15-150 Fall 2023
Lecture 5

- These notes accompany and do not replace the lecture content.
  Some material from class appear only on the lecture slides.
  Some of the examples given here were omitted in class because of time.
  You should be sure to read the slides and these notes.
- New this semester: a preamble discussing equational and evaluational reasoning in a bit more detail, with examples.

1 Preamble

We talked previously in class about two alternative (complementary) ways to think about program behavior: *evaluationally* and *equationally*. It's good to get familiar with these methods, and to be aware of subtle differences. Most notably, the one-step evaluation relation $\Rightarrow$ is directed from left to right, whereas equivalence (or equality) $=$ is symmetric and compositional. These differences influence the way we reason about code behavior in the two styles.

For evaluational reasoning there are laws that explain the order in which sub-expressions get evaluated, such as (assuming sequential evaluation):

> "$e_1 + e_2$ first evaluates e_1 to a value v_1, then e_2 to a value v_2, then returns the value of $v_1 + v_2$".

This can also be expressed as:

> If $e_1 \Rightarrow^* v_1$ and $e_2 \Rightarrow^* v_2$ then $e_1 + e_2 \Rightarrow^* v_1 + e_2 \Rightarrow^* v_1 + v_2 \Rightarrow v$, where v is the numeral for $v_1 + v_2$.

This law is deducible from the *rules* for the one-step evaluation relation given in class. We won't concern ourselves here with *how* to derive the law, but it really is just a logical consequence of what the rules and definitions say.

It's a little awkward, but proper, that we distinguish between the expression $v_1 + v_2$ and its value, the corresponding numeral; syntactically, $2+2$ and 4 are distinct. After all, evaluation produces syntactic values, and syntactic values of type `int` are numerals. $2 + 2$ isn't a numeral.

Recall that we introduced a collection of rules for `=>`, including:

- In the scope of the declaration `fun f(x)=e` the expression `f` evaluates to (the function value) `fn x => e`.
- If e_1 evaluates to the function value `fn x => e` and e_2 evaluates to the value v, then $e_1 e_2$ evaluates to the expression obtained by substituting v for x in e. (This substituted expression may need further evaluation.)

Again we wrote these out in "English" but the rules can also be expressed using $\Rightarrow$ exactly as we did in class.

As an example, suppose we make the following declaration:

```
fun exp (n) = if n=0 then 1 else 2 * exp(n-1)
```

In the scope of this declaration, we have

$$\texttt{exp} \Rightarrow \texttt{fn n => if n=0 then 1 else 2 * exp(n-1)}$$

The right-hand-side is a function value. Let's call it `M` for short. So we have `exp` $\Rightarrow$ `M`. Using the evaluation rules for application and so on, we also have the following sequence of evaluation steps for the expression `exp 3`:

```
exp 3  ⇒  M 3
       ⇒  if 3=0 then 1 else 2 * exp(3-1)
       ⇒  if false then 1 else 2 * exp(3-1)
       ⇒  2 * exp(3-1)
       ⇒  2 * M(3-1)
       ⇒  2 * M 2
```

showing that `exp 3` $\Rightarrow^*$ `M 3` $\Rightarrow^*$ `2 * M 2`.
But we don't get `exp 3` $\Rightarrow^*$ `2 * exp 2`, because the application `exp(3-1)` steps to `M(3-1)`. Maybe you don't think this is a big deal, but the point is that we should be accurate in what we claim.

Note also that for any value `v` of type `int` we have

```
exp v  ⇒  M v
       ⇒  if v=0 then 1 else v * exp(v-1)
```

but when we apply `exp` to an *expression* `e` that isn't a *value* (like `2+2`) the order of evaluation is different and we do *not* get that

```
exp e  ⇒  M e
       ⇒  if e=0 then 1 else e * exp(e-1)
```

because the substitution rule is only allowed when the argument expression is a value.

Now let's think equationally. In equational reasoning we work with a mathematically defined notion of equality or equivalence for values of the same type. The nature of equivalence of course depends on the type. For `int` this is just the obvious one: integer values are equivalent iff they are the same integer. Two function values of the same type are equivalent iff they map equivalent arguments to equivalent results. We then extend equivalence to (well-typed) expressions of the same type in the obvious way: expressions are equivalent iff they evaluate to equivalent values, or both fail to terminate.

It's not hard to see that at each type the designated notion of equivalence satisfies the "expected" properties, including:

$$\begin{array}{rll} e = e & \text{always} & \\ e_1 = e_2 & \text{implies} & e_2 = e_1 \\ e_1 = e_2 \;\&\; e_2 = e_3 & \text{implies} & e_1 = e_3 \\ e_1 = e \;\&\; e_2 = e & \text{implies} & e_1 = e_2 \end{array}$$

Moreover, equivalence is *compositional*. For example, we have all well-typed instances of the following:

$$\begin{array}{rll} e_1 = e_2 & \text{implies} & e_1 + e = e_2 + e \\ e_1 = e_2 & \text{implies} & e + e_1 = e + e_2 \\ e_1 = e_2 & \text{implies} & e\ e_1 = e\ e_2 \\ e_1 = e_2 & \text{implies} & e_1\ e = e_2\ e \\ e_1 = e_2 & \text{implies} & \texttt{fn x =>}\ e_1 = \texttt{fn x =>}\ e_2 \end{array}$$

As you should expect, the following properties hold, for integer expressions:

$$\begin{array}{rcl} e + 0 & = & e \\ e_1 + e_2 & = & e_2 + e_1 \\ e_1 + (e_2 + e_3) & = & (e_1 + e_2) + e_3 \\ e * 1 & = & e \\ e_1 * e_2 & = & e_2 * e_1 \\ e_1 * (e_2 * e_3) & = & (e_1 * e_2) * e_3 \end{array}$$

You can also check that the following statement is true; we'll use it below. If $e_1 =$ `fn x => e` and $e_2 = v$ (and the type of v is suitable as argument to this function), then $e_1\ e_2 =$ (`fn x => e`) v, which is also equal to the expression obtained by substituting v for x in e.

