Exploring Languages with Interpreters and Functional Programming Chapter 29

H. Conrad Cunningham

04 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

29 Models of Reduction

29.1 Chapter Introduction

TODO:

- Complete introduction and other missing pieces.

- Redraw LaTeX figures so they appear in formats other than LaTeX/PDF.
- Remove or explain any unnecessary redundancies between this chapter and chapters 8, 9, etc.
- Consider whether to replace the use of Haskell as pseudo-math notation.
- Check section breaks and titles.

29.2 Big-O and Efficiency

We state efficiency (i.e., time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the list-reversing functions `rev` and `reverse` that we have looked at several times. We stated that the number of steps required to evaluate `rev xs` is, in the worst case, "on the order of" `n^2` where `n` denotes the length of list `xs`. We let the number of steps be our measure of time and write

> T(`rev xs`) = O(`n^2`)

to mean that the time to evaluate `rev xs` is bounded by some (mathematical) function that is proportional to the square of the length of list `xs`.

Similarly, we write

> T(`reverse xs`) = O(`n`)

to mean that the time (i.e., number of steps) to evaluate `reverse xs` is bounded by some function that is proportional to the length of `xs`.

Note: These expressions are not really equalities. We write the more precise expression

> T(`reverse xs`)

on the left-hand side and the less precise expression O(`n`) on the right-hand side.

For short lists, the performance of `rev` and `reverse` are similar. But as the lists get long, `rev` requires considerably more steps than `reverse`.

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time as the size of the input approaches infinity (i.e., gets large). We often do worst case analyses of time. Such analyses are usually easier to do than average-case analyses.

29.3 Reduction

29.3.1 Definition

The terms *reduction*, *simplification*, and *evaluation* all denote the same process: rewriting an expression in a "simpler" equivalent form. That is, they involve two kinds of replacements:

- the replacement of a subterm that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters. (This is sometimes called β-reduction.)
- the replacement of a primitive application (e.g., `+` or `*`) by its value. (This is sometimes called δ-reduction.)

29.3.2 Redexes

The term *redex* refers to a subterm of an expression that can be reduced.

An expression is said to be in *normal form* if it cannot be further reduced.

Some expressions cannot be reduced to a value. For example, `1/0` cannot be reduced; an error message is usually generated if there is an attempt to evaluate (i.e., reduce) such an expression.

For convenience, we sometimes assign the value $\perp$ (pronounced "bottom") to such error cases to denote that their values are undefined. Remember that this value cannot be manipulated within a computer.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

- **leftmost redex first**—where the leftmost reducible subterm in the expression text is reduced before any other subterms are reduced
- **rightmost redex first**—where the rightmost reducible subterm in the expression text is reduced before any other subterms are reduced

The redex can also be selected based on whether or not it is contained within another redex:

- **outermost redex first**—where a reducible subterm that is not contained within any other reducible subterm is reduced before one that is contained within another
- **innermost redex first**—where a reducible subterm that contains no other reducible subterm is reduced before one that contains others

29.3.3 AOR and NOR

The two most often used reduction orders are:

- **applicative order reduction (AOR)**—where the leftmost innermost redex is reduced first
- **normal order reduction (NOR)**—where the leftmost outermost redex is reduced first.

To see the difference between AOR and NOR consider the following functions:

```
fst :: (a,b) -> a
fst (x,y) = x

sqr :: Int -> Int
sqr x = x * x
```

Now consider the following reductions.

First, reduce the expression with **AOR**:

```
  fst (sqr 4, sqr 2)
⟹ { sqr }
  fst (4*4, sqr 2)
⟹ { * }
  fst (16, sqr 2)
⟹ { sqr }
  fst (16, 2*2)
⟹ { * }
  fst (16, 4)
⟹ { fst }
  16
```

Thus AOR requires 5 reductions.

Second, reduce the expression with **NOR**:

```
  fst (sqr 4, sqr 2)
⟹ { fst }
  sqr 4
⟹ { sqr }
  4*4
⟹ { * }
  16
```

Thus NOR requires 3 reductions.

