Tuples

Scala has special syntax for pairs and larger tuples.

A pair of two values x and y is written (x, y).

If x is of type T and y is of type U, then (x, y) is of the tuple-type (T, U).

More generally, if $x_i : T_i$, then the tuple $(x_1, ..., x_n)$ is of type $(T_1, ..., T_n)$.

Tuples and lists both aggregate data, but there are two important differences.

- The elements of a tuple can be of different types, whereas the elements of a list are all of the same type.
- The number of elements of a tuple is fixed by its type, whereas the length of a list is not.

The elements of an n-tuple are accessed by selectors *_1*, ..., *_n*.

Example:

```
> val xy = (2, 3)
val xy : scala.Tuple2 = (2,3)
> xy._1 + xy._2
5
```

The Tuple Class

Like all other types, tuples are represented as classes in Scala.

For instance the *2*-tuple type (T_1, T_2) is represented as the class type *scala.Tuple2*$[T_1, T_2]$.

The *Tuple2* class is defined as follows.

```
package scala with {
    class Tuple2[a, b](x: a, y: b) with {
        def _1: a = x
        def _2: b = y
        override def toString = "(" + _1 + "," + _2 + ")"
    }
}
```

Explanations:

- The ***package*** clause plays a role similar to Java: It places *Tuple2* in the package *scala*.
- *Tuple2* is parameterized with type parameters $[a, b]$ and value parameters (x, y).
- It defines *access functions* *_1*, *_2*.
- It also *overrides* the *toString* function of *Object*.

Larger tuples can be defined analogously (but are currently missing from the predefined classes).

Function *zip*

The *zip* method in *List* combines two lists into a list of pairs.

```
abstract class List[a] with {
  ...
  def zip[b](that: List[b]): List[(a, b)] =
    if (this.isEmpty || that.isEmpty) []
    else (this.head, that.head) :: (this.tail zip that.tail)
```

Example: Using *zip* and *fold*, we can define the scalar product of two lists as follows.

```
def scalarProduct(xs: List[Double], ys: List[Double]): Double =
  (xs zip ys)
  .map(xy => xy._1 * xy._2)
  .fold(x: Double, y: Double => x + y)(0)
```

More on Fold and Reduce

Exercise: Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as fold operations.

```
def mapFun[a, b](xs: List[a], f: (a)b): List[b] =
    xs.foldRight(y: a, ys: List[b] ⇒  ??  )([])

def concatFun[a](xs: List[a], ys: List[a]): List[a] =
    ??  .foldRight(cons[a])(  ??  )

def lengthFun[a](xs: List[a]): Int =
    xs.foldRight(  ??  )(0)
```

Here, *cons* is predefined in the *List* file as

```
def cons[a](x: a, xs: List[a]): List[a] = x :: xs
```

Nested Mappings

We can extend the higher-order list functions to include many computations that are normally expressed as nested loops.

Example: Given a positive integer n, find all pairs of positive integers i, j, where $1 \leq j < i \leq n$ such that $i + j$ is prime.

For example, if $n = 6$, the pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i + j$	3	5	5	7	7	7	11

A natural way to do this is:

- Generate the sequence of all pairs (i, j) of integers such that $1 \leq j < i \leq n$.

- Filter the pairs such that $i + j$ is prime.

A natural way to generate the sequence of pairs is:

- Generate all integers between 1 and n for i. This can be packaged using the function

```
def range(lo: Int, hi: Int): List[Int] =
    if (lo > hi) [] else lo :: range(lo + 1, hi)
```

- For each integer i, generate the list of pairs $(i, 1), \ldots, (i, i-1)$. This can be achieved by a combination of *range* and *map*:

```
range(1, i-1) map (x => (i, x))
```

- Finally, combine all sublists using *foldRight* with :::.

Putting everything together gives the following expression:

```
range(1, n)
    .map(i => range(1, i-1).map(x => (i, x)))
    .foldRight(xs: List[(Int, Int)], ys: List[(Int, Int)] => xs ::: ys)([])
    .filter(pair => isPrime(pair._1 + pair._2))
```

Function *flatMap*

The combination of mapping and then concatenating sublists resulting from the map is so common that we introduce a special method for it in *List.scala*:

```
abstract class List[a] with {
    ...
    def flatMap[b](f: (a)List[b]): List[b] = {
        map(f).foldRight(xs: List[b], ys: List[b] ⇒ xs ::: ys)([])
    }
}
```

With *flatMap*, our expression could have been written more concisely as follows.

```
range(1, n)
    .flatMap(i ⇒ range(1, i−1).map(x ⇒ (i, x)))
    .filter(pair ⇒ isPrime(pair._1 + pair._2))
```

Q: What is a concise way to define *isPrime*? (Hint: use *forall* in *List*).

