Continuations

COMP 105 Assignment

Due Tuesday, February 27, 2018 at 11:59PM

Contents

Overview 1
Setup 1
Dire Warnings 2
Reading Comprehension (10 percent) 2

Programming and Language Design (90 percent) 4
  Language-design problem ........................................... 4
  Representing Boolean formulas ...................................... 4
  Programming problems .............................................. 4

What and how to submit 8

Avoid common mistakes 9

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

Overview

The purpose of this assignment is to give you additional experience with higher-order, polymorphic
functions and to give you practice using continuations for a backtracking search problem. The assign-
ment 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 interac-
tively, you may find it helpful to use ledit, as in the command
In this assignment, you may find the $\texttt{\&trace}$ feature especially useful. It is described in the Scheme homework\textsuperscript{1}.

**Dire Warnings**

The μScheme programs you submit must not use any imperative features. **Banish** \texttt{set}, \texttt{while}, \texttt{println}, \texttt{print}, \texttt{printu}, and \texttt{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 \texttt{begin} and \texttt{println} while debugging, but they must not appear in any code you submit. As a substitute for assignment, use \texttt{let} or \texttt{let*}.

**Except as noted below, do not define helper functions at top level.** Instead, use \texttt{let} or \texttt{letrec} to define helper functions. When you do use \texttt{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 be exactly as described in each question. Misnamed functions earn No Credit.**

**Reading Comprehension (10 percent)**

These questions are meant to guide you through the readings that will help you complete the assignment. Keep your answers brief and simple.

As usual, you can download the questions\textsuperscript{2}.

1. Read section 2.12.3, which starts on page 154. Assume $y$ is in the domain of $\rho$. Recall that evaluating a lambda in some environment creates a closure using that same environment. What is the result of the call to $f$ in the program below? Pick either A or B.

```
(val f (lambda () y))
(val y 10)
(f)
```

(A) It returns 10
(B) An error is raised: Run-time error: name y not found

---

\textsuperscript{1}scheme.html#diagnostic-tracking
\textsuperscript{2}/cqts.continuations.txt
Assume $y$ is not in the domain of $p$. What is the result of the call to $f$ in the same program? Pick either A or B.

(A) It returns 10
(B) An error is raised: Run-time error: name $y$ not found

You are ready to start problem 44.

2. Set aside an hour to study the conjunctive-normal-form solver in section 2.10.1, which starts on page 140. This will help you a lot in solving problem S.

(a) Look at code chunk 144a on page 144. In English, describe how (one-solution $f$) produces the answer $((x \ #t) \ (y \ #f))$. Walk through each function call, what the input to the function is, how the input is processed, and what the output of the function call is.

It will help us read your answer and should help you answer this question if each function call is formatted similar to the examples below. But if the formatting gets too cumbersome just make sure you keep the four key parts of each call (function, input, processing, and output) separate. You can use the examples below to start your answer for one-solution.

function: one-solution
input: formula is $((x \ y \ z) \ ((not \ x) \ (not \ y) \ (not \ z)) \ (x \ y \ (not \ z)))$
processing: none
output: what find-cnf-true-assignment returns for the whole formula with no current solutions, a failure continuation that returns no-solution, and a success continuation that returns the first solution

function: find-cnf-true-assignment
input: the arguments mentioned in the previous output, where disjunctions is bound to the formula
processing: disjunctions is not null, so this function calls find-disjunction-true-assignment
output: what find-disjunction-true-assignment returns for the first disjunction $((x \ y \ z)$ with no current solutions, the same failure continuation, and a success continuation that checks the remaining disjunctions $(((not \ x) \ (not \ y) \ (not \ z)) \ (x \ y \ (not \ z)))$. If checking the remaining disjunctions fails we want to try to find another solution to $(x \ y \ z)$, so we pass resume as the failure continuation; otherwise if checking succeeds with an answer then that answer is good for the entire search, so we pass succeed as the success continuation

(b) Look at code chunk 144c. As you did with 144a, describe how (one-solution '((x) ((not x)))) produces the answer no-solution. Format this answer like you did for the previous part.

You are ready to start problem S.
Programming and Language Design (90 percent)

You will explore an alternative semantics for \texttt{val} (44), and you will solve three problems related to Boolean formulas: recognize a formula (F), evaluate a formula (E), and a solve a formula (S). You will also submit test cases for the solver (T). Problems F and E require algebraic laws.

Language-design problem

44. Operational semantics and language design. Do all parts of exercise 44 on page 222 of Build, Prove, and Compare. Be sure your answer to part (b) compiles and runs under uscheme.

