VDM*: Mathematics for Computer Science

Andrew Butterfield

May 10, 2001

Contents

1 Class 1 ................................................. 4
   1.1 A Bank Account Model ......................... 4

2 Class 2 ................................................. 7
   2.1 Sets ........................................... 7
   2.2 Sets Values and Types ....................... 7
   2.3 Spelling Checker Dictionary ............... 8
   2.4 Exercises 1 ................................ 11

3 Class 3 ................................................. 12
   3.1 Set Cardinality ................................ 12
   3.2 Set Mapping .................................. 13
   3.3 Set Reduction ............................... 13
   3.4 Function Composition ...................... 14
   3.5 Exercises 2 ................................ 14

4 Class 4 ................................................. 15
   4.1 Sequences ................................... 15
   4.2 Dictionary Sequence Model .................. 16
   4.3 Sequence Reduction ......................... 16
   4.4 Sequence Mapping ............................ 16
   4.5 Initial Dictionary ........................... 17
   4.6 Relating Abstract and Concrete .......... 17
   4.7 Operations on Sequence Dictionaries ..... 18

5 Class 5 ................................................. 19
   5.1 Proof Principles .............................. 19
   5.2 Proof by Induction ........................... 20
   5.3 Exercises 3 .................................. 22
6 Class 6

7 Class 7
   7.1 Mathematical Structures ........................................ 23
   7.2 Semigroups .......................................................... 23
   7.3 Monoids .............................................................. 24
   7.4 Groups ................................................................. 25
   7.5 Morphisms ............................................................ 25
      7.5.1 Semigroup Homomorphisms ................................... 25
      7.5.2 Monoid Homomorphisms ...................................... 26
   7.6 Are Homomorphisms Unique? ....................................... 27
   7.7 Exercises 4 ........................................................... 27

8 Class 8
   8.1 Fully Defining Homomorphisms ................................... 28
   8.2 Composing Homomorphisms ........................................ 28
   8.3 More Homomorphisms ............................................... 28
   8.4 Structure/Morphism Summary ...................................... 31
   8.5 Extending Dictionary Model ...................................... 31
   8.6 Maps ................................................................. 31

9 Class 9
   9.1 Maps and their Morphisms ........................................ 32
      9.1.1 Domain ......................................................... 33
      9.1.2 Map Removal .................................................. 33
      9.1.3 Map Restriction ............................................... 33
      9.1.4 Map Mapping .................................................. 33
   9.2 Map Model of Dictionary ........................................... 34
      9.2.1 Dictionary State .............................................. 34
      9.2.2 Lookup Operation ............................................ 34
      9.2.3 Insert Operation ............................................... 35
      9.2.4 Remove Operation ............................................. 36
      9.2.5 Joining Maps .................................................. 36
   9.3 Exercises 5 ........................................................ 36

10 Class 10
    10.1 Map Range ....................................................... 37
    10.2 Update Operation ................................................ 38
    10.3 Extend Operation ................................................ 39

11 Class 11 ............................................................... 40
11.1 Health Informatics ........................................ 40
11.2 Patients and Doctors ...................................... 40
11.3 Patient-GP Actions ......................................... 41
11.3.1 Registering with a GP .................................. 42
11.3.2 Adding a New GP ........................................ 42
11.3.3 Patient Count ........................................... 43
11.3.4 Patients per GP .......................................... 43

12 Class 12 .................................................. 44
12.1 Registry Merge ............................................ 44
12.2 Registry Add ................................................ 45
12.3 A Common Theme: Resolution ............................ 46
12.4 Indexing a Binary Operator ............................... 46
12.5 Properties of Indexing .................................... 46

13 Class 13 .................................................. 47
13.1 Healthcare Elaboration: Consultants ...................... 48
13.2 Initial State of System .................................... 48
13.3 Incorporating Registry Operations ........................ 48
13.4 Consultant Referrals ...................................... 49
13.5 Proof Principle: Case Analysis ........................... 49
13.6 Proof Attempt ............................................. 50
13.6.1 Case 1 — c not in κ .................................. 50
13.6.2 Case 2 — c in κ ........................................ 50
13.6.3 A Conceptual Note ...................................... 51
13.7 Exercises 6 ............................................... 51

A Solutions to Exercises ..................................... 51
A.1 Exercises 1 ............................................... 51
A.2 Exercises 2 ............................................... 52
A.3 Exercises 3 ............................................... 53
A.4 Exercises 4 ............................................... 55
A.5 Exercises 5 ............................................... 56
A.6 Exercises 6 ............................................... 56

B Definitions (Recursive) ................................... 58
B.1 Set Definitions ............................................. 58
B.2 Sequence Definitions ..................................... 59
B.3 Set & Sequence Properties ............................... 61
B.4 Set & Sequence Examples ................................ 62
C Structures and their Morphisms

C.1 Definition of Structures .................................. 62
  C.1.1 Semigroup ........................................... 62
  C.1.2 Monoid ............................................. 63
  C.1.3 Group .............................................. 64
  C.1.4 Morphism Composition ................................ 64
  C.1.5 Morphism Definition ................................ 65
C.2 Booleans .................................................. 65
  C.2.1 Boolean Structures .................................. 65
  C.2.2 Boolean Morphisms .................................. 65
C.3 Numbers ................................................... 65
  C.3.1 Number Structures .................................. 65
  C.3.2 Number Morphisms .................................. 65
C.4 Sets ........................................................ 66
  C.4.1 Set Structures ...................................... 66
  C.4.2 Set Morphisms ...................................... 66
C.5 Sequences .................................................. 66
  C.5.1 Sequence Structures .................................. 66
  C.5.2 Sequence Morphisms .................................. 66
C.6 Maps ........................................................ 67
  C.6.1 Map Structures ...................................... 67
  C.6.2 Map Morphisms ...................................... 67
C.7 Mixed Morphisms ........................................... 67

1 Class 1

The emphasis in this course is on the three “Ms”:

- Mathematics
- Method
- Modelling

Mathematics and Method are tools to achieve the goal, which is the Modelling of systems — the task that engineers do as part of the design process.

An overview of the syllabus would be:

- Mathematics — Sets, Sequences, Maps, Laws, Proof, Mathematical Structure, Indexing, Graphs
- Method — State, Operations, Key Properties (Invariants, Preconditions, Postconditions), Proof Obligations, Refinement, Retrieval Relations.

Initially we shall use some simple models to introduce the notation associated with modelling, and to revise basic notions such as set and function notation.

1.1 A Bank Account Model

We start by presenting a simple model, that of a single bank account, and use this to illustrate key ideas. We shall model the State of the system by the amount of the balance in the account. We shall impose a State Invariant that requires the balance to be non-negative.

Let $a$ denote a bank account (i.e. its balance). We shall view the balance as a real number, and we shall view the system as having type “$\text{Acct}$”. The key concept being introduced here is of mathematical values (such as $a$ above) having well-defined types (such as $\text{Acct}$):

- **Value**: $a$ — current value of account balance
- **Type**: $\text{Acct}$ — set of all possible accounts

We supply all this information as the following shorthand:

$$a \in \text{Acct} = \mathbb{R}$$

which reads as “$a$ is an $\text{Acct}$, which is modelled by a real number”.

We want the balance to be non-negative ($a \geq 0.0$). This is a key invariant (unchanging) property of the system. We shall describe an invariant by a function that maps a system into a boolean value (i.e. either $\text{True}$ or $\text{False}$), according as to whether or not the property in question actually holds.

Such functions are called Predicates, i.e. functions from some type $X$ to the set of boolean values ($\mathbb{B} = \{\text{FALSE, TRUE}\}$). We denote such a (predicate) function by

$$X \to \mathbb{B}$$

We introduce a system invariant by defining a function whose name is that of the system type, prefixed by “$\text{inv}$-”. For our bank account we write:

$$\text{inv-Acct} : \text{Acct} \to \mathbb{B}$$

$$\text{inv-Acct}(a) \equiv a \geq 0.0$$

The symbol $\equiv$ is read as “is defined equal to”.

Next, we introduce the notion of Operations, which capture ways in which the system can change and evolve. We shall consider two operation at present:

- **Deposit** — places money into the account
- **Withdraw** — remove money from the account

5
We introduce the notion of money as being a real number:

\[ m \in \text{Money} = \mathbb{R} \]

Let us consider the Deposit operation. This takes a money amount \( m \) and an account \( a \) and returns a new balance, i.e. a modified account.

\[
\text{Deposit} : (\text{Money} \times \text{Acct}) \to \text{Acct}
\]

The notation \((\text{Money} \times \text{Acct})\) denotes the set of all pairs whose first element is in \(\text{Money}\) and whose second element is in \(\text{Acct}\). We define the deposit operation as

\[
\text{Deposit}(m, a) \overset{\Delta}{=} a + m
\]

(Note, we have implicitly assumed that \( m \geq 0.0 \) — we should introduce an invariant for \(\text{Money inv-Money}(m) \overset{\Delta}{=} m \geq 0.0\).

We can define the withdrawal operation in a similar manner:

\[
\text{Withdraw} \quad : \quad (\text{Money} \times \text{Acct}) \to \text{Acct}
\]

\[
\text{Withdraw}(m, a) \overset{\Delta}{=} a - m
\]

What happens if \( m > a \)?

We end up with an account with a negative balance, which we have rules out with our invariant. We need to prevent the withdrawal from being used to take out too much money. We need a Precondition for Withdrawal.