In the scope of declaration `fun f(x)=e` the following equation holds:

$$\texttt{f} = \texttt{fn x => e}$$

This equation amounts to "unfolding" the recursive function declaration, and it implies that the function behaves exactly as you would expect: any call to `f` in the function body `e` is a "recursive call".

Armed with all these basic facts about equivalence/equality, let's look again at the example. Suppose we make the following declaration:

```
fun exp (n) = if n=0 then 1 else 2 * exp(n-1)
```

In the scope of this declaration, we have

$$\texttt{exp} = \texttt{fn n => if n=0 then 1 else 2 * exp(n-1)}$$

using the unfolding equation. Again let's use the name `M` as an abbreviation for the function value on the right. So we have `exp` = `M`. We also have the following equational derivation for the expression `exp 3`:

$$\begin{array}{rcl} \texttt{exp 3} & = & \texttt{M 3} \\ & = & \texttt{if 3=0 then 1 else 2 * exp(3-1)} \\ & = & \texttt{if false then 1 else 2 * exp(3-1)} \\ & = & \texttt{2 * exp(3-1)} \end{array}$$

mimicking the evaluational steps as before. Note that we've appealed silently here to referential transparency. For example, since `exp` = `M` it follows that `exp 3` = `M 3`. So far this derivation looks just like what we did when we analyzed the evaluation steps for `exp 3` , but thinking equationally, we can now take a different path. Indeed, we see that (since `3 - 1 = 2`)

$$\begin{array}{rcl} \texttt{2 * exp(3-1)} & = & \texttt{2 * exp(2)} \\ & = & \texttt{2 * M 2} \end{array}$$

so that we get `exp 3 = 2 * exp 2` (and we also have `M 3 = 2 * M 2`). Again, to emphasize what's different: we did *NOT* have `exp 3` $\Rightarrow^*$ `2 * exp 2`.

In case you are worried about this "discrepancy" between the two styles of reasoning, there's no need. We actually have `exp 3 = 8` and `exp 2 = 4` and `8 = 2 * 4`. Also of course `M 3 = 8` and so on.

Each style of reasoning (evaluational, equational) gives an accurate way to explain program behavior, provided you pay attention to any side conditions governing evaluation rules or constraining equations: most importantly, some of them require one or more of the expressions to be a (syntactic) value.

Given the importance of *expressions-that-evaluate-to-a-value* it's worth giving them a suggestively positive-sounding name: we'll say that e is *valuable* if it evaluates to some value. (The type guarantee of ML tells us that if e has type `t`, and e evaluates to v, then v also has type `t`.) One nice property of valuable expressions is that it's very easy to reason about their behavior when used in a program. For example, recall that we know that

$$\texttt{exp} = \texttt{fn n => if n=0 then 1 else 2 * exp(n-1)}.$$

We already saw that whenever `v` is a value of type `int`, i.e. a numeral, we can derive

```
exp v  =  (fn n => if n=0 then 1 else 2 * exp(n-1)) v
       =  if v=0 then 1 else 2 * exp(v-1)
```

Now let `e` be an expression of type `int` that evaluates to `v`. Then we also have `e = v`, and we can say:

```
exp e  =  (fn n => if n=0 then 1 else 2 * exp(n-1)) e
       =  (fn n => if n=0 then 1 else 2 * exp(n-1)) v
       =  if v=0 then 1 else 2 * exp(v-1)
       =  if e=0 then 1 else 2 * exp(e-1)
```

Since `exp` and `e` here are valuable, `exp e` is equal to the result of substituting `e` for `n` in function body. This is true, even though the equational rule for application used in the second line above needs a value `v`.

Finally, as we mentioned in class, evaluation is consistent with equational reasoning, in the following sense:

- If e evaluates to value v, then $e = v$.
- If e fails to terminate, there's no value such that $e = v$

I hope this discussion helps to clarify what's needed when you reason about functional program behavior. Once you get used to it, especially the equational style, it becomes easier to do "math-like" analysis of code. Just be careful about *valuability*, and only ever do substitution into a function body when the argument expression has a value,.

Equational reasoning lets us think of code in mathematical terms, working with standard mathematical notions such as partial functions. Evaluational reasoning gives us a way to focus on order-of-evaluation, crucial when we need to count steps. The two approaches are complementary — they fit together nicely.

Now we're ready for the major topic of this lecture ... and a big theme of the semester: reasoning about efficiency of functional code.

2 Introduction

We introduce techniques for analyzing or estimating the runtime of functional programs, with emphasis on asymptotic analysis, under various assumptions about evaluation strategy. We deal mainly with *work* (assuming sequential evaluation) but we also talk about *span* (allowing for parallel evaluation of independent code). Both work and span will be explained more fully in later lectures, where we will consider some more interesting examples as well.

You may already be familiar with basic concepts of asymptotic analysis and "big-O" notation, but in any case we give a brief recap of the main ideas. The main math techniques (such as recurrences) are relevant both for work and span.

Main points

- Primitive operations (like arithmetic) count as constant time, or one single step. In calculating work for a compound expression we *add* the work for sub-expressions, as they will get evaluated consecutively (on a single processor). To calculate *span* we *add* the span for *dependent* sub-expressions (dependency forces them to get evauated one after the other) but we take the *maximum* of the spans for independent sub-expressions (which may be computed simultaneously).

- To evaluate $e_1 @ e_2$ we evaluate e_1 and e_2 to get two list values (can be done in parallel), then prepend the first list on the front of the second (with a bunch of cons operations). In a tuple expression like (e_1, e_2, e_3) the component sub-expressions e_1, e_2 and e_3 are independent, so can be evaluated in parallel.

- We show how to obtain a *recurrence relation* for the runtime of an ML function when applied to an argument with a given size. The shape of the recurrence is based on the way the function is defined. We show how to find exact solutions to recurrences, or an asymptotic approximation when an exact solution is not needed or not feasible. We list asymptotic solutions for some common recurrence relations.

