t` if `v` is a list of values, each of which satisfies `A?`. Otherwise, `(list-of? A? v)` returns `#f`.

Hints:

- The forms of data for arbitrary µScheme values can be deduced from the definition of values in Figure 2.1 on page page~93.

  Function `list-of?` must correctly handle fully general S-expressions like `(cons 'CS 105)`.

- Provided `A?` is a good predicate, `(list-of? A? v)` never causes a checked run-time error, no matter what `v` is. This property distinguishes `list-of?` from `all?`.
- Every one of the primitive type predicates, like `symbol?` and `function?`, is a good predicate to pass to `list-of?`.
- For testing, I encourage you to define and use the following additional predicate:

  ```
  (define value? (_) #t) ;; tell if the argument is a value
  ```

  You can then identify any list of values with `(list-of? value? v)`.
- Test with several different predicates. For each one, write `check-assert` tests with both true and false results.
- Remember that for every A, the empty list is a list of A.

Related reading: In *Build, Prove, and Compare*, section~2.2, especially figure 2.1. For a refresher about list structure, the second *Lesson in Program Design* (Scheme values).

Laws: Function `list-of?` needs algebraic laws. The last law may include the side condition "`v` has none of the forms above."

Design techniques for type predicates

Functions `list-of?` (in problem L above) and `formula?` (in problem F below) are both examples of *type predicates*. Other examples include primitive functions like `number?`, `symbol?`, and `null?`. In general, a type predicate is a function that takes *any* value and returns true or false. There are two ways to approach the design of such a predicate, through its algebraic laws:

1. You can break down the "any value" input into all of its forms, as described in figure 2.1 on page~93, and write one law for each form. This approach is exactly the approach you've used to design other problems, but it often results in laws and code that feel bloated and unnatural.
2. You can break down the *values you are looking for* into all of their forms, and write an algebraic law for each form. **You must then add a final law** that matches *any* input form, with a side condition saying that the argument has none of the forms of the previous laws. The right-hand side of that final law is `#f`.

The second approach is preferable, but you must **avoid a common mistake**: if you forget to add that final law, your type predicate might not do the right thing when presented with an unexpected value.

F. *Recognizing formulas*. Define a function `formula?`, which when given an *arbitrary* µScheme value, returns `#t` if the value represents a Boolean formula and `#f` otherwise. Use the definition of formulas shown above.

A straightforward solution that uses `if` expressions and "how were you formed?" questions takes 10 lines of µScheme. If you want to get clever with short-circuit operators or `cond`, you can cut that in half.

Hints:

- Function `formula?` must analyze every possible µScheme value, no matter what its form, and must always return `#t` or `#f`. Every possible form of µScheme value must therefore be handled by some case. (If it turns out that one case can handle multiple forms of input, that is OK. It is even good practice.)
- Function `formula?` must identify, out of all possible values, which ones are formulas. Every possible form of formula must therefore be handled by a case that returns `#t`.

- Just because some clown puts a list of values in a record, it doesn't mean that the record holds a list of formulas. If you need to confirm the presence of a list of formulas, try using function `list-of?` with `formula?`—that's what it is for.
- Because `formula?` must analyze every possible μScheme value, it is not possible to violate its contract!

Related reading: Section~2.2, which starts on page~93. The definition of `equal?` in section 2.3. The definition of *LIST(A)* in section 2.4.2.

Laws: Function `formula?` needs algebraic laws. I encourage you to use the second approach to design as sketched in the box above—this approach will enable you to focus your laws on the interesting forms.

E. *Evaluating formulas*. Define a function `eval-formula`, which takes two arguments: an S-expression f and an environment `env`.

- The S-expression f is meant to represent a Boolean formula; if f does not represent a formula, `eval-formula` must halt with a checked run-time error.
- The environment `env` is an association list in which each key is a symbol and each value is a Boolean. When f is a formula, the caller must guarantee that every variable of f is bound in `env`.

Assuming that f does represent a formula, if f is satisfied in environment `env`, then `(eval-formula` f `env)` returns `#t`; if f is not satisfied, `eval-formula` returns `#f`. Evaluation is defined by induction:

- The result of evaluating a symbol is the result of looking that symbol up in the environment.
- The result of evaluating a record `(make-not` f`)` is the complement of the result of evaluating f.
- The result of evaluating a record `(make-or` *fs*`)` is true if and only if the list of formulas *fs* contains a formula that evaluates to true.
- The result of evaluating a record `(make-and` *fs*`)` is true if and only if every formula in the list of formulas *fs* evaluates to true.

Our model solution is 10 lines, and it is structurally similar to a straightforward implementation of `formula?`.

You must document your solution with algebraic laws. As usual, we recommend you write the laws before you write the code.

