Sequences

Klaus Draeger

Lecture 7: Sequences

One property of sets is that all the following are equal:

$$\begin{aligned} & \{\, 0,\ 1,\ 2,\ 5,\ 23,\ 99 \,\} \\ = & \{\, 1,\ 99,\ 5,\ 2,\ 23,\ 0 \,\} \\ = & \{\, 2,\ 1,\ 99,\ 5,\ 2,\ 23,\ 1,\ 0,\ 1 \,\} \end{aligned}$$

Sometimes it is necessary to:

- distinguish values of a set by **position**,
- or to **permit duplicate** values,
- or to impose some **ordering** on the values.

None of these can be achieved using a flat **set.**
In such cases a **sequence** is the correct structure to use.

Defining Sequences

- For any set **S** there is a set **seq(S)** of S-valued sequences.
- A sequence is defined by enclosing its values in **square brackets []**
- Example: for **Colours = {red, yellow, green}** we have [red,green,red,red] : seq(Colours)
- Note that a sequence can be empty: []
- Like for functions, seq(S) has **specialised subsets**:
  - seq1(S) – the set of **non-empty** sequences
  - iseq(S) – the set of **injective** (non-repeating) sequences

Defining Sequences

- The defining property of sequences is that they are:
  - **Indexed** by numbers 1,. . . ,n (for some n)
  - That is, they are **functions f : NAT1+->S** with the additional constraint that **dom(f)** is an **initial segment** 1..n of NAT1
- So we have the equality **seq(S) = ???**

Defining Sequences

- The defining property of sequences is that they are:
  - **Indexed** by numbers 1,. . . ,n (for some n)
  - That is, they are **functions f : NAT1+->S**
    with the additional constraint that **dom(f)** is an **initial segment** 1..n of NAT1
- So we have the equality
  seq(S) = {f | f:NAT1+->S & #(n).(n:NAT & dom(f)=1..n)}
  which we can simplify using **n = ???**

Defining Sequences

- The defining property of sequences is that they are:
  - **Indexed** by numbers 1,. . . ,n (for some n)
  - That is, they are **functions f : NAT1+->S** with the additional constraint that **dom(f)** is an **initial segment** 1..n of NAT1
- So we have the equality **seq(S) = {f | f:NAT1+->S & #(n).(n:NAT & dom(f)=1..n)}** which we can simplify using **n = card(f)**: **seq(S) = {f | f:NAT1+->S & dom(f)=1..card(f)}**

Working with Sequences: Access

- Since sequences are functions,
  we can get the **n**th element by **evaluating** at **n**
- If colSequence = [red,green,green,blue,red]
  then colSequence(1) = red and colSequence(4) = blue
- The **first** and **last** operators return the first/last element:
  first(colSequence) = red, last(colSequence) = red
- How do these differ from **min** and **max**?

Working with Sequences: Access

- Since sequences are functions,
  we can get the **n**th element by **evaluating** at **n**
- If colSequence = [red,green,green,blue,red]
  then colSequence(1) = red and colSequence(4) = blue
- The **first** and **last** operators return the first/last element:
  first(colSequence) = red, last(colSequence) = red
- In general, for any sequence **seq**,
  first(seq) = seq(1) and last(seq) = seq(card(seq))
- None of these are defined for the empty sequence [].

Working with Sequences: Truncation

- **front** and **tail** complement the first and last operators:
  - **front(seq)** is **seq** with its last element removed
  - **tail(seq)** is **seq** with its first element removed
  - Examples:
    - **front([2,3,5,7,11]) = [2,3,5,7]**
    - **tail([2,3,5,7,11]) = [3,5,7,11]**
  - Like first and last, they are undefined for the empty sequence.
- How could we re-create these if they did not exist?
  - To define front(seq):