In this example NOR requires fewer steps because it avoids reducing the unneeded second component of the tuple.

The number of reductions is different, but the result is the same for both reduction sequences.

In fact, this is always the case. If any reduction terminates (and not all do), then the resulting value will always be the same.

(Consequence of) Church-Rosser Theorem: If an expression can be reduced in two different ways to two normal forms, then these normal forms are the same (except that variables may need to be renamed).

The *diamond property* for the reduction relation $\rightarrow$ states that if an expression `E` can be reduced to two expressions `E1` and `E2`, then there is an expression `N` which can be reached (by repeatedly applying $\rightarrow$) from both `E1` and `E2`. We use the symbol $\overset{*}{\rightarrow}$ to represent the *reflexive transitive closure* of $\rightarrow$. (`E` $\overset{*}{\rightarrow}$ `E1` means that `E` can be reduced to `E1` by some finite, possibly zero, number of reductions.)

$$\begin{tikzcd}
 & \mathrm{E} \arrow[dl, "\overset{*}{\rightarrow}"'] \arrow[dr, "\overset{*}{\rightarrow}"] & \\
\mathrm{E1} \arrow[dr, "\overset{*}{\rightarrow}"'] & & \mathrm{E2} \arrow[dl, "\overset{*}{\rightarrow}"] \\
 & \mathrm{N} &
\end{tikzcd}$$

Some reduction orders may fail to terminate on some expressions. Consider the following functions:

```
answer :: Int -> Int
answer n = fst (n+n, loop n)

loop :: Int -> [a\]
loop n = loop (n+1)
```

First, reduce the expression with **AOR**:

```
answer 1
```

$\Longrightarrow$ { answer }

```
fst (1+1,loop 1)
```

$\Longrightarrow$ { + }

```
fst (2,loop 1)
```

$\Longrightarrow$ { `loop` }

```
fst (2,loop (1+1))
```

$\Longrightarrow$ { + }

```
fst (2,loop 2)
```

$\Longrightarrow$ { `loop` }

```
fst (2,loop (2+1))
```

$\Longrightarrow$ { + }

```
fst (2,loop 3)
```

$\Longrightarrow \cdots$ Does not terminate normally

Second, reduce the expression with **NOR**:

```
answer 1
```

$\Longrightarrow$ { `answer` }

```
fst (1+1,loop 1)
```

$\Longrightarrow$ { `fst` }

```
1+1
```

$\Longrightarrow$ { + }

```
2
```

Thus NOR requires 3 reductions.

If an expression `E` has a normal form, then a normal order reduction of `E` (i.e., leftmost outermost) is guaranteed to reach the normal form (except that variables may need to be renamed).

29.3.4 Concepts related to AOR and NOR

There are several concepts in functional programming languages related to AOR:

- ***Applicative order reduction (AOR)*** Reduce leftmost innermost redex first.
- ***Eager evaluation*** Evaluate any expression that can be evaluated regardless of whether the result is ever needed. (For example, arguments of a function are evaluated before the function is called.)
- ***Strict semantics*** A function is only defined if all of its arguments are defined. For example, multiplication is only defined if both of its operands are defined, `5 * \bot = \bot`.

- ***Call-by-value parameter passing*** Evaluate the argument expression and bind its value to the function's parameter.

Similarly, there are several concepts in functional programming languages related to NOR:

- ***Normal order reduction (NOR)*** Reduce leftmost outermost redex first.
- ***Lazy evaluation*** Do not evaluate an expression unless its result is needed.
- ***Nonstrict (lenient) semantics*** A function may have a value even if some of its arguments are undefined. (For example, tuple construction is not strict in either parameter. That is, `(⊥,x)` $\neq \bot$ and `(x,⊥)` $\neq \bot$.)
- ***Call-by-name parameter passing*** Pass the unevaluated argument expression to the function; evaluate it upon each reference.

Note that in the absence of side-effects (e.g., when we have referential transparency, call-by-name gives the same result as call-by-value.

In general, call-by-name parameter passing is inefficient. However, a referentially transparent language can replace call-by-name parameter passing with the equivalent, but more efficient, *call-by-need* method.

