HW05

Homework 5: Symbolic Evaluation of Boolean Expressions

Due: Wednesday, Oct 14, 2020 at 11:59PM

200 pts.

Overview

Write a Racket function `reduce` that reduces boolean expressions (represented in Racket notation) to simplified form. For the purposes of this assignment, boolean expressions are Racket expressions constructed from:

- the boolean constants `true` and `false` ;
- boolean variables (represented by symbols other than `true`, `false`, `not`, `and`, `or`, `implies`, and `if`) that can be bound to either `true` or `false`.
- the unary operator `not` .
- the binary operators `and`, `or`, and `implies`, and
- the ternary operator `if`.

The course staff is providing functions `parse` and `unparse` in the file `parse.rkt` that convert boolean expressions in Racket notation to a simple inductively defined type called `BoolExp` and vice-versa. The coding of `parse` and `unparse` is not difficult, but it is tedious (like most parsing) so the course staff is providing this code rather than asking students to write it. The Racket primitive `read: -> RacketExp` is a procedure of no arguments that reads a Racket expression from the console. DrRacket pops up an input box to serve as the console when `(read)` is executed.

These parsing functions rely on the following Racket data definitions. Given

```
(define-struct Not (arg))
(define-struct And (left right))
(define-struct Or (left right))
(define-struct Implies (left right))
(define-struct If (test conseq alt))
```

a `BoolExp` is either:

- a boolean constant `true` and `false` ;
- a symbol S representing a boolean variable;
- `(make-Not X)` where X is a `BoolExp`;
- `(make-And X Y) where` X and Y are `BoolExps`;
- `(make-Or X Y) where` X and Y are `BoolExps`;
- `(make-Implies X Y)` where X and Y are `BoolExps`; or
- `(make-If X Y Z)` where X , Y , and Z are `BoolExps`.

A BoolRacketExp is either:

- a boolean constant `true` or `false`;
- a symbol S;
- `(list 'not X)` where X is a `BoolRacketExp` ;
- `(list op X Y)` where op is `'and` , `'or` , or `'implies` where X and Y are `BoolRacketExps`;
- `(list 'if X Y Z)` where X, Y, and Z are `BoolRacketExps`.

The provided functions `parse` and `unparse` have the following signatures.

```
parse: BoolRacketExp -> BoolExp
unparse: BoolExp -> BoolRacketExp
```

The course staff is also providing a very simple test file\ for the `eval` and `reduce` functions and a file\ containing a sequence of raw input formulas (to be parsed by `parse` function in parse.ss\). A good solution to this problem will include much more comprehensive test data for all functions, including some much larger test cases for `reduce`. The `normalize` function is difficult to test on large data because the printed output for some important normalized trees (represented as DAGs (Directed Acyclic Graphs) in memory) is so large.

Given a parsed input of type `BoolExp` , the simplification process consists of following four phases:

- Conversion to `If` form implemented by the function `convertToIf`.
- Normalization implemented by the function `normalize`.
- Symbolic evaluation implemented by the function `eval`.
- Conversion back to conventional `boolean` form implemented by the function `convertToBool`.

A description of each of these phases follows. The `reduce` function has type `BoolRacketExp -> BoolRacketExp`.

Conversion to `if` form

A boolean expression (`BoolExp`) can be converted to `if` form (a boolean expression where the only constructor is `make-If`) by repeatedly applying the following rewrite rules in any order until no rule is applicable.

```
(make-Not  X)           =>        (make-If  X  false true)
(make-And  X  Y)        =>        (make-If  X  Y  false)
(make-Or  X  Y)                =>        (make-If  X  true  Y)
(make-Implies  X  Y)        =>        (make-If  X  Y  true)
```

In these rules, `X` and `Y` denote arbitrary `BoolExps}.` The conversion process always terminates (since each rewrite strictly reduces the number of logical connectives excluding `make-If` and yields a unique answer independent of the order in which the rewrites are performed. This property is called the Church-Rosser property, after the logicians (Alonzo Church and Barkley Rosser) who invented the concept.

Since the reduction rules for this phase are Church-Rosser, you can write the function `convertToIf` using simple structural recursion. For each of the boolean operators `And`, `Or`, `Not`, `Implies`, and `If`, reduce the component expressions first and then applying the matching reduction (except for `If` for which there is no top-level reduction).

The following examples illustrate the conversion process:

```
(check-expect  (convertToIf (make-Or (make-And 'x 'y) 'z))    (make-If (make-If 'x 'y false) true 'z))
(check-expect  (convertToIf (make-Implies 'x (make-Not 'y))   (make-If 'x (make-If 'y false  true) true))
```

We suggest simply traversing the tree using the structural recursion template for type `BoolExp` and converting all structures (other than `If`) to the corresponding `if` structures.

Write an inductive data definition and template for boolean formulas in `if` form, naming this type `ifExp`. (Note: `make-If` is the only constructor, other than variables and constants, for `ifExp`.

The provided function `parse: input -> BoolExp` takes a Racket expression and returns the corresponding `BoolExp`.

Normalization

An `ifExp` is *normalized* iff every sub-expression in `test` position is either a variable (symbol) or a constant (`true` or `false`). We call this type `NormIfExp` .

For example, the `ifExp` (`make-If (make-If X Y Z) U V)`) is not a `NormIfExp` because it has an `If` construction in `test` position. In contrast, the equivalent `ifExp` (`make-If X (make-If Y U V) (make-If Z U V)`) is normalized and hence is an `NormIfExp`.