- Sometimes the efficiency of a function can be improved by introducing an "accumulator", or by computing extra information. Mathematical insight may also lead to more efficient code. We give some examples.

3 Asymptotic analysis

We will focus today on *asymptotic analysis* of the *work* done during evaluation of functional programs. This kind of analysis predicts how long it will take to run your code on really big inputs, without actually running it. It is one of the main tools used to choose between different algorithms for the same problem. Underlying this kind of analysis is the assumption that primitive operations (such as arithmetic and boolean operators, or cons-ing an item onto a list) take constant time and that we don't care about (and don't need to know) the precise value of these constants. Moreover we only really "care" about what happens with "large" arguments, because after all one could easily re-design a function to do something fancy for a few small arguments, and that would likely have no significant effect on how the function works on large arguments.

We also want our analysis to be robust. So "work" isn't going to be expressed in units like micro-seconds, seconds, minutes, hours, or days; it's not sensible to claim that your piece of code takes 33 milliseconds to produce the result 42. Rather we will show how to prove assertions like "the work to evaluate $f(L)$ is proportional to n^2 when L is a list of integer values of length n". This will enable us to make predictions about the way code speeds up or slows down when we change one piece for another. Implicitly, work represents the number of "basic" steps needed to evaluate the piece of code, and we usually express it as some function of the "size" of some parameter (typically, the argument to some relevant function).

big-O classification

Asymptotic analysis is based on big-O classifications: $\texttt{O}(1)$ or "constant time"; $\texttt{O}(n)$ or "linear"; $\texttt{O}(n^2)$ or "quadratic"; $\texttt{O}(\mathit{log}\ n)$, or "logarithmic"; and so on. As we have said, big-O abstracts away from constant factors. So an algorithm with running time proportional to $50000n^3$ is $\texttt{O}(n^3)$ and so is an algorithm with running time $2n^3$. In fact constant factors sometimes do make a difference, practically, especially for low input sizes; but usually the behavior when inputs get very large is more significant. And we would clearly prefer a running time of $50000n^3$ to a running time of 2^n, since $2^n > 50000n^3$ for all large enough values of n. Thus we say that $\texttt{O}(n^3)$ is better than or faster than $\texttt{O}(2^n)$.

More rigorously, for two functions f, g of type `int -> int` we say that

"f is $\mathtt{O}(g)$" if there is a (real-valued) constant c and an integer N such that for all $n \geq N$, $|f(n)| \leq c|g(n)|$.

When the values of $f(n)$ and $g(n)$ are always non-negative (e.g. when they represent running times of code fragments!) we can elide the absolute value signs and just say "f is $\mathtt{O}(g)$" when "for all $n \geq N$, $f(n) \leq c * g(n)$". Similarly we only usually care about non-negative values of n, because in our analysis n usually stands for some measure of "argument size".

We often say "for sufficiently large n" as an abbreviation for "for all $n \geq N$, for some N".

We usually simplify and write something like $30n^2 + 4000n + 1$ is $\mathtt{O}(n^2)$, rather than naming the functions (e.g. "let $f(n) = 30n^2 + 4000n + 1 \ldots$").

We may take advantage of well known results about big-O notation, for instance the fact that "constants don't matter". At the end of the notes for today we summarize some key results.

Comments

We say that "f is $\mathtt{O}(g)$". Some people use "$f = \mathtt{O}(g)$" or "$f \in \mathtt{O}(g)$". Sometimes we'll write something like $f(n) = \mathtt{O}(n) + n^2$ to mean that there is a function $g(n)$ belonging to $\mathtt{O}(n)$ such that $f(n) = g(n) + n^2$. Note that in this case it follows that f is actually $\mathtt{O}(n^2)$.

It is common to use notation like $\mathtt{O}(n^2) \subset \mathtt{O}(n^3)$ to indicate the (true) fact that every function that is $\mathtt{O}(n^2)$ is also $\mathtt{O}(n^3)$. The use of the "proper inclusion" symbol $\subset$ emphasizes the (also true) fact that it isn't true that every function that is $\mathtt{O}(n^3)$ is also $\mathtt{O}(n^2)$. (Indeed, it is pretty obvious that the function $f(n) = n^3$ is $\mathtt{O}(n^3)$ but not $\mathtt{O}(n^2)$.) We may also write (true) statements such as $\mathtt{O}(n^2) = \mathtt{O}(n^2 + 543n + 42)$ and $\mathtt{O}(n^2 + 2^n) = \mathtt{O}(2^n)$ to indicate when two big-O classes contain exactly the same functions.

4 Examples

In these examples we sometimes omit the type annotations that we have insisted on previously, because we want to focus on runtime analysis and all of our code is known to be well-typed; you should continue to obey the requirements in your own code development, until we tell you otherwise! As an exercise, you can figure out how to put type annotations into our examples. In any case, our code runs perfectly well without the extra type information. As we will see shortly, ML can do a lot of type inference in the background.

We also relax the requirement to include REQUIRES and ENSURES comments, again because the focus here is on runtime, not on correctness. Nevertheless we try to give clear informal specifications, and we indicate how you could prove correctness. Moreover, sometimes we *need* to know something about the applicative behavior of a function to justify our runtime analysis; in such cases it will be vital to have a good specification for that function! Indeed, in many of the examples, we do make assertions about the applicative behavior of our functions, which talk about the results produced by a function when applied to an argument. You can use the proof techniques from the previous lectures to fill in the details, if you so desire. (And some of the examples will already have been covered in class.)

4.1 Powers of 2

Here is a simple ML function `exp` for calculating powers of 2.

```
(* exp : int -> int *)
fun exp (n:int):int = if n=0 then 1 else 2 * exp (n-1)
```

It is easy to prove by induction that for all $n \geq 0$, `exp` $n = 2^n$.