We designate an operation precondition by prefixing pre- to the operation name. It is a predicate defined on the operation inputs:

\[
\text{pre-Withdraw} \quad : \quad (\text{Money} \times \text{Acct}) \to \mathbb{B}
\]

\[
\text{pre-Withdraw}(m, a) \overset{\Delta}{=} m \leq a
\]

We only perform an operation if its precondition is true.

to recap:

<table>
<thead>
<tr>
<th>Declare function</th>
<th>( f : \text{Inputs} \to \text{Outputs} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define its value</td>
<td>( f(i) \overset{\Delta}{=} \text{expression for output} )</td>
</tr>
<tr>
<td>value ( \in \text{type} )</td>
<td>( \text{value is arbitrary member of type} )</td>
</tr>
<tr>
<td>value : ( \text{type} )</td>
<td>( \text{value is specific member of type} )</td>
</tr>
<tr>
<td>value ( \overset{\Delta}{=} \ldots )</td>
<td>( \text{definition of specific value} )</td>
</tr>
</tbody>
</table>

In general we need a richer expression language than just numbers and arithmetic operations. For example, we need conditional expressions, whose value depends on the truth of some condition or value of some controlling expression. The simplest form of conditional expression is the binary “\( \text{if} \ldots \text{then} \ldots \text{else} \)” form

\[
\text{if } c \text{ then } e_1 \text{ else } e_2
\]

If condition \( c \) is true, then the whole expression has the same value as \( e_1 \), otherwise it has the same value as \( e_2 \). We can express this as the following laws:

\[
\begin{align*}
\text{if} \ \text{false} \ \text{then} \ e_1 \ \text{else} \ e_2 &= e_2 \\
\text{if} \ \text{true} \ \text{then} \ e_1 \ \text{else} \ e_2 &= e_1
\end{align*}
\]
An alternative notation for the above is the *McCarthy Conditional Form*:

\[ c \rightarrow e_1, e_2 \]

More elaborate conditionals, with more cases, are captured by “case” or “switch” expressions:

\[
\begin{align*}
    c_1 & \rightarrow e_1 \\
    c_2 & \rightarrow e_2 \\
    \vdots & \rightarrow \vdots \\
    c_n & \rightarrow e_n \\
    \rightarrow & \quad e_{n+1}
\end{align*}
\]

The conditions \( c_i \) are checked in order \((1, 2, \ldots, n)\). The value of the case-expression as a whole is the value of the right-hand side expression \( e_i \) that corresponds to the first condition to be true. If none of the conditions is true, then the whole expression has the same value as \( e_{n+1} \) (the “default” or “otherwise” case).

\section{2 Class 2}

\subsection{Sets}

Sets are unordered collections of values, were any given value occurs at most once (no duplicates).

\[ \{1, 2, 3\} = \{3, 2, 1\} = \{1, 3, 2, 3, 2, 1, 2\} \]

All of the above represent the same set, that containing 1, 2 and 3.

\[ \{1, 2, 3\} \neq \{2, 3\} \neq \{4, 5\} \]

Sets are equal only if they have exactly the same contents.

We can ask if a given value is in a given set, often using the terminology “is a member of”, or “is an element of”, the given set. If \( S \) denotes the given set, and \( x \) a possible element, we write \( x \in S \) to express the statement “\( x \) is an element of \( S \)”, or the question “is \( x \) an element of \( S \)?”.

A key relationship between sets is that of *subset* - we say that a set \( S \) “is a subset” of set \( T \) \((S \subseteq T)\) if every element of \( S \) is also an element of \( T \).

We can combine sets in a variety of ways. *Set Union* \((\cup)\) joins two sets to give all the elements in either:

\[ x \in S \cup T \text{ iff } x \in S \text{ or } x \in T \]

*Set Intersection* \((\cap)\) joins two sets to give all the elements in both:

\[ x \in S \cap T \text{ iff } x \in S \text{ and } x \in T \]

*Set Difference* \((\setminus)\) joins two sets to give all the elements in the first that are not in the second:

\[ x \in S \setminus T \text{ iff } x \in S \text{ and } x \notin T \]
2.2 Set Values and Types

Sets are strongly typed in our approach, which means we distinguish between sets of integers, sets of characters, sets of reals, sets of sets of integers, and so on.

We denote the set of all integer values by $\mathbb{Z}$. We can view $\mathbb{Z}$ as the “type” of integers, so, if $n$ is an integer we can say

\[ n \in \mathbb{Z} \]

What about sets as values - what type does $\{-3, 0, 2\}$ have? We use the notation $\mathcal{P}\mathbb{Z}$ to stand for “set of integers” or “integer-set”. The symbol $\mathcal{P}$ denotes the “powerset”, or set of all subsets of a given set. A set of integers is a subset of the set of all integers

\[ \{-3, 0, 2\} \subseteq \mathbb{Z} \]

This is the same as saying that such a set is an element of the set of all subsets of the set of all integers. So we can now say:

\[ \{-3, 0, 2\} : \mathcal{P}\mathbb{Z} \]

The key concept here is that we view types as the set of permissible values that may be denoted by a variable or expression of that type.

Given the above, we can now give the type of the set union operator as:

\[ \sqcup : \mathcal{P}A \times \mathcal{P}A \to \mathcal{P}A \]

Here $A$ denotes an arbitrary type. The statement says that union is a binary infix operator taking two sets of an arbitrary given type to a result set of the same type. The placeholders $(\_\_\_\_)$ on either side of the union symbol indicate that the normal use of the symbol is as an infix operator, i.e. written in between its two arguments.

We can give types to the other set operations in a similar style:

\[ \sqcap : \mathcal{P}A \times \mathcal{P}A \to \mathcal{P}A \]
\[ \setminus : \mathcal{P}A \times \mathcal{P}A \to \mathcal{P}A \]
\[ \in : A \times \mathcal{P}A \to B \]

2.3 Spelling Checker Dictionary

We shall now use sets to build a model of a more complex system, in this case a *Spelling Checker Dictionary*. Such a dictionary checks for correct spelling only, and as a consequence contains correctly spelled words, but no definitions — such a dictionary is used for word-processor “spell-checkers” and would suit players of the board game “Scrabble”.

We shall introduce words — the type $\text{Word}$, and a typical word $w$:

\[ w \in \text{Word} \]

We shall not say anymore about what words are (strings of characters). Here we simply state that a type of words exists, and it contains distinguishable
elements (words). Introducing types like this without giving further definitions is often quite useful. It allows us to defer decisions about the precise nature of words until some later stage of our modelling. Note that all we can do with words at present is to ask if they are equal or not. In particular, we have no notion of an “alphabetical” or “lexicographic” ordering at this stage.

We shall model a dictionary \( \delta \in \text{Dict} \) as a Set of Words:

\[
\delta \in \text{Dict}_0 = \mathcal{P} \text{Word}
\]

Here we use the ‘0’ subscript to indicate that this is our initial model — later we will produce more elaborate models of the dictionary.

We shall view \( \text{Dict}_0 \) as our system state, and we shall introduce an invariant requiring the words to conform to UK spelling

\[
\text{inv-Dict}_0 : \text{Dict}_0 \rightarrow \mathbb{B}
\]

\[
\text{inv-Dict}_0(\delta) \equiv \text{all words have UK spelling}
\]

(We shall see how to express this mathematically later on...)

Given our definition of a system state and its invariant, we shall want to introduce the notion of an Initial State \( (\delta_0) \):

\[
\delta_0 : \text{Dict}_0
\]

\[
\delta_0 \equiv \emptyset
\]

Here, \( \emptyset \) denotes the empty set, the set with no elements, often also written as \{\}.

We now encounter the concept of Proof Obligation for the first time. A proof obligation is a property of the model that we are required by the method to show holds true for the model. In this case, having defined a state, invariant, and an initial state, we are obliged to prove that the initial state satisfies the system state invariant property:

\[
\text{inv-Dict}_0(\delta_0) = \text{TRUE}
\]

This will be easily shown to be the case, once we have developed a bit more mathematical notation and technique.

We want the following three operations:

- Insert — put word into dictionary.
- Lookup — see if a word is in the dictionary.
- Remove — take a word out of the dictionary.

We shall start by considering the Insert operation \( \text{Ins}_0 \), with the precondition that the word is not already in the dictionary, and that is uses UK spelling. Note that we also label the operation \text{Ins} with a 0 subscript, to indicate that we are talking about our initial model.

This operation takes a word and (pre-existing) dictionary and returns a (modified) dictionary that has the word added in. We could say it has the type:

\[
\text{Ins}_0 : \text{Word} \times \text{Dict}_0 \rightarrow \text{Dict}_0
\]
However, we are going to change its form a little:

\[ \text{Ins}_0 : \text{Word} \rightarrow (\text{Dict}_0 \rightarrow \text{Dict}_0) \]

We are now defining the insert operation to be a function that takes a word as input, and returns a function as a result. The returned function is one that inputs a dictionary and gives back a dictionary as a result. In other words, \( \text{Ins}_0(w) \) is a function that changes an arbitrary dictionary by adding the word \( w \) into it.

To get a better feeling for what has just happened, let us look at a much simpler example. We have taken a function of the form

\[
(A \times B) \rightarrow C
\]

and replaced it with

\[
A \rightarrow (B \rightarrow C)
\]

which we can write as \( A \rightarrow B \rightarrow C \), by assuming that the function arrow “\( \rightarrow \)" associates to the right. This conversion is known as “currying”, after the logician Haskell B. Curry.

Consider real number addition \( (\_ + \_ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}) \) We can say that \( 3 + 4 = 7 \), for example.

Now consider the following definition:

\[
\text{plus} : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}
\]

\[
\text{plus}(a)(b) \equiv a + b
\]

We can say that \( \text{plus}(3)(4) = 3 + 4 = 7 \). However, we can also talk about the value \( \text{plus}(3) \). This value has type \( \mathbb{R} \rightarrow \mathbb{R} \), and denotes the function that adds 3 to its real number argument.

\[
\text{plus}(3)(x) = 3x
\]

This “currying” approach to defining functions of more than one argument is the basis of most modern functional programming languages such as Miranda, Haskell and Clean.

Our interest in curry arises because it is often useful to talk about partially applied functions, and currying makes the notation much simpler.

In particular, when defining operations on the system state in our method, we shall use the form:

\[ \text{Ins}_0 : \text{Word} \rightarrow \text{Dict}_0 \rightarrow \text{Dict}_0 \]

Here the first argument type (\( \text{Word} \)) denotes the parameters or control inputs to the operation, while the function result (\( \text{Dict}_0 \rightarrow \text{Dict}_0 \)) indicates that the operation is a State Transformer, taking an input state and returning a transformed result state.