In the call-by-need method, the unevaluated argument expression is passed to the function as in call-by-name. The first reference to the corresponding parameter causes the expression to be evaluated; subsequent references just use the value computed by the first reference. Thus the expression is only evaluated when needed and then only once.

Consider the `sqr` program again.

```
sqr x = x \* x
```

First, reduce the expression with **AOR**:

```
 sqr (4+2)
```

$\Longrightarrow$ `{ + }`

```
 sqr 6
```

$\Longrightarrow$ `{ sqr }`

```
 6 * 6
```

$\Longrightarrow$ `{ * }`

```
 36
```

Thus AOR requires 3 reductions.

Second, reduce the expression with **NOR**:

```
     sqr (4+2)
⟹ { sqr }
     (4+2) * (4+2)
⟹ { + }
     6 * (4+2)
⟹ { + }
     6 * 6
⟹ { * }
36
```

Thus NOR requires 4 reductions.

Here NOR is less efficient than AOR. What is the problem?

The argument `(4+2)` is reduced twice because the parameter appeared twice on the right-hand side of the definition.

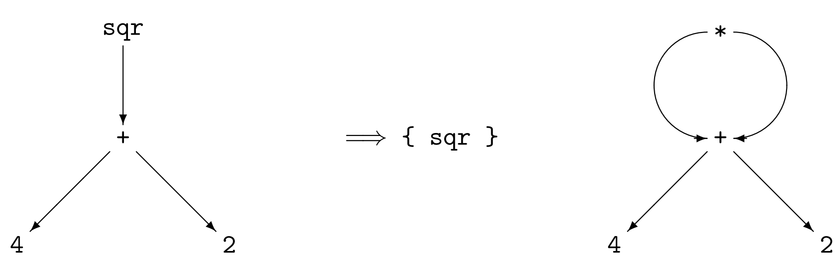
29.3.5 String and graph reduction

The rewriting strategy we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

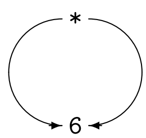
A more efficient alternative is *graph reduction*. In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subterms of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated, it need not be evaluated again. Thus leftmost outermost (i.e., normal order) graph reduction is a technique for implementing call-by-need parameter passing.

The Haskell interpreter uses a graph reduction technique.

Consider the leftmost outermost graph reduction of the expression `sqr (4+2)`.



$\Longrightarrow$ { + }



$\Longrightarrow$ { * }

36

Note: In a graph reduction model, normal order reduction never performs more reduction steps than applicative order reduction. It may perform fewer. And, like all outermost reduction techniques, it is guaranteed to terminate if any reduction sequence terminates.

As we see above, parameters that repeatedly occur on the right-hand side introduce shared components into the expression graph. A programmer can also introduce shared components into a function's expression graph by using **`where`** or **`let`** to define new symbols for subexpressions that occur multiple times in the defining expression. This potentially increases the efficiency of the program .

Consider a program to find the solutions of the following equation:

$$a * x^2 + b * x + c = 0$$

Using the quadratic formula the two solutions are:

$$\frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Expressing this formula as a Haskell program to return the two solutions as a pair, we get:

```
roots :: Float -> Float -> Float -> (Float,Float)
roots a b c = ( (-b-d)/e, (-b+d)/e )
    where d = sqrt (sqr b - 4 * a * c)
          e = 2 * a
```

Note the explicit definition of local symbols for the subexpressions that occur multiple times.

Function `sqr` is as defined previously and `sqrt` is a primitive function defined in the standard prelude.

In one step, the expression `roots 1 5 3` reduces to the expression graph shown on the following page. For clarity, we use the following in the graph:

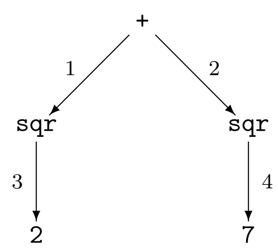
- `tuple-2` denotes the pair forming operator `( , )`.
- `div` denotes division (on `Float`).
- `sub` denotes subtraction.
- `neg` denotes unary negation.