Let $W_{\texttt{exp}}(n)$ be the running time (or "work") of `exp` n, for $n \geq 0$. We assume (as usual) that arithmetic and boolean operations take constant time. It should then be clear from the structure of the function definition that there are (non-negative) constants c_0, c_1 such that

$$\begin{aligned} W_{\texttt{exp}}(0) &= c_0 \\ W_{\texttt{exp}}(n) &= c_1 + W_{\texttt{exp}}(n-1), \text{ for } n > 0. \end{aligned}$$

Note that the program design determines the form of the recurrence: the recursive call happens only when the argument is non-zero, and it happens

after the test expression gets evaluated, so we add. Think of c_0 as representing the number of basic operations needed to evaluate `exp(0)` down the the value 1, and c_1 as the number of basic operations needed when $n > 0$, to get from $\texttt{exp}(n)$ to $\texttt{2} * \texttt{exp}(n-1)$. (Using the evaluation rules for our programming language we could actually calculate what these constants are: the number of `=>` steps taken, in each case. But the details don't really matter so it is convenient just to use named constants with unspecified values in the analysis.)

Using this recurrence relation it is easy to prove, by induction on n, that for all $n \geq 0$, $W_{\texttt{exp}}(n) = n * c_1 + c_0$. Exercise: prove this.

This result, that for all $n \geq 0$, $W_{\texttt{exp}}(n) = n * c_1 + c_0$, is called a *closed form* solution of the recurrence relation. This closed form makes it obvious that $W_{\texttt{exp}}(n)$ is *linear* in n.

It is also easy to show, using this closed form, that $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$. Here is a sketch of the details. We know that there are (positive) constants c_0 and c_1 such that $W_{\texttt{exp}}(n) = n * c_1 + c_0$, for all $n \geq 0$. Pick c to be $c_1 + 1$ and let $N = c_0$. Then for all $n \geq N$ we have

$$W_{\texttt{exp}}(n) = n * c_1 + c_0 \leq n * (c_1 + 1) = c * n.$$

Thus, according to the definition of big-O, $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$. In other words, the running time for `exp` n is linear in n.

Actually, it can be convenient to make a simplifying assumption about these "unknown" constants. It should be clear that for *any* non-negative constants $\texttt{c}_0$ and $\texttt{c}_1$, the function $f(n) = n * \texttt{c}_1 + \texttt{c}_0$ is $\texttt{O}(n)$. The choice of constants makes no difference to this fact. So we could have made an arbitrary decision to choose $\texttt{c}_0 = \texttt{c}_1 = 1$ and taken the recurrence defining $W_{\texttt{exp}}$ to be

$$\begin{aligned} W_{\texttt{exp}}(0) &= 1 \\ W_{\texttt{exp}}(n) &= 1 + W_{\texttt{exp}}(n-1), \text{ for } n > 0. \end{aligned}$$

We would have then been able to show that $W_{\texttt{exp}}(n) = n + 1$ for $n \geq 0$, and hence that $W_{\texttt{exp}}(n)$ is $\texttt{O}(n)$ as before.

Having shown that $W_{\texttt{exp}}(n) = n * \texttt{c}_1 + \texttt{c}_0$, now let's see how that connects with evaluation steps and how we can do more sophisticated runtime analysis using this information.

Firstly, assuming that the constants were chosen to match up with how many `=>` evaluation steps really happen, we can safely say that:

For all values $n \geq 0$,

$$\texttt{exp}(n) \Longrightarrow^{(n\texttt{c}_1+\texttt{c}_0)} k, \quad \text{where } k \text{ is the value of } 2^{\texttt{n}}$$

Here we annotate the evaluation symbol not with `*` but with an indication of the number of steps taken, i.e. we write $=>^{(m)}$ where m is the number of steps, rather than $=>^*$, which we used to mean in some finite number of steps. (Also I use the nicer looking notation $\Longrightarrow$ instead of `=>` simply because it is available in LaTeX math mode!)

Using the above property, we can now answer questions about code that uses the function `exp`. For example, when $n \geq 0$, what is the work needed to evaluate the expression $\texttt{exp}(\texttt{exp}(n))$? We can figure this out as follows, using the evaluation rules to guide us. We have:

$$\begin{array}{lll} \texttt{exp}(\texttt{exp}(n)) & \Longrightarrow^{(1)} & (\texttt{fn x } \ldots)(\texttt{exp}(n)) \\ & \Longrightarrow^{(n\texttt{c}_1+\texttt{c}_0)} & (\texttt{fn x } \ldots)(k) \\ & \Longrightarrow^{(k\texttt{c}_1+\texttt{c}_0-1)} & m \end{array}$$

where k is the integer value of 2^n and m is the integer value of 2^k, so $m = 2^{2^n}$. The third line in this derivation is justified because we know from the above property that

$$\texttt{exp}(k) \Longrightarrow^{(k\texttt{c}_1+\texttt{c}_0)} m,$$

and this computation begins with the step

$$\texttt{exp}(k) \Longrightarrow^{(1)} (\texttt{fn x } \ldots)(k),$$

so the rest of this computation looks like

$$(\texttt{fn x } \ldots)(k) \Longrightarrow^{(k\texttt{c}_1+\texttt{c}_0-1)} m.$$

So the overall conclusion here is that the number of steps to evaluate $\texttt{exp}(\texttt{exp}(n))$, the *work* for this expression, is

$$1 + (n\texttt{c}_1 + \texttt{c}_0) + (k\texttt{c}_1 + \texttt{c}_0 - 1) = n\texttt{c}_1 + 2^n\texttt{c}_1 + \texttt{c}_0.$$

In this formula the linear term grows less quickly than the exponential term, and we therefore deduce that the work for this expression is $\texttt{O}(2^n)$.

4.2 Powers of 2, faster

Now let's define a (more efficient) function that takes advantage of some simple mathematical facts about powers of 2. Specifically whenever $n > 0$, either n is even, and $2^n = (2^{n \texttt{ div } 2})^2$; or n is odd, and $2^n = 2 * 2^{n-1}$.