We can now give a full formal definition of the insert operation, starting by describing the precondition:

\[ \text{pre-Ins}_0 : \text{Word} \rightarrow \text{Dict}_0 \rightarrow \mathbb{B} \]

\[
\text{pre-Ins}_0(w) \delta \equiv \text{isUK}(w) \land w \not\in \delta
\]
Here “isUK” is a predicate (which we shall not define here) which tells if a word has UK spelling or not. We are also using the following notation for boolean (logical) operators:

\[ \land \text{ Logical-And} \]
\[ \lor \text{ Logical-Or} \]
\[ \neg \text{ Logical-Not} \]

We can now define the operation itself

\[ \text{Ins}_0 : \text{Word} \rightarrow \text{Dict}_0 \rightarrow \text{Dict}_0 \]
\[ ON_{\text{Ins}_0}(w) \delta \Delta = \delta \cup \{w\} \]

The operator \( \_ \cup \_ : \mathcal{P}A \times \mathcal{P}A \rightarrow \mathcal{P}A \) is set extension. It behaves like set union, but is only defined for pairs of sets that do not have any elements in common (disjoint sets). Using it here, instead of set union, serves to emphasise the fact that the dictionary set has got larger by one element.

\[ S \cup T \equiv S \cup T, \text{ iff } S \cap T = \emptyset \]

We can now give a definition of the dictionary lookup operation (\( \text{Lkp}_0 \)), which differs in that it does not change the state, but instead performs a query on that state. The first argument is the query input or parameter (Word), and a function is returned that takes a state and returns a query result value, in this case a simple boolean outcome:

\[ \text{Lkp}_0 : \text{Word} \rightarrow \text{Dict}_0 \rightarrow \mathbb{B} \]
\[ \text{Lkp}_0(w) \delta \Delta = w \in \delta \]

An alternative notation to the infix \( w \in \delta \) is to use the curried form \( \chi(w) \delta \). This is the so-called Characteristic Function:

\[ \chi : A \rightarrow \mathcal{P}A \rightarrow \mathbb{B} \]

In short we could say that

\[ \text{Lkp}_0 = \chi \]

The removal operation (\( \text{Rem}_0 \)) is defined as

\[ \text{Rem}_0 : \text{Word} \rightarrow \text{Dict}_0 \rightarrow \text{Dict}_0 \]
\[ \text{Rem}_0(w) \delta \Delta = \delta \setminus \{w\} \]

or

\[ \setminus \{w\} \delta \]

The symbol \( \setminus \) denotes the curried form of \( \setminus \) with arguments reverse (for reasons that will become clear later on).

\[ \setminus : \mathcal{P}A \rightarrow \mathcal{P}A \rightarrow \mathcal{P}A \]
\[ \setminus(S)T \equiv T \setminus S \]

The function \( \setminus(S) \) remove elements in \( S \) from its argument.

Note also we have a “restriction” operator, which is a curried form of set intersection:

\[ \setminus : \mathcal{P}A \rightarrow \mathcal{P}A \rightarrow \mathcal{P}A \]
\[ \setminus(S)T \equiv T \cap S \]
We have curried forms for quite a few traditional set operators:

\[
\begin{array}{c}
\text{Traditional} & \in & \cap & \setminus \\
\text{Curried} & \chi & \triangleleft & \Rightarrow
\end{array}
\]

### 2.4 Exercises 1

Assume

\[
\text{Word} = \{\text{ask, borrow, model, curry, color, colour}\},
\]

that is \(\text{UK}(\text{color} = \text{False})\), is \(\text{UK}\) returns true for all other words, and the initial dictionary is empty.

**Q1.1** Show dictionary contents after the following operations:

- \(\text{Ins}_0(\text{borrow})\)
- \(\text{Ins}_0(\text{curry})\)
- \(\text{Ins}_0(\text{color})\)
- \(\text{Ins}_0(\text{colour})\)
- \(\text{Ins}_0(\text{borrow})\)

**Q1.2** What happens if we then do:

- \(\text{Rem}_0(\text{borrow})\)
- \(\text{Rem}_0(\text{ask})\)

**Q1.3** Write down a precondition for the remove operation that requires the word to be present.

*(Q1.4 Optional)* Prove that the insert operation preserves the invariant.

### 3 Class 3

We now consider other set operations.

#### 3.1 Set Cardinality

Set cardinality \((\#)\) returns the number of elements in a set. We can provide a recursive definition

\[
\begin{align*}
\# : \mathcal{P} A & \rightarrow \mathbb{N} \\
\#\emptyset & \equiv 0 \\
\#(S \cup \{w\}) & \equiv \#S + 1
\end{align*}
\]

Here we are using the set extend operator \((\cup)\), defined only for disjoint sets, in a recursive definition pattern \((S \cup \{w\})\), in order to break a non-empty set into a singleton set \((\{w\})\) and remainder \((S)\).

\(S \cup \{w\}\) requires that \(w \notin S\). We can give an example:
\[ \#\{a, b, c\} = (\text{defn. of } \# \text{, rec. case with } S = \{a, b\}, w = c) \]
\[ \#\{a, b\} + 1 \]
\[ (\text{defn. of } \# \text{, rec. case with } S = \{a\}, w = b) \]
\[ (\#\{a\} + 1) + 1 \]
\[ (\text{defn. of } \# \text{, rec. case with } S = \emptyset, w = a) \]
\[ ((\#\emptyset + 1) + 1) + 1 \]
\[ (0 + 1) + 1 \]
\[ (\text{arithmetically}) \]
\[ 3 \]

An important property of \( \# \) with respect to set extend is
\[ \#(S \cup T) = \#S + \#T \]
provided \( S \) and \( T \) are disjoint. This should be contrasted with using set union:
\[ \#(S \cup T) = \#S + \#T - \#(S \cap T) \]

3.2 Set Mapping

Set Mapping \((\mathcal{P} f)\) takes a set \((S : \mathcal{P} A)\) and applies a function \(f : A \rightarrow B\) to every element of the set, returning a new set of the results \((\mathcal{P} f S : \mathcal{P} B)\), e.g.
\[ \mathcal{P} f \{a_1, a_2, a_3\} = \{f(a_1), f(a_2), f(a_3)\} \]

Note that we are overloading the \(\mathcal{P}\) symbol
\[ \mathcal{P} : (A \rightarrow B) \rightarrow \mathcal{P} A \rightarrow \mathcal{P} B \]
\[ \mathcal{P} f \emptyset \overset{\Delta}{=} \emptyset \]
\[ \mathcal{P} f (S \cup \{a\}) \overset{\Delta}{=} \mathcal{P} f S \cup \{f(a)\} \]

For an example, consider the “sign function” \((\text{sgn} : \mathbb{Z} \rightarrow \mathbb{Z})\) which maps an integer to -1, 0, or +1 according to whether or not it is negative, zero or positive:
\[ \text{sgn}(z) \overset{\Delta}{=} \begin{cases} 
-1 & \text{if } z < 0 \\
0 & \text{if } z = 0 \\
+1 & \text{if } z > 0 
\end{cases} \]

We can map this across a set of integers:
\[ \mathcal{P} \text{sgn} \{1, 2, 3\} = \{1\} \]

This example illustrates clearly why, in the recursive case of the definition of \(\mathcal{P}\), we use set extend in the pattern, but set union in the right-hand side. In general, the function may map distinct set elements to the same value.

Recursive definitions allow us to use induction for proofs — however the results are only valid for finite (countable?) sets.
3.3 Set Reduction

Given a set $S : \mathcal{P}A$, we can repeatedly apply a (associative, commutative) binary operator $\cdot : A \times A \to A$ to reduce the set down to a single value:

$$\gamma : (A \times A \to A) \to \mathcal{P}A \to A$$

For example, $+/S$ returns the sum of all the elements in set $S : \mathcal{P}\mathbb{R}$

$$+/\{a_1, a_2, a_3\} = a_1 + (a_2 + a_3) \quad +/\{a\} = a \quad +/\{a_1, a_2\} = a_1 + a_2$$

What happens if the set is empty? ($+/\emptyset = ?$). An obvious answer is that it should return 0, in the case that we are reducing with addition! What should do in the general case, given an arbitrary binary operator $\cdot$?

Consider the following example: let $p$ be a predicate ($p : A \to \mathbb{B}$). If we map $p$ across a set, we get a set of booleans:

$$\mathcal{P}p\{a_1, \ldots, a_n\} = \{p(a_1) \cup \ldots \cup p(a_n)\} = \{\text{False} \text{ or } \{\text{True} \text{ or } \{\text{False, True}$$

We then reduce the boolean set with logical and:

$$\wedge/\{\text{False, True} \} = \text{False} \quad \wedge/\{\text{False} \} = \text{False} \quad \wedge/\{\text{True} \} = \text{True}$$

The composition of these two operations (map $p$, then reduce with $\wedge$)

$$\wedge/(\mathcal{P}pS)$$

effectively asks does $p$ hold for all elements of $S$?

In predicate logic, this is often written as

$$\forall x \in S \cdot p(x)$$

We can define a function $\forall$ that behaves similarly:

$$\forall : (A \to \mathbb{B}) \to \mathcal{P}A \to \mathbb{B}$$

$$\forall(p)S \overset{\text{def}}{=} \wedge/(\mathcal{P}p)$$

We can now give a proper definition of the invariant for $\text{Die}^0$:

$$\text{inv-Die}^0(\delta) \overset{\text{def}}{=} \forall(\text{is UK})\delta$$

3.4 Function Composition

Often we prefer to use function composition notation, i.e.

$$(f \circ g)(x) \text{ instead of } f(g(x)) \quad \text{i.e. } (^/\circ \mathcal{P}p)S \text{ instead of } \wedge/(\mathcal{P}pS)$$

We read

$$(f \circ g)(x)$$

as “apply $f$ after $g$ to $x$”.

The expression “$(^/\circ \mathcal{P}p)$” denotes the process that reduces using logical-And, after mapping predicate $p$. Note that this expression does not need to mention a specific set $S$, and so is a quite generic description of the process, rather than of a particular instance of its use.
3.5 Exercises 2

Q2.1 Show the step by step evaluation of $P_{\text{sgn}}\{-2, 2, 4\}$ using the recursive definition.

Q2.2 Let $S : \mathbb{P} \mathbb{R}$, what does $\times / S \text{ compute}$ (if $\times$ denotes multiplication) ?

Q2.3 Evaluate $(\neg / \circ P_{\text{isOdd}})\{2, 3, 1, 4\}$

Q2.4 What should be the value of (i) $\neg / \emptyset$ ? (ii) $\neg / \emptyset$ ?

Q2.5 Write down a recursive definition for $\times / S$ (using $\{a\}$ as the base case).

Q2.6 Evaluate $\neg / \{1, 3, 7\}$ where $-$ is subtraction (if you can ! what goes wrong ?).

Q2.7 It makes sense that $\neg / \emptyset$ should equal 0, and that $\neg / \emptyset$ should equal 1. What is the common and key concept here?

4 Class 4

We shall develop another model of the spell-checking dictionary, based on sequences, and then explore its relationship with the set-based model.