Reasoning About Lists

Recall the concatenation operation for lists:

```
class List[a] with {
    ...
    def ::: (that: List[a]) =
        if (isEmpty) that
        else head :: (tail ::: that)
}
```

We would like to verify that concatenation is associative, with the empty list $[\,]$ as left and right identity:

$$
\begin{aligned}
(xs \mathbin{:::} ys) \mathbin{:::} zs &= xs \mathbin{:::} (ys \mathbin{:::} zs) \\
xs \mathbin{:::} [] &= xs \quad = [] \mathbin{:::} xs
\end{aligned}
$$

Q: How can we prove statements like the one above?

A: By structural induction over lists.

Reminder: Natural Induction

Recall the proof principle of natural induction:

To show a property $P(n)$ for all numbers $n \geq b$:

1. Show that $P(b)$ holds (base case).
2. For arbitrary $n \geq b$ show:

   if $P(n)$ holds, then $P(n+1)$ holds as well

   (induction step).

Example: Given

```
def factorial(n: Int): Int =
    if (n == 0) 1
    else n * factorial(n-1)
```

show that, for all $n \geq 4$,

$$factorial(n) \geq 2^n$$

$\boxed{\textbf{Case } 4}$ is established by simple calculation of $factorial(4) = 24$ and $2^4 = 16$.

$\boxed{\textbf{Case } n{+}1}$ We have for $n \geq 4$:

$$
\begin{array}{ll}
 & factorial(n+1) \\
= & (\textit{by the second clause of factorial(*)}) \\
 & (n+1) * factorial(n) \\
\geq & (\textit{by calculation}) \\
 & 2 * factorial(n) \\
\geq & (\textit{by the induction hypothesis}) \\
 & 2 * 2^n .
\end{array}
$$

Note that in our proof we can freely apply reduction steps such as in (*) anywhere in a term.

This works because purely functional programs do not have side effects; so a term is equivalent to the term it reduces to.

The principle is called *referential transparency*.

Structural Induction

The principle of structural induction is analogous to natural induction:

In the case of lists, it is as follows:

To prove a property $P(xs)$ for all lists xs,

1. Show that $P(\,[\,]\,)$ holds (base case).
2. For arbitrary lists xs and elements x show:

   if $P(xs)$ holds, then $P(x :: xs)$ holds as well

   (induction step).

Example

We show $(xs \mathbin{:::} ys) \mathbin{:::} zs \;=\; xs \mathbin{:::} (ys \mathbin{:::} zs)$ by structural induction on xs.

$\boxed{\textbf{Case } [\,]}$ For the left-hand side, we have:

$$
\begin{array}{ll}
 & ([\,] \mathbin{:::} ys) \mathbin{:::} zs \\
= & (\textit{by first clause of } :::) \\
 & ys \mathbin{:::} zs
\end{array}
$$

For the right-hand side, we have:

$$
\begin{array}{ll}
 & [\,] \mathbin{:::} (ys \mathbin{:::} zs) \\
= & (\textit{by first clause of } :::) \\
 & ys \mathbin{:::} zs
\end{array}
$$

So the case is established.

Case $x :: xs$

For the left-hand side, we have:

$$
\begin{array}{ll}
 & ((x :: xs) ::: ys) ::: zs \\
= & (\textit{by second clause of } :::) \\
 & (x :: (xs ::: ys)) ::: zs \\
= & (\textit{by second clause of } :::) \\
 & x :: ((xs ::: ys) ::: zs) \\
= & (\textit{by the induction hypothesis}) \\
 & x :: (xs ::: (ys ::: zs))
\end{array}
$$

For the right-hand side, we have:

$$
\begin{array}{ll}
 & (x :: xs) ::: (ys ::: zs) \\
= & (\textit{by second clause of } :::) \\
 & x :: (xs ::: (ys ::: zs))
\end{array}
$$

So the case (and with it the property) is established.

Exercise: Show by induction on xs that $xs ::: [\,] \;=\; xs$.

Example (2)

As a more difficult example, consider function

```
abstract class List[a] with {
  ...
  def reverse: List[a] =
    if (isEmpty) []
    else tail.reverse ::: [head]
}
```

We would like to prove the proposition that

$$xs.reverse.reverse \;=\; xs \quad .$$

We proceed by induction over xs. The base case is easy to establish:

$$\begin{array}{ll} & [\,].reverse.reverse \\ = & (\textit{by first clause of } reverse) \\ & [\,].reverse \\ = & (\textit{by first clause of } reverse) \\ & [\,] \end{array}$$

For the induction step, we try:

$$
\begin{array}{ll}
 & (x :: xs).reverse.reverse \\
= & (\textit{by second clause of } reverse) \\
 & (xs.reverse ::: [x]).reverse
\end{array}
$$

There's nothing more we can do to this expression, so we turn to the right side:

$$
\begin{array}{ll}
 & x :: xs \\
= & (\textit{by induction hypothesis}) \\
 & x :: xs.reverse.reverse
\end{array}
$$

The two sides have simplified to different expressions.

So we still have to show that

$$(xs.reverse ::: [x]).reverse \;=\; x :: xs.reverse.reverse$$

Trying to prove this directly by induction does not work.

Instead we have to *generalize* the equation to:

$$(ys ::: [x]).reverse \;=\; x :: ys.reverse$$

This equation can be proved by a second induction argument over ys. (See blackboard).