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

(* fastexp : int -> int *)
fun fastexp (n:int):int =
   if n = 0 then 1 else
   if (n mod 2 = 0) then square (fastexp (n div 2))
                    else 2 * fastexp (n-1)
```

Again it is easy to prove that for all $n \geq 0$, `fastexp` $n = 2^n$.

Now let $W_{\texttt{fastexp}}(n)$ be the runtime of `fastexp` n, for $n \geq 0$. Again the structure of the function definition tells us that there are constants k_0, k_1, k_2 such that:

$$\begin{array}{lcll} W_{\texttt{fastexp}}(0) & = & k_0 & \\ W_{\texttt{fastexp}}(n) & = & k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) & \text{if } n > 0 \text{ and } n \text{ even} \\ W_{\texttt{fastexp}}(n) & = & k_2 + W_{\texttt{fastexp}}(n-1) & \text{if } n > 0 \text{ and } n \text{ odd} \end{array}$$

Hence, because $n - 1$ is even and non-negative when n is odd and positive, and in such a case $(n-1)$ `div` 2 is equal to n `div` 2, we actually have:

$$\begin{array}{lcll} W_{\texttt{fastexp}}(0) & = & k_0 & \\ W_{\texttt{fastexp}}(n) & = & k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) & \text{if } n > 0 \text{ and } n \text{ even} \\ W_{\texttt{fastexp}}(n) & = & k_2 + k_1 + W_{\texttt{fastexp}}(n \texttt{ div } 2) & \text{if } n > 0 \text{ and } n \text{ odd} \end{array}$$

Since we only care about the *asymptotic* runtime, we lose no generality by expanding out the case for $n = 1$, setting all constants to 1, and working with the recurrence relation given by

$$\begin{array}{lcl} T_{\texttt{fastexp}}(0) & = & 1 \\ T_{\texttt{fastexp}}(1) & = & 1 \\ T_{\texttt{fastexp}}(n) & = & 1 + T_{\texttt{fastexp}}(n \texttt{ div } 2) \text{ for } n > 1. \end{array}$$

$T_{\texttt{fastexp}}$ defined this way is obviously not the same function as $W_{\texttt{fastexp}}$ as given above, but it can be shown that these two functions have the same asymptotic behavior. It's much easier to find a closed form for $T_{\texttt{fastexp}}$.

Indeed this recurrence for $T_{\texttt{fastexp}}$ is *exactly the same recursive pattern* as we used in lab to define the logarithm function `log : int -> int`, and we already proved in lab that this function computes logarithms in base 2. So we can get a closed form for $T_{\texttt{fastexp}}(n)$ immediately: For all $n \geq 1$, $T_{\texttt{fastexp}}(n) = \texttt{log}_2(n)$. Recall that $log_2\ n$ is the largest non-negative integer k such that $2^k \leq n$.

This doesn't imply that $W_{\texttt{fastexp}}(n)$ is also equal to $\texttt{log}_2(n)$ — it couldn't be, because its recurrence relation mentions k_0, k_1, k_2. But we said that $W_{\texttt{fastexp}}$ and $T_{\texttt{fastexp}}$ have the same asymptotic behavior. That means that $W_{\texttt{fastexp}}(n)$ is in the same $\texttt{O}$-class as $T_{\texttt{fastexp}}(n)$. Hence $W_{\texttt{fastexp}}(n)$ is $\texttt{O}(log_2\ n)$.

Recall another well known property of big-O notation: $\texttt{O}(log_2\ n)$ means the same as $\texttt{O}(log_3\ n)$, and so on. The choice of logarithmic base makes no difference to big-O classification. We simply say that $T_{\texttt{fastexp}}(n)$ is $\texttt{O}(log\ n)$.

4.3 Powers of 2, faster or slower

Here is yet another exponentiation function, based on the facts that for $n > 1$, if n is even then $2^n = (2^{n\ \texttt{div}\ 2})^2$ and if n is odd then $2^n = 2(2^{n\ \texttt{div}\ 2})^2$. We give this function a different name, so we can compare it with the previous functions.

```
(* pow : int -> int *)
fun pow 0 = 1
 |  pow 1 = 2
 |  pow n = let
              val k = pow(n div 2)
            in
              if n mod 2 = 0 then k*k else 2*k*k
            end
```

Notice that we use a local variable `k` to hold the value returned by the recursive call, and this variable gets used twice (in each branch). This fact turns out to be crucial in our analysis!

Again it is easy to prove by induction on n that for all $n \geq 0$, `pow` $n = 2^n$. Indeed this function does compute powers of 2. How about its running time, when applied to a non-negative integer?

In each recursive call, the argument gets halved. So we should expect logarithmic running time. Our recurrence analysis confirms this. Let $W_{\texttt{pow}}(n)$

be the runtime of `pow` n, for $n \geq 0$. The function definition tells us that there are constants c_0, c_1, c_2 such that:

$$\begin{array}{rcl} W_{\texttt{pow}}(0) & = & c_0 \\ W_{\texttt{pow}}(1) & = & c_1 \\ W_{\texttt{pow}}(n) & = & c_2 + W_{\texttt{pow}}(n \texttt{ div } 2) \text{ if } n > 1 \end{array}$$

This is essentially the same recurrence as the one for $W_{\texttt{fastexp}}$, so the runtime of `pow` n is $\texttt{O}(log\ n)$, the same as for `fastexp`(n), asymptotically.

The use of a local variable in the above function definition, to save and re-use the value returned by the recursive call, is crucial for efficiency. Here is a bad version that makes redundant recursive calls. Compare the code with that of `pow`.

