Mathematically Rigorous Software Design

Review of mathematical prerequisites

2002 September 27

Part 1: Boolean algebra

1. Define the Boolean functions and, or, not, implication ($\Rightarrow$), equivalence ($\Leftrightarrow$) and equals (=) by truth tables.

2. In an expression the various functions are evaluated in the following order unless otherwise indicated by parentheses:

$\uparrow$ (exponentiation)
+, - (sign)
*, / (multiplication, division)
+, - (addition, subtraction)
<, >, =, $\leq$, $\geq$, $\neq$ (relations)
not (also written $\neg$)
and (also written $\wedge$)
or (also written $\vee$)
$\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$ (logical implications)

Prove that the following equalities hold for all x, y, z $\in$ $\mathbb{B}$ ($\mathbb{B}$ = {false, true}). Base your proofs on your answer to question 1 above.

2.1. (x **and** y) = (y **and** x)
2.2. (x **or** y) = (y **or** x)

2.3. (x **and** (y **and** z)) = ((x **and** y) **and** z)
2.4. (x **or** (y **or** z)) = ((x **or** y) **or** z)

2.5. (x **and** (y **or** z)) = ((x **and** y) **or** (x **and** z))
2.6. (x **or** (y **and** z)) = ((x **or** y) **and** (x **or** z))

2.7. (x **and not** x) = false
2.8. (x **and** false) = false
2.9. (x **and** x) = x
2.10. (x **and** true) = x

2.11. (x **or** false) = x
2.12. (x **or** x) = x
2.13. (x **or** true) = true

2.14. (x **or not** x) = true

2.15. (x **or** (x **and** y)) = x
2.16. (x **or** (**not** x **and** y)) = (x **or** y)

2.17. (**not** (**not** x)) = x

2.18. (**not** (x **and** y)) = ((**not** x) **or** (**not** y))
2.19. (**not** (x **or** y)) = ((**not** x) **and** (**not** y))

2.20. (x ⇒ y) = (**not** (x **and not** y))
2.21. (x ⇒ y) = ((**not** x) **or** y)
2.22. (x ⇒ y) = ((**not** y) ⇒ (**not** x))
2.23. (z **and** (x ⇒ y)) = (z **and** ((z **and** x) ⇒ y))

2.24. (x = y) = ((x **and** y) **or** (**not** x **and not** y))

2.25. (x⇒(y=z)) = ((x **and** y)=(x **and** z))
2.26. (x⇒(y=z)) ⇒ ((x **and** y)=(x **and** z))

What is the practical significance of 2.26?

3. Let B, C and D be Boolean variables or functions (i.e. functions with values in 𝔹 = {false, true}). The function F is defined as follows:

F = C, if B = true,
= D, if B = false

Write an equivalent expression for F using only B, C, D, and the Boolean functions **and**, **or** and **not**. Prove that your expression for F satisifes the definition above.

4. Simplify or expand the following expressions:

4.1: x **and** (y ⇒ z)
4.2: x **or** (y ⇒ z)
4.3: (x **and** y) ⇒ z
4.4: (x **or** y) ⇒ z

4.5: -x>0 **and** x<0 **or** x>0 **and not** x<0
4.6: -x≥0 **and** x<0 **or** x≥0 **and not** x<0
4.7: -x<0 **and** x<0 **or** x<0 **and not** x<0
4.8: -x≤0 **and** x<0 **or** x≤0 **and not** x<0

4.9: (w **or** x) **and** (y **or** z)

4.10: [ia≤na **or** ib≤nb] **and** [ib>nb **or** ia≤na **and** A(ia)≤B(ib)]
4.11: [ia≤na **or** ib≤nb] **and** [ib>nb **or** ia≤na **and** ib≤nb **and** A(ia)≤B(ib)]

4.12: **not** (ib>nb **or** ia≤na **and** A(ia)≤B(ib))

5. Show that

{**not** (ia≤na **and** [ib>nb **or** A(ia)≤B(ib)])}
= {ib≤nb **and** [ia>na **or** A(ia)>B(ib)]}

when (ia≤na or ib≤nb).

(end)

Mathematically Rigorous Software Design

Review of mathematical prerequisites

Part 2: Notation

1. The notation f.x is often used in place of the more classical form f(x). The dot (.) is interpreted as an infix operator with the meaning "functional application", i.e. the application of the function f to the argument x.

2. Proofs are often written in the following format:

expression 1
=
expression 2
=
expression 3

etc. This is defined to mean (expression 1 = expression 2) and (expression 2 = expression 3), etc. Similarly,

expression 1
⇒
expression 2
⇒
expression 3

is defined to mean (expression 1 ⇒ expression 2) and (expression 2 ⇒ expression 3).