4.1 Sequences

We now look at sequences, which are collections of elements where order is significant, and duplicate values are allowed:

$$\{1, 2, 3\} \neq \langle 3, 2, 1 \rangle \neq \{1, 3, 2, 3, 2, 1, 2\}$$

We denote the type “sequence of $A$s”, or “$A$-list” by

$$A^*$$

This is the “Kleene-star” notation, denoting the set of all finite sequences built from elements drawn from $A$. If

$$A = \{a, b, c\}$$

then

$$A^* = \{A, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle a, b \rangle, \langle b, a \rangle, \langle a, c \rangle, \langle c, a \rangle, \langle b, c \rangle, \langle c, b \rangle, \langle a, b, c \rangle, \ldots \}$$

The empty sequence ($\langle \rangle$) is usually denoted $\Lambda$. Note that $A^*$ is an infinite set, even if $A$ is finite.

We can build sequences inductively using the empty sequence ($\Lambda$) and a sequence constructor (often called “cons” or denoted as “;”). The constructor combines an element of $A$ with a sequence from $A^*$ to give a new sequence in $A^*$ which has the element stuck on to the front of the sequence. The notation

$$\langle a, b, c \rangle$$

is syntactic sugar for

$$a : b : c : \Lambda$$
where “cons” (:) associates to the right.

We typically use $\sigma$ to denote sequences.

We can define functions recursively on sequences, using $\Lambda$ as the base case, and $\alpha: \sigma$ as the recursive step case.

A frequent operator defined for sequences is the concatenation operator ($\cdot$) which joins together two sequences to give a new longer one

$$\langle 1, 2, 3 \rangle \cdot \langle 3, 2, 1 \rangle = \langle 1, 2, 3, 3, 2, 1 \rangle$$

We can define concatenation in terms of “cons” as follows:

$$\cdot : A^* \times A^* \rightarrow A^*$$

$\Lambda \cdot \tau \equiv \tau$

$\langle x: \sigma \rangle \cdot \tau \equiv x : (\sigma \cdot \tau)$

4.2 Dictionary Sequence Model

We could model a spell-checker dictionary as a sequence of words, rather than a set — this is closer to a possible computer program based representation. We will call the new dictionary system $\text{Dict}_1$ to distinguish it from the set-based model.

$\sigma \in \text{Dict}_1 = \text{Word}^*$

$\text{inv- Dict}_1(\sigma) \equiv \forall (\text{is UK}) \sigma$

Note that we are overloading the $\forall$ symbol,

$$\forall : (A \rightarrow B) \rightarrow A^* \rightarrow B$$

here it applies to sequences of elements, rather than sets, and returns true only if the predicate holds true for all elements of the sequence. We can define $\forall$ on sequences as

$$\forall(p) \equiv \land / p^*$$

We see that reduction is also overloaded to apply to sequences, and that $f^*$ ought to play a similar role with sequences, as $\mathcal{P} f$ does with sets.

4.3 Sequence Reduction

$\ast$ / reduces a sequence using binary operator $\ast$:

$$\ast / \langle 1, 2, 3, 4, 5 \rangle = 1 + 2 + 3 + 4 + 5 = 15$$

Unlike with set reduction, the operator need not be commutative, or associative. However, if the operator is not associative, we need to distinguish between left- and right-reduction ($\ast_l$ and $\ast_r$, resp.):

$$\ast_l / \langle 1, 3, 7 \rangle = (1 - 3) - 7 = -9$$

$$\ast_r / \langle 1, 3, 7 \rangle = 1 - (3 - 7) = +5$$

16
4.4 Sequence Mapping

Given $f : A \rightarrow B$ and $\sigma : A^*$ then $f^*\sigma$ has type $B^*$ and is the result of applying $f$ to every element of $\sigma$

$$f^*(a, b, c) = (f(a), f(b), f(c))$$

We notice the same parallel between the use of a symbol as a type constructor, and as a higher-order function that maps across elements of the corresponding type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>$\mathcal{P}A$</td>
</tr>
<tr>
<td>Sequence</td>
<td>$A^*$</td>
</tr>
</tbody>
</table>

4.5 Initial Dictionary

The initial (empty) dictionary is modelled by the empty sequence:

$$\sigma_0 : \text{Dict}_1$$
$$\sigma_0 \equiv \Lambda$$

4.6 Relating Abstract and Concrete

We are going to describe how to capture the key relationship between a model of an abstract concept, and a model of a more concrete form of the concept. We shall do this by defining a Retrieval Function which maps from concrete instances of the model to the corresponding abstract instance:

<table>
<thead>
<tr>
<th>Abstract</th>
<th>$\text{Dict}_0$</th>
<th>$\mathcal{P}\text{Word}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>$\text{ret}_0$</td>
<td>$\text{elem}$</td>
</tr>
<tr>
<td>Concrete</td>
<td>$\text{Dict}_1$</td>
<td>$\text{Word}^*$</td>
</tr>
</tbody>
</table>

Each sequence can be viewed as representing a set:

- $(1, 2, 3)$ represents $\{1, 2, 3\}$
- $(3, 2, 1)$ represents $\{1, 2, 3\}$
- $(1, 3)$ represents $\{1, 3\}$

A given set may have many representations. A given sequence represents exactly one set. We can describe the relationship between Abstract and Concrete as a function $\text{Concrete} \rightarrow \text{Abstract}$. We refer to these functions as retrieval functions.

For the dictionary models, we have a function that retrieves the abstract set-based model, from the concrete sequence based one.

$$\text{ret}_0 : \text{Dict}_1 \rightarrow \text{Dict}_0$$

The retrieval function can be defined as

$$\text{ret}_0(\sigma) \equiv \text{set of elements in } \sigma$$
$$\equiv \text{elem } \sigma$$

17
Here we have introduced a function \( \text{elems} \) that relates sequences to sets:

\[
\begin{align*}
\text{elems} & : A^* \to \mathcal{P}A \\
\text{elems} \Lambda & \equiv \emptyset \\
\text{elems}(a: \sigma) & \equiv \{a\} \cup (\text{elems} \sigma) \\
\text{elems}(1, 2, 3) & = \{1, 2, 3\} \\
\text{elems}(3, 2, 1, 3, 2) & = \{1, 2, 3\}
\end{align*}
\]

Once we have two models related this way, a whole new collection of proof obligations are introduced. The first of these states that, for any concrete model instance that satisfies the concrete invariant, the retrieval of it to the corresponding abstract instance, must satisfy the abstract invariant:

\[
\left[ i_{\text{inv-Dict}}_1(\sigma) \Rightarrow i_{\text{inv-Dict}}_0(\text{retr-Dict}^\prime_1(\sigma)) \right]
\]

This check of invariant preservation by retrieval is a basic form of “sanity-check”, ensuring that the concrete model still faithfully captures the key properties of the abstract form.

This reduces, in the case of the dictionary models to showing that

\[
\forall(isUK)\sigma \Rightarrow \forall(isUK)(\text{elems} \sigma)
\]

where the first “\(\forall\)” is that defined on sequences, whereas the second is that defined for sets.

### 4.7 Operations on Sequence Dictionaries

We can define corresponding operation that work on sequence based dictionaries. For insertion:

\[
\begin{align*}
\text{Ins}_1 & : \text{Word} \to \text{Dict}_1 \to \text{Dict}_1 \\
\text{pre-Ins}_1(w)\sigma & \equiv w \notin \text{elems} \sigma \\
\text{Ins}_1(w)\sigma & \equiv w : \sigma
\end{align*}
\]

For lookup:

\[
\begin{align*}
\text{Lkp}_1 & : \text{Word} \to \text{Dict}_1 \to \mathbb{B} \\
\text{Lkp}_1(w)\sigma & \equiv w \in \text{elems} \sigma \\
\text{or} & \\
\text{Lkp}_1(w)\Lambda & \equiv \text{FALSE} \\
\text{Lkp}_1(w)(x : \sigma) & \equiv w = x \lor \text{Lkp}_1(w)\sigma
\end{align*}
\]

The alternative form shows how we can define our operators using recursion or pre-existing operators. For remove:

\[
\begin{align*}
\text{Rem}_1 & : \text{Word} \to \text{Dict}_1 \to \text{Dict}_1 \\
\text{Rem}_1(w)\Lambda & \equiv \Lambda \\
\text{Rem}_1(w)(x : \sigma) & \equiv w = x \to \text{Rem}_1(w)\sigma \\
& \quad \text{if } w \neq x \to x : \text{Rem}_1(w)\sigma
\end{align*}
\]

18
How do these three operators relate to \( \text{Ins}_0 \), \( \text{Lkp}_0 \) and \( \text{Rem}_0 \)? How do we develop the correct form of a **Operation Correctness Proof Obligation**? The key is the following commuting diagram:

\[
\begin{array}{ccc}
\text{Dict}_0 & \xrightarrow{\text{Ins}_0(w)} & \text{Dict}_0 \\
\downarrow{\text{retr-Dict}_1^0} & & \downarrow{\text{retr-Dict}_1^0} \\
\text{Dict}_1 & \xrightarrow{\text{Ins}_1(w)} & \text{Dict}_1 \\
\end{array}
\]

\( \text{inv-Dict}_1 \land \text{pre-Ins}_1(w) \)

We want the same outcome to occur if we start at the bottom-left corner, and either go first up and then to the right, or first to the right and then up. This gives us:

\[
\text{inv-Dict}_1 \land \text{pre-Ins}_1(w) \Rightarrow \text{retr-Dict}_1^0 \circ \text{Ins}_1(w) = \text{Ins}_0(w) \circ \text{retr-Dict}_1^0
\]

In words, if the concrete invariant holds, and the concrete operation precondition is true, then retrieval after performing the concrete operation gives the same result as performing the abstract operation after retrieval.

## 5 Class 5

We have seen a number of **Proof Obligations** associated with the method. This would seem an opportune time to introduce the notion of **Proof**.