```
(* badexp : int -> int *)
fun badpow 0 = 1
 |  badpow 1 = 2
 |  badpow n = let
                 val k2 = badpow(n div 2) * badpow(n div 2)
               in
                 if n mod 2 = 0 then k2 else 2*k2
               end
```

Let $W_{\texttt{badpow}}(n)$ be the runtime of `badpow` n, for $n \geq 0$. Then (again, from the function definition) we can derive the recurrence

$$\begin{array}{rcl} W_{\texttt{badpow}}(0) & = & 1 \\ W_{\texttt{badpow}}(1) & = & 1 \\ W_{\texttt{badpow}}(n) & = & 1 + 2 * W_{\texttt{badpow}}(n \texttt{ div } 2) \text{ if } n > 1 \end{array}$$

If n is a power of 2, say $n = 2^k$, we have $W_{\texttt{badpow}}(2^k) = 2 * W_{\texttt{badpow}}(2^{k-1}) + 1$. Expanding out a few examples, we see that

$$\begin{array}{rcl} W_{\texttt{badpow}}(2^0) & = & 1 \\ W_{\texttt{badpow}}(2^1) & = & 1 + 2W_{\texttt{badpow}}(2^0) = 1 + 2 = 3 \\ W_{\texttt{badpow}}(2^2) & = & 1 + 2W_{\texttt{badpow}}(2^1) = 1 + 2 + 4 = 7 \end{array}$$

These examples suggest that for $k \geq 0$, $W_{\texttt{badpow}}(2^k) = 2^{k+1} - 1$. Indeed this can be shown by induction on k. So $W_{\texttt{badpow}}(2^k)$ is $\texttt{O}(2^k)$, and it can further be shown that for generally $W_{\texttt{badpow}}(n)$ is $\texttt{O}(n)$, so `badpow` has *linear* runtime!

Clearly, we should prefer `pow`, with logarithmic running time, over `badpow`, with linear runtime.

This simple example shows that attention to detail and careful design can improve efficiency.

4.4 General exponentiation

We can easily derive a function for computing b^n, where $n \geq 0$ and b is an integer. The main idea is that when n is even and greater than 2, $b^n = (b^2)^{n \texttt{ div } 2}$. We were unable to take advantage of this particular kind of math fact earlier, because we were fixated on computing powers of 2. By tackling a more general task we actually make it possible to exploit results from math to develop a faster algorithm.

```
(* gexp : int * int -> int *)
fun gexp (b, 0) = 1
 |  gexp (b, 1) = b
 |  gexp (b, n) =
      let
         val k = gexp (b*b, n div 2)
      in
          if n mod 2 = 0 then k else b*k
      end
```

It is (again!) easy to prove by induction on n that for all b and all $n \geq 0$, $\texttt{gexp}(b, n) = b^n$. The runtime of $\texttt{gexp}(b, n)$ is $\texttt{O}(log\ n)$. We can easily adapt the analysis for `pow` to show this.

4.5 Fibonacci numbers

Here is a simple ML definition that corresponds to the usual mathematical presentation of the Fibonacci series. For $n \geq 0$ we represent the n^{th} Fibonacci number as the value of `fib` n.

```
(* fib : int -> int *)
fun fib 0 = 1
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2)
```

If we use this function in the ML interpreter window we will see that `fib` 42 takes a very long time to return its result; and `fib` 43 raises the `Overflow` exception, because the 43^{rd} Fibonacci number is too large.

Let $W_{\texttt{fib}}(n)$ be the running time for `fib`(n). Then, choosing the relevant constants to be 1, we obtain the recurrence relation

$$\begin{aligned} W_{\texttt{fib}}(0) &= 1 \\ W_{\texttt{fib}}(1) &= 1 \\ W_{\texttt{fib}}(n) &= 1 + W_{\texttt{fib}}(n-1) + W_{\texttt{fib}}(n-2) \text{ for } n > 1 \end{aligned}$$

While this recurrence doesn't seem easily solvable (at least, not explicitly), it is obvious that `fib`$(n) \leq W_{\texttt{fib}}(n)$ for all $n \geq 0$. And mathematicians have proven that `fib`(n) is exponential in n, so $W_{\texttt{fib}}$ has *at least* exponential running time. No wonder `fib` 42 is so slow! It can be shown that $W_{\texttt{fib}}(n)$ is actually `O(fib`(n)`)`, so `fib` indeed has exponential running time.

We can speed up the code by computing two Fibonacci numbers in each iteration; we introduce a helper function that does this.

```
(* fastfib : int -> int *)
(* Local function loop : int * int * int -> int *)
(* Counts from n down to 0 *)
fun fastfib n =
  let
    fun loop(i, a, b) = if i=0 then a else loop(i-1, b, a+b)
  in
    loop(n, 1, 1)
  end
```

Let $W_{\texttt{loop}}(n)$ be the running time for `loop`(n, a, b), when $n \geq 0$. (Clearly the running time does not depend on the values of a and b.) We have, from the function definition, that there are constants c_0, c_1 such that

$$\begin{aligned} W_{\texttt{loop}}(0) &= c_0 \\ W_{\texttt{loop}}(i) &= c_1 + W_{\texttt{loop}}(i-1) \text{ for } i > 0 \end{aligned}$$

Hence $W_{\texttt{loop}}(n)$ is `O`(n). And therefore so is the running time of `fastfib` n.

Actually, `fastfib 42` returns the result very quickly, but `fastfib 43` raises the `Overflow` exception because the 43rd Fibonacci number is too large.

Note that the functions `fib` and `fastfib` are extensionally equivalent, even though their runtimes differ significantly.

Exercise: prove that `fib` = `fastfib`. You will need to state and prove a suitable specification for `loop`.

Warning

Introducing an accumulator to improve efficiency is a widely useful technique. As we will see later, depending on the setting, you may want to choose the accumulator to be a list, an integer, a function, or a value of some other type. But this technique is not a panacea: the trick doesn't always truly improve runtime! Consider:

