Exponentials

15-150 Principles of Functional Programming (M21)

Jacob Neumann

June 2021

1 Exponentials

The (base-2) **exponential function** is a mathematical function

$$2^{(-)} : \mathbb{N} \to \mathbb{N}$$

sending n to 2^n. This operation is uniquely characterized by the following property:

Defn. $2^{(-)}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n. && \text{(for all } n\text{)} \end{aligned}$$

This characterization allows us to elegantly translate this function into Standard ML.

Defn. `exp`

`exp : int -> int`

REQUIRES: `n` $\geq$ `0`

ENSURES: `exp n` $\cong$ $2^{\texttt{n}}$

```
fun exp (0:int):int = 1
  | exp n = 2 * exp(n-1)
```

The proof that this function evaluates to a value (and specifically *the correct value*) when applied to natural number values furnishes our first example in this course of proofs by **weak induction**.

Thm. 1
A

Prop. 1 For every valuable expression `e : int` whose value is nonnegative, `exp(e)` is valuable.

Proof. By hypothesis, `e` $\hookrightarrow$ `v` for some `v` $\geq$ `0`. It suffices to prove `exp(v)` valuable, which we do by weak induction on `v`.

BC `v=0`

$$\texttt{exp 0} \implies \texttt{1} \qquad \textbf{Defn. } \texttt{exp}$$

So `exp 0` valuable.

IS `v=n+1` for some $n \geq 0$
IH `(exp n)` $\hookrightarrow$ `v'` for some value `v':int`
WTS: `exp(n+1)` $\hookrightarrow$ `v''` for some value `v'':int`

$$\begin{aligned}
\texttt{exp(n+1)} &\Longrightarrow \texttt{2 * exp(n)} && \textbf{Defn. } \texttt{exp} \\
&\Longrightarrow \texttt{2 * v'} && \textbf{IH} \\
&\Longrightarrow \texttt{v''} && \text{(for some value } \texttt{v''}\text{, by totality of } *\text{)}
\end{aligned}$$

so our induction carries through. □

The totality of `*` is required in the last step, to guarantee that `2*v'` is valuable. This establishes that `exp(n)` is valuable for all natural numbers `n`. We can also prove this function *correct* as well, using **Defn.** $2^{(-)}$.

Prop. 2 For every value `n : int` such that $n \geq 0$

$$\texttt{exp(n)} \quad \cong \quad 2^{\texttt{n}}$$

Proof. By weak induction on `n`.

BC `n=0`

$$\begin{aligned}
\texttt{exp 0} &\cong 1 && \textbf{Defn. } \texttt{exp} \\
&\cong 2^0 && \textbf{Defn. } 2^{(-)}
\end{aligned}$$

IH `(exp n)` $\hookrightarrow 2^{\texttt{n}}$ for some value $n \geq 0$
WTS: `exp(n+1)` $\cong 2^{\texttt{n+1}}$

$$\begin{aligned}
\texttt{exp(n+1)} &\cong \texttt{2 * exp(n)} && \textbf{Defn. } \texttt{exp} \\
&\cong 2 * 2^{\texttt{n}} && \textbf{IH} \\
&\cong 2^{\texttt{n+1}} && \textbf{Defn. } 2^{(-)}
\end{aligned}$$

as desired. □

2 Faster Implementation

Though we haven't yet developed the tools to demonstrate this precisely, we can tell intuitively by looking at the evaluation traces that `exp` is a *linear time* function: the number of steps it takes to evaluate `exp(n)` is (approximately) proportional to `n`:

```
    exp 10
⟹ 2 * exp 9
⟹ 2 * 2 * exp 8
⟹ 2 * 2 * 2 * exp 7
⟹ 2 * 2 * 2 * 2 * exp 6
⟹ 2 * 2 * 2 * 2 * 2 * exp 5
⟹ 2 * 2 * 2 * 2 * 2 * 2 * exp 4
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * exp 3
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * exp 2
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * exp 1
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * exp 0
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 1
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 4
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 2 * 8
⟹ 2 * 2 * 2 * 2 * 2 * 2 * 16
⟹ 2 * 2 * 2 * 2 * 2 * 32
⟹ 2 * 2 * 2 * 2 * 64
⟹ 2 * 2 * 2 * 128
⟹ 2 * 2 * 256
⟹ 2 * 512
⟹ 1024
```

This evaluation trace has 21 steps, that is, $2(10) + 1$.[1] If we similarly traced out `exp 11`, it would have $23 = 2(11) + 1$ steps, `exp 17` would take $35 = 2(17) + 1$ steps, and `exp 1000000` would have two-million-and-one steps.