### 5.1 Proof Principles

The key principle of our proof technique, is that of the “substitution of equals for equals”, also known as the principle of **Referential Transparency**, or **Equational Reasoning**. If we know that \( e_1 = e_2 \), then we can safely substitute one for the other, **no matter in what context they actually occur**.

\[
\text{BigExpression}(e_1) = \text{BigExpression}(e_2)
\]

An expression of the form \( e_1 = e_2 \) is often called an “identity”. Many of our mathematical laws are expressed as such identities, e.g. the law which state that sequence concatenation is associative:

\[
\rho \cdot (\sigma \cdot \tau) = (\rho \cdot \sigma) \cdot \tau
\]

Here, \( \rho, \sigma \) and \( \tau \) stand for arbitrary sequences, so the law as expressed above is in fact shorthand for:

“For all \( \rho, \sigma, \tau \in A^* \) we have \( \rho \cdot (\sigma \cdot \tau) = (\rho \cdot \sigma) \cdot \tau \)”

We would like to point out that Predicate Calculus, and similar logics, do not follow this principle. Neither do imperative or object-oriented languages, when it comes to reasoning about them. Pure functional programming languages,
like Miranda, Haskell and Clean do support this principle. The principle is of interest because it makes reasoning easier to do.

The proof style reflects the use of equality. To prove \( e_1 = e_2 \) for given \( e_1 \) and \( e_2 \) we generally try one of the following things:

1. Simplify \( e_1 \) using known identities until it equals \( e_2 \).
2. Simplify \( e_2 \) using known identities until it equals \( e_1 \).
3. Simplify both \( e_1 \) and \( e_2 \) using known identities until they both some other expression \( e' \).
4. Another approach is to try and use Induction.

Consider proving

\[
((\sigma_1 \circ \sigma_2) \circ \sigma_3) \circ \sigma_4 = \sigma_1 \circ (\sigma_2 \circ (\sigma_3 \circ \sigma_4))
\]

using the first approach:

\[
\begin{align*}
((\sigma_1 \circ \sigma_2) \circ \sigma_3) \circ \sigma_4 &= \text{(associativity of } \circ, \text{ rhs } \mapsto \text{lhs)} \\sigma_1 \circ (\sigma_2 \circ (\sigma_3 \circ \sigma_4)) \\
&= \quad \text{(associativity of } \circ, \text{ rhs } \mapsto \text{lhs)} \\
&= \sigma_1 \circ (\sigma_2 \circ (\sigma_3 \circ \sigma_4))
\end{align*}
\]

In step 1, we used the associativity identity with the following bindings:

\[
\rho \mapsto (\sigma_1 \circ \sigma_2), \sigma \mapsto \sigma_3, \tau \mapsto \sigma_4
\]

In step 2, the binding was:

\[
\rho \mapsto \sigma_1, \sigma \mapsto \sigma_2, \tau \mapsto (\sigma_3 \circ \sigma_4)
\]

5.2 Proof by Induction

We would like to prove certain key identities/laws from basic definitions. As a lot of these definitions are recursive, the natural way to do the corresponding proofs is by induction.

For \( \text{Sels} \), the base case is the empty set (\( \emptyset \)), and the inductive step case is an arbitrary set extended with a new element (\( S \cup \{x\} \)).

For \( \text{Seqs} \), the base case is the empty sequence (\( \Lambda \)), and the inductive step case is an arbitrary sequence with a new element “consed” to the front (\( x : \sigma \)).

To show that a property \( p \) hold for an arbitrary set \( S \), we need to show that it holds for \( \emptyset \), and, if \( p \) holds for \( S \), it also holds for \( S \cup \{x\} \) (where clearly \( x \notin S \)).

\[
\forall S: p(S) = \text{True} \iff p(\emptyset) = \text{True} \land p(S) = \text{True} \Rightarrow p(S \cup \{x\}) = \text{True}
\]

For sequences, a similar condition can be stated:

\[
\forall \sigma: p(\sigma) = \text{True} \iff p(\Lambda) = \text{True} \land p(\sigma) = \text{True} \Rightarrow p(x : \sigma) = \text{True}
\]

20
As an example, we prove the following
\[ \text{len}(\sigma \sim \tau) = \text{len } \sigma + \text{len } \tau \]
by induction, on the first sequence (\textit{sigma}), so we have
\[ p(\sigma) \equiv (\text{len}(\sigma \sim \tau) = \text{len } \sigma + \text{len } \tau) \]

**Base Case** \( p(\Lambda) \)
\[ \text{len}(\Lambda \sim \tau) = \text{len } \Lambda + \text{len } \tau \]
We prove this by simplifying both sides to the same expression. First the left-hand side (lhs):
\[
\begin{align*}
\text{len}(\Lambda \sim \tau) &= (\text{definition of } \sim) \\
&= \text{len } \tau
\end{align*}
\]
Now, the right-hand side (rhs):
\[
\begin{align*}
\text{len } \Lambda + \text{len } \tau &= (\text{defn. of len}) \\
&= 0 + \text{len } \tau \\
&= (\text{arithmetic}) \\
&= \text{len } \tau
\end{align*}
\]
Both sides are equal.

**Inductive Step** \( p(\sigma) \Rightarrow p(x : \sigma) \) Assuming that
\[ \text{len}(\sigma \sim \tau) = \text{len } \sigma + \text{len } \tau \]
prove
\[ \text{len}((x : \sigma) \sim \tau) = \text{len } (x : \sigma) + \text{len } \tau \]
This is a complex thing to be shown. We could use the identity
\[ A \Rightarrow B = \neg A \lor B \]
to transform the above, and then try to simplify it all down to \textit{true}. However, a much simpler form would be to use the assumption as a valid identity in the proof of the consequence. In other words, to prove
\[ A_1 = A_2 \Rightarrow B_1 = B_2 \]
we simply prove \( B_1 = B_2 \) to be true, in a context where \( A_1 = A_2 \) is taken to be a true identity. This approach fits best with equational reasoning.
So, given
\[ \text{len}(\sigma \sim \tau) = \text{len } \sigma + \text{len } \tau \]
for \textit{fixed value of } \sigma, we prove
\[ \text{len}((x : \sigma) \sim \tau) = \text{len } (x : \sigma) + \text{len } \tau \]
As for the base case, we reduce both lhs and rhs to the same form. Starting with the lhs:
\[
\begin{align*}
\text{len}(x : \sigma \vdash \tau) \\
&= \text{(defn. of } \vdash \text{)} \\
\text{len}(x : (\sigma \vdash \tau)) \\
&= \text{(defn. of } \text{len}) \\
&= 1 + \text{len}(\sigma \vdash \tau)
\end{align*}
\]

For the rhs:
\[
\begin{align*}
\text{len}(x : \sigma) + \text{len } \tau \\
&= \text{(defn. of length, associativity of +)} \\
&= 1 + \text{len } \sigma + \text{len } \tau \\
&= \text{(assumption, rhs } \mapsto \text{ lhs)} \\
&= 1 + \text{len}(\sigma \vdash \tau)
\end{align*}
\]

Q.E.D.

5.3 Exercises 3

Q3.1 Prove
\[
\#(S \sqcup T) = \#S + \#T
\]
given extra laws:
\[
\begin{align*}
(S \sqcup \{x\}) \sqcup T &= (S \sqcup T) \cup \{x\} \\
S \sqcup T &= T \sqcup S \\
S \cup (T \cup U) &= (S \cup T) \cup U \\
\emptyset \sqcup S &= S
\end{align*}
\]
where all sets are disjoint.

Q3.2 Prove
\[
\text{elems}(\sigma \vdash \tau) = \text{elems } \sigma \cup \text{elems } \tau
\]

6 Class 6

Now, let us prove a model-related property, namely that “Retrieval preserves Invariant”:
\[
\forall \sigma : \text{inv-Dict}_1(\sigma) \Rightarrow \text{inv-Dict}_0(\text{retr-Dict}_1^0(\sigma))
\]

If we replace “method functions” by their definitions we get
\[
P(\sigma) \equiv \forall (\text{isUK})\sigma \Rightarrow \forall (\text{isUK})(\text{elems } \sigma)
\]

We prove this by induction over \(\sigma\).

Base Case \(P(\Lambda) \equiv \forall (\text{isUK})\Lambda \Rightarrow \forall (\text{isUK})(\text{elems } \Lambda)\)

We simplify the expression to TRUE.
\[
\forall (s : \text{US}) \Lambda \Rightarrow \forall (s : \text{US}) (\text{elem} s \Lambda) \\
= \quad \text{(defn. elem)} \\
\forall (s : \text{US}) \Lambda \Rightarrow \forall (s : \text{US}) (\text{elem} \emptyset) \\
= \quad \text{(defn. \forall on Sets)} \\
\forall (s : \text{US}) \Lambda \Rightarrow \text{TRUE} \\
= \quad \text{(defn. \forall on Sequences)} \\
\text{TRUE} \Rightarrow \text{TRUE} \\
= \quad \text{(defn. \Rightarrow)} \\
\text{TRUE}
\]

**Inductive Step** \( P(\sigma) \Rightarrow P(w : \sigma) \)

\[
(\forall (s : \text{US}) \Rightarrow \forall (s : \text{US}) (\text{elem} \sigma)) \Rightarrow (\forall (s : \text{US}) (w : \sigma) \Rightarrow \forall (s : \text{US}) (\text{elem} (w : \sigma)))
\]

We have a implications whose both sides are themselves (nested) implications, i.e and expression of the form \((A \Rightarrow B) \Rightarrow (C \Rightarrow D)\) We could attempt to simplify this as it stands, or by taking the hypothesis as given, so showing \(C \Rightarrow D\) given \(A \Rightarrow B = \text{TRUE}\). Alternatively, we can exploit a law known as the **monotonicity of \&**:\[
(A \Rightarrow B) \Rightarrow (C \land A \Rightarrow C \land B)
\]

\[
\forall (s : \text{US}) (\sigma \Rightarrow \forall (s : \text{US}) (\text{elem} \sigma)) \\
= \quad \text{(monotonicity of \&)} \\
\text{US}(w) \land \forall (s : \text{US}) (\sigma \Rightarrow \text{US}(w) \land \forall (s : \text{US}) (\text{elem} \sigma)) \\
= \quad \text{(defn. of \forall for sequences)} \\
\text{US}(w) (w : \sigma) \Rightarrow \text{US}(w) \land \forall (s : \text{US}) (\text{elem} \sigma) \\
= \quad \text{(defn. of \forall for sets)} \\
\forall (s : \text{US}) (w : \sigma) \Rightarrow \forall (s : \text{US}) (\text{elem} \{w\} \cup \text{elem} \sigma) \\
= \quad \text{(defn. of elem)} \\
\forall (s : \text{US}) (w : \sigma) \Rightarrow \forall (s : \text{US}) (\text{elem} (w : \sigma))
\]

Isn’t induction wonderful! Recursion and Induction form a Foundation of a lot of our mathematics, but suffer from being very low-level. We want a higher level of reasoning using “nice” properties. However, we have a lot of properties, and would like to avoid having to memorise them all. We could list them all out, but then we have a searching problem. What we really would like is some form of “organising principle” for structuring our collection of properties.