```
(* exp' : int * int -> int *)
fun exp' (n, a) = if n=0 then a else exp' (n-1, 2*a);
fun Exp n = exp'(n, 1);
```

`exp'` is obtained from `exp` by adding an accumulator integer argument, but has the same asymptotic behavior. And `Exp` is extensionally equivalent to `exp` from before. The runtime for `exp'(n, a)` is also `O(n)`, just like the runtime for `exp(n)`.

5 big-O classes

- `O`(1), or constant time
- `O`$(log\ n)$, or logarithmic
- `O`(n), or linear
- `O`(n^2), or quadratic
- `O`(n^3), or cubic
- `O`(2^n), `O`(3^n), ... exponentials (each is a different class)

6 Some useful facts

The following facts can help explain common terminology:

- $\mathtt{O}(log_2\ n)$ is the same class of functions as $\mathtt{O}(log_{10}\ n)$. In fact the base of the logarithm makes no difference to the class of functions, so we usually just write $\mathtt{O}(log\ n)$ and refer to "logarithmic" time.
- A function is called *polynomial time* if it is $\mathtt{O}(n^k)$ for some $k \geq 0$.
- A polynomial function with highest power k is $\mathtt{O}(n^k)$.
- A linear function of n has the form $\alpha n + \beta$, for some constants α and β. Every linear function is $\mathtt{O}(n)$.
- Every function that is $\mathtt{O}(n^2)$ is also $\mathtt{O}(n^3)$, but the converse fails.
- Every function that is $\mathtt{O}(2^n)$ is also $\mathtt{O}(3^n)$, but the converse fails.
- A function of n is said to be *exponential time* if it is $\mathtt{O}(k^n)$ for some constant k.
- Every polynomial time function is also exponential time. (We're not going to make any claims about the converse!)
- If $f(n)$ is $\mathtt{O}(g(n))$, then $\mathtt{O}(f(n) + g(n))$ is the same as $\mathtt{O}(g(n))$.

7 Common recurrences

It's common to use sloppy notation and write something like $\mathtt{O}(g(n))$ in a recurrence, in a place where an actual function of n is intended. For example if we write $W(n) = \mathtt{O}(n) + W(n-1)$ we mean that there is some function $f(n) \in \mathtt{O}(n)$ such that $W(n) = f(n) + W(n-1)$. It doesn't make any difference, asymptotically. For any non-zero linear function f the solution to this recurrence is going to be $\mathtt{O}(n^2)$.

We give the clause for $n > 0$. In each case c and/or k are constants. We mention in each case the most informative time-complexity class of the solution to the recurrence relation.

- $T(n) = T(n\ \mathtt{div}\ 2) + c$, or $T(n) = T(n\ \mathtt{div2}) + \mathtt{O}(1)$
  $T(n)$ is $\mathtt{O}(log\ n)$

- $T(n) = T(n-1) + c$, or $T(n) = T(n-1) + \mathtt{O}(1)$
  $T(n)$ is $\mathtt{O}(n)$

- $T(n) = 2 * T(n \texttt{ div } 2) + c$, or $T(n) = 2 * T(n \texttt{ div } 2) + \mathtt{O}(1)$
  $T(n)$ is $\mathtt{O}(n)$

- $T(n) = T(n-1) + c * n$, or $T(n) = T(n-1) + \mathtt{O}(n)$
  $T(n)$ is $\mathtt{O}(n^2)$

- $T(n) = 2 * T(n \texttt{ div } 2) + c * n$, or $T(n) = 2 * T(n \texttt{ div } 2) + \mathtt{O}(n)$
  $T(n)$ is $\mathtt{O}(n \, log \, n)$

- $T(n) = k * T(n-1) + c$, $k > 1$
  $T(n)$ is $\mathtt{O}(k^n)$
  This is also the case for $T(n) = k * T(n-1) + \mathtt{O}(1)$.

8 Guessing or estimating solutions

Often it is easy to expand out a few examples and look for a pattern from which we can guess a solution. Here we sketch how to justify some of the above facts about common recurrences. Since we are only sketching the ideas we won't provide completely rigorous inductive proofs.

- $T(n) = T(n \texttt{ div } 2) + c$
  For $n = 2^m$ with $m > 0$ we have

  $$T(2^m) = T(2^{m-1}) + c = T(2^{m-2}) + c + c$$

  and it looks like $T(2^m) = T(2^{m-k}) + kc$, for $0 \leq k \leq m$. In particular, $T(2^m) = T(0) + mc$. This suggests that $T(2^m)$ is $\mathtt{O}(m)$. Extrapolating to all values of n, we expect that $T(n)$ is $\mathtt{O}(log \; n)$.

- $T(n) = 2 * T(n \texttt{ div } 2) + c$
  For $n = 2^m$ with $m > 0$ we have

  $$T(2^m) = 2T(2^{m-1}) + c = 2(2T(2^{m-2}) + c) + c = 2^2 T(2^{m-2}) + (2+1)c$$

  and it looks like

  $$T(2^m) = 2^k T(2^{m-k}) + (2^{k-1} + \cdots + 2 + 1)c$$

for $0 \leq k \leq m$. In particular, putting $k = m$,

$$T(2^m) = 2^m T(1) + (2^{m-1} + \cdots + 2 + 1)c.$$

Note that $2^{m-1} + \cdots + 2 + 1 = 2^m - 1$ is $\mathtt{O}(2^m)$. So this analysis suggests that $T(2^m)$ is $\mathtt{O}(2^m)$. Extrapolating to all values of n, we expect that $T(n)$ is $\mathtt{O}(n)$. When extrapolating like this we are usually appealing to the fact that when $2^k \leq n < 2^{k+1}$ we have $T(2^k) \leq T(n) \leq T(2^{k+1})$, so if $T(2^k)$ is approxiately ck we get $ck \leq T(n) \leq c(k+1)$, from which we see that $T(n) \leq c\mathtt{log}_2\ n + c$. Hence $T(n)$ is $\mathtt{O}(\mathtt{log}\ n)$.

This result is consistent with the table of standard recurrences given earlier.

9 Going forward

- Get used to deriving recurrences for your recursive function designs and choosing more efficient designs when available.
- Be aware of how you could solve recurrences inductively, either exactly or asymptotically. Practice on examples.
- Learn to recognize commonly occurring recurrences: this can save you a lot of time!
- You won't need to work extensively with the internal details of big-O notation, although we have given you some insights that show the way.

10 Self-test

1. Let `e` be a well-typed expression of type `int` such that `e` $\Rightarrow^*$ `42`. Which of the following statements are true?

   (a) `(fn x => [x,x,x])e` $\Rightarrow$ `[42,42,42]`

   (b) `(fn x => [x,x,x])e` $\Rightarrow^*$ `[42,42,42]`

   (c) `[42,42,42]` $\Rightarrow^*$ `(fn x => [x,x,x])e`

   (d) `(fn x => [x,x,x])e` $=$ `[42,42,42]`

   (e) `[42,42,42]` $=$ `(fn x => [x,x,x])e`

2. Suppose we make the following declarations:

   ```
   fun loop (x:int) : int = loop x;
   fun zero (y:int) : int = 0;
   ```

   Which of the following statements are true (in scope of these declarations)?

   (a) `zero (loop 42)` $\Rightarrow^*$ `0`

   (b) `loop (zero 42)` $\Rightarrow^*$ `loop 0`

   (c) `loop 0` $=$ `loop 1`

   (d) `zero (loop 42)` $=$ `0`

3. Let $W(n)$ be defined by the following recurrence, in which c_0 and c_1 are positive constants:

$$\begin{aligned} W(0) &= c_0 \\ W(n) &= c_1 n + W(n-1), \text{ for } n > 0, \end{aligned}$$

   Prove by induction that for all $n \geq 0$, $W(n) = c_0 + \frac{1}{2}n(n+1)c_1$. Explain why this implies that $W(n)$ is $\mathtt{O}(n^2)$.

4. Using the definition of "f is $\mathtt{O}(g)$", show that when c_0 and c_1 are integers and $c_1 \neq 0$, the function $f(n) = c_0 + nc_1$ is not $\mathtt{O}(1)$.

5. Show that the function $3n^2 + 4$ is $\mathtt{O}(8n^3)$.

6. Consider the function W_1 given by the following recurrence, on non-negative integer arguments:

$$\begin{aligned} W_1(0) &= 1 \\ W_1(n) &= 3W_1(n-1)+1 \quad \text{for } n > 0 \end{aligned}$$

   (a) Draw a table showing the values of $W_1(n)$ for $n = 0, 1, 2, 3, 4$.

   (b) Augment your table to show the values of $W_1(k+1) - W_1(k)$ for $k = 0, 1, 2, 3$.

   (c) Prove, by induction on n, that for all $n \geq 0$, $W_1(n) = \frac{1}{2}(3^{n+1} - 1)$.

   (d) Deduce that $W_1(n)$ is $\mathtt{O}(3^n)$.

7. Now consider the function W_2 given by the following recurrence, on non-negative integer arguments:

$$\begin{aligned} W_2(0) &= 1 \\ W_2(1) &= 1 \\ W_2(n) &= 3W_2(n-2)+1 \quad \text{for } n > 1 \end{aligned}$$

   (a) Draw a table showing the values of $W_2(n)$ for $n = 0, 1, 2, 3, \ldots, 8$.

   (b) Prove by induction on n that for all $n \geq 0$, $W_2(n) = W_1(n \texttt{ div } 2)$, where `div` is integer division, so that $3 \texttt{ div } 2 = 1$ and so on.

   (c) Deduce that $W_2(n)$ is $\mathtt{O}(3^{n \texttt{ div } 2})$.

8. Is $\mathtt{O}(3^{n \texttt{ div } 2}) \subseteq \mathtt{O}(3^n)$? Is $\mathtt{O}(3^n) \subseteq \mathtt{O}(3^{n \texttt{ div } 2})$? Say why.

9. Suppose we are given the following property of the function `fastexp`:

   For all values $n \geq 1$,

$$\texttt{fastexp}(n) \Longrightarrow^{(\mathtt{c}_0 log_2(n) + \mathtt{c}_1)} k,$$

   where k is the value of $2^{\mathtt{n}}$, $\mathtt{c}_0$ and $\mathtt{c}_1$ are unspecified constants, $\mathtt{c}_0 \neq 0$, and $log_2(n)$ is the (integer) logarithm base 2 of n.

   (a) How many steps does it take to evaluate the expression

$$\texttt{fastexp}(\texttt{fastexp}(n)),$$

   when $n \geq 1$? Give a big-O classification for the work of this expression.

(b) From the above property of `fastexp`, does it follow that $W_{\texttt{fastexp}}(n)$ is $\texttt{O}(log_2(n))$? How about $\texttt{O}(n)$?

10. Consider the following function definitions:

