MAU11602
Lecture 7
2026-02-05

Referential transparency with variables

What should referential transparency mean when variables can appear in expressions?

If two expressions evaluate to the same value then we can replace one with the other whenever it appears as a subexpression. For example, `+ + 1 2 3` and `+ 1 + 2 3` both evaluate to 6, so I can replace `< + + 1 2 3 4` by `< + 1 + 2 3 4` .

Can I replace `+ + x y z` by `+ x + y z`?

Neither expression has a value as it stands, but once values are assigned to `x`, `y`, and `z` they will evaluate to the same value.

We'd like referential transparency to cover this case as well, but there are some subtleties here.

A subtlety

`let val x = 1 in + x 3 end` and `let val y = 2 in + y 2` both evaluate to 4 so I should be able to replace one with the other anywhere it appears as a subexpression, and I can.

But wait, in the previous example I had to say that `+ + x y z` and `+ x + y z` evaluate to the same value no matter what values are assigned to `x`, `y` and `z`. In this example I didn't need to assign any values.

The `x` and `y` in `let val x = 1 in + x 3 end` and `let val y = 2 in + y 2 end` are *bound*, while the `x`, `y`, and `z` in `+ + x y z` and `+ x + y z` are *free*.

For testing referential transparency we only need to consider replacing free occurrences of variables, not bound occurrences.

Could the same variable occur free and bound in an expression?

Bound and free occurrences

Could the same variable occur free and bound in an expression?

Yes! Consider `+ x let val x = 1 in + x x end`.

`let val x = 1 in + x x end` evaluates to 2, `+ x let val x = 1 in + x x end` should be equivalent to `+ x 2`. If I want to say that's equivalent to `+ 2 x` then I need both to evaluate to the same value for all choices of `x`.

So `+ x let val x = 1 in + x x end` is equivalent to `+ 2 x` because when I substitute any value for the first `x` in the first expression and the (only) `x` in the second expression I get the same value.

I don't substitute for the other three occurrences of `x` in

`+ x let val x = 1 in + x x end`

Substituting for the second `x` would give me something that's not even an expression!

Substituting for the third or fourth `x` would give me the wrong answer.

`+ 2 let val x = 1 in + 2 2 end` is not equivalent to `+ 2 2`.

We need definitions which make the first `x` free and the others bound in the expression `+ x let val x = 1 in + x x end`.

Comments

We can only evaluate expressions with no free variables, so
`+ x let val x = 1 in + x x end` can't be evaluated directly, but it could be a subexpression in a larger expression which can be evaluated, like
`let val x = 3 in + x let val x = 1 in + x x end end`.
The `x` after the first + is free in `+ x let val x = 1 in + x x end` but bound in the larger expression.
The problems come from the fact that we have an inner let and an outer let, both setting `x`. These `x`'s have different values.
Can we just not do this? We can always replace the variable in a let expression and all free occurrences inside the let expression with a fresh variable.
So we can replace `let val x = 1 in + x x end` by
`let val y = 1 in + y y end` and hence can replace
`let val x = 3 in + x let val x = 1 in + x x end end` by
`let val x = 3 in + x let val y = 1 in + y y end end`.
This process is known as α conversion. Using it we can make all variables distinct.

More comments

Using α conversion we can make all variables distinct.
Should just we do this?
Yes! The expression `let val x = 3 in + x let val x = 1 in + x x end end` is considered bad style.
Why not just forbid it?
That would break referential transparency. Since `let val y = 1 in + y y end` evaluates to the same value as `let val x = 1 in + x x end` I should be able to substitute the first for the second in
`let val x = 3 in + x let val y = 1 in + y y end end`, which gives
`let val x = 3 in + x let val x = 1 in + x x end end`.
It's a bad idea to make such a substitution, but it's also a bad idea to forbid it.
Occasionally this is even useful. Induction requires taking a statement with a free variable n and proving the same statement with all free occurences of n replaced by $n+1$. We can express this as `let val n = n + 1 in ... end`.

Rules for free variables

Like everything else, the rules for identifying free occurences of a variable in an expression are based on its subexpressions.

For every expression other than let expressions, the free variables are the union of the free variables of its immediate subexpressions.

For expressions of the form `let val` v = e_1 in e_1 the v and all occurrences of v in e_2 are bound. Any occurrences of v in e_1 are free, as are all free occurrences of any other variable in e_1 or e_2.

We will introduce other binders later.