Working with Sequences: Truncation

- **front** and **tail** complement the first and last operators:
  - **front(seq)** is **seq** with its last element removed
  - **tail(seq)** is **seq** with its first element removed
  - Examples:
    - **front([2,3,5,7,11]) = [2,3,5,7]**
    - **tail([2,3,5,7,11]) = [3,5,7,11]**
  - Like first and last, they are undefined for the empty sequence.
- How could we re-create these if they did not exist?
  - To define front(seq): **restrict** seq removing the last index
    front(seq) = (dom(seq) - card(seq)) <| seq
    e.g. front([2,3,5,7,11]) = (1..4) <| [2,3,5,7,11]
  - To define tail(seq):

Working with Sequences: Truncation

- **front** and **tail** complement the first and last operators:
  - **front(seq)** is **seq** with its last element removed
  - **tail(seq)** is **seq** with its first element removed
  - Examples:
    - **front([2,3,5,7,11]) = [2,3,5,7]**
    - **tail([2,3,5,7,11]) = [3,5,7,11]**
  - Like first and last, they are undefined for the empty sequence.
- How could we re-create these if they did not exist?
  - To define front(seq): **restrict** seq removing the last index
    front(seq) = (dom(seq) - card(seq)) <| seq
    e.g. front([2,3,5,7,11]) = (1..4) <| [2,3,5,7,11]
  - To define tail(seq): apply a **shift** (a restriction of **succ**)
    tail(seq) = (dom(seq) <| succ) ; seq
    e.g. tail([2,3,5,7,11]) = [2,3,4,5] ; [2,3,5,7,11]

Working with Sequences: Insertion

- The opposite of the truncations are the front and end **insertions.**
- **x -> seq** is the sequence **seq** with **x** inserted at the **front** so **->** is a function in **(X * seq(X)) –> seq(X)** with **x -> [**$z_1, \ldots, z_n$**] = [x**, $z_1, \ldots, z_n$**]**
- **seq <- y** is the sequence **seq** with **y** inserted at the **back** so **<-** is a function in **(seq(X) * X) –> seq(X)** with **[**$z_1, \ldots, z_n$**] <- y = [**$z_1, \ldots, z_n$,**y]**
- This lets us express the relationship between first/last/front/tail as
  - seq = front(seq) <- last(seq)
  - seq = first(seq) -> tail(seq)

Sequence Example: Stack

- Our goal is to model a **stack** of natural numbers.
- It should have the following operations:
  - **Push** – pushes a number onto the stack
  - **Pop** – pops (i.e. removes) and returns the number at the top of the stack
    – this requires the stack to be **non-empty**
  - **IsEmpty** – returns **Yes** if the stack is empty; otherwise returns **No**
- To do this, we need to
  - represent the stack itself
    – a sequence works for this
  - Figure out how to perform the operations

Sequence Example: Stack

```
MACHINE Stack

  SETS
    ANSWER = { Yes, No }

  VARIABLES
    stack

  INVARIANT
    stack : seq( NAT )

  INITIALISATION
    stack := []

  OPERATIONS
    res <-- isEmpty = ?
```

Sequence Example: Stack

```
//...

    res <-- isEmpty =
      IF stack = []
      THEN
        res := Yes
      ELSE
        res := No
      END
    END;

    push(num) = ?

    pop = ?
```

Sequence Example: Stack

```
// using the back of the sequence as the top

  push(num) =
    PRE
      num : NAT
    THEN
      stack := stack <- num
    END;

  res <-- pop =
    PRE
      stack /= []
    THEN
      stack := front(stack) ||
      res := last(stack)
    END
  END
```

Doing More with Sequences

- Beyond the operators discussed so far, we may want to:
  - **Combine** two sequences
    For example, combine [1,2,3] and [3,2,1] into [1,2,3,3,2,1]
    Similar to **concatenating** two strings in Java
  - **Split** a sequence at some position
    For example, split [1,2,3,4,5] into [1,2,3] and [4,5]
  - **Reverse** a sequence

  These are built-in, and we will see how to define others:
  - **Insert** or **delete** values in anywhere
  - **Filter** a sequence

Concatenation

- The concatenation operator ^ combines two sequences: [1,3,5] ^ [2,4,5] = [1,3,5,2,4,5]
- Note how the insertions are special cases:
  - x -> seq = [x] ^ seq
  - seq <- z = seq ^ [z]
- Why not just use the existing union (\/) operator?

