• Last lecture I introduced functional programming using the language HOPE.
  • This allows us to write functions like:
    \begin{verbatim}
    dec map : (alpha -> beta) # list(alpha) -> list(beta);
    --- map(f, nil) <= nil;
    --- map(f, x :: l) <= f(x) :: map(f, l);
    \end{verbatim}
  • This lecture will look at the mathematical underpinning of functional languages.
  • But first we will look at some more typical things one can do with them.

• One important property of functional languages is the way they handle parameters.
  • We are used to passing parameters using \textit{call-by-value}.
  • You may even have come across \textit{call-by-reference}.
  • Call by value is good for efficiency.
  • It may, however, result in redundant computation.
  • So we also consider \textit{call-by-need} where arguments are passed unevaluated and evaluated when required.

• The advantage of call-by-need is that we waste no computation.
  • The disadvantage is the expense of implementation.
  • In general with functional languages we distinguish between \textit{eager} and \textit{lazy} evaluation.
  • Eager evaluation does everything as soon as possible without worrying whether it is useful.
  • Lazy evaluation only does things when absolutely necessary.
  • They roughly correspond to call-by-value and call-by-need.

• However, there is more to eagerness and laziness than just these efficiency issues.
  • Their effect permeates the whole of functional programming.
  • In particular they determine what it is possible to do at the extreme limits of the language.
  • (For example, handling infinite data structures.)
  • Let’s start by considering the limitations that \textit{strictness} imposes on a language.

• One such is:
  \begin{verbatim}
  --- f(x, y) <= if x < 10 then x else y
  \end{verbatim}
  • This does not always need the value of \(y\), but does need the value of \(x\).
  • It is thus strict in \(x\).
  • And non-strict in \(y\).
  • Thus when \(x < 10\), we waste computation evaluating \(y\) since there is no need to know its value.

• As an extreme, consider:
  \begin{verbatim}
  f(4, <non-terminating expression>)
  \end{verbatim}
  • Here, an eager implementation would cause the program to fail, whereas a lazy one would give us 4.
  • Of course, this does not help us when we have:
    \begin{verbatim}
    f(<non-terminating expression>, 4)
    \end{verbatim}

• To see how laziness can eliminate redundant computation, consider the function:
  \begin{verbatim}
  dec reduce : (alpha # beta # list(alpha) # beta)
  --- reduce(f, b, nil) <= b;
  --- reduce(f, b, x::l) <= f(x, reduce(f, b, l));
  \end{verbatim}
  • This function takes a function as an argument and applies it to reduce a list to a single element.
  • \(b\) is what you use as an argument when you get to the end of the list.
  • Thus \texttt{reduce(+, 0, L)} sums the elements of \(L\) and \texttt{reduce(*, 1, L)} computes their product.
Writing \( b \) for:
\[
\text{lambda}(el, isthere) => \text{if} \ isthere \ \text{then} \ \text{true} \\
\text{else} \ (1 = el)
\]
we can use \( \text{reduce}(b, \text{false}, \text{List}) \) to test if 1 is in \( \text{List} \).
Consider doing this for the list \( [1, 3, 5, 7] \).
For eager evaluation we would get:
\[
\text{reduce}(b, \text{false}, [1, 3, 5, 7])
\]
\[
\text{b}(1, \text{b}(3, \text{b}(5, \text{b}(7, \text{false})))))
\]
\[
\text{b}(1, \text{b}(3, \text{false}))
\]
\[
\text{b}(1, \text{false})
\]
\[
\text{true}
\]
If the implementation were lazy, we would get:
\[
\text{reduce}(b, \text{false}, [1, 3, 5, 7])
\]
\[
\text{b}(1, \text{false})
\]
since in this case the second argument never has to be evaluated.

\[\text{Lambda calculus}\]

The lambda calculus is the calculus of anonymous functions.
It provides a means of representing functions and a means of transforming them.
Let’s consider a very simple function:
\[
\text{--- double}(x) <= 2 * x
\]
We write this in lambda notation as:
\[
\lambda x. 2 \times x
\]
Dropping the name anonymises the function.
Note that we use the prefix form of the \( \times \) function.

