A Fast Exponentiation Algorithm

Previously: an example of *a proof about an algorithm* (binary search)
In this episode: another such proof, for an exponentiation algorithm

What’s new:

- properties of exponentiation (**)
- a “fast” exponentiation algorithm
  - why it’s fast
  - why it’s correct
- and, to do the proof:
  - introducing *strong induction*
  - a few lemmas, about simple natural number functions:
    - half, even, odd, square, **
  - a variant of ordinary induction (one of many possible)
- an attempt at optimization
Mathematical background

First, define the exponentiation operator, **

```plaintext
extend-module N {
  open Times
  declare **: [N N] -> N [400 [int->nat int->nat]]
  module Power {
    assert* def := [(x ** zero = one)
                    (x ** S n = x * x ** n)]
    define [if-zero if-nonzero] := def
  } # close module Power
} # close module N
```

In conventional mathematical notation:

\[ x^0 = 1; \]
\[ x^{n+1} = x \cdot x^n. \]
Defining a “fast” exponentiation algorithm

Using the defining equations, $x^n$ requires $n - 1$ multiplications. It’s possible to compute $x^n$ with only $\log_2 n$ multiplications, using

$$n = 2\lfloor n/2 \rfloor, \text{ if } n \text{ is even;}$$

$$n = 2\lfloor n/2 \rfloor + 1, \text{ if } n \text{ is odd.}$$

Thus, if $n$ is even,

$$x^n = x^{2\lfloor n/2 \rfloor}$$

$$= (x^{\lfloor n/2 \rfloor})^2,$$

and if $n$ is odd,

$$x^n = x^{2\lfloor n/2 \rfloor}+1$$

$$= x^{2\lfloor n/2 \rfloor} \cdot x$$

$$= (x^{\lfloor n/2 \rfloor})^2 \cdot x.$$
The algorithm

\[
x^n = \begin{cases} 
(x^{n/2})^2 & \text{where } n \text{ is even}; \\
(x^{n/2})^2 \cdot x & \text{where } n \text{ is odd}
\end{cases}
\]

Using this formula recursively and grounding it with the \( n = 0 \) case:

```plaintext
extend-module N {
  declare fast-power: [N N] -> N [[int->nat int->nat]]
  module fast-power {
    assert def :=
      (fun
        [(fast-power x n) =
          [one when (n = zero)
          (square (fast-power x half n)) when (n =/= zero & even n)
          ((square (fast-power x half n)) * x) when (n =/= zero & ~ even n)]])
    define [if-zero nonzero-even nonzero-odd] := def
  } # close module fast-power
} # close module N
```
Strong induction

Proving that \((\text{fast-power } x \ n) = x^n\) is most readily done using “strong induction.”

**Principle .1: Strong Induction for Natural Numbers**

To prove \(\forall n . P(n)\) where \(n\) ranges over the natural numbers, it suffices to prove:

\[
\forall n . [\forall k . k < n \Rightarrow P(k)] \Rightarrow P(n).
\]

The assumption \([\forall k . k < n \Rightarrow P(k)]\) is called the **strong induction hypothesis**.

The strong induction hypothesis assumes \(P(k)\) for *all* preceding values \(k = 0, \ldots, n - 1\).

Just what is needed for proofs about recurrence relations that recur back to one or more values other than \(n - 1\).
Understanding strong induction

• Why is there no basis case?
• Does it have to be that “strong”?
• Is it really “stronger” than ordinary induction?
Why do a formal proof about fast-power?

- Check the details rigorously.
- Develop new tools: lemmas about basic mathematical functions.
- “Warm-up” for proof about a more subtle exponentiation algorithm — see Section 15.2 of the textbook.
- Practice with logic principles, including new “strong induction.”
Properties of half

extend-module N {
  declare half: [N] -> N [[int->nat]]

module half {
  assert* def :=
  [(half zero = zero)]
  [(half S zero = zero)]
  [(half S S n = S half n)]

  define [if-zero if-one nonzero-nonone] := def
}

Here are a couple of simple properties of half that we will need:

define double := (forall n . half (n + n) = n)

define times-two := (forall n . half (two * n) = n)
Another variant of induction

**Principle .2: Induction for Natural Numbers — Variant**

To prove $\forall n . P(n)$ where $n$ ranges over the natural numbers, it suffices to prove:

1. *First Basis Case*: $P(0)$.
2. *Second Basis Case*: $P(1)$.
3. *Induction Step*: $\forall n . P(n) \Rightarrow P(n + 2)$. 
Why it works

