Equational Reasoning

Pete Manolios
Northeastern

Logic and Computation, 10/28/2019

Complexity Analysis

$$\sum_{i=0}^{k} i$$

```
(definec sum (k :nat) :nat
  (if (= k 0)
       0
      (+ k (sum (- k 1)))))
```

What is the time complexity?

Input (k)	Aritmetic Ops	Complexity (k)	Input Size (n)	Complexity (n)
2	2(2)	O(k)	1	$O(2^n)$
32	2(32)		5	
1024	2(1024)		10	
32768	2(32768)		15	
1048576	2(k)		20	

Exponential time because k requires log(k) bits to represent

Complexity Analysis

- With SAT, no one has come up with a polynomial time algorithm
- What about sum? Can we do better?

```
(definec fsum (k :nat) :nat
  (/ (* k (+ k 1)) 2))
```

What is the time complexity?

Input (k)	Aritmetic Ops	Complexity (k)	Input Size (n)	Complexity (n)
2	3	O(1)	1	O(1)
32	3		5	
1024	3		10	
32768	3		15	
1048576	3		20	

Constant time, so exponentially better than sum!

Reasoning About Arithmetic

```
(definec sum (k :nat) :nat
  (if (= k 0)
      0
    (+ k (sum (- k 1)))))
```

```
(definec fsum (k :nat) :nat
  (/ (* k (+ k 1)) 2))
```

- We want to prove that a more clever version is equivalent

```
(implies (natp k)
         (equal (sum k)
                (fsum k)))
```

- How? By “mathematical induction” (think about 1800)

Induction Proof

```
(definec sum (k :nat) :nat
  (if (= k 0)
       0
     (+ k (sum (- k 1)))))
```

```
(definec fsum (k :nat) :nat
  (/ (* k (+ k 1)) 2))
```

```
Conjecture: (natp k) ⇒ (sum k) = (fsum k)
```

- Base case:

```
(natp k) ∧ k = 0  ⇒  (sum k) = (/ (* k k+1) 2)
```

- Induction step:

```
   (natp k) ∧ k ≠ 0 ∧
   [(natp k-1) ⇒ (sum k-1) = (/ (* k-1 k) 2)]
⇒ (sum k) = (/ (* k k+1) 2)
```

ACL2s Demo

- Show that sum takes exponential time
- The importance of tail recursion
- fsum to the rescue

Lessons Learned

- Algorithmic complexity is vitally important: consider big-data, Web
- Take algorithms as soon as possible
- As a computer scientist, *always* think about complexity
- But, correctness is most important: fast, but wrong is not good
  - Planes, trains and automobiles (not the movie) crash
  - Wrong simulation results for weather, nuclear testing, experiments…
  - Correctness is mostly what we care about in this class
- Powerful idea: define correctness using simplest definitions (the spec)
- Then define efficient implementation and prove equivalence
- Allows one to reason using the spec, but execute using efficient code

Comparison with C & Java

- Suppose that we write this code in an imperative language like C or Java
- Let's see a DEMO
- What happened?

Limited Precision!

- C, Java, etc. do not have arbitrary precision arithmetic
- So sum, fsum are not equivalent!
- We get a negative number because most languages use fixed-bit arithmetic

Finding Bugs

- You could have tested your program 1K times and not found errors
- We knew what we were looking for and so we found an error
- Is this a problem in practice? Yes. See http://googleresearch.blogspot.no/2006/06/extra-extra-read-all-about-it-nearly.html

Reasoning About C/Java

- Can we reason about C/Java code?
- We don’t have a theorem prover for these languages
- But, we can reason about them!
- Use ACL2s to model arithmetic in C/Java
    - Let’s say that the spec is that fsum should be equal to sum
    - We can use property-based testing
    - DEMO