We read this lambda expression as follows.
The \( \lambda \) we read as “The function of”.
The dot we read as “which returns”.
So, the whole thing is:
The function of \( x \) which returns \( x \) times 2.
Of course it is very similar to:
\[
\text{lambda} \ x \ => \ 2 \times x
\]

The \( x \) in the lambda abstraction is called the bound variable.
The \( \lambda \) corresponds to the idea of a formal parameter.
The bit of the lambda abstraction to the right of the dot is the body.
The body can be any valid lambda expression, so it can be another lambda abstraction.
\[
\lambda x. \lambda y. (\times y)/2
\]
“The function of \( x \) which returns the function of \( y \) which returns the sum of \( x \) and \( y \) multiplied by 2.”

This is just the lambda calculus version of:
\[
\text{lambda} \ x. \ => \ \text{lambda} \ y. \ => \ (x + y) \times 2
\]
All lambda calculus functions have just a single argument.
So multi-argument functions become multiple applications of single-argument functions.
This is known as “currying”.
Although we should write brackets between the different functions:
\[
(\lambda x. (\lambda y. (\times y)/2))
\]
by convention we don’t.

When we call a lambda function we place it in brackets before its argument.
Thus calling:
\[
\lambda x. 2 \times x
\]
on the value 4 is done by writing:
\[
(\lambda x. 2 \times x)4
\]
and we call:
\[
(\lambda x. (\lambda y. (\times y)/2))
\]
with \( y \) as 2 and \( x \) as 3 by:
\[
(\lambda x. (\lambda y. (\times y)/2) \ 2) \ 3
\]

This is all there is to the syntax of the lambda calculus.
The BNF is:
\[
<\text{exp}> ::= \lambda <\text{id}> . <\text{exp}> | <\text{id}> \\
| <\text{exp}> <\text{exp}> | <\text{exp}> <\text{id}> | \text{con}
\]
\[
<\text{id}> ::= \text{any} \ \text{identifier}
\]
\[
<\text{con}> ::= \text{constant}
\]
There is a surprising amount that you can put together with such a simple syntax.

The syntax shows us how to build valid lambda expressions.
But how do we evaluate them?
We have conversion rules which do this.
The first rule is the simplest.
Constants evaluate to constants.
Other functions are reduced using the \( \delta \)-rules.
These allow us to replace function applications with their values.
For example:

\[ +1 3 \rightarrow_\eta 4 \]

we read this as “+ 1 3 reduces to 4”.

To do this we have to have the arguments of the function be themselves already reduced.

So we cannot directly reduce:

\[ s(1 2)(-4 1) \]

Instead we have to reduce each argument of the outer * first.

So we have:

\[ s(1 2)(-4 1) \]

\[ \rightarrow_\eta (+1 2)/4 \]

\[ \rightarrow_\eta +3 3 \]

\[ \rightarrow_\eta 9 \]

This reduction then evaluates simple functions.

To evaluate lambda abstractions we need a \( \beta \)-reduction

A \( \beta \)-reduction replaces the value of the bound variables with the values they are called with.

Thus evaluating:

\[ (\lambda x \cdot x x)2 \]

we have:

\[ (\lambda x \cdot x x)2 \rightarrow_\eta \star 2 2 \]

\[ \rightarrow_\eta 4 \]

This kind of reduction might end up having to be repeated.

For example:

\[ (((\lambda x. \lambda y. x + y)7)8) \]

will reduce as:

\[ (((\lambda x. \lambda y. x + y)7)8) \]

\[ \rightarrow_\eta (\lambda y. +7 y)8 \]

\[ \rightarrow_\eta +7 8 \]

\[ \rightarrow_\eta 15 \]

However, we have to be careful in doing this.

Consider:

\[ \lambda x. (\lambda x.x)(+1 x) \]

Here we have two distinct \( x \)s.