The normalization process, implemented by the function `normalize: ifExp -> NormIfExp` eliminates all `if` constructions that appear in `test` positions inside `if` constructions. We perform this transformation by repeatedly applying the following rewrite rule (to any portion of the expression) until it is inapplicable:

```
(make-If  (make-If  X  Y  Z)  U  V)  =>         (make-If  X  (make-If  Y  U  V) (make-If Z  U  V)).
```

This transformation always terminates and yields a unique answer independent of the order in which rewrites are performed. The proof of this fact is left as an optional exercise.

In the `normalize` function, it is critically important not to duplicate any work, so the order in which reductions are made really matters. Do **NOT** apply the normalization rule above unless `U` and `V` are already normalized, because the rule duplicates both `U` and `V`. If you reduce the `consequent` and the `alternative` (`U` and `V` in the left hand side of the rule above) before reducing the `test`, `normalize` runs in linear time (in the number of nodes in the input); if done in the wrong order it runs in exponential time in the worst case. (And some of our test cases will exhibit this worst case behavior.)

Hint: define a sub-function head-normalize that takes three `NormIfExps` `X`, `Y`, and `Z` and constructs a `NormIfExp` equivalent to `(makeIf X Y Z)`. This help function processes `X` because the `test` position must be a variable or a constant, yet `X` can be an arbitrary `NormIfExp`. In contrast, `(head-normalize X Y Z)` never even inspects `Y` and `Z` because they are already normalized and the normalizing transformations performed in `head-normalize` never place these expressions in `test` position.

The following examples illustrate how the `normalize` and `head-normalize` functions behave:

```
(check-expect (head-normalize 'x 'y 'z) (make-If 'x 'y 'z))
(check-expect (head-normalize true 'y 'z) (make-If true 'y 'z))
(check-expect (head-normalize false 'y 'z) (make-If false 'y 'z))
(check-expect (head-normalize (make-If 'x 'y 'z) 'u 'v) (make-If 'x (make-If 'y 'u 'v) (make-If 'z 'u 'v)))
(check-expect (head-normalize (make-If 'x (make-If 'yt 'yc 'ya) (make-If 'zt 'zc 'za)) 'u 'v)
(make-If 'x (make-If 'yt (make-If 'yc 'u 'v) (make-If 'ya 'u 'v)) (make-If 'zt (make-If 'zc 'u 'v) (make-If 'za
'u 'v))))

(check-expect (normalize true) true)
(check-expect (normalize false) false)
(check-expect (normalize 'x) 'x)
(check-expect (normalize (make-If 'x 'y 'z)) (make-If 'x 'y 'z))
(check-expect (normalize (make-If (make-If 'x 'y 'z) 'u 'v)) (make-If 'x (make-If 'y 'u 'v) (make-If 'z 'u 'v)))
```

Once a large formula has been normalized, do not try to print it unless you know that the formula is small! The printed form can be exponentially larger than the internal representation (because the internal representation can share subtrees).

Before you start writing `normalize`, write the template corresponding to the inductive data definition of `NormIfExp`.

Symbolic Evaluation

The symbolic evaluation process, implemented by the function `eval: NormIfExp environment -> NormIfExp`, reduces a `NormIfExp` to simple form. In particular, it reduces all tautologies (expressions that are always true) to `true` and all contradictions (expressions that are always false) to `false`.

Symbolic evaluation applies the following rewrite rules to an expression until none is applicable (with one exception discussed below):

```
(make-If  true  X  Y)           =>        X
(make-If  false  X  Y)           =>        Y
(make-If  X  true  false)  =>        X
(make-If  X  Y  Y)            =>        Y
(make-If  X  Y  Z)           =>        (make-If  X  Y[X <- true\]  Z[X <- false])
```

The notation `M[X <- N]` means `M` with all occurrences of the symbol `X` replaced by the expression `N`. It is very costly to actually perform these substitutions on `NormIfExp` data. To avoid this computational expense, we simply maintain a list of bindings which are pairs consisting of symbols (variable names) and boolean values {`true`, `false`. The following data definition definition formally defines the `binding` type.

A `binding` is a pair `(make-binding s v)` where `s` is a symbol (a variable) and `v` is a boolean value (an element of { `true`, `false` }.

An `environment` is a `binding-list`.

When the `eval` function encounters a variable (symbol), it looks up the symbol in the environment and replaces the symbol it's boolean value if it exists.

These rewrite rules do not have the Church-Rosser property. The last two rewrite rules are the spoilers; the relative order in which they are applied can affect the result in some cases. However, the rewrite rules do have the Church-Rosser property on expressions which are tautologies or contradictions.

If the final rule is applied only when `X` actually occurs in either `Y` or `Z`, then the symbolic evaluation process is guaranteed to terminate. In this case, every rule either reduces the size of the expression or the number of variable occurrences in it.

We recommend applying the rules in the order shown from the top down until no more reductions are possible (using the constraint on the final rule). Note that the last rule should only be applied once to a given subexpression.

Conversion to Boolean Form

The final phase converts an expression in (not necessarily reduced or normalized) `If` form to an equivalent expression constructed from variables and { `true`, `false`, `And`, `Or`, `Not`, `Implies`, `If`. This process eliminates every expression of the form

```
(make-If X Y Z)
```

where one of the arguments {`X`, `Y`, `Z` is a constant { `true`, `false` }.

Use the following set of reduction rules to perform this conversion

```
(make-If  X  false true)  =>  (make-Not X)
(make-If  X  Y  false)    =>  (make-And  X  Y)
(make-If  X  true  Y)     =>  (make-Or  X  Y)
(make-If  X  Y  true)     =>  (make-Implies  X  Y)
```

where `X` , `Y` , and `Z` are arbitrary `If` forms. This set of rules is Church-Rosser, so the rules can safely be applied using simple structural recursion.

Points Dsitribution

- convertToIf (10%)
- normalize (20%)
- eval (20%)
- convertToBool (10%)
- reduce (40%)