If we visualize the original induction formulation like this:

```
0 1 2 3 4 5 ... 
```

our variant can correspondingly be visualized:

```
0 1 2 3 4 5 ... 
```

In Athena, this “two-step” variant requires no new machinery. by-induction is sufficient: it allows subcasing of the proof in any way that exhausts the entire set of natural numbers.
Proof of \((\forall n . \text{half} (n + n) = n)\)

by-induction double {
    zero => (!chain [(half (zero + zero))
        -->(half zero) [Plus.right-zero]
        -->(zero) [if-zero]])
}

| (S zero) =>
    (!chain [(half (S zero + S zero))
        -->(half S (S zero + zero)) [Plus.right-nonzero]
        -->(half S S (zero + zero)) [Plus.left-nonzero]
        -->(half S S zero) [Plus.right-zero]
        -->(S half zero) [nonzero-nonone]
        -->(S zero) [if-zero]])
}

| (S (S m)) =>
    let {IH := (half (m + m) = m)}
        (!chain
            [(half (S S m + S S m))
            -->(half S (S S m + S m)) [Plus.right-nonzero]
            ...
            -->(S S m) [IH]])
}
Proof of \texttt{half.times-two := (forall n . half (two \times n) = n)}

conclude \texttt{times-two}

pick-any \texttt{x}

\texttt{(!chain [(half (two \times x))
--> (half (x + x)) [Times\textunderscore two\textunderscore times]
--> x [double]]})

Exercises in the textbook prove other simple properties of \texttt{half}:

\begin{verbatim}
define twice := (forall x . two \times half S S x = S S (two \times half x))
define two-plus := (forall x y . half (two \times x + y) = x + half y)
\end{verbatim}

and the textbook contains proofs of ordering properties:

\begin{verbatim}
define less-S := (forall n . half n < S n)
define less := (forall n . n /= zero ==> half n < n)
\end{verbatim}
Properties of odd and even

extend-module N {
    declare even, odd: [N] -> Boolean [[int->nat]]

module EO {
    assert* even-definition := [(even x <=> two * half x = x)]
    assert* odd-definition := [(odd x <=> two * (half x) + one = x)]

Some lemmas:

    define even-zero := (even zero)
    define odd-one := (odd S zero)
    define even-S-S := (forall n . even S S n <=> even n)
    define odd-S-S := (forall n . odd S S n <=> odd n)
    define odd-if-not-even := (forall n . ~ even n => odd n)
    define not-odd-if-even := (forall n . even n => ~ odd n)
    define even-iff-not-odd := (forall n . even n <=> ~ odd n)
    define not-even-if-odd := (forall n . odd n => ~ even n)
    ...
    define even-square := (forall n . even n <=> even square n)
}
# close module EO
Proof of \((\forall n . \sim \text{even } n \implies \text{odd } n)\)

by-induction odd-if-not-even {
  zero => (!chain [\(\sim \text{even zero}\)]
                \implies (\text{even zero} \& \sim \text{even zero}) [\text{augment}]
                \implies (\text{odd zero}) [\text{prop-taut}])
  | (S zero) =>
    assume (\sim \text{even S zero})
    (!chain<-
       [(\text{odd S zero})
        \implies (\text{two} \cdot (\text{half S zero}) + \text{one} = \text{S zero}) [\text{odd-definition}]
        \implies (\text{S (two} \cdot \text{half S zero}) = \text{S zero}) [\text{Plus.right-one}]
        \implies (\text{S (two} \cdot \text{zero}) = \text{S zero}) [\text{half.if-one}]
        \implies (\text{S zero} = \text{S zero}) [\text{Times.right-zero})])
  | (S (S m)) =>
    let {IH := (\sim \text{even m} \implies \text{odd m})}
    (!chain [(\sim \text{even S S m})
               \implies (\sim \text{even m}) [\text{even-S-S}]
               \implies (\text{odd m}) [IH]
               \implies (\text{odd S S m}) [\text{odd-S-S}])
} # close module EO
Proof of \((\forall x . \text{even } x \iff \text{even square } x)\)

More challenging (a starred exercise). Does not require induction; it can be done with equation chaining and proof by contradiction.

```plaintext
extend-module EO {
  conclude even-square
  pick-any x
  let {right := assume (even x)
    conclude (even square x)
    ...
    left := assume (even square x)
    (!by-contradiction (even x)
    assume hyp := (~ even x)
    ...
    A := conclude (two * (half square x) + one = square x)
    ...
    (!absurd
    (!chain-> [A ==> (odd square x) [odd-definition]])
    (!chain-> [(even square x)
      ==> (~ odd square x) [not-odd-if-even]])
    (!equiv right left)}
} # close module EO
```
Properties of \( ** \)

extend-module Power {
  define Plus-case := (forall m n x . x ** (m + n) = x ** m * x ** n)
  define left-one := (forall n . one ** n = one)
  define right-one := (forall n . n ** one = n)
  define right-two := (forall n . n ** two = n * n)
  define left-times := (forall n x y . (x * y) ** n = x ** n * y ** n)
  define right-times := (forall m n x . x ** (m * n) = (x ** m) ** n)
  define two-case := (forall n . square n = n ** two)
}
# close module Power
## Correctness property of fast-power

```
declare fast-power : [N N] -> N [[int->nat int->nat]]

module fast-power {

    assert axioms :=

        (fun

            [(fast-power x n) =

                [one when (n = zero)   
                 (square (fast-power x half n)) when (n =/= zero & even n)  
                 ((square (fast-power x half n)) * x) when (n =/= zero & ~ even n)]]]

    define [if-zero nonzero-even nonzero-odd] := axioms

    define correctness := (forall n x . (fast-power x n) = x ** n)
```
Proof of correctness

We use strong induction.

