Scheme representations of terms and statements

In the /home/reseda/discrete-structures/code/discrete directory on MathLAN, you will find four new Scheme libraries that can be used to parse and evaluate terms and statements of the predicate calculus: relations.ss, predicate-scanner.ss, predicate-parser.ss, and predicate-evaluator.ss.

1. The predicate-parser.ss file gives record definitions for the kinds of expressions that can occur in terms and statements of the predicate calculus. Read through these and acquaint yourself with the names of the constructors and accessors that they define.

The parser also provides and exports the parse procedure, which can be used to convert any string that represents a statement of the predicate calculus into a structured record of the appropriate statement type. In writing strings that represent statements beginning with quantifiers, take care to leave a space after the variable of quantification, thus:

"∀x P(x)"

Without the space, the parser would read the variable of quantification as xP and then be puzzled to find just (x), which is not a statement, afterwards. Inserting the space between x and P enables the parser to recognize them as separate identifiers.

2. Use the parse procedure to parse the string ‘∃x R(x, c)’. What kind of a statement does parse return? How can one extract the inner, unquantified statement ‘R(x, c)’ from this record?

Assignments

Recall from the “Semantics of the Predicate Calculus” reading that an assignment to one or more statements in the predicate calculus supplies a denotation to each variable, function name, and predicate in those statements, all constructed from the values in a “universe of discourse.” For example, to construct an assignment for the statements ‘P(t(x))’ and ‘∃x R(x, c)’, we chose {red, blue, green} as our universe of discourse, and we selected the following denotations:

- ‘x’: red
- ‘c’: green
- ‘t’: the function f such that f(red) = blue, f(blue) = blue, and f(green) = red
- ‘P’: {⟨green⟩, ⟨blue⟩}
- ‘R’: {⟨green, green⟩, ⟨green, red⟩, ⟨blue, blue⟩, ⟨red, blue⟩, ⟨red, green⟩}

Of course, this is not the only possible assignment that can be constructed relative to this universe of discourse. (In fact, there are 995327 others.)

3. Construct at least one more such assignment.

Representing Assignments

Now open the predicate-evaluator.ss library in a new DrRacket window. In it, you’ll find the definition of a record type called assignment, with four fields: universe, variables, functions, and predicates. The first field will contain a set of the values in the universe of discourse. (In our example, we would place the set containing the symbols red, blue, and green in this field when constructing the assignment record.)
The other three fields will be association lists — that is, lists of pairs in which the car of each pair is a “key” (in this case, a string representing a simple term, function name, or predicate name) and the cdr is a value associated with that key (in this case, the denotation of that name). These are like the association lists that we used in implementing the semantics of the propositional calculus in the lab on evaluating Boolean expressions, where the keys were characters representing Boolean variables and the associated values were Booleans. In predicate calculus, of course, the denotations for function names and predicates have more structure.

Having separate association lists for simple terms, function names, and predicate names will make lookups more efficient. At any point in its work, the evaluator will be able to determine from the structure of the statement that it is working on which of the three kinds of identifiers it needs a denotation for.

The denotation of a predicate, its extension, is supposed to be a set of sequences of values in the universe. We’ll use lists to represent sequences, so that the extension of a predicate is simply a set of lists of values in the universe. Thus, for instance, the denotation of ‘\(P\)’ in our example assignment would be represented by the set \{\((\text{green}), (\text{blue})\)\} — a set of lists of length 1, reflecting the fact that the valence of ‘\(P\)’ is 1.

The data structure that represents the denotation of a function name is a relation, as defined in the \((\text{discrete relations})\) library. The relation must be a functional one, as defined in the lab on functions — in the underlying set of pairs, no two pairs should have the same car.

In that lab, we assumed that all of our functions would apply to single arguments; we implemented a function as a relation in which the car of any member of the underlying set of pairs was the argument to the function and the corresponding cdr was the value. But in the predicate calculus, the valence of a function doesn’t have to be 1; it can be any natural number. In this setting, therefore, the car of any member of the underlying set of pairs should be a sequence of argument values, and the cdr should be the (single) value that the function yields when applied to the arguments listed in that sequence.

So, for instance, we might construct the denotation of a function \(\text{mid}\) of valence 2, in our universe of three colors, like this:

\[
\text{(make-relation (set (cons (list 'red 'red) 'red) )}
\text{ (cons (list 'red 'green) 'blue) )}
\text{ (cons (list 'red 'blue) 'green) )}
\text{ (cons (list 'green 'red) 'blue) )}
\text{ (cons (list 'green 'green) 'green) )}
\text{ (cons (list 'green 'blue) 'red) )}
\text{ (cons (list 'blue 'red) 'green) )}
\text{ (cons (list 'blue 'green) 'red) )}
\text{ (cons (list 'blue 'blue) 'blue) ))}
\]

Each call to \text{cons} takes a sequence of arguments and the value that the function should yield when given those arguments. (For instance, this function yields \text{green} when applied to the arguments \text{blue} and \text{red}.)

4. How will we represent the denotation of ‘\(t\)’ in our example assignment?

5. How will we represent the denotation of ‘\(R\)’ in our example assignment?

6. Invoke the \text{make-assignment} procedure to construct our example assignment, giving it as arguments the values that it should place in its four fields. Invoke it again to construct the assignment that you described in the solution to exercise 2 above.
The Structure of the Evaluator

Reflecting the fact that an assignment in the predicate calculus associates values with terms as well as with statements, the evaluator is actually implemented as two procedures, `evaluate-term` to determine the denotations of terms and `evaluate-statement` to determine the truth-values of statements. The `predicate-evaluator.ss` file contains the complete definition of `evaluate-term`. Study this definition. How does it determine which member of the universe of discourse will be associated with a function term such as ‘t(t(x))’?

The `predicate-evaluator.ss` file also contains all the easy and straightforward parts of the definition of the `evaluate-statement` procedure, but I’ve replaced the action part of some of the `cond`-clauses with `(assert #f)`. The missing actions deal with two of the three kinds of statements that don’t correspond to anything in the propositional calculus: atomic statements, and statements beginning with ‘∃’.

7. Create a file named `my-predicate-evaluator.ss` and copy into it the source code for the `(discrete predicate-evaluator)` library. In the first line of the `define-library`-expression, change the name of the library to `(discrete my-predicate-evaluator)`

8. Remove the `(assert #f)` clauses, fill in the blanks to get a working evaluator, and try it out! As an initial test, confirm that the example assignment you built in exercise 5 above does not satisfy the premiss ‘P(t(c))’ of the inference used as an example in the reading, but does satisfy the conclusion ‘∃x (R(x, c) → P(x))’. What truth-values do these statements have in the other assignment you constructed above?

Hints: (1) Atomic statements are somewhat analogous to function terms in the way that they evaluate their arguments. (2) Existential statements are analogous to universal ones. What is the semantic change needed?

I am indebted to Marcel Champagne 2017 and Dennis Chan 2019 for calling my attention to a careless error in an earlier version of this document.

Copyright © 2013, 2014, 2016, 2018 John David Stone

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit

http://creativecommons.org/licenses/by-sa/4.0/deed.en

or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.