The key idea is to use Mathematical Structure and Morphisms to help in choosing the key properties we would like to establish. We then use induction over the recursive definitions of values and types to establish these properties.

## 7 Class 7

### 7.1 Mathematical Structures

Mathematical Structures are composed from

- one of more given Sets
• one or more functions defined over those sets\(^1\)
• one or more distinguished elements in the sets
• some key properties

In what follows, we shall limit our attention to structures with one given set \((A)\), a function which is a binary operator over the given set \((\ast : A \times A \rightarrow A)\), and zero or more other distinguished elements or unary functions.

### 7.2 Semigroups

We want structures that catch common useful properties. We shall start with the most important two, namely

• the binary operator is total and closed, i.e. \(a_1 \ast a_2\) is defined for all \(a_1, a_2\) in \(A\), and the result is also in \(A\).

• the binary operator is associative, i.e., for all \(a_1, a_2\) and \(a_3\) in \(A\), we have

\[
(a_1 \ast (a_2 \ast a_3)) = ((a_1 \ast a_2) \ast a_3)
\]

So, given

• Set \(A\)
• Binary Operator \(\ast : A \times A \rightarrow A\)
• Property: \(\ast\) is associative

we say that the structure \((A, \ast)\) is a “Semigroup”. If the context requires it, we may emphasise this by declaring

\[
SGrp(A, \ast)
\]

Examples of Semigroups:

\[
(N, +) \ (N, \times) \ (\mathbb{Z}, +) \ (\mathbb{Z}, \times) \ (\mathbb{Q}, +) \ (\mathbb{Q}, \times) \ (\mathbb{R}, +) \ (\mathbb{R}, \times) \\
(\mathbb{B}, \land) \ (\mathbb{B}, \lor) \ (P. \cup) \ (P. \cap) \ (A^*, \ast)
\]

### 7.3 Monoids

The next property of interest is that of the existence, or otherwise, of an identity element for the binary operator, i.e., an element \(e : A\) such that, for any \(a \in A\):

\[
e \ast a = a = a \ast e
\]

So, given

• Set \(A\)

\(^1\)If the functions map from some combination of those sets into a single such set, then the structures are said to be “algebraic”. If the functions map from a single set to some combination of them, then the structures are “co-algebraic”. All of the structures mentioned in this document are algebraic.
• Binary Operator \( \star : A \times A \to A \)
• Property: \( \star \) is associative
• Distinguished Element \( e : A \)
• Property: \( e \) is identity for \( \star \)

we say that the structure \((A, \star, e)\) is a “Monoid”. If the context requires it, we may emphasise this by declaring

\[
\text{Mon}(A, \star, e)
\]

Examples of Monoids:

\[
(N, +, 0) \quad (\mathbb{N}, \times, 1) \quad (\mathbb{Z}, +, 0) \quad (\mathbb{Z}, \times, 1) \quad (\mathbb{Q}, +, 0) \quad (\mathbb{Q}, \times, 1) \quad (\mathbb{R}, +, 0.0) \quad (\mathbb{R}, \times, 1.0) \\
(\mathbb{B}, \land, \text{True}) \quad (\mathbb{B}, \lor, \text{False}) \quad (\mathcal{P}(A, \cup, \emptyset) \quad (\mathcal{P}(A, \cap, A) \quad (A^*, \ominus, A)
\]

7.4 Groups

Finally, we look at the property concerning of the existence, or otherwise, of an inverse elements for the binary operator, i.e. given \( a : A \), the existence of an element (usually denoted \( a^{-1} : A \)) such that

\[
a^{-1} \star a = e = a \star a^{-1}
\]

So, given

• Set \( A \)
• Binary Operator \( \star : A \times A \to A \)
• Property: \( \star \) is associative
• Distinguished Element \( e : A \)
• Property: \( e \) is identity for \( \star \)
• Function: \( -1 : A \to A \)
• Property \( a^{-1} \) is inverse of \( a \)

we say that the structure \((A, \star, e, -1)\) is a “Group”. If the context requires it, we may emphasise this by declaring

\[
\text{Grp}(A, \star, e, -1)
\]

Examples of Groups:

\[
(\mathbb{Z}, +, 0, -) \quad (\mathbb{Q} \setminus 0, \times, 1, -1)
\]

7.5 Morphisms

We now introduce a very powerful concept — that of structure preserving functions or **Morphisms**.
7.5.1 Semigroup Homomorphisms

Consider two semigroups, \((A, \ast)\) and \((B, \circ)\), and a function \(h : A \rightarrow B\), such that
\[
h(a_1 \ast a_2) = h(a_1) \circ h(a_2)
\]
In this case, we say that \(h\) is a “\textit{Semigroup Homomorphism}”, which we can declare as follows:
\[
h : (A, \ast) \rightarrow (B, \circ)
\]
Essentially such a homomorphism preserves the semigroup structure in the sense that we can either first apply one binary operator and then the homomorphism, or else apply the homomorphism twice and then use the second operator, both approaches leading to the same outcome. We can picture this as a commuting diagram:

\[
\begin{array}{ccc}
A \times A & \xrightarrow{\ast} & A \\
\downarrow{h \times h} & & \downarrow{h} \\
B \times B & \xrightarrow{\circ} & B
\end{array}
\]

Examples of semigroup homomorphisms:

\[
\begin{align*}
\log & : (\mathbb{R}, \times) \rightarrow (\mathbb{R}, +) \\
\text{len} & : (A^*, \subset) \rightarrow (\mathbb{N}, +) \\
\text{elems} & : (A^*, \subset) \rightarrow (\mathcal{P}A, \cup) \\
\mathcal{P}f & : (\mathcal{P}A, \cup) \rightarrow (\mathcal{P}B, \cup) \\
\mathcal{S}(S) & : (\mathcal{P}A, \cup) \rightarrow (\mathcal{P}A, \cup) \\
\mathcal{S}(S) & : (A^*, \subset) \rightarrow (A^*, \subset)
\end{align*}
\]

The latter three are \textit{endomorphisms}, i.e. homomorphisms from a semigroup into itself.

This approach to describing functions is very compact. For example, consider the declaration of \text{elems} above. This is in effect shorthand for the following properties:

\[
\begin{align*}
\cup & \text{ is associative} \\
\subset & \text{ is associative} \\
\text{elems} & : A^* \rightarrow \mathcal{P}A \\
\text{elems}(\sigma \subset \tau) & = \text{elems} \sigma \cup \text{elems} \tau
\end{align*}
\]

7.5.2 Monoid Homomorphisms

Consider two monoids, \((A, \ast, e)\) and \((B, \circ, e')\), and a function \(h : A \rightarrow B\), such that
\[
h(a_1 \ast a_2) = h(a_1) \circ h(a_2) \quad \text{and} \quad h(e) = e'
\]
In this case, we say that \(h\) is a “\textit{Monoid Homomorphism}”, which we can declare as follows:
\[
h : (A, \ast, e) \rightarrow (B, \circ, e')
\]
We can picture this as a commuting diagram:

\[
\begin{array}{ccc}
A \times A & \xrightarrow{\kappa} & A \\
\downarrow{h \times h} & & \downarrow{h} \\
B \times B & \xrightarrow{\circ} & B \\
\end{array}
\]

Examples of monoid homomorphisms:

\[
\begin{align*}
\log & : (\mathbb{R}, \times, 1) \to (\mathbb{R}, +, 0) \\
\text{len} & : (A^*, \land, A) \to (\mathbb{N}, +, 0) \\
\text{elements} & : (A^*, \land, A) \to (\mathcal{P}A, \cup, \emptyset) \\
\mathcal{P}f & : (\mathcal{P}A, \cup, \emptyset) \to (\mathcal{P}B, \cup, \emptyset) \\
\mathfrak{g}(S) & : (\mathcal{P}A, \cup, \emptyset) \to (\mathcal{P}A, \cup, \emptyset) \\
\mathfrak{g}(S) & : (A^*, \land, A) \to (A^*, \land, A)
\end{align*}
\]

In fact - all of the semigroup homomorphisms mentioned previously are in fact monoid homomorphisms as well.

The definition of a group homomorphism is left as an exercise.

### 7.6 Are Homomorphisms Unique?

Consider the declaration above of \text{len} as a monoid homomorphism. This gives rise to the following properties:

\[
\begin{align*}
\land : A^* \times A^* & \to A^* \\
\lor & \text{ is associative} \\
\Lambda \lor \sigma = \sigma \lor \Lambda & = 0 + n = n + 0 \\
\text{len}(\sigma \lor \tau) & = \text{len}(\sigma) + \text{len}(\tau) \\
\text{len}(\Lambda) & = 0
\end{align*}
\]

Is this enough to define \text{len} uniquely?

To examine this, let's assume that \( A = \mathbb{N} \), and consider the following homomorphism:

\[
\text{sum} : (\mathbb{N}^*, \land, A) \to (\mathbb{N}, +, 0)
\]

with the recursive definition:

\[
\begin{align*}
\text{sum} \Lambda & \triangleq 0 \\
\text{sum}(n : \sigma) & \triangleq n + \text{sum}(\sigma)
\end{align*}
\]

We can we that both \text{len} and \text{sum} form homomorphisms from sequences of naturals to naturals. Clearly we need more information to distinguish between them.

### 7.7 Exercises 4

**Q4.1** Give a definition of a Group Homomorphism.
Q4.2 Is log a group homomorphism?
\[ (\mathbb{R}, \times, 1^{-1}) \to (\mathbb{R}, +, 0, -) \]
If not, why not? How would change things to get a homomorphism?

Q4.3 Is \( \leq (\mathbb{R}) \) a homomorphism \( (\mathcal{P} A, \cup) \to (\mathcal{P} A, \cap) \)?

Q4.4 Is \( \leq (\mathbb{R}) \) a homomorphism \( (\mathcal{P} A, \cap) \to (\mathcal{P} A, \cup) \)?

Q4.5 Is either \( \leq (\mathbb{R}) \) or \( \leq (\mathbb{R}) \) a homomorphism \( (\mathcal{P} A, \cup, \emptyset) \to (\mathcal{P} A, \cap, A) \)?