3. The notation (**op** i : r.i : exp.i) means

exp.i1 **op** exp.i2 **op** exp.i3 ...

where **op** is any operator (function) satisfying the commutative and associative laws and where i1, i2, i3, etc. are all the values of i for which r.i is true. The expression (function) exp need not be Boolean; its range may be any set consistent with the definition of the particular **op**.

The above definition can be extended to cover situations in which **op** is not commutative or not associative (or both). If **op** is not commutative, then the order in which the elements i1, i2, etc. appear must be defined. Usually, this order will be the order defined on the set to which the values of i1, i2, etc. belong. This set will often be specified within the Boolean function r. If **op** is not associative, the grouping (implied parentheses) must be specified, e.g. left to right.

If quantification over the empty set is specified (i.e. if there is no value for i for which r.i is true), then the value of the entire expression is defined to be the identity element of the operator **op**. (Why is this convention appropriate and convenient?)

$(\wedge\ i : i \in S : exp.i)$, (**A** i : i∈S : exp.i) and $(\forall\ i : i \in S : exp.i)$ are often written instead of $(\forall i \in S : exp.i)$ or other similar forms. Similarly, $(\vee\ i : i \in S : exp.i)$, (**E** i : i∈S : exp.i) and $(\exists\ i : i \in S : exp.i)$ are often written for $(\exists i \in S : exp.i)$ or other similar forms.

4. Many target groups of readers are not particularly familiar with mathematical symbols such as $\wedge$, $\vee$, $\forall$, $\exists$, etc. They find the resulting, often rather dense, expressions difficult to read and understand. Readability can often be facilitated by using more descriptive notation such as **and**, **or**, etc. instead of the mathematical symbols. Splitting an expression over two or more lines and indenting appropriately can often improve readability of long expressions considerably.

5. The familiar Σ notation can be generalized to commutative and associative operations (functions) other than addition in the obvious manner, e.g.

$$\mathbf{and}_{i=1}^{n}\ exp.i$$

etc.

Formally, $\mathbf{op}_{i=1}^{n}\ exp.i$ means $(\mathbf{op}\ i : i \in \mathbf{Z} \wedge 1 \leq i \leq n : exp.i)$.

(end)

Mathematically Rigorous Software Design

Review of mathematical prerequisites

Part 3: Images and preimages of a function

1. Let F be a function with domain Xd and range Yr; F maps any element of Xd to some element of Yr. Further, let X1 and Xd be subsets of a set X. Similarly, let Y1 and Yr be subsets of a set Y.

The *image* of X1 under F is defined as the set of all elements of Y to which F maps elements of X1. Somewhat more precisely, the image of X1 under F is the set of all F.x for which $x \in X1$ and F.x is defined. Formally, the image of X1 under F is the set

$$(\cup\, x : x \in X1 \cap Xd : \{F.x\})$$

The image of X1 under F is written F.X1 and is a subset of Yr (and hence of Y). The function referred to in the expression F.X1 (where $X1 \subseteq X$) is, strictly speaking, a different function than the one referred to in the expression F.x (where $x \in X$). The same notation is usually used, however, since the two functions are so closely related.

The *preimage* of Y1 under F is defined as the set of all elements of X which F maps to elements of Y1. Somewhat more precisely, the preimage of Y1 under F is the set of all x for which F.x is defined and in Y1. Formally, the preimage of Y1 under F is the set

$$(\cup\, x : x \in Xd \wedge F.x \in Y1 : \{x\})$$

The preimage of Y1 under F is written $F^{-1}.Y1$ and is a subset of Xd (and hence of X).

The range Yr of F is the image of X under F. The domain Xd of F is the preimage of Y under F.

2. The formation of images and preimages are monotonic operations:

$$X1 \subseteq X2 \Rightarrow F.X1 \subseteq F.X2$$
$$Y1 \subseteq Y2 \Rightarrow F^{-1}.Y1 \subseteq F^{-1}.Y2$$

The reverse implications are not, however, generally true.

3. The images and preimages of intersections and unions are equal to the intersections and unions of images and preimages:

$F.(X1 \cap X2) = (F.X1) \cap (F.X2)$ [= valid only if F is a one-to-one function, otherwise $\subseteq$]
$F.(X1 \cup X2) = (F.X1) \cup (F.X2)$

$F^{-1}.(Y1 \cap Y2) = (F^{-1}.Y1) \cap (F^{-1}.Y2)$

$F^{-1}.(Y1 \cup Y2) = (F^{-1}.Y1) \cup (F^{-1}.Y2)$

where X1 and X2 are subsets of X and Y1 and Y2 are subsets of Y (cf. above).

4. Image formation and preimage formation are not inverses of each other, that is, **neither**

$F^{-1}.(F.X1) = X1$

nor

$F.(F^{-1}.Y1) = Y1$

holds in general.

$F^{-1}.(F.X1)$ will not contain elements x in X1 for which F.x is undefined. $F^{-1}.(F.X1)$ will contain elements not in X1 which are mapped to values in F.X1.