Exercise: Is it the case that $(xs\ drop\ m)\ at\ n\ =\ xs\ at\ (m + n)$ for all natural numbers m, n and all lists xs?

Structural Induction on Trees

Structural induction is not restricted to lists; it works for arbitrary trees.

The general induction principle is as follows.

To show that property $P(t)$ holds for all trees of a certain type,

- Show $P(l)$ for all leaf trees.
- For every interior node t with subtrees s_1, ..., s_n, show that $P(s_1) \wedge \ldots \wedge P(s_n) \Rightarrow P(t)$.

Example: Recall our definition of *IntSet* with operations *contains* and *incl*:

```
abstract class IntSet with {
    abstract def incl(x: Int): IntSet
    abstract def contains(x: Int): Boolean
}
```

```
class Empty extends IntSet with {
    def contains(x: Int): Boolean = False
    def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
class NonEmpty(elem: Int, left: Set, right: Set) extends IntSet with {
    def contains(x: Int): Boolean =
        if (x < elem) left contains x
        else if (x > elem) right contains x
        else True
    def incl(x: Int): IntSet =
        if (x < elem) NonEmpty(elem, left incl x, right)
        else if (x > elem) NonEmpty(elem, left, right incl x)
        else this
}
```

What does it mean to prove the correctness of this implementation?

Laws of IntSet

One way to state and prove the correctness of an implementation is to prove laws that hold for it.

In the case of *IntSet*, three such laws would be:

For all sets s, elements x, y:

$$\begin{array}{lcll} \mathit{Empty}\ \mathit{contains}\ x & = & \mathit{False} & \\ (s\ \mathit{incl}\ x)\ \mathit{contains}\ x & = & \mathit{True} & \\ (s\ \mathit{incl}\ x)\ \mathit{contains}\ y & = & s\ \mathit{contains}\ y & \textbf{if}\ x \neq y \end{array}$$

(In fact, one can show that these laws characterize the desired data type completely).

How can we establish that these laws hold?

Proposition 1: $\mathit{Empty}\ \mathit{contains}\ x = \mathit{False}$.

Proof: By the definition of *contains* in *Empty*.

Proposition 2: $(xs\ incl\ x)\ contains\ x = True$

Proof:

Case $Empty$

$$
\begin{array}{ll}
 & (Empty\ incl\ x)\ contains\ x \\
= & (\textit{by definition of}\ incl\ \textit{in}\ Empty) \\
 & NonEmpty\,(x,\ Empty,\ Empty)\ contains\ x \\
= & (\textit{by definition of}\ contains\ \textit{in}\ NonEmpty) \\
 & True
\end{array}
$$

Case $NonEmpty\,(x,\ l,\ r)$

$$
\begin{array}{ll}
 & (NonEmpty\,(x,\ l,\ r)\ incl\ x)\ contains\ x \\
= & (\textit{by definition of}\ incl\ \textit{in}\ NonEmpty) \\
 & NonEmpty\,(x,\ l,\ r)\ contains\ x \\
= & (\textit{by definition of}\ contains\ \textit{in}\ Empty) \\
 & True
\end{array}
$$

$\boxed{\textbf{Case}\ \mathit{NonEmpty}\,(y,\ l,\ r\,)\ \text{where}\ y < x}$

$$
\begin{array}{ll}
 & (\mathit{NonEmpty}\,(y,\ l,\ r\,)\ \mathit{incl}\ x\,)\ \mathit{contains}\ x \\
= & (\textit{by definition of}\ \mathit{incl}\ \textit{in}\ \mathit{NonEmpty}) \\
 & \mathit{NonEmpty}\,(y,\ l,\ r\ \mathit{incl}\ x\,)\ \mathit{contains}\ x \\
= & (\textit{by definition of}\ \mathit{contains}\ \textit{in}\ \mathit{NonEmpty}) \\
 & (r\ \mathit{incl}\ x\,)\ \mathit{contains}\ x \\
= & (\textit{by the induction hypothesis}) \\
 & \mathit{True}
\end{array}
$$

$\boxed{\textbf{Case}\ \mathit{NonEmpty}\,(y,\ l,\ r\,)\ \text{where}\ y > x}$ is analogous.

Proposition 3: If $x \neq y$ then $xs\ \mathit{incl}\ y\ \mathit{contains}\ x\ =\ xs\ \mathit{contains}\ x$.

Proof: See blackboard.

Exercise

Say we add a *union* function to *IntSet*:

```
class IntSet with {
    ...
    def union(other: IntSet): IntSet
}
class Expty extends IntSet with {
    ...
    def union(other: IntSet) = other
}
class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet with {
    ...
    def union(other: IntSet): IntSet = l union r union other incl x
}
```

The correctness of *union* can be subsumed with the following law:

Proposition 4: $(xs\ union\ ys)\ contains\ x\ =\ xs\ contains\ x\ ||\ ys\ contains\ x$.

Show Proposition 4 using structural induction on *xs*.