```
fun upto (i:int, j:int) : int list =
    if i>j then [ ] else i :: upto (i+1, j);
fun sum [ ] = 0
 | sum (x::L) = x + (sum L)
```

The work to evaluate `upto(1,n)` for an integer value `n` is `O(n)`. The work to evaluate `A@B` when `A` and `B` are list values is `O(length(A))`. For each of the following expressions, give a big-O estimate of the work to evaluate the expression. Be as accurate as you can.

(a) `sum(upto(1,n) @ upto(1,n))`

(b) `let val L = upto(1,n) in L @ L end`

(c) `sum(upto(1,n)) + sum(upto(1,n))`

11. Define an ML function `foo : int -> int` such that for all `n` ≥ 0, `foo(n)` returns the same integer result as does each of the expressions (a), (b), (c) above, but the work to evaluate `foo(n)` is $O(1)$.

12. Recall that the span of a tuple expression like (e_1, e_2) is the *maximum* of the span of e_1 and the span of e_2. Derive recurrences for the work and span to evaluate `f n`, when `f` is the following function:

```
fun f n =
   if n=0 then 1 else
       let val (x, y) = (f (n-1), f(n-1)) in x * y end
```

    Is the work the same as the span, asymptotically? Find a way to define a function `g` that is equivalent to `f`, but `g`'s span is the same as its work, asymptotically. Comment on the relative virtues of these two functions.

13. Consider the following ways to define a function `bar : int -> int` and check that the functions all have the same applicative behavior. What are the work and span for `bar n` in each case (for `n` ≥ 0)?

```
fun bar 0 = 1
|   bar n = (fn x:int => x*x) (bar (n-1))

fun bar 0 = 1
|   bar n = let val y = bar (n-1) in y*y end

fun bar 0 = 1
|   bar n = bar (n-1) * bar (n-1)

fun bar n = if n >= 0 then 1 else bar (n-1)
```