Concatenation

- The concatenation operator ^ combines two sequences:
  [1,3,5] ^ [2,4,5] = [1,3,5,2,4,5]
- Note how the insertions are special cases:
  - x -> seq = [x] ^ seq
  - seq <- z = seq ^ [z]
- Why not just use the existing union (\/) operator?
  - \/ works at the set level
  - At this level [1,3,5] and [2,4,5] are simply
    {1|->1,2|->3,3|->5} and {1|->2,2|->4,3|->5}
  - Their union is
    {1|->1,2|->3,3|->5,1|->2,2|->4,3|->5}
    This is not a function (e.g. 1|->1 **and** 1|->2)
    and therefore also not a sequence

We **can** re-define ^ though, see next tutorial!

Splitting sequences

- The operators /|\ (up-arrow) and \|/ (down-arrow) split a sequence at a given position:
  For any **n** between 0 and card(seq),
  - **seq** /|\ **n** returns the first **n** elements of **seq**
  - **seq** \|/ **n** returns the rest of **seq**
    i.e. everything except the first **n** elements
- If c = [red, orange, yellow, green, blue, purple]:
  - c /|\ 2 = [red, orange]
  - c \|/ 2 = [yellow, green, blue, purple]
- So for all **n**, (seq/|\n) ^ (seq\|/n) = seq
- Again, let's see how we could re-define these ourselves:

Splitting sequences

- The operators /|\ (up-arrow) and \|/ (down-arrow) split a sequence at a given position:
  For any **n** between 0 and card(seq),
  - **seq** /|\ **n** returns the first **n** elements of **seq**
  - **seq** \|/ **n** returns the rest of **seq**
    i.e. everything except the first **n** elements
- If c = [red, orange, yellow, green, blue, purple]:
  - c /|\ 2 = [red, orange]
  - c \|/ 2 = [yellow, green, blue, purple]
- So for all **n**, (seq/|\n) ^ (seq\|/n) = seq
- Again, let's see how we could re-define these ourselves:
  - seq/|\n = (1..n) <| seq
  - seq\|/n : see tutorial!

Adding and Removing

- With these operators in place we can insert values wherever we want:
- To **insert** a value at position **n** in **seq**,
  - Take the first **n-1** elements of **seq**,
  - add the new value,
  - then append the remaining values of **seq**.

  e.g. we can insert **3** in the middle of **seq = [1,2,4,5]** using
  ((seq/|\2)<-3) ^ (seq\|/2) = ([1,2]<-3) ^ [4,5] = [1,2,3,4,5]
- To **remove** the element at position **n**,
  - Take the first **n-1** elements of **seq**,
  - and append the values of **seq** past the **n**th:

  To remove the 4th entry in seq = [1,2,3,4,5],
  use (seq/|\3) ^ (seq\|/4) = [1,2,3] ^ [5] = [1,2,3,5]

Filtering

- Often we want to remove elements based on their **value**, not **position**
- For example, for a list of numbers **seq = [1,2,3,4,3,2,1,4,2,1,3]**
  filter out the even numbers,
  leaving just the odd ones: **[1,3,3,1,1,3]**
- So suppose we have a sequence **seq** and subset **filter**, how to define a restriction of **seq** to values from **filter**?

Filtering

- Often we want to remove elements based on their **value**, not **position**
- For example, for a list of numbers **seq = [1,2,3,4,3,2,1,4,2,1,3]**
  filter out the even numbers,
  leaving just the odd ones: **[1,3,3,1,1,3]**
- So suppose we have a sequence **seq** and subset **filter**, how to define a restriction of **seq** to values from **filter**?
  - Get the set of indices for elements in the filter
    e.g. the indices of odd numbers are {1,3,5,7,10,11}
  - Define a sequence **select** containing them in order
    e.g. [1,3,5,7,10,11]
  - Compose them (select ; seq)

  We will do this in ProB.

Filtering