CS 3EA3 - Lecture Notes
Quantification

Marina Mansour

Friday, February 3, 2017

Consider the for loop:

```
total := e;
for (int i = 0; i < n; i++)
    total := total ⊕ A[i];
```

What does this for loop do? It adds all the elements in the array A one by one.

This program is a general schema and works for **all interpretations** of $\oplus$ and `e`.

Now consider the definition of a *Monoid:*

Monoid : (M, $\oplus$, e)
$\oplus$: M $\to$ M $\to$ M
Where:

- $\oplus$ is an arbitrary binary operator that is associative.
  $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
- e is an identity operator.
  $x \oplus e = x = e \oplus x$

This can be applied in many domains:

- if $\oplus$ is and ($\wedge$) :
  then e is true
  and post-condition: $\{total = (\forall\, i : 0 \,..\, n-1 \cdot A[i])\}$
- if $\oplus$ is or ($\vee$) :
  then e is false
  and post-condition: $\{total = (\exists\, i : 0 \,..\, n-1 \cdot A[i])\}$
- if $\oplus$ is plus ($\oplus$) :
  then e is $\equiv$
  and post-condition: $\{total = (\sum\, i : 0 \,..\, n-1 \cdot A[i])\}$

In fact,

$$(\oplus \texttt{i} : \mathbb{Z} \mid \texttt{a} \leq \texttt{i} \leq \texttt{b} \cdot \texttt{f(i)})$$

is applicable in many programming languages such as Python, SQL, and many others.

Since it is abstract, it offers the advantage of being more efficient (because it can be executed in parallel in some compilers thus optimized) compared to a for-loop.

Translate the above-mentioned for-loop to a recursive function:

```
F ( unsigned int N )
{
    if ( N == 0)
        return e;
    else
        return F (N - 1) ⊕ A[N];
}
```

Note: Composition (;) is not symmetric. For programs A and B, A ; B $\neq$ B ; A

Now, let's define the recursive function F_N:

$$\begin{aligned} F_N &= (\oplus\, i \colon \mathbb{Z} \mid 0 \leq i \leq N \cdot f_i) \\ F_0 &= e \\ F_N &= F_{N-1} \oplus F_N \end{aligned}$$

Interestingly, the property $0 \leq i \leq$ N can be replaced with any other property.
→ Memorise the quantifier properties in section 11.4 of the course text.

The $\forall$ and $\exists$ quantifiers are handy and show up in lattices naturally.

Note: Leibniz Rule
→ $x = y \Rightarrow f(x) = f(y)$

The material in the table below is drawn from the book: *The Mathematics of Programming: An Inaugural Lecture Delivered Before the University of Oxford on 17 October 1985* by C. A. R. Hoare.

Table 1: Connecting Properties in Mathematics and Programming

Numbers	**Programs**
1. Associativity $a * (b * c) = (a * b) * c$	**1. Associativity** A ; (B ; C) = (A ; B) ; C
2. Identity $a * 1 = a$	**2. Identity** A ; SKIP = A
3. Zero $a * 0 = 0$	**3. Zero** A ; ABORT = ABORT
4. Distributivity $(a + b) * c = (a * c) + (b * c)$	**4. Distributivity** (A $[]$ B) ; C = (A ; C) $[]$ (B ; C)
5. Least Upper Bound $x \uparrow y$ = the greater of x and y	**5. Choice** A $[]$ B = do A or do B
6. Monotonicity $a \leq b \wedge c \leq d \Rightarrow a * c \leq b * d$	**6. Refinement by Parts** A $\sqsubseteq$ B $\wedge$ C $\sqsubseteq$ D $\Rightarrow$ A ; C $\sqsubseteq$ B ; D

Notes: (4) In mathematics, order of operations is in effect.
In programming, sequential execution is in effect.

(6) A $\sqsubseteq$ B means that A is a refinement of B.
This means that all side effects of A are also side effects of B.

Notes on Galois Connections:

If you have A and C but **not** B, from A ; B $\sqsubseteq$ C, you can derive:

$$\text{A ; B} \sqsubseteq \text{C} \mapsto \text{B} \sqsubseteq \text{C} \div \text{A} \mapsto wp(A,\ C)$$

Notes Weakest Precondition and Side effects:

What does it mean for programs A and B to be equal?
A = B if and only if both programs produce identical side effects.
It also means:

$$\forall\ p \cdot wp(A,\ p) \equiv wp(B,\ p)$$

where p is any property of A and B