Related reading: The initial basis of μScheme. For guidance on the structure of an evaluator, the Impcore evaluator or the μScheme evaluator in chapter 1 or 2. (You could also look at the implementation of a μScheme evaluator *in μScheme*, which you would find in the book's online Supplement, in section~E.1.4, which starts on page~S131, but that section has so much detail that it may be easier just to figure out on your own how to structure an evaluator written in μScheme.)

Laws: Function `eval-formula` needs algebraic laws. The only permissible side condition should be something like "where `x` is a symbol."

Your laws should cover only those inputs for which `eval-formula`'s contract calls for it to return `#t` or `#f`. Do not include laws for malformed formulas or for any inputs that violate the contract. (In the code, you must include a case that causes a checked run-time error when the formula is malformed. But do not include any cases for inputs that violate the contract.)

T. *Testing SAT solvers*. Create three test cases to test solutions to problem **S**. Your test cases will be represented by six `val` bindings, to variables `f1`, `s1`, `f2`, `s2`, `f3`, and `s3`.

- Value `f1` should be a Boolean formula as described above. For example,

  ```
  (val f1 (make-not 'x))
  ```

  would be acceptable.

- Value `s1` should be an association list that represents a satisfying assignment for formula `f1`. If no satisfying assignment exists, value `s1` should be the symbol `no-solution`. For example,

  ```
  (val s1 '((x #f)))
  ```

  solves formula `f1` above.

- Values `f2`, `s2`, `f3`, and `s3` are similar: two more formulas and their respective solutions.

As another example, if I wanted to code the test formula

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y \vee \neg z),$$

I might write

```
(val f1 (make-and
           (list3 (make-or (list3 'x 'y 'z))
                  (make-or (list3 (make-not 'x) (make-not 'y) (make-not 'z)))
                  (make-or (list3 'x 'y (make-not 'z))))))
(val s1 '((x #t) (y #f)))
```

As a second test case, I might write

```
(val f2 (make-and (list2 'x (make-not 'x))))  ; x and not x
(val s2 'no-solution)
```

Put your test cases into a file `solver-tests.scm`, which should you create initially using the template.

In comments in your test file, explain *why* these particular test cases are important—your test cases must not be too complicated to be explained. Consider different combinations of the various Boolean operators.

Design test cases that will find bugs in solvers. (We will run every submitted solver on every submitted test case.)

Related reading: The example formulas, satisfying assignments, and a formula without a solution on pages 141 and 142 (at the very end of section 2.10.2).

Laws: This problem includes only test cases and results, not any code. No laws are needed.

S. *SAT solving using continuation-passing style*. Write a function `find-formula-true-asst`, which searches for a *satisfying* assignment—that is, a mapping of variables to Booleans that makes the formula true. Remember De Morgan's laws, one of which is mentioned on page~133.

Function `find-formula-true-asst` must take three parameters: a formula, a failure continuation, and a success continuation. A call to

```
(find-formula-true-asst f fail succ)
```

searches for an assignment that satisfies formula `f`. If it finds a satisfying assignment, it calls `succ`, passing both the satisfying assignment (as an association list) and a resume continuation. If it fails to find a satisfying assignment, it calls `fail`. Notes:

- The failure continuation does not accept any arguments.
- The success continuation accepts two arguments: the first is the current (and perhaps partial) solution, and the second is a resume continuation. A resume continuation, like a failure continuation, does not accept any arguments.
- Formulas may be nested with one kind of operator under another.

Our model solution to this exercise is under 50 lines of μScheme.

You'll be able to use the ideas in section~2.10.2, but not the code. Instead, try using `letrec` to define the following mutually recursive functions:

- Calling `(find-formula-asst formula bool cur fail succeed)` extends assignment `cur` to find an assignment that makes the single `formula` equal to `bool`.
- Calling `(find-all-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes every formula in the list `formulas` equal to `bool`.
- Calling `(find-any-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes any one of the `formulas` equal to `bool`.
- Calling `(find-formula-symbol x bool cur fail succeed)`, where `x` is a symbol, does one of three things:
  - If `x` is bound to `bool` in `cur`, succeeds with environment `cur` and resume continuation `fail`
  - If `x` is bound to `(not bool)` in `cur`, fails
  - If `x` is not bound in `cur`, extends `cur` with a binding of `x` to `bool`, then succeeds with the extended environment and resume continuation `fail`

In all the functions above, `bool` is `#t` or `#f`. Before defining each of the functions, complete the following algebraic laws:

```
(find-formula-asst x            bool cur fail succeed) == ...,
                                                  where x is a symbol
(find-formula-asst (make-not f)  bool cur fail succeed) == ...
(find-formula-asst (make-or  fs) #t   cur fail succeed) == ...
(find-formula-asst (make-or  fs) #f   cur fail succeed) == ...
(find-formula-asst (make-and fs) #t   cur fail succeed) == ...
(find-formula-asst (make-and fs) #f   cur fail succeed) == ...

(find-all-asst '()         bool cur fail succeed) == ...
(find-all-asst (cons f fs) bool cur fail succeed) == ...

(find-any-asst '()         bool cur fail succeed) == ...
(find-any-asst (cons f fs) bool cur fail succeed) == ...

(find-formula-symbol x bool cur fail succeed) == ..., where x is not bound in cur
(find-formula-symbol x bool cur fail succeed) == ..., where x is bool in cur
(find-formula-symbol x bool cur fail succeed) == ..., where x is (not bool) in cur
```

Include the completed laws in your solution.

