Equational Reasoning

Pete Manolios
Northeastern

Logic and Computation, 2/11/2019

Complexity Analysis

```
(definec sum (n :nat) :nat
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))
```

- What is the time complexity of this function?
- Build a table with inputs and #arithmetic operations
- It takes time exponential in the size of the input because n requires log(n) bits to represent
- For example if the input is 10^n, that is of length n, so the size is n
- But sum takes $2*10^n$ operations, so $O(10^n)$ number of ops
- With SAT, no one could come up with a polynomial time algorithm
- What about sum?

Complexity Analysis

```
(definec sum (n :nat) :nat
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))
```

```
(defunc fsum (n :nat) :nat
  (/ (* n (+ n 1)) 2))
```

- What is the time complexity of fsum?
- Build a table with inputs and #arithmetic operations
- It always takes 3 operations so O(1) number of ops
- In contrast to SAT, we found an efficient algorithm!
- In fact, fsum algorithmic is exponentially better than sum

Reasoning About Arithmetic

```
(definec sum (n :nat) :nat
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))
```

```
(defunc fsum (n :nat) :nat
  (/ (* n (+ n 1)) 2))
```

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

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

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

Exponential Improvement

```
(definec sum (n :nat) :nat
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))
```

```
(defunc fsum (n :nat) :nat
  (/ (* n (+ n 1)) 2))
```

- Base case:

  (natp n) ∧ (equal n 0) ⇒ (sum n) = (/ (* n n+1) 2)

- Induction step:

  (natp n) ∧ n ≠ 0 ∧

  [(natp n-1) ⇒ (sum n-1) = (/ (* n-1 n) 2)]

  ⇒ (sum n) = (/ (* n n+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