Related reading: Rules for evaluating definitions in section 2.12.3, 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:

\begin{verbatim}
(record not [arg])
(record or [args])
(record and [args])
\end{verbatim}

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

Programming problems

F. Recognizing formulas. Define a function \texttt{formula?}, which when given an arbitrary S-expression, returns \#t if the S-expression represents a Boolean formula and \#f otherwise. Follow the above definition of formulas exactly. (In case mixing records and S-expressions doesn’t sound proper, recall that record operations in \( \mu \)-Scheme are syntactic sugar for operations S-expressions. This means passing one of your formula records to \texttt{formula?} is within the function’s contract.)

A straightforward solution that uses if expressions and “how were you formed?” questions takes 10 lines of \( \mu \)-Scheme. If you want to get clever with short-circuit operators, you can cut that in half.

Your solution must include algebraic laws. If possible, supplement your algebraic laws with a case at the end saying, “When none of the laws above applies, return (you fill in this part).” This supplement will obviate the need to define a law for every possible form of S-expression—you can focus your laws on just the interesting forms.

Related reading: The definition of \texttt{equal?} in section 2.3. The definition of \( LIST(A) \) in section 2.6.
E. Evaluating formulas. Define a function eval-formula, which takes two arguments: a formula \( f \) and an environment \( env \). If the formula is satisfied in the given environment, \((\text{eval-formula} \ f \ \text{env})\) returns \#t\; otherwise it 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 \((\text{make-not} \ f)\) is the complement of the result of evaluating \( f \).
- The result of evaluating a record \((\text{make-or} \ \text{fs})\) is true if and only if the list of formulas \( \text{fs} \) contains a formula that evaluates to true.
- The result of evaluating a record \((\text{make-and} \ \text{fs})\) is true if and only if every formula in the list of formulas \( \text{fs} \) evaluates to true.

It is part of the contract of \((\text{eval-formula})\) that \((\text{eval-formula} \ f \ \text{env})\) may be called only when every variable in formula \( f \) is bound in \( env \).

My solution is 10 lines, and it is structurally similar to my 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. And if you are uncertain about structure, the implementation of either the Impcore evaluator or the μScheme evaluator. (You could also look at the implementation of a μScheme evaluator in μScheme, which you would find in section 2.15.4, which starts on page 180, 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.)

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

- Value \texttt{f1} should be a Boolean formula as described above. For example,
  \[
  \text{(val f1 (make-not 'x))}
  \]
  would be acceptable.
- Value \texttt{s1} should be an association list that represents a satisfying assignment for formula \texttt{f1}. If no satisfying assignment exists, value \texttt{s1} should be the symbol \texttt{no-solution}. For example,
  \[
  \text{(val s1 '((x #t) (y #f)))}
  \]
  solves formula \texttt{f1} above.
- Values \texttt{f2}, \texttt{s2}, \texttt{f3}, and \texttt{s3} are similar: two more formulas and their respective solutions.

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

\[
(x \lor y \lor z) \land (\neg z \lor \neg y \lor \neg z) \land (x \lor y \lor \neg z),
\]

I might write

\[
\text{(val f1 (make-and}
\text{(list3 (make-or (list3 'x 'y 'z)))}
\text{(make-or (list3 (make-not 'x) (make-not 'y) (make-not 'z)))}
\text{(make-or (list3 'x 'y (make-not 'z)))))})
\]

\[
\text{(val s1 '(((x #t) (y #f))}}
\]

5
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 the template at http://www.cs.tufts.edu/comp/105/homework/sat_solver_template.scm.

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 and satisfying assignments on page 144 (at the very end of section 2.10.1).

S. SAT solving using continuation-passing style. Write a function find-formula-true-asst which, given a satisfiable formula in the form, finds 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 131.

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.

My solution to this exercise is under 50 lines of μScheme.

You’ll be able to use the ideas in section 2.10.1, 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:

\textsuperscript{3}/sat_solver_template.scm
– 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, we recommend completing 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 (procedure? 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)

You can download all the tests⁴. You can run them at any time with

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

This problem is (forgive me) the most satisfying problem on the assignment.

Related reading: Section 2.10 on continuation passing, especially the CNF solver in section 2.10.1.

What and how to submit

You must submit five files:

- A README file containing
  - The names of the people with whom you collaborated

---

⁴/solver-interface-tests.scm
– The numbers of the problems that you solved

• A cqs.continuations.txt containing the reading-comprehension questions with your answers edited in

• A PDF file semantics.pdf containing the solution to Exercise 44. 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 44 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 F, E, and S. You must precede each solution by a comment that looks like something like this:

;;
;; Problem F
;;

• 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

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.

---

5. cqs.continuations.txt