Equational Reasoning

Garlan & Wing Handout 2 8 September 1997

There is a very simple, but powerful subset of first-order predicate logic called *equational logic*. This handout explains the essence of equational logic and equational-style proofs and how they are relevant to software engineering.

In this handout and the next I will be copying examples and statements verbatim (or nearly so) from Gries and Schneider's book, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993. When I do, I'll reference it as [GS].

1 Motivation

In junior high school you probably remember solving word problems like [GS]:

> Mary has twice as many apples as John. Mary throws half her apples away, because they are rotten, and John eats one of his. Mary still has twice as many apples as John. How many apples did Mary and John have initially?

You solved this word problem by introducing variables, m and j, to stand for the number of apples Mary initially has and the number of apples John initially has, respectively; then you wrote two equations to relate the two numbers:

$$m = 2 \times j$$
$$m/2 = 2 \times (j - 1)$$

Then you solved for m and j to find a solution to the problem. In the process of solving for m and j you used properties about $=$, e.g., moving quantities from one side of the $=$ sign to the other. I am going to remind you of these properties in this handout. You also used properties about $-$, $\times$, and $/$; these are called *algebraic* properties; more on this later.

2 Equational Logic

2.1 Syntax

The syntax of an equational logic is very simple. The basic kind of formulae you write is of the form:

$$e_1 = e_2$$

where e_1 and e_2 are quantifier-free expressions, usually typed. e_1 and e_2 may or may not have variables in them. It's important that e_1 and e_2 are quantifier-free because any variables that do appear in e_1 and e_2 are implicitly universally quantified.

You can think of the $=$ (equal) symbol as just a very special predicate symbol in a predicate logic. We very well could have introduced a word for this predicate, e.g., *equal*, and have written *equal*(e_1, e_2) instead of the equation above. Here, we are appealing to your intuition by using the symbol = and treating it as a binary infix operator rather than a prefix operator.

In a typed logic, the expressions on the right and left hand sides of = have to be of the same type for the whole expression to make sense. You can write

$m = 2 \times j$
$69 \times 4 = 23 \times 12$

since for both equations the right and left hand sides both evaluate to integers but writing

$blue = 2 \times j$
$69 \times 4 = true$

is utter nonsense.

In a typed logic, we usually assume that for each type there is a special = symbol but we don't usually introduce different notation for each. In other words, I don't write $=_{int}$ and $=_{color}$ to distinguish between equality for integers and equality for colors. There is often one important exception, and that is for boolean expressions. You will see that many people use the $\equiv$ symbol (pronounced "is equivalent to") to denote equality between boolean expressions.[1] So you can write things like

$p \vee q \equiv q \vee p$
$p \Rightarrow q \equiv \neg(p \wedge \neg q)$

2.2 Semantics

Here are three **AXIOMS** about =:

- Reflexivity: $x = x$
- Symmetry: $x = y \equiv y = x$
- Transitivity: $(x = y \wedge y = z) \Rightarrow x = z$

Here is the critical **INFERENCE RULE** for =:

- Substitution of Equals for Equals:

$$\frac{a = b}{e[a/x] = e[b/x]}$$

The axioms should be familiar and obvious. The inference rule is simple, but extremely powerful. It says that if I know two entities (a and b) are equal then in the same expression (e), I can substitute one (say, a) for some variable x and get a new expression, $e[a/x]$, and I can substitute the other (say, b) for the same variable x and get another new expression, $e[b/x]$, and that the two new expressions are *equal*.

[Aside 1: Here is an example of how mathematical notation is much more concise than English!]

In other words, this rule allows us to substitute equals for equals in an expression without changing the value of that expression.

[Aside 2: It is because of this substitution rule that purely functional languages enjoy the property called *referential transparency*].

[1] Not to be confused with Woodcock and Loomes's use of $\equiv$ for $\Leftrightarrow$.

3 Equational Proofs

The substitution of equals for equals rule gives us a method for showing that two expressions are equal:

$$
\begin{array}{ll}
 & e[a/x] \\
= & \%\ a = b \\
 & e[b/x]
\end{array}
$$

where I wrote the rationalization for the proof step after the % character.

Here is an application of the substitution rule for the rotten apple problem [GS]:

$$
\begin{array}{ll}
 & m/2 = 2 \times (j-1) \\
= & \qquad \%\ m = 2 \times j \\
 & (2 \times j)/2 = 2 \times (j-1)
\end{array}
$$

where e is $x/2 = 2 \times (j-1)$, a is m, and b is $2 \times j$. (Notice here that the (boolean) expression e itself is an equation between two integer expressions.)

