Questions

Bookmark this page

Not

1/1 point (graded)

Alyssa and Ben are studying the recursive datatype for Boolean formulas.

Ben suggests changing the `Not` variant from `Not(formula:Formula)` to `Not(var:Variable)`. Which of these formulas should Alyssa offer as a counterexamples?

- □ $(P \vee Q) \wedge (\neg P \vee R)$
- □ $(P \vee Q) \wedge (P \vee \neg R)$
- □ $(P \vee Q) \wedge (\neg P \vee \neg R)$
- ☑ $(P \vee Q) \wedge \neg (P \vee R)$
- ☑ $\neg ((P \vee Q) \wedge (P \vee R))$

✓

Explanation
With this change, Ben can only represent negated variables, not negated (sub-)formulas.

Submit

Show Answer

Answers are displayed within the problem

Pair

1/1 point (graded)

When we write `Environment = ImList<Variable x Boolean>`, we mean that an `Environment` is implemented as an `ImList` of two-element tuples (a.k.a. pairs) where the first element is a `Variable` and the second is a `Boolean`.

Then we wrote statements like:

```
lookup(Cons((var, val), rest), x) = if var = x then val else lookup(rest, x)
```

which says: when we look up *x* in an `Environment` whose first pair is *(var, val)*, if *var* is *x*, return *val*, otherwise keep looking for *x* in the `rest` of the `Environment` (which is just an `ImList` of these pairs).

Unlike Python, Java does not have tuples. What could we do instead to store these tuples?

- □ Environment = ImList< List< Variable, Boolean>>
- □ Environment = ImList< List< String, Boolean>>
- ☑ Environment = ImList< List< Object>>
- □ Environment = ImList< Variable[]>
- □ Environment = ImList< Boolean[]>

☐ Environment = ImList< String[]>

☑ Environment = ImList< Object[]>

✓

Explanation

The first two options are not valid Java: `List` s have only one type of thing in them. And we need to store two different types, so the `Object` list and array are the only options that work.

Submit

Show Answer

Answers are displayed within the problem

Pair problems

0/1 point (graded)

The solutions for storing pairs in the previous question are pretty bad. What do they *not* provide?

☑ static checking of the types of objects in the pair

☑ dynamic checking of the types of objects in the pair

☑ static checking of the number of objects in the pair

☑ dynamic checking of the number of objects in the pair

☑ static checking that we only access valid indices in the pair

☐ dynamic checking that we only access valid indices in the pair

Explanation

Lists and arrays provide dynamic out-of-bounds checking, but that's it.
Storing these pairs as sequences of `Object` s is not a good design.

Submit

Show Answer

Pair progress

4/4 points (graded)

Let's define an abstract datatype `Pair<T, U>` to store pairs of values. This is another generic type: the first element of the pair is of some unknown type we'll call `T` , and the second is of unknown type `U` . For Boolean formula environments, we will use `Pair<Variable, Boolean>` .

This ADT will be characterized by two operations: `first` , to retrieve the first element in the pair, and `second` , to retrieve the second element.

To work on the concrete implementation, we'll write a datatype definition:

```
Pair<T, U> = Pair(first:T, second:U)
```

Pair has only one concrete variant.

Fill in the blanks to define operations:

```
first : FIRST_INPUTS -> FIRST_OUTPUTS
second : SECOND_INPUTS -> SECOND_OUTPUTS
```

FIRST_INPUTS

● Pair< T, U>

○ T

○ U

○ first

○ t

○ u

✓

FIRST_OUTPUTS

○ Pair< T, U>

● T

○ U

○ first

○ t

○ u

✓

SECOND_INPUTS

● Pair< T, U>

○ T

○ U

○ second

○ t

○ u

✓

SECOND_OUTPUTS

○ Pair< T, U>

○ T

● U

○ second

○ t

○ u

✓

Explanation

These function definitions are in terms of types, so names like `first` , `t` , etc. are out. The input is a `Pair` (probably these will be instance methods, and it will be `this`), and the output is the first or second element, which are of arbitrary, unknown types `T` and `U` .

Submit

Show Answer

Answers are displayed within the problem

Previous Next

Open Learning Library
About
Accessibility
All Courses
Why Support MIT Open Learning?
Help

Connect
Contact
Twitter
Facebook

Privacy Policy Terms of Service