$F.(F^{-1}.Y1)$ does not contain elements of Y1 outside the range of F. $F^{-1}.Y1$ contains only elements which F maps into Y1, so it is true that

$F.(F^{-1}.Y1) \subseteq Y1$

It is recommended that the student prove the above statements and find counterexamples of those statements purported not to be true.

(end)

Mathematically Rigorous Software Design

Review of mathematical prerequisites

Part 4: Boolean series

1. The value of the empty **and** series, e.g.

$$\mathbf{and}_{i=1}^{0}\ \text{exp.i}$$

is defined to be true. The value of the empty **or** series is defined to be false. (Why?)

2. A term may be taken out of a Boolean series only if the series is not empty. E.g. the equality

$$[\mathbf{and}_{i=1}^{n}\ \text{exp.i}] = [\text{exp.n}\ \mathbf{and}_{i=1}^{n-1}\ \text{exp.i}]$$

is true in general only if $1 \leq n$.

To take a term out of a series which may be empty, one must in effect make a case distinction, e.g. by **and**-ing the series in question with a tautology (a universally true expression) which distinguishes between an empty and a non-empty series. In the above example, a suitable such expression is ($n<1$ **or** $1 \leq n$).

The resulting generally valid expressions for removing a term from a series are:

$$[\mathbf{and}_{i=a}^{b}\ \text{exp.i}] = [b<a\ \mathbf{or}\ a \leq b\ \mathbf{and}\ \text{exp.b}\ \mathbf{and}_{i=a}^{b-1}\ \text{exp.i}]$$

$$[\mathbf{or}_{i=a}^{b}\ \text{exp.i}] = [a \leq b\ \mathbf{and}\ (\text{exp.b}\ \mathbf{or}_{i=a}^{b-1}\ \text{exp.i})]$$

(end)

Mathematically Rigorous Software Design

Review of mathematical prerequisites

Part 5: Extending the domain of a function

1. One sometimes wants to extend the definition of a function for arguments not in the domain of the original function. The general pattern of such an extension is as follows.

Let f be a function with domain D and range R. Furthermore let D' be a superset of D. The function f' is defined in the following general way:

$$\begin{aligned} f'.x &= f.x, \quad \text{if } x \in D \\ &= ?, \quad \text{otherwise} \end{aligned}$$

where ? represents any arbitrary value. One often useful convention is to replace the ? above with a value (an element) undef which is neither in D nor in R. Other conventions, in which other values are substituted for ? above, are also useful in specific contexts.

2. In modelling mathematically some types of run time errors, e.g. those resulting from references to undeclared variables, one can extend the definitions of the Boolean functions **and**, **or** and **not** over the set {false, undef, true} in different ways. See *The Spine of Software*, p. 271 ff.

(end)

Mathematically Rigorous Software Design

Review of mathematical prerequisites

Part 6: Proofs by induction

Often one wants to prove that a proposition (expression, Boolean function) E with an integer argument is true for all values of its argument. *Proof by induction* is one structural form for such a proof. In such a proof, one proves that (1) the proposition is true for some particular value of its argument and (2) the truth of the proposition for one value of its argument implies the truth of the proposition for the next value of the argument. Together, these two steps prove that the proposition is true for every value of its argument greater than or equal to the particular value used in part 1 of the proof.

Part 1 of the proof is called the *base* case. Part 2 is called the *inductive step*.

Typically the theorem to be proved is of the form:

> **Theorem:** If ... (a given hypothesis) is true, then E.n is true for all integer values of $n \geq 0$.

Often the given hypothesis consists of an iterative definition of a function and the proposition E.n states that the value of the function for the argument n is given by a closed (non-iterative) formula.

The proof by induction of a theorem stated in the above form consists of the two parts mentioned above, i.e. of proofs of the following lemmata:

> **Lemma 1:** E.0 is true.
>
> **Lemma 2:** E.n $\Rightarrow$ E.(n+1) for any (each, every, all) integer $n \geq 0$.

In the proofs of these two lemmata one may, of course, use (assume the truth of) the hypothesis of original theorem.

In some cases, lemma 2 above cannot be proved because the truth of E.n alone does not imply the truth of E.(n+1). Sometimes the truth of all of the propositions E.0, E.1, ... E.n is needed to prove that E.(n+1) is true. In such cases, the following alternative form of lemma 2 must be proved:

> **Lemma 2 alternative:** $(\wedge\ i : i \in \mathbf{Z} \wedge 0 \leq i \leq n : E.i) \Rightarrow E.(n+1)$ for any (each, every, all) integer $n \geq 0$.

If one is not careful to structure and formulate an attempted proof by induction clearly and correctly, mistakes are often made. It is worthwhile to write every proof by induction painstakingly precisely and accurately. The time taken to formulate the proof correctly on the

first attempt is saved by eliminating the necessity to find and correct errors later and to rewrite the proof.

(end)