[1] If we had made different choices in how much evaluation to show, we would get another number of steps. For instance, if we write the intermediate step

```
exp 10 ⟹ 2 * exp(10-1) ⟹ 2 * exp 9
```

and likewise throughout the trace, then we would get 31 steps instead of 21. But the point still stands: the number of steps is proportional to the initial input.

It turns out that we can perform this calculation faster, using some clever optimizations justified by basic number theory. Specifically, we'll make use of the following identities:

$2^{(-)}$ opt.

$$2^n = \left(2^{\lfloor n/2 \rfloor}\right)^2 \qquad (n \text{ even})$$

$$2^n = 2 \cdot \left(2^{\lfloor n/2 \rfloor}\right)^2 \qquad (n \text{ odd})$$

Here, $\lfloor x \rfloor$ denotes the largest integer less than or equal to x. For n even, $\lfloor n/2 \rfloor$ is just $n/2$ (since $n/2$ is an integer), whereas for n odd, $\lfloor n/2 \rfloor = {}^{n}/_{2} - {}^{1}/_{2}$. Either way,

Fact 3

$$\texttt{n div 2} \quad \cong \quad \left\lfloor \frac{\texttt{n}}{2} \right\rfloor . \qquad (\texttt{n} \geq 0)$$

This motivates the following declarations:

Defn. `pow`

`pow : int -> int`

REQUIRES: `n` $\geq$ `0`

ENSURES: `pow(n)` $\cong$ `exp(n)`

```
fun even (n:int):bool = (n mod 2)=0

fun square (x:int):int = x * x

fun pow (0:int):int = 1
  | pow n = case (even n) of
               true => square(pow(n div 2))
             | false => 2 * pow(n-1)
```

We could have written the odd case as `2*square(pow(n div 2))`, but it will be easier to prove as-is.

We can observe from the traces that this is more efficient:

```
    pow 10
==> square(pow(5))
==> square(2 * pow(4))
==> square(2 * square(pow(2)))
==> square(2 * square(square(pow(1))))
==> square(2 * square(square(2 * pow 0)))
==> square(2 * square(square(2 * 1)))
==> square(2 * square(square(2)))
==> square(2 * square(2 * 2))
==> square(2 * square(4))
==> square(2 * 4 * 4)
```

```
⟹ square(32)
⟹ 32 * 32
⟹ 1024
```

This trace of `pow(10)` has 13 steps. A similar trace of `pow(20)` would have 16 steps, a trace of `pow(40)` would have 19, and a trace of `pow(10485760)` would only have 73 steps![2] Conclusion: `pow` is way more efficient than `exp`. Now we just need to prove that `pow` satisifes its spec: that it indeed behaves the exact same on natural numbers as `exp`.

3 Proving the faster version

Recall the notion of **referential transparency**: if two pieces of code are extensionally-equivalent, then they are *interchangeable*: `e` $\cong$ `e'` means that `e'` can be put in place of `e` in any piece of code, without changing its behavior at all. The situation with `exp` and `pow` is a paradigm example of where this principle is useful: it is, in general, much quicker to evaluate `pow(n)` than `exp(n)`. So, if we can prove that `pow(n)` $\cong$ `exp(n)`, then we can replace all the `exp(n)`'s in our code with `pow(n)` and take advantage of the improved speed, and we'd be assured that doing so would not affect anything whatsoever about the code. More precisely, the properties of `exp` articulated in **Prop. 1** and **Prop. 2** will hold of `pow` as well. We'll have *proven* it.

3.1 Numerical Lemmas

We'll need a couple lemmas to write this proof. First of all, the correctness of `pow` relies on the assumption that arithmetic in SML (specifically `div`) works correctly. So we'll explicitly and precisely articulate what properties we're relying on – they turn out to be quite modest.

Lemma 4 For any valuable `n` whose value is nonnegative and **even**,

$$\texttt{n} \cong \texttt{(n div 2) + (n div 2)}$$

Lemma 5 For any value `n` $>$ `0`,

$$\texttt{(n div 2)} \text{ is valuable} \quad \text{and} \quad 0 \leq \texttt{(n div 2)} < \texttt{n}$$

Both of these lemmas are true: `div` implements integer division in SML correctly, including the properties demanded by these lemmas.

