MAU22C00 Lecture 16

John Stalker

Trinity College Dublin

Lists

We saw one way to define ordered pairs last time, as Kuratowski pairs $\{\{x\}, \{x, y\}\}$.

Unfortunately the obvious generalisation doesn't give ordered triples.

We'd like not just ordered pairs and triples but n-tuples and lists.

How do you implement lists in a low level programming language, e.g. C?

There are multiple options, of which the simplest is singly linked lists.

A list element is a structure with two fields. The first holds the data, an item in the list, and the second is a list, the remainder of the original list after removing that item.

A list is a pointer to a list element.

These definitions are mutually recursive but good programming languages allow that.

You can't get much lower level than set theory! It's possible to implement singly linked lists in set theory.

Defining lists within set theory

You can find the details in the notes. You're not responsible for them but if you want to read them then these comments may be helpful.

The role of the list elements is played by Kuratowski pairs, with the data as the left (x) element of the pair and the remainder as the right (y) element of the pair.

The role of lists themselves is played by sets of pairs, more or less.

You need to order the sets of pairs, so you can identify the head of the list.

Isn't ordering the members of a set the problem we were trying to solve? Isn't this circular?

Note quite. The generalised Kuratowski construction ran into problems for lists of length greater than two due to repeated elements. When we're ordering the members of a set that's not an issue.

So the real definition of a list is as a Kuratowski chain (generalisation of pairs) of Kuratowski pairs (which can be considered as non-empty Kuratowski chains with at most two elements).

Do you care?

How do you deal with lists in a programming language which doesn't provide them?

- Write an implementation of lists, providing at least the following operations:
  - Check whether its argument is a list.
  - Check whether it's empty.
  - Append an element to a list.
  - Extract the first element of the list and the remainder.
- From now on you can behave as if the language provided them natively.

This approach works in mathematics as well. In fact it works *better* in mathematics than in programming.

In programming you might find that singly linked lists are inefficient in some use cases and might need to switch to a different implementation.

If so you'd better make sure you used only the public interface described above, not the internal details of the implementation.

In mathematics the only measure of efficiency that matters is the work required to prove that your implementation provides the required interface. After you've done that efficiency is never an issue again.

You don't care

Once you have an implementation of lists which is proved to work you can forget the internal implementation details.

In fact you *should* forget the internal implementation details and use only the public interface.

More complicated operations, e.g. reversing a list or concatenating lists, should be defined in terms of that interface, just as in programming.

This is a general organising principle, not limited to lists. We now have two ways to implement ordered pairs, as Kuratowski pairs or as lists with two items. We need to pick one–I chose lists of length two–and then never worry about it again.

We'll meet the same issue with the natural numbers. They aren't a primitive element of our set theory. If we want them we need to implement them. The implementation needs to provide a particular interface, given by the Peano axioms. Once we've done that we should never refer to the implementation details again.

Stacks

Remember the interface I described above?

- Check whether its argument is a list.
- Check whether it's empty.
- Append an element to a list.
- Extract the first element of the list and the remainder.

Does this remind you of anything?

This is essentially the interface for a stack, with some of the words changed, e.g. "list" for "stack", "append" for "push", and "extract" for "pop".

This is one reason why I wanted to make sure we had a working implementation of lists.

The other was formal languages, which are defined as sets of lists of tokens.

Cartesian products, binary relations

Once we have ordered pairs, however we've decided to implement them, we can define Cartesian products as sets of ordered pairs.

$A \times B$ is the set of ordered pairs (x, y) with $x \in A$ and $y \in B$.

That's not actually a proper definition. We need a set which is large enough that it at least contains all of those pairs and then we need to apply selection to it. Obtaining such a set is rather tedious and I won't do it.

We use the shorthand notation A^2 for $A \times A$.

I'll also use A^3 for lists of length three all of whose items are in A.

Subsets of $A \times B$ are called binary relations from A to B. Subsets of A^2 are called binary relations on A.

Subsets of A^3 are called ternary relations on A. They do come up, e.g. "... is the sum/product of ... and ...", but not nearly as often as binary relations.

From now on, relation, unless stated otherwise, means binary relation.

Properties of relations

Some properties a (binary) relation R from A to B can have:

- left total: for every $x \in A$ there is a $y \in B$ such that $(x, y) \in R$
- right total: for every $y \in B$ there is an $x \in A$ such that $(x, y) \in R$
- left unique: for every $y \in B$ there is at most one $x \in A$ such that $(x, y) \in R$
- right unique: for every $x \in A$ there is at most one $y \in B$ such that $(x, y) \in R$

Properties which only make sense when $A = B$:

- reflexive: for every $x \in A$ we have $(x, x) \in R$
- transitive: if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$
- symmetric: if $(x, y) \in R$ then $(y, x) \in R$
- antisymmetric: if $(x, y) \in R$ and $(y, x) \in R$ then $x = y$

Combinations of properties

A relation which left total and right unique is called a function.

A function is called injective if it is also left unique.

A function is called surjective if it is also right total.

This is different from the usual functional notation. We write $(x, y) \in F$ instead of $y = f(x)$. Essentially we're identifying functions with their graphs.

This is an extensional definition of functions, not an intensional one. In other words, functions are completely characterised by their values at each value of their arguments, independently of how the value is calculated, or even whether it can be.

A relation which is reflexive, transitive and symmetric is called an equivalence relation.

A relation which is reflexive, transitive and antisymmetric is called a partial order.

If for all $x \in A$ and $y \in A$ we have $(x, y) \in R$ or $(y, x) \in R$ then it's called a total order.

Examples

Functions from N, the set of natural numbers[1], to itself:

- $f(x) = x + 1$ or, in our notation, the set of $(x, y) \in N^2$ such that $y = x + 1$. This function is injective but not surjective.
- $f(x) = x/2$, the way most computer languages handle integer division, i.e. throwing away the remainder. As a subset of N^2 this is the pairs $(2 \cdot k, k)$ and $(2 \cdot k + 1, k)$ for all $k \in N$. This function is surjective but not injective.

Relations on N:

- (x, y) such that $x = y$. This is an equivalence relation. Perhaps surprisingly it is also a partial order. It is not a total order.
- (x, y) such that $x \leq y$. This is a partial order, and indeed a total order, but not an equivalence relation.
- (x, y) such that $x < y$. This is a partial order, but not a total order or an equivalence relation.

[1] assuming, for the moment, there is such a set

Operations on relations

$R \circ S$ is the set of (x, z) such that there is a y such that $(x, y) \in S$ and $(y, z) \in R$.

The order is chosen to agree with the traditional (bad) order in which we write composition of functions.

R^{-1} is the set of (x, y) such that $(y, x) \in R$.

R^{-1} of a function is generally not a function. It is if and only if R is both injective and surjective. In this case it is what's normally called the inverse function.

We have the following useful properties:

$$(R \circ S) \circ T = R \circ (S \circ T) \qquad {R^{-1}}^{-1} = R \qquad (R \circ S)^{-1} = (S^{-1}) \circ (R^{-1})$$

Relation to finiteness

If A is a finite set and F is an injective function on A then F is a surjective function.

This is proved by set induction.

If A is a finite set and R is a partial order on A then A has a minimal member and a maximal member.

This is proved by set induction.

Both of these properties can fail for infinite sets. In fact they often provide a quick way of showing that a set is infinite.

The first property is sometimes taken as the definition of finite sets. This is called the Dedekind definition. It's a bit more intuitive than the Tarski definition, but it's quite hard to prove some properties of finite sets from this definition, like the fact that subsets of finite sets are finite.