Free and bound variables in mathematics

The terms free and bound variables predate computing. You're already familiar with the idea, if not the terms.
Are $\prod_{i=1}^{n}(i+k)$ and $\prod_{j=1}^{n}(j+k)$ equivalent?
Does it make sense to substitute a value for i in the first or for j in the second?
Does it make sense to substitute a value for k or n?
Yes, no, and yes, respectively. But why?
The two are equivalent by α conversion.
Substituting for i or j doesn't make sense because they are bound variables.
Substituting for k and n does make sense because they are free variables.
$\prod$ is a binder, much like `let`.
We use a variety of terms to describe this, like dummy indices.
As long as we're substituting values for (free) variables things are fairly straightforward, but what if we want to substitute expressions?

Capture

What happens when I substitute j for k in $\prod_{i=1}^{3}(i+k) = \prod_{j=1}^{3}(j+k)$?

$$\prod_{i=1}^{3}(i+j) = (1+j)(2+j)(3+j)$$

$$\prod_{j=1}^{3}(j+j) = (1+1)(2+2)(3+3) = 48.$$

Is $(1+j)(2+j)(3+j) = 48$ for all j?
No, they're not equal for any j.
Note that I'm substituting for a free variable on both sides, which ought be be okay.
This is an example of *variable capture*.
This can't happen when substituting values, only expressions.

Evading capture

We have a few options:
- Only allow substitution of values, not expressions.
This is what I did in the reduction rules for `let` expressions.
- Allow substitution of expressions in general, but forbid them when the bound variable occurs in the expression we're substituting.
- Allow substitution of expressions in general, but forbid them when the bound variable occurs *free* in the expression we're substituting.
We only substitute for free occurrences of the variable, so bound occurrences can't result in capture.
- Allow substitution of expressions in general, but force an α conversion when the bound variable occurs *free* in the expression we're substituting.
If you want to be able to substitute arbitrary expressions for variables then this is the least restrictive sound rule.
Minor nuisance: the result of substitution is now only defined up to α-conversion.

An example

The identity

$$\prod_{m=1}^{\infty}(1-q^m)=\sum_{n=-\infty}^{\infty}(-1)^n q^{(3n^2-n)/2}$$

holds for complex $q \in (-1,1)$.
If we want to apply this to $q=\frac{1}{m^2+n^2+2}$ we write

$$\prod_{k=1}^{\infty}\left(1-\left(\frac{1}{m^2+n^2+2}\right)^k\right)=\sum_{k=-\infty}^{\infty}(-1)^k\left(\frac{1}{m^2+n^2+2}\right)^{(3k^2-k)/2},$$

not

$$\prod_{m=1}^{\infty}\left(1-\left(\frac{1}{m^2+n^2+2}\right)^m\right)=\sum_{n=-\infty}^{\infty}(-1)^n\left(\frac{1}{m^2+n^2+2}\right)^{(3n^2-n)/2}.$$

Note that I also needed to add parentheses. Because infix.

Rules for substitution

The linear way of writing expressions is just a way to encode trees. Everything really happens at the tree level. To figure out whether you need to add parentheses, think about whether leaving them out would give you the correct tree.

This is mostly, but not entirely, an infix problem.

Substitution in an expression is mostly just substitution in the subexpressions, until you reach the leaves, but binders are special.

Look at the variable the binder binds. If it's the same as the variable you're substituting for, don't substitute within the body of the binding expression.

If the bound variable occurs free in the expression you're substituting, first substitute a fresh variable for it in the body, then perform the substitution.

Fresh means not used elsewhere.

These rules apply in programming languages and mathematics.

Scope

Variables have a *scope*, within which they are meaningful.
Binders introduce a new scope, within which there's a new variable. All variables in the outer scope remain in the inner scope, except if the new variable has the same name as the old one then the old one's no longer accessible until you revert to the old scope.
In

$$2 + \sum_{i=3}^{5} \left(i + \prod_{j=7}^{11} (i + j) \right)$$

the scope of i is the summand expression $i + \prod_{j=7}^{11}(i + j)$.
The scope of j is the factor $i + j$ in the product.
i is still in scope in the $i + j$, but j is not in scope in the part of the summand outside the product.
Standard mathematical notation does a poor job of delimiting scope.
It lacks an equivalent for the `end` matching the `let` in our let expressions.

Back to our language

Substituting expressions for variables is *not* part of our reduction semantics.

We *evaluate* the right hand side of the = and then substitute the *value* in for the variable in the body.

Since we're substituting values, not expressions, we don't need rules for avoiding variable capture.

We do still need to understand free and bound variables because we only substitute for free occurrences of a variable.

We also need the more complicated substitution rules if we want to use referential transparency to predict the values of expressions without strictly following the reduction rules.