Q4.6 (Optional) Is \( \leq (\mathbb{R}) \) a homomorphism \( (\mathcal{P} A, \cup, \emptyset) \to (\mathcal{P} A, \cap, \emptyset) \)?

8 Class 8

8.1 Fully Defining Homomorphisms

Consider:

\[
\begin{align*}
\text{len} & : (A^*, \neg, A) \to (\mathbb{N}, +, 0) \\
\text{sum} & : (\mathbb{N}^*, \neg, A) \to (\mathbb{N}, +, 0)
\end{align*}
\]

We want to distinguish between them, when \( A = \mathbb{N} \).

We consider where do carrier sets come from? Being more specific, we ask, can we generate a carrier set in some way using a small number of elements and the operations of the structure. In other words, can every element of the carrier be built up in this way?

We can generate \( \mathbb{N} \) from \( \{0, 1\} \) using +.

We can generate \( A^* \) from \( \{A\} \cup \{a \mid a \in A\} \), using \( \neg \).

The sets above are known as generator sets.

Provided each element is generated uniquely, or the operator is both commutative and associative, we can complete a definition of a homomorphism by giving its action on the elements of the generator set.

For \( \text{len} \) and \( \text{sum} \) we need to show how they act for singleton list elements:

\[
\begin{align*}
\text{len}(a) & \equiv 1 \\
\text{sum}(n) & \equiv n
\end{align*}
\]

We can generate \( \mathcal{P} A \) from \( \{\emptyset\} \cup \{\{a\} \mid a \in A\} \), using \( \cup \).

This approach does not work very well with Real numbers — there is no minimal generator set — any open interval containing 0 will generate \( \mathbb{R} \) using +.

The key principle is that a homomorphism is fully defined by identifying the source and target structures, and showing its outcome when applied to any generator element.

8.2 Composing Homomorphisms

An important property of homomorphisms is, given

\[
\begin{align*}
h_1 & : (A, \ast, e) \to (B, \circ, e') \\
h_2 & : (B, \circ, e') \to (C, \ast, e'')
\end{align*}
\]

28
is that their composition is also a homomorphism:

\[ h_2 \circ h_1 : (A, \ast, e) \to (C, \cdot, e') \]

### 8.3 More Homomorphisms

The mapping and reducing operators we introduced for sets and sequences also give rise to homomorphisms. Mapping is homomorphic regardless of the function being mapped. If \( f : A \to B \), then

\[
P f : (P A, \cup, \emptyset) \to (P B, \cup, \emptyset) \\
P f(a) \equiv \{f(a)\} \\
f^* : (A^*, \preceq, \Lambda) \to (B^*, \preceq, \Lambda) \\
f^*(a) \equiv \{f(a)\}
\]

When is \( ^* / \) homomorphic? For sequences, we require:

\[ ^*/(\sigma \cdot \tau) = ^*/\sigma \cdot ^*/\tau \]

which implies that \( \ast \) must be associative. For sets, a similar consideration,

\[ ^*/(S \cup T) = ^*/\sigma \cdot ^*/\tau = ^*/(T \cup S) = ^*/\tau \cdot ^*/\sigma \]

forces us to stipulate that \( \ast \) in this case must be associative and commutative. For a monoid homomorphism we need to be able to define \( ^*/\Lambda \) and \( ^*/\emptyset \), and we find that \( \ast \) needs to have an identity element \( e \), and we define

\[ ^*/\Lambda \equiv e \quad \text{and} \quad ^*/\emptyset \equiv e \]

If \( \ast \) is a monoid operator, it satisfies the requirements for sequence reduction, so we can define sequence reduction as a monoid homomorphism as follows:

\[ ^*/ : (A^*, \preceq, \Lambda) \to (A, \ast, e) \\
^*/(a) \equiv a \]

If \( \ast \) is a commutative monoid operator, it satisfies the requirements for set reduction, so we can define set reduction as a monoid homomorphism as follows:

\[ ^*/ : (P A, \cup, \Lambda) \to (A, \ast, e) \\
^*/(a) \equiv a \]

Given this approach using homomorphisms defined on generator elements, there is a case for moving from the “cons” notation for lists (i.e. \( x : \sigma \)) to a notation using concatenation and singleton sequences \((x) \preceq \sigma\). The inductive step of a proof on sequences would then read as

\[ P(\sigma) \Rightarrow P((x) \preceq \sigma) \]

For example, in an earlier example we had the expression

\[(^*/ \circ \text{PisUK})(x : \sigma)\]

We would now rewrite this, and could immediately make the following proof step:
\[(\lor \circ \text{PisUK})(\{x\} \land \sigma)\]
\[= (\lor \circ Pf \text{ is comp. of homomorphisms}) \land (\lor \circ \text{PisUK})\sigma\]

Generally, we find that reducing after mapping is itself a homomorphism. Two notable examples are the “for all” and “exists” predicates:

\[\forall(p)S \equiv (\lor \circ \mathcal{P}p)S \quad \forall(p) \text{ is a homomorphism}\]
\[\exists(p)S \equiv (\lor \circ \mathcal{P}p)S \quad \exists(p) \text{ is a homomorphism}\]

Sticking with the boolean theme, another notable homomorphism is logical negation:

\[\neg : (\mathbb{B}, \land, \text{True}) \leftrightarrow (\mathbb{B}, \lor, \text{False})\]
\[\neg \text{True} = \text{False} \quad \neg \text{False} = \text{True}\]
\[\neg(a \land b) = \neg a \lor \neg b \quad \neg(a \lor b) = \neg a \land \neg b\]

Logical negation is a homomorphism both ways between logical-And and logical-Or. This statement is just a re-phrasing of De Morgan’s Laws.

We can use the properties, for instance, to prove

\[\neg \circ \forall(p) = \exists(\neg \circ p)\]

This result is often stated in predicate calculus as

\[\neg \forall x \cdot P(x) \equiv \exists x \cdot (\neg P(x))\]

The proof relies on the fact, that all we need to do to prove two homomorphism equal, is to show they have the outcome on generator elements. If \(p : A \rightarrow \mathbb{B}\), then both the left- and right-hand sides of the equation above denote homomorphisms from

\[(\mathcal{P} A, \lor, \land) \rightarrow (\mathbb{B}, \lor, \text{False})\]

The generator is \(\{a\}\) for all \(a \in A\). So we prove:

\[\neg \circ \forall(p) | \{a\} = (\exists(\neg \circ p) | \{a\}\]

Proof: Simplify lhs:

\[\neg (\forall(p) | \{a\}\]
\[= (\text{defn. of } \forall)\]
\[\neg (\lor \circ \mathcal{P}p) | \{a\}\]
\[= (\text{defn. of } \mathcal{P})\]
\[\neg [\mathcal{P}(\neg \circ p)] | \{a\}\]
\[= (\text{defn. of } \mathcal{P})\]
\[\neg p(a)\]

Simplify rhs:

\[\exists(\neg \circ p) | \{a\}\]
\[= (\text{defn. of } \exists)\]
\[\mathcal{P}(\neg \circ p) | \{a\}\]
\[= (\text{defn. of } \mathcal{P})\]
\[\neg p(a)\]

\[\neg \circ p\]

\[30\]
8.4 Structure/Morphism Summary

Structures and Morphisms provide a way to give compact
— descriptions of properties;
— definitions of functions.

However, not everything forms a nice structure or is a homomorphism. But it
is best to use such structures and morphisms when there is a choice.

8.5 Extending Dictionary Model

The idea now is to extend our dictionary model to include word definitions
(WordDef). We shall view a word definition as simply being a sequence of
words (the text of the definition):

\[ d \in \text{WordDef} = \text{Word}^* \]

We want to associate definitions with words in the dictionary.

\[ D \in \text{Dict}_2 = \text{associate a definition with each word} \]

To do this we need to introduce a new type, the type of Maps.

8.6 Maps

We describe a map as associating two sets. One set is called the Source or
Domain Set S. The other set is called the Target, Range or Co-Domain set T.

We write a map from S to T as

\[ S \rightarrow T \]

A map is a function from S to T which associates at most one value of T with
each value in S. In most cases these maps are partial finite functions.

Consider a concrete example, a map associating an age with persons:

\[
\begin{align*}
\text{Person} & = \{\text{AB, AJ, PG}\} \\
\text{Age} & = \mathbb{N} \\
\alpha & \in \text{Person} \rightarrow \text{Age}
\end{align*}
\]

In the first instance, we might have an empty maps (\emptyset) which has no associations
in it. We could also have a singleton map, giving an age to one person only:

\[
\{\text{AB} \mapsto 39\}
\]

Such a singleton map is often referred to a a “maplet”. We could have ages
assigned to two people:

\[
\{\text{AB} \mapsto 39, \text{PG} \mapsto 34\}
\]

Note all of the maps are partial functions on Person, as none has an entry for
the age of AJ.

31
The first map operator we introduce is called “override” (†). It joins together tow maps. If both maps have associations for the same Source element, then the second takes precedence. So, for example we have

\[
\{ AB \mapsto 39, PG \mapsto 34 \} \vdash \{ AB \mapsto 21 \} = \{ AB \mapsto 21, PG \mapsto 34 \}
\]

The override operator is associative, with identity θ and so forms a monoid:

\[
\text{Mon}(S \rightarrow T, \vdash, \theta)
\]

The generator set includes the empty map, and all (singleton) maplets.

We can now introduce the “domain” functions (dom) which returns a set of all the Source elements for which the map makes an association:

\[
\text{dom}\{ AB \mapsto 39, PG \mapsto 34 \} = \{ AB, PG \}
\]

This function is a homomorphism, so we can define it as

\[
\text{dom} : (S \rightarrow T, \vdash, \theta) \rightarrow (\text{PS}, \cup, \emptyset)
\]

\[
\text{dom}\{ s \mapsto t \} \equiv \{ s \}
\]

Remember, this captures the following properties

\[
\text{dom} \theta = \emptyset
\]

\[
\text{dom}(\mu_1 \vdash \mu_2) = \text{dom} \mu_1 \cup \text{dom} \mu_2
\]

9 Class 9

9.1 Maps and their Morphisms

We have introduced the notation

\[
\mu \in A \rightarrow B
\]

as introducing µ as a map or function from elements of A to elements of B. We don’t state at this point if the function is total, partial, finite or infinite.

We have the empty map (θ) which does not map any element of A to an element of B.