We also need to require that our helper function `even` behaves properly. This is typical in proving code: helper functions correspond to lemmas, in that we usually need lemmas to guarantee that our helpers do the right thing.

Lemma 6 `even` is total, and for any value `n:int`,

[2]Take a moment to appreciate how unfathomably large a number `pow(10485760)` is: *2 to the power 10 million, 485 thousand, 760*. SML has no hope of calculating a value that big, but it's pretty cool that in theory we could calculate it in just 73 steps.

- if `n` is even, `(even n)` $\Longrightarrow$ `true`
- if `n` is odd, `(even n)` $\Longrightarrow$ `false`.

Again, if we believe that `mod` and integer equality are implemented correctly in SML, then this is true.

3.2 exp Lemma

The final ingredient we'll need: a simple arithmetical property about `exp`. Recall that `exp n` implements 2^{n}, and so inherits all relevant properties of exponentials. In particular, the property that

$$2^n \cdot 2^k = 2^{n+k} \qquad \text{for all } n \in \mathbb{N}$$

is also possessed by `exp`. We prove this fact straight from **Defn. exp**, by induction (of course!).

Lemma 7 For all *valuable expressions* `n,k : int` whose values are nonnegative,

`exp(n) * exp(k)` $\cong$ `exp(n+k)`

Proof. Let `k` be arbitrary and fixed, and write v_n for the value `n` evaluates to. We proceed by weak induction on v_n.

BC $v_n = 0$

$$
\begin{aligned}
\texttt{exp n * exp k} &\cong \texttt{exp 0 * exp k} && (v_n=0)\\
&\cong \texttt{1 * exp k} && \textbf{Defn. exp}\\
&\cong \texttt{exp k} && \text{(math)}\\
&\cong \texttt{exp(0 + k)} && \text{(math)}\\
&\cong \texttt{exp(n + k)} && (v_n=0)
\end{aligned}
$$

IS $v_n = v+1$ for some value $v \geq 0$
IH `exp(v) * exp(k)` $\cong$ `exp(v+k)`
WTS: `exp(n) * exp(k)` $\cong$ `exp(n+k)`

$$
\begin{aligned}
\texttt{exp n * exp k} &\cong \texttt{exp(v+1) * exp(k)} && (v_n=v+1)\\
&\cong \texttt{2 * exp(v) * exp(k)} && \textbf{Defn. exp}\\
&\cong \texttt{2 * exp(v+k)} && \textbf{IH}\\
&\cong \texttt{exp((v+k)+1)} && (\textbf{Defn. exp}, \texttt{v+k} \text{ valuable})\\
&\cong \texttt{exp((v+1)+k)} && \text{(math)}\\
&\cong \texttt{exp(n+k)} && (v_n=v+1)
\end{aligned}
$$

Where `v+k` is valuable because we assumed `v` is a value and `k` is valuable. $\square$

3.3 The Proof

Finally, we come to the correctness claim for `pow`.

Thm. 8 For all values `n:int` with $n \geq 0$,

$$\texttt{pow(n)} \quad \cong \quad \texttt{exp(n)}$$

Note the pattern of recursion utilized by `pow`: while in the `n` odd case the only recursive call made is to `pow(n-1)`, in the even case the recursive call is to `pow(n div 2)`. Given the tight connection between the form of a recursive function and its inductive correctness proof, this suggests to us that a weak inductive hypothesis will not suffice. So we'll prove this by strong induction.

Proof. By strong induction on `n`.

BC `n=0`

$$\texttt{exp 0} \Longrightarrow 1 \qquad \text{Defn. exp}$$
$$\texttt{pow 0} \Longrightarrow 1 \qquad \text{Defn. pow}$$

so, since `pow 0` and `exp 0` evaluate to the same value, they are extensionally equivalent.