The following unit tests will help make sure your function has the correct interface:

```
(check-assert (function? find-formula-true-asst))     ; correct name
(check-error (find-formula-true-asst))                ; not 0 arguments
(check-error (find-formula-true-asst 'x))             ; not 1 argument
(check-error (find-formula-true-asst 'x (lambda () 'fail)))   ; not 2 args
(check-error
  (find-formula-true-asst 'x (lambda () 'fail) (lambda (c r) 'succeed) 'z)) ; not 4 args
```

These additional checks also probe the interface, but they require at least a little bit of a solver—enough so that you call the success or failure continuation with the right number of arguments:

```
(check-error (find-formula-true-asst 'x (lambda () 'fail) (lambda () 'succeed)))
    ; success continuation expects 2 arguments, not 0
(check-error (find-formula-true-asst 'x (lambda () 'fail) (lambda (_) 'succeed)))
    ; success continuation expects 2 arguments, not 1
(check-error (find-formula-true-asst
                  (make-and (list2 'x (make-not 'x)))
                  (lambda (_) 'fail)
                  (lambda (_) 'succeed)))
    ; failure continuation expects 0 arguments, not 1
```

And here are some more tests that probe if you can solve a few simple formulas, and if so, if you can call the proper continuation with the proper arguments.

```
(check-expect   ; x can be solved
   (find-formula-true-asst 'x
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'succeed)

(check-expect   ; x is solved by '((x #t))
   (find-formula-true-asst 'x
                           (lambda () 'fail)
                           (lambda (cur resume) (find 'x cur)))
   #t)

(check-expect   ; (make-not 'x) can be solved
   (find-formula-true-asst (make-not 'x)
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'succeed)

(check-expect   ; (make-not 'x) is solved by '((x #f))
   (find-formula-true-asst (make-not 'x)
                           (lambda () 'fail)
                           (lambda (cur resume) (find 'x cur)))
   #f)
```

```
(check-expect   ; (make-and (list2 'x (make-not 'x))) cannot be solved
   (find-formula-true-asst (make-and (list2 'x (make-not 'x)))
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'fail)
```

All the tests can be downloaded. Once downloaded, they can be run at any time with

```
-> (use solver-interface-tests.scm)
```

This problem is the most satisfying problem on the assignment. (I'll see myself out.)

Related reading: Section 2.10 on continuation passing, especially the CNF solver in section 2.10.2.

Laws: Complete the laws given in the problem, and include the completed laws with your solution. Place laws for each helper function near the definition of that function.

What and how to submit

You must submit four files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - The identifying numbers (or letters) of the problems that you solved
- A PDF file `semantics.pdf` containing the solution to Exercise **46**. If you already know LaTeX, by all means use it. Otherwise, write your solution by hand and scan it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.

  N.B. Part of your solution to Exercise 46 includes μScheme code, which we ask you to compile to make sure that it works. We nevertheless want you to include the code in your PDF along with your semantics and your explanation—*not* in one of the other files.
- A file `solution.scm` containing the solutions to Exercises **L**, **F**, **E**, and **S**. Each function should be accompanied by a brief contract, algebraic laws as specified, and unit tests. Precede each solution by a comment that looks like something like this:

  ```
  ;;
  ;; Problem L
  ;;
  ```
- A file `solver-tests.scm` containing the definitions of formulas `f1`, `f2`, and `f3` and the definitions of solutions `s1`, `s2`, and `s3`, which constitutes your answer to Exercise **T**.

As soon as you have the files listed above, run `submit105-continuations` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Avoid common mistakes

It's a common mistake to define a version of `list-of?` or `formula?` that causes a checked run-time error on some inputs. These functions must always return Booleans.[1]

[1] Function `list-of?` requires a good predicate, but I've never seen a student make a mistake with the predicate.

The most common mistakes on this assignment have to do with the Boolean-formula solver in problem **S**. They are

- It's easy to handle fewer cases than are actually present in the exercise. You can avoid this mistake by considering all ways the operators `and`, `or`, and `not` can be combined pairwise to make formulas.
- It's easy to write near-duplicate code that handles essentially similar cases multiple times. This mistake is harder to avoid; I recommend that you look at your cases carefully, and if you see two pieces of code that look similar, try abstracting the similar parts into a function.
- It's easy to write code with the wrong interface—but if you use the unit tests above, they should help.