The application `roots 1 5 3` reduces to the following expression graph:

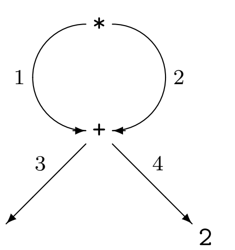
(Drawing Not Currently Available)

We use the total number of *arguments* as the measure of the *size* of a term or graph.

Example: `sqr 2 + sqr 7` has size 4.



Example: `x * x where x = 7 + 2` has size 4.



Note: This size measure is an indication of the size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different from the way we normally think of space complexity in an imperative algorithms class, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may first appear. Remember that data structures such as lists and tuples are themselves *expressions* built by applying constructors to simpler data.

29.4 Head Normal Form

Sometimes we need to reduce a term but not all the way to normal form.

Consider the expression `head (map sqr [1..7])` and a normal order reduction.

```
     head (map sqr [1..7])
⟹ { [1..7] }
     head (map sqr (1:[2..7]))
⟹ { map.2 }
     head (sqr 1 : map sqr [2..7])
⟹ { head }
     sqr 1
⟹ { sqr }
     1 * 1
⟹ { * }
     1
```

Note that the expression `map sqr [1..7]` was reduced but not all the way to normal form. However, any term that is reduced must be reduced to *head normal form*.

A term is in *head normal form* if:

- it is not a redex
- it cannot become a redex by reducing any of its subterms

If a term is in normal form, then it is in head normal form, but not vice versa.

Any term of form `(e1:e2)` is in head normal form, because regardless of how far `e1` and `e2` are reduced, no reduction rule applies to `(e1:e2)`. The cons operator is the primitive list constructor; it is not defined in terms of anything else.

However, a term of form `(e1:e2)` is only in normal form if both `e1` and `e2` are in their normal forms.

Similarly, any term of the form `(e1,e2)` is in head normal form. The tuple constructor is a primitive operation; it is not defined in terms of anything else.

However, a term of the form `(e1,e2)` is in normal form only if both `e1` and `e2` are.

Whether a term needs to be reduced further than head normal form depends upon the context.

Example: In the reduction of the expression `head (map sqr [1..7])`, the term `map sqr [1..7]` only needed to be reduced to head normal form, that is, to the expression `sqr 1 : map sqr [2..7]`.

However, `appendChan stdout (show (map sqr [1..7])) exit done` would cause reduction of `map sqr [1..7]` to normal form.

29.5 Pattern Matching

For reduction using equations that involve pattern matching, the leftmost outermost (i.e., normal order) reduction strategy is not, by itself, sufficient to guarantee that a terminating reduction sequence will be found if one exists.

Consider function `zip'`.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (a:as) (b:bs) = (a,b) : zip' as bs
zip'  _      _       = []
```

Now consider a leftmost outermost (i.e., normal order) reduction of the expression `zip' (map sqr []) (loop 0)`, where `sqr` and `loop` are as defined previously.

```
zip' (map sqr []) (loop 0)
```

$\Longrightarrow$ { `map.1`, to determine if first arg matches `(a:as)` }

```
zip' [] (loop 0)
```

$\Longrightarrow$ { `zip'.2` }

```
[]
```

Alternatively, consider a rightmost outermost reduction of the same expression.

```
zip' (map sqr []) (loop 0)
```

$\Longrightarrow$ { `loop`, to determine if second arg matches `(b:bs)` }

```
zip' (map sqr []) (loop (0+1))
```

$\Longrightarrow$ { + }

```
zip' (map sqr []) (loop 1)
```

$\Longrightarrow$ { `loop` }

```
zip' (map sqr []) (loop (1+1))
```

$\Longrightarrow$ { + }

```
zip’ (map sqr []) (loop 2)
```

$\Longrightarrow \cdots$ Does not terminate normally

Pattern matching should not cause an argument to be reduced unless absolutely necessary; otherwise nontermination could result.

Pattern-matching reduction rule: Match the patterns left to right. Reduce a subterm only if required by the pattern.

In `zip’ (map sqr []) (loop 0)` the first argument must be reduced to head normal form to determine whether it matches `(a:as)` for the first leg of the definition. It is not necessary to reduce the second argument unless the first argument match is successful.

Note that the second leg of the definition, which uses two anonymous variables for the patterns, does not require any further reduction to occur in order to match the patterns.

The expressions