There is the inner one, in the \((+1 x)\), and the one referred to in the outer \( \lambda x \).

Thus it would be wrong to do a \( \beta \)-reduction for the argument \( 1 \) as:

\[ (\lambda x. (\lambda x.x)(+1 x))1 \]

\[ \rightarrow (\lambda x.1)(+1 1) \]

\[ \rightarrow 1 \]

We have to be careful not to substitute inside an abstraction if the bound variable has the same name as the variable being substituted.

The easiest way to do avoid this is to do \( \sigma \)-conversion.

This is to rename the bound variables to make them have different names.

The same thing as standardising variables in logic.

Applying this to our previous example, gives:

\[ \lambda x. (\lambda y.y)(+1 x) \]

There is one other form of reduction

\( \eta \)-reduction allows the reduction:

\[ \lambda x. E x \rightarrow_\eta E \]

if \( x \) does not occur free in \( E \) since:

\[ (\lambda x. E x)A \]

is just

\[ E A \]

This is not widely used.

(It is a compile-time feature rather than a run-time one.)

Sometimes we have a choice in the order we apply reductions.

This can have a big impact on the result.

Consider:

\[ (\lambda x. \lambda y y)((\lambda z. z)(\lambda z. z)) \]

Here we end up either trying to reduce:

\[ (\lambda z. z)(\lambda z. z) \]

or

\[ (\lambda x. \lambda y y)((\lambda z. z)(\lambda z. z)) \]

which does not terminate.

If we reduce the other we get:

\[ (\lambda x. \lambda y y)((\lambda z. z)(\lambda z. z)) \]

\[ \rightarrow (\lambda z. z)(\lambda z. z) \]

\[ \rightarrow (\lambda z. z)(\lambda z. z) \]

1

which does not terminate.

If we try to reduce the first one we get:

\[ (\lambda z. z)(\lambda z. z) \]

\[ \rightarrow (\lambda z. z)(\lambda z. z) \]

\[ \rightarrow (\lambda z. z)(\lambda z. z) \]

\[ \rightarrow \lambda y \lambda y \]

which does terminate.

So, order matters.
We call the expression being reduced the redex. Then we have:

- The leftmost redex is the one whose \( \lambda \) is to the left of all other redexes in the expression.
- The rightmost is similarly defined.
- The outermost redex is a redex not contained in any other redex.
- The innermost redex is one which contains no other.

We can then define two ways of reducing expressions.

- Applicative order reduction (AOR) says you should always reduce the leftmost innermost redex first.
- Normative order reduction (NOR) says you should always reduce the leftmost outermost redex first.

So in our example,

\[
(\lambda z. z) (\lambda z. z)
\]

is the leftmost innermost, and

\[
(\lambda x. \lambda y. y)((\lambda z. z) (\lambda z. z))
\]

is the leftmost outermost.

A lambda expression is said to be in normal form when it can be reduced no more.

Thus:

\[
\lambda y. y
\]

is the normal form of:

\[
(\lambda x. \lambda y. y)((\lambda z. z) (\lambda z. z))
\]

It turns out that there is something special about the normal form, and normal order reduction.

- Basically we can use NOR to get the same normal form expression whatever way we do the reduction.

The Church-Rosser theorem has as a consequence:

- If an expression \( E \) can be reduced in two different ways to two normal forms, then these normal forms are the same up to alphabetical equivalence.
- The last bit means you can change variable names to make them identical.
- The standardization theorem says:
  - If an expression \( E \) has a normal form then reducing the leftmost outermost redex at each stage in the reduction of \( E \) guarantees to reach that normal form (up to alphabetical equivalence).

Thus the normal form is unique.

- (Which is exactly what we want— it would be a shame if different implementations of the same function could give us different results.)
- We also have the diamond property for reductions.
  - If you can reduce \( E \) to \( E_1 \) and \( E_2 \) by applying any reduction operation several times, then by applying the same operation some more, you can reduce both \( E_1 \) and \( E_2 \) down to some expression \( N \).