More generally, suppose you want to prove two expressions, e_0 and e_n, are equal, then an equational style proof would typically look like:

$$
\begin{array}{ll}
 & e_0 \\
= & \quad \%\ \text{explanation of why } e_0 = e_1 \\
 & e_1 \\
= & \quad \%\ \text{explanation of why } e_1 = e_2 \\
 & e_2 \\
\vdots & \\
= & \quad \%\ \text{explanation of why } e_{n-2} = e_{n-1} \\
 & e_{n-1} \\
= & \quad \%\ \text{explanation of why } e_{n-1} = e_n \\
 & e_n
\end{array}
$$

Then we appeal to the transitivity property of = to conclude that e_0 equals e_n.[2] The "explanations" often appeal to the substitution of equals for equals rule, but sometimes they appeal to the algebraic properties of operators that appear in e_i. For example, in the rotten apple problem, you might find it useful to appeal to the algebraic property from arithmetic that relates $\times$ and $/$: $2 \times x/2 = x$.

Some people prefer to put the explanation on the same line as the "righthand" side of the = symbol to save space:

$$
\begin{array}{ll}
e_0 & \\
= e_1 & \quad \%\ \text{explanation of why } e_0 = e_1 \\
\vdots & \\
= e_n & \quad \%\ \text{explanation of why } e_{n-1} = e_n
\end{array}
$$

Finally, some people prefer to work with both sides of the equation at the same time. At each step in the proof you find some subexpression of either the lefthand or righthand side of the equation and replace it with an equivalent one; you can keep rewriting both sides of the equation, justifying each step as before. [3] You can stop when the expressions on the left and right hand sides of the equation are identical (syntactically the same).

[2] The symmetry property says that we can do the proof backwards. Notice that if = were replaced by $\Rightarrow$ we could not do that!

[3] It's also possible to reduce both sides of the equation at the same time, but since you'd really be doing two proof steps simultaneously, you need to be careful to justify both steps. We do not recommend this procedure.

4 Equational Theories

An equational theory is a set of formulae, each of the form $e_i = e_j$. Each formula is either an axiom or it is provable using the axioms for equality and the substitution rule of equals for equals.

5 What Does Any of This Have to do with Software Engineering?

There are many reasons that we are teaching you a bit about equational reasoning. First, it's a very simple and powerful proof technique, one with which you are familiar (in a different guise) from high school algebra and trigonometry. Because of its restricted set of formulae, axioms, and proof rules, equational reasoning can be semi-automated. Many theorem provers embody equational reasoning in terms of a semantics based on *rewrite rules* where equations are treated unidirectionally (e.g., the lefthand side *rewrites* to the righthand side). These theorem provers are used in the specification and verification of programs, hardware, and software systems.

Second, there is a class of models called *algebras* that are models for equational theories. An algebra is a pair, $\langle V, F\rangle$, of a set of values V and a set of functions, F. The equations in the theory determine when two values in the algebra are equal.

- Algebras are the foundational models of *abstract data types* (ADT's). From modern programming practice, we know it's good to structure programs around ADT's. One of the linchpins of all object-oriented methodologies is design using ADT's; thus, object-oriented design methods and programming languages owe their existence to algebraic models.

- Algebras are also the foundational model for some notions of *concurrent processes*. Later this term when we study CSP, we will look at one of these process algebras more carefully.

Finally, even though you may not build algebraic models of your systems, it's important to have in your arsenal a sense of *algebraic properties* of the entities in your system. Some standard algebraic properties of an operation on a system entity (like an object or a process) include: idempotency, reflexivity, symmetry, transitivity, commutativity, associativity, distributivity, existence of an identity element, and existence of an inverse relation or function. You are comfortable with algebraic properties of numbers, e.g., the commutativity and associativity of addition and multiplication for integers; it's just that there are similarly relevant algebraic properties that one can rely on, reason with, or prove about entities in computer hardware and software systems too.

For example, with respect to ADT's, you might want rely on the idempotency property of the *insert(x)* method on a set object; that is, if you insert x into the set multiple times you don't change the value of set object. Notice this property does not hold of a bag or stack or sequence or queue.

In the context of concurrent programs, it's reassuring to know that the parallel composition operator ($||$) is commutative, so that $P \mid\mid Q$ and $Q \mid\mid P$ denote the very same concurrent system. Notice of course that sequential composition (;) is not at all commutative (in most programming languages); thus, $P; Q$ and $Q; P$ will usually yield two different behaviors.

Thus, operations on entities like objects or processes that are modeled in terms of algebras enjoy algebraic properties.