```
zip' (map sqr [1,2,3]) (map sqr [1,2,3])
```

and

```
zip' (map sqr [1,2,\]) []
```

both require their second arguments to be reduced to head normal form in order to determine whether the arguments match `(b:bs)`.

Note that the first does match and, hence, enables the first leg of the definition to be used in the reduction. The second expression does not match and, hence, disables the first leg from being used. Since the second leg involves anonymous patterns, it can be used in this case.

- Normal order graph reduction `e_{0}` $\Longrightarrow$ `e_{1}` $\Longrightarrow$ `e_{2}` $\Longrightarrow \cdots \Longrightarrow$ `e_{n}`
- Time = number of reduction steps (n)
- Space = size of the largest expression graph `e_{i}`

Most lazy functional language implementations more-or-less correspond to graph reduction.

29.6 Reduction Order and Space

It is always the case that the number of steps in an *outermost graph reduction* $\leq$ the number of steps in an *innermost reduction* of the same expression.

However, sometimes a combination of innermost and outermost reductions can save on space and, hence, on implementation overhead.

Consider the following definition of the factorial function. (This was called `fact3` in Chapter 4.)

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Now consider a normal order reduction of the expression `fact 3`.

```
 fact 3
```

$\Longrightarrow$ { `fact.2` }

```
 3 * fact (3-1)
```

$\Longrightarrow$ { -, to determine pattern match }

```
 3 * fact 2
```

$\Longrightarrow$ { `fact.2` }

```
 3 * (2 * fact (2-1))
```

$\Longrightarrow$ { -, to determine pattern match }

```
 3 * (2 * fact 1)
```

$\Longrightarrow$ { `fact.2` }

```
 3 * (2 * (1 * fact (1-1))) MAX SPACE!
```

$\Longrightarrow$ { -, to determine pattern match }

```
 3 * (2 * (1 * fact 0))
```

$\Longrightarrow$ { `fact.1` }

```
 3 * (2 * (1 * 1))
```

$\Longrightarrow$ { * }

```
 3 * (2 * 1)
```

$\Longrightarrow$ { * }

```
 3 * 2
```

$\Longrightarrow$ { * }

```
 6
```

We define the following measures of the

- **Time:** Count reduction steps. 10 for this example.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression. 4 binary operations, 1 unary operation, hence size is 9 for this example.

  In general, 1 multiplication for each `n > 01` plus 1 subtraction and one application of `fact`. Thus `2n + 3` arguments. O(`n`).

Note that function `fact` is strict in its argument. That is, evaluation of `fact` *always* requires the evaluation of its argument.

Since the value of the argument expression `n-1` in the recursive call is eventually needed (by the pattern match), there is no reason to delay evaluation of the expression. That is, the expression could be evaluated eagerly instead of lazily. Thus any work to save this expression for future evaluation would be avoided.

Delaying the computation of an expression incurs overhead in the implementation. The delayed expression and its calling environment (i.e., the values of variables) must be packaged so that evaluation can resume correctly when needed. This packaging—called a *closure*, *suspension*, or *recipe*—requires both space and time to be set up.

Furthermore, delayed expressions can aggravate the problem of *space leaks*.

The implementation of a lazy functional programming language typically allocates space for data dynamically from a memory *heap*. When the heap is exhausted, the implementation searches through its structures to recover space that is no longer in use. This process is usually called *garbage collection*.

However, sometimes it is very difficult for a garbage collector to determine whether or not a particular data structure is still needed. The garbage collector thus retains some unneeded data. These are called space leaks.

Aside: Picture bits of memory oozing out of the program, lost to the program forever. Most of these bits collect in the bit bucket under the computer and are automatically recycled when the interpreter restarts. However, in the past a few of these bits leaked out into the air, gradually polluting the atmosphere of functional programming research centers. Although it has not be scientifically verified, anecdotal evidence suggests that the bits leaked from functional programs, when exposed to open minds, metamorphose into a powerful intellectual stimulant. Many imperative programmers have observed that programmers who spend a few weeks in the vicinity of functional programs seem to develop a permanent distaste for imperative programs and a strange enhancement of their mental capacities.

Aside continued: As environmental awareness has grown in the functional programming community, the implementors of functional languages have begun to develop new leak-avoiding designs for the language processors and garbage collectors. Now the amount of space leakage has been reduced considerably. Although it is still a problem. Of course, in the meantime a large community of programmers have become addicted to the intellectual stimulation of functional programming. The number of addicts in the USA is small, but growing. FP traffickers have found a number of ways to smuggle their illicit materials into the country. Some are brought in via the Internet from clandestine archives in Europe; a number of professors and students are believed to be cultivating a domestic supply. Some are smuggled from Europe inside strange red-and-white covered books (but that source is somewhat lacking in the continuity of supply). Some are believed hidden in Haskell holes; others in a young nerd named Haskell's pocket protector. (Haskell is Miranda's younger brother; she was the first one who had any comprehension about FP.)

Aside ends: Mercifully.

Now let's look at a tail recursive definition of factorial.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = fact' (f*n) (n-1)
```