IS $n > 0$
IH $\texttt{pow(i)} \cong \texttt{exp(i)}$ for all $0 \leq i < n$
WTS: $\texttt{pow(n)} \cong \texttt{exp(n)}$

Break into two cases: `n` even and `n` odd. We'll start with odd.

$$\begin{aligned}
\texttt{pow(n)} &\cong \texttt{2 * pow(n-1)} && \text{Defn. pow, Lemma 6} \\
&\cong \texttt{2 * exp(n-1)} && \text{IH} \\
&\cong \texttt{exp(n)} && \text{Defn. exp}
\end{aligned}$$

For the even case,

$$\begin{aligned}
&\texttt{pow(n)} \\
&\cong \texttt{square(pow(n div 2))} && \text{Defn. pow, Lemma 6} \\
&\cong \texttt{square(exp(n div 2))} && \text{IH, Lemma 5} \\
&\cong \texttt{(exp(n div 2)) * (exp(n div 2))} && \text{(defn. square, Lemma 5, Prop. 1)} \\
&\cong \texttt{exp((n div 2) + (n div 2))} && \text{Lemma 7, Lemma 5} \\
&\cong \texttt{exp n} && \text{Lemma 4}
\end{aligned}$$

and we're done. □

3.4 Details

Here, we explain each significant step from the preceding proof in greater detail.

- From `n` odd:

  `pow(n)` $\cong$ `2 * pow(n-1)` — Defn. `pow`, Lemma 6

  We need Lemma 6 to guarantee that, since `n` is odd, `even n` will evaluate to `false`, so we'll go into the false-branch of the `case` expression in the definition of `pow`.

- From `n` even:

  `pow(n)` $\cong$ `square(pow(n div 2))` — Defn. `pow`, Lemma 6

  We need Lemma 6 to guarantee that, since `n` is even, `even n` will evaluate to `true`, so we'll go into the true-branch of the `case` expression in the definition of `pow`.

- From `n` even:

  `square(pow(n div 2))`
  $\cong$ `square(exp(n div 2))` — IH, Lemma 5

  We need Lemma 5 here to guarantee for us that `n div 2` is valuable – call its value `i`. Lemma 5 furthermore tells us that `i` is a natural number less than `n`, hence `i` is within the scope of quantification in the inductive hypothesis. So, more fully, we have this reasoning:

  `square(pow(n div 2))`
  $\cong$ `square(pow(i))` — (`n div 2` $\hookrightarrow$ `i`)
  $\cong$ `square(exp(i))` — (IH, $0 \le$ `i` $<$ `n` by Lemma 5)
  $\cong$ `square(exp(n div 2))`. — (`n div 2` $\hookrightarrow$ `i`)

- From `n` even:

  `square(exp(n div 2))`
  $\cong$ `(exp(n div 2)) * (exp(n div 2))`
  (defn. `square`, Lemma 5, Prop. 1)

  Recall that `square` is `fn x => x * x`. So in this equivalence, we're saying that `square` applied to the expression `exp(n div 2)` is equivalent to the body of `square`, namely `x * x`, with both instances of `x` replaced by `exp(n div 2)`. If `exp(n div 2)` were a *value*, this would just be an evaluation step:

  `(fn x => x * x) v` $\Longrightarrow$ `v * v` — if `v` is a value.

But `exp(n div 2)` isn't a value! However, we're in luck: it's enough that `exp(n div 2)` is *valuable*:

$$\texttt{(fn x => x * x) e} \quad \cong \quad \texttt{e * e} \qquad\qquad \text{if } \texttt{e} \text{ is valuable.}$$

Note that is is an extensional equivalence, *not* an evaluation step: SML will evaluate `exp(n div 2)` to a value *before* substituting it into the body of the function, not substitute it unevaluated like written here. But the valuability guarantees that we can evaluate `exp(n div 2)` before substituting, or subtitute before evaluating, and get the same result. We referred to this as the "valuable-stepping principle" in lecture.

We therefore need to justify that `exp(n div 2)` is valuable. Lemma 5 tells us that `n div 2` is valuable and nonnegative. Prop. 1 takes this assumption and derives that `exp(n div 2)` is valuable, as needed.

- From `n` even:

$$\begin{aligned} &\texttt{(exp(n div 2)) * (exp(n div 2))} \\ \cong\ &\texttt{exp((n div 2) + (n div 2))} && \text{Lemma 7, Lemma 5} \end{aligned}$$

Notice that the statement of Lemma 7 demands valuable, nonnegative expressions. Lemma 5 tells us that, since $\texttt{n} > 0$, `(n div 2)` is indeed a nonnegative, valuable expression.

4 Tail-Recursive Implementation

5 Work Analysis