Available in Athena as a binary method

strong-induction.principle that takes the following arguments:

1. The sentence $p$ that we are seeking to derive.

2. A unary method $M$ that derives the strong induction step of the proof.

Given these two arguments, an application of

strong-induction.principle will derive $p$. 
Proof of correctness

```plaintext
define[^ sq hf] := [fast-power square half]
define step :=
  method (n)
  assume ind-hyp := (forall m . m < n == > forall x . x ^ m = x ** m)
  conclude (forall x . x ^ n = x ** n)
  pick-any x
    (!two-cases
      assume (n = zero)
      (!chain [(x ^ n)
        --> one [if-zero]
        <-- (x ** zero) [Power.if-zero]
        <-- (x ** n) [(n = zero)]])
      assume (n =/= zero)
      ...
    )
```

Proof of correctness, continued

assume (n /= zero)

let {p1 :=

conclude p := (forall x . x ^ hf n = x ** hf n)

(!chain-> [(n /= zero)

==>(hf n < n) [half.less]

==> p [ind-hyp]]);

p2 := conclude (sq (x ^ hf n) = x ** (two * hf n))

(!chain

[(sq (x ^ hf n))

--> (sq (x ** hf n)) [p1]

--> ((x ** hf n) *

(x ** hf n)) [square.def]

<-- (x ** ((hf n) + hf n)) [Power.Plus-case]

<-- (x ** (two * hf n)) [Times.two-times]]})

(!two-cases

assume (even n)

...}

assume (~ even n)

...}
Proof of correctness, continued

(!two-cases
  assume (even n)
  (!chain
    [(x ^ n)
     --> (sq (x ^ hf n)) [nonzero-even]
     --> (x ** (two * hf n)) [p2]
     --> (x ** n) [EO.even-definition]]))

assume (~ even n)
  let {_ := (!chain-> [(~ even n) ==> (odd n)
                       [EO.odd-if-not-even]])}
  (!chain
    [(x ^ n)
     --> ((sq (x ^ hf n)) * x) [nonzero-odd]
     --> (((x ** (two * hf n)) * x) [p2]
     <-- (((x ** (two * hf n)) * (x ** one)) [Power.right-one]
     <-- ((x ** ((two * hf n) + one)) [Power.Plus-case]
     --> (x ** n) [EO.odd-definition]]))))
Proof of correctness, continued

With the step method thus defined, the proof is completed with:

```
(!strong-induction.principle correctness step)
```

Or, we could write the whole proof as an application of strong-induction.principle with the step method defined inline:

```
(!strong-induction.principle correctness
  method (n)
  ... body of the above step method)
```
A potential optimization, using tail-recursion

```plaintext
extend-module N {

declare fast-power-accumulate: [N N N] -> N [[int->nat int->nat int->nat]]

module fast-power-accumulate {

define fpa := fast-power-accumulate

assert axioms :=

(fun

    [(fpa r x n) =
     [r when (n = zero)
      (fpa r (x * x) (half n)) when (n =/= zero & even n)
      (fpa (r * x) (x * x) (half n)) when (n =/= zero & ~ even n)]])

define [if-zero nonzero-even nonzero-odd] := axioms

define correctness := (forall n r x . (fpa r x n) = r * x ** n)
}
# close module fast-power-accumulate
}
# close module N
```
If we still want an exponentiation function with the same two-argument interface as before:

```plaintext
extend-module N {
  extend-module fast-power {
    define fpa := fast-power-accumulate
    assert* definition := [((fast-power x n) = (fpa one x n))]
  }
  # close module fast-power
} # close module N
```
Is it really an optimization?

The definition of fast-power-accumulate is tail-recursive, and therefore is equivalent to a loop.

- Does that make it necessarily more efficient than the original embedded-recursion version?
- Perhaps surprisingly, and unfortunately, the answer is no.
- It can be \textit{less efficient} in some cases.
- See Section 12.7 of the textbook for an explanation.
- See Section 15.2 for a true optimization (and a generalization).
Recap

• New algorithm correctness proof example, fast-power
• New induction principle: strong induction
• New “two-step” variant of ordinary induction
• Building up library of axioms and lemmas for natural number functions: half, even, odd, square, **
• An attempt at optimization …