Because of the Tail Recursion Theorem, we know that `fact' 1 n = fact n` for any natural `n`.

Now consider a normal order reduction of the expression `fact' 1 3`.

`fact' 1 3`

$\Longrightarrow$ { `fact'.2` }

`fact' (1 * 3) (3 - 1)`

$\Longrightarrow$ { -, to determine pattern match }

`fact' (1 * 3) 2`

$\Longrightarrow$ { `fact'.2` }

`fact' ((1 * 3) * 2) (2 - 1)`

$\Longrightarrow$ { -, to determine pattern match }

`fact' ((1 * 3) * 2) 1`

$\Longrightarrow$ { `fact'.2` }

`fact' (((1 * 3) * 2) * 1) (1 - 1)` MAX SPACE!

$\Longrightarrow$ { -, to determine pattern match }

`fact' (((1 * 3) * 2) * 1) 0`

$\Longrightarrow$ { `fact'.1` }

`((1 * 3) * 2) * 1`

$\Longrightarrow$ { * }

`(3 * 2) * 1`

$\Longrightarrow$ { * }

```
     6 * 1
⟹ { 6 }
     6
```

- **Time:** Count reduction steps. 10 for this example, same as for `fact`.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3*n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression. 4 binary operations, 1 two-argument function, hence size is 10 for this example.

  In general, 1 multiplication for each `n > 0` plus 1 subtraction and one application of `fact'`. Thus `2*n+4` arguments. O(`n`).

Note that function `fact'` is strict in both arguments. The second argument of `fact'` is evaluated immediately because of the pattern matching. The first argument's value is eventually needed, but its evaluation is deferred until after the `fact'` recursion has reached its base case.

Perhaps we can improve the space efficiency by forcing the evaluation of the first argument immediately as well. In particular, we try a combination of outermost and innermost reduction.

```
     fact' 1 3
⟹ { fact'.2 }
     fact' (1 * 3) (3 - 1)
⟹ { *, innermost }
     fact' 3 (3 - 1)
⟹ { -, to determine pattern match }
     fact' 3 2
⟹ { fact'.2 }
     fact' (3 * 2) (2 - 1)
⟹ { *, innermost }
     fact' 6 (2 - 1)
⟹ { -, to determine pattern match }
     fact' 6 1
⟹ { fact'.2 }
     fact' (6 * 1) (1 - 1)
⟹ { *, innermost }
     fact' 6 (1 - 1)
```

$\Longrightarrow$ { -, to determine pattern match }

```
fact’ 6 0
```

$\Longrightarrow$ { `fact’.1` }

```
6
```

- **Time:** Count reduction steps. 10 for this example. Same as for previous two reduction sequences.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3*n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression.

  For any `n > 0`, the longest expression consists of one multiplication, one subtraction, and one call of `fact'`. Thus the size is constantly 6. O(1).

How to decrease space usage and implementation overhead.

1. The compiler could do *strictness analysis* and automatically force eager evaluation of arguments that are always required.

   This is done by many compilers. It is sometimes a complicated procedure.

2. The language could be extended with a feature that allows the programmer to express strictness explicitly.

In Haskell, reduction order can be controlled by use of the special function `strict`.

A term of the form `strict f e` is reduced by first reducing expression `e` to head normal form, and then applying function `f` to the result. The term `e` can be reduced by normal order reduction, unless, of course, it contains another call of `strict`.

The following definition of `fact’` gives the mixed reduction order given in the previous example. That is, it evaluates the first argument eagerly to save space.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = (strict fact' (f*n)) (n-1)
```

29.7 Choosing a Fold

Remember that earlier we defined two folding operations. Function `foldr` is a backward linear recursive function that folds an operation through a list from the tail (i.e., right) toward the head. Function `foldl` is a tail recursive function that folds an operation through a list from the head (i.e., left) toward the tail.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

The first duality theorem (as given in the Bird and Wadler textbook [1]) states the circumstances in which one can replace `foldr` by `foldl` and vice versa.

If $\oplus$ is a associative binary operation of type `t -> t` with identity element `z`, then:

First duality theorem: If $\oplus$ is a associative binary operation of type `t -> t` with identity element `z`, then:

```
foldr (⊕) z xs = foldl (⊕) z xs
```

Thus, often we can use either `foldr` or `foldl` to solve a problem. Which is better?

We discussed this problem before, but now we have the background to understand it a bit better.

Clearly, eager evaluation of the second argument of `foldl`, which is used as an accumulating parameter, can increase the space efficiency of the folding operation. This optimized operation is called `foldl’` in the standard prelude.

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z []     = z
foldl' f z (x:xs) = strict (foldl' f) (f z x) xs
```

Suppose that `op` is *strict in both arguments* and can be computed in O(1) time and O(1) space. (For example, + and $*$ have these characteristics.) If `n = length xs`, then both `foldr op i xs` and `foldl op i xs` can be computed in O(`n`) time and O(`n`) space.

However, `foldl’ op i xs`) requires O(`n`) time and O(1) space. The reasoning for this is similar to that given for `fact’`.

Thus, in general, `foldl’` is the better choice for this case.

Alternatively, suppose that `op` is *nonstrict in either argument*. Then `foldr` is usually more efficient than `foldl`.

As an example, consider operation `||` (i.e., logical-or). The `||` operator is strict in the first argument, but not in the second. That is, `True || x = True` without having to evaluate `x`.

Let `xs = [x_1, x_2, x_3, ... x_n]` such that $(\exists i : 1 \leq i \leq n ::$ `x_i == True` $) \wedge (\forall j : 1 \leq < i ::$ `x_i == False` $))$

Suppose `x_i` is the minimum i satisfying the above existential.

```
foldr (||) False xs
```

$\Longrightarrow$ { many steps }

```
x_1 || (x_2 || ( ... || (x_i || ( ... || (x_n || False) ... )
```

Because of the nonstrict definition of `||`, the above can stop after the `x_i` term is processed. None of the list to the right of `x_i` needs to be evaluated.

However, a version which uses `foldl` must process the entire list.

```
foldl (||) False xs
```

$\Longrightarrow$ { many steps }

```
( ... ( False || x_i) || x_2) || ... ) || x_i) || ... ) || x_n
```

In this example, `foldr` is clearly more efficient than `foldl`.

29.8 What Next?

TODO

29.9 Exercises

TODO

29.10 Acknowledgements

TODO History of chapter in FP class.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I adapted and revised this chapter from Chapter 13 of my *Notes on Functional Programming with Haskell* [2]. I had included some some of this discussion in Chapter 8 in 2016 and later.

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [1:6.1–6.3], [3:6], and other functional programming sources.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

29.11 Terms and Concepts

TODO

29.12 References

[1] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/notes/haskell_notes.pdf

[3] Anthony J. Field and Peter G. Harrison. 1988. *Functional programming.* Addison-Wesley, Boston, Massachusetts, USA.