The first binary operator for combining maps is override (†), which has the effect of the second map overwriting the first. With the following generator set

\[
\{ \emptyset \} \cup \{ \{ a \mapsto b \} \mid a \in A, b \in B \}
\]

we can generate all the elements of the following monoid

\[
\text{Mon}(A \rightarrow B, \vdash, \theta)
\]

The following examples illustrate the behaviour of override:

\[
\theta \vdash \{ a \mapsto b \} = \{ a \mapsto b \}
\]

\[
\{ a_1 \mapsto b_1 \} \vdash \{ a_2 \mapsto b_2 \} = \{ a_1 \mapsto b_1, a_2 \mapsto b_2 \}
\]

\[
\{ a_1 \mapsto b_1, a_2 \mapsto b_2 \} \vdash \{ a_1 \mapsto b_3 \} = \{ a_1 \mapsto b_3, a_2 \mapsto b_2 \}
\]

Associate with maps we have a number of useful homomorphisms.
9.1.1 Domain

The homomorphism \( \text{dom} \) returns all the source elements which are mapped

\[
\text{dom} : \quad \{ \text{function from } A \to B \} \to \{ \text{set of all } a \in A \}
\]

\[
\text{dom} \{ a \mapsto b \} \equiv \{ a \}
\]

9.1.2 Map Removal

The endomorphism \( \triangledown(S) \) removes any mappings from elements of \( S \)

\[
\triangledown(S) : \quad \{ \text{function from } A \to B \} \to \{ \text{function from } A \to B \}
\]

\[
\triangledown(S) \{ a \mapsto b \} \equiv a \in S \to \{ a \mapsto b \}, \theta
\]

Example:

\[
\triangledown(\{ a_1, a_2 \}) \{ a_1 \mapsto b_1, a_2 \mapsto b_2, a_3 \mapsto b_3 \} = \{ a_3 \mapsto b_3 \}
\]

9.1.3 Map Restriction

The endomorphism \( \triangledown(S) \) keeps only mappings from elements of \( S \)

\[
\triangledown(S) : \quad \{ \text{function from } A \to B \} \to \{ \text{function from } A \to B \}
\]

\[
\triangledown(S) \{ a \mapsto b \} \equiv a \in S \to \{ a \mapsto b \}, \theta
\]

Example:

\[
\triangledown(\{ a_1, a_2 \}) \{ a_1 \mapsto b_1, a_2 \mapsto b_2, a_3 \mapsto b_3 \} = \{ a_1 \mapsto b_1, a_2 \mapsto b_2 \}
\]

Note that \( \triangledown \) and \( \triangledown \) have very similar definitions for sets, sequences and maps! They are homomorphisms from a monoid into itself, i.e endomorphisms.

9.1.4 Map Mapping

We can also map functions over maps — we need two functions, one to operate on the Domain, the other on the Range, so taking

\[
\{ a_1 \mapsto b_1, a_2 \mapsto b_2 \} \to \{ f(a_1) \mapsto g(b_1), f(a_2) \mapsto g(b_2) \}
\]

The mapping operator is written

\[
(f \mapsto g)
\]

following the tradition that mapping uses the same notation as is used to declare the type. We can define map mapping as a homomorphism:

\[
(f \mapsto g) : \quad \{ \text{function from } A \to B \} \to \{ \text{function from } C \to D \}
\]

\[
(f \mapsto g) \{ a \mapsto b \} \equiv \{ f(a) \mapsto g(b) \}
\]

For this to work, we need \( f : A \to C \) and \( g : B \to D \), and one other restriction. The function \( f \) must be injective, i.e, for all \( a_1, a_2 \) in \( A \), we have

\[
f(a_1) = f(a_2) \Rightarrow a_1 = a_2
\]
In other words, $f$ must map distinct values to distinct values.

We finish on mapping with the following table, which shows how the type notation is used in the value world as denoting the appropriate form of mapping:

<table>
<thead>
<tr>
<th>Type</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets</td>
<td>$\mathcal{P} A$</td>
</tr>
<tr>
<td>Sequences</td>
<td>$A^*$</td>
</tr>
<tr>
<td>Maps</td>
<td>$A \rightarrow B$</td>
</tr>
</tbody>
</table>

### 9.2 Map Model of Dictionary

We now turn our attention to building our model of a dictionary with definitions, using maps. This will lead us to applying what we know at present, as well as leading us to consider some non-homomorphic aspects of maps.

#### 9.2.1 Dictionary State

We can now state that our dictionary is a mapping from words to word definitions:

$$d \in \text{WordDef} = \text{Word}^*$$
\[\mu \in \text{Dict}_2 = \text{Word} \rightarrow \text{WordDef}\]

We will continue with the invariant which requires each word given a definition to have UK spelling, but we won’t extend this requirement to the actual definitions themselves.

$$i_{in-\text{Dict}_2} : \text{Dict}_2 \rightarrow \mathbb{B}$$
\[i_{in-\text{Dict}_2}(\mu) \triangleq \forall (\text{UK}) (\text{dom } \mu)\]

An example of the invariant in use is:

\[i_{in-\text{Dict}_2}\{\text{hello} \mapsto d_1, \text{color} \mapsto d_2\}\]
\[= \forall (\text{UK}) (\text{dom } \{\text{hello} \mapsto d_1, \text{color} \mapsto d_2\})\]
\[= \forall (\text{UK}) \{\text{hello, color}\}\]
\[= \text{FALSE}\]

Note that the invariant is itself a homomorphism, being the composition of same.

The initial state is simply the empty dictionary:

$$\mu_0 : \text{Dict}_2$$
\[\mu_0 \triangleq \emptyset\]

#### 9.2.2 Lookup Operation

We first define the normal spell-checker lookup, i.e. is the word in the dictionary?

$$\text{Lkp}_2 : \text{Word} \rightarrow \text{Dict}_2 \rightarrow \mathbb{B}$$
\[\text{Lkp}_2(w) \mu \triangleq w \in \text{dom } \mu\]
But we can do more — we are now in a position to lookup a word in a dictionary with a view to obtaining its definition. So, we introduce a new version of lookup ($\text{Lkp}_2'$), which returns the definition:

$$\text{Lkp}_2' : \text{Word} \rightarrow \text{Dict}_2 \rightarrow \text{WordDef}$$

$$\text{Lkp}_2'(w) \mu \equiv \mu(w)$$

Here we have treated the map as a function, and used function application notation, to denote the operation of actually using the map to “map” a given value.

So, $\mu(a)$ is Map Application — it looks up the mapping source for the value $a$ and returns the corresponding map target value. E.g:

$$\{a_1 \mapsto b_1, a_2 \mapsto b_2\}(a_1) = b_1$$

Map Application has type:

$$\{\_\} : (A \rightarrow B) \rightarrow A \rightarrow B$$

The problem with map application is that is is partial, is only defined when $a \in \text{dom } \mu$. So the following applications are undefined:

$$\{a_1 \mapsto b_1\}(a_2) \quad \text{and} \quad \theta(a)$$

(We sometimes denote an undefined result by $\bot$, often pronounced “bottom”).

The consequence of this is that the second form of lookup needs a precondition to ensure that the word is present:

$$\text{pre-Lkp}_2' : \text{Word} \rightarrow \text{Dict}_2 \rightarrow \text{B}$$

$$\text{pre-Lkp}_2'(w) \mu \equiv w \in \text{dom } \mu$$

Note that the precondition for the second form of lookup is effectively that the first form returns true!

Finally, observe that map application is not a homomorphism.

### 9.2.3 Insert Operation

We can define an operation to insert words as:

$$\text{Ins}_2 : \text{Word} \times \text{WordDef} \rightarrow \text{Dict}_2 \rightarrow \text{Dict}_2$$

$$\text{Ins}_2(w, d) \mu \equiv \mu \cup \{w \mapsto d\}$$

But this raises a variety of questions. Do we want to override an existing definition? Do we want to extend it instead? Should we be restricted to only inserting new words?

For now, we shall assume that we keep the restriction that the word must be new:

$$\text{pre-Ins}_2 : \text{Word} \times \text{WordDef} \rightarrow \text{Dict}_2 \rightarrow \text{Dict}_2$$

$$\text{pre-Ins}_2(w, s) \mu \equiv w \notin \text{dom } \mu$$

$$\text{Ins}_2(w, d) \mu \equiv \mu \cup \{w \mapsto d\}$$
9.2.4 Remove Operation

We remove a word and its definition:

\[
\text{Rem}_2 : \text{Word} \to \text{Dict}_2 \setminus \text{Dict}_2 \\
\text{Rem}_2(w)\mu \equiv \emptyset \{w\} \mu
\]

9.2.5 Joining Maps

The problem when combining maps occurs when the two maps have overlapping domains. So the main issue when combining \(\mu_1\) and \(\mu_2\) is what to do with the elements of \(\text{dom} \mu_1 \cap \text{dom} \mu_2\)?

The override operator is one approach — the second map takes precedence in the event of a clash

\[
(\mu_1 \updownarrow \mu_2)(a) = \mu_2(a) \text{ if } a \in \text{dom} \mu_2
\]

There are other possibilities.

Map Extend (\(\|\)). The expression \(\mu_1 \| \mu_2\) is only defined if \(\text{dom} \mu_1 \cap \text{dom} \mu_2 = \emptyset\) — this is analogous to set extend. This can be used to split maps for recursive definitions in the same way as extend is used for sets.

Map Glueing (\(\|\)). The expression \(\mu_1 \| \mu_2\) is only defined if for all \(a \in \text{dom} \mu_1 \cap \text{dom} \mu_2\) we have \(\mu_1(a) = \mu_2(a)\). This is defined only if the two maps agree on their common intersection.

Another approach is to resolve a conflict by combining the conflicting target values using an operator — we will address this approach later.

9.3 Exercises 5

Q5.1 Let

\[
\mu_1 = \{3 \mapsto "a", 4 \mapsto "z", 5 \mapsto "e" \} \\
\mu_2 = \{1 \mapsto "a", 2 \mapsto "x", 5 \mapsto "e" \}
\]

Evaluate the following:

(i) \(\text{dom}\{1 \mapsto "a", 2 \mapsto "b", 3 \mapsto "a" \}\)

(ii) \(\mu_1 \updownarrow \mu_2\)

(iii) \(\mu_2 \| \mu_1\)

(iv) \(\mu_2 \| \{3 \mapsto "a" \}\)

(v) \(\mu_1(4)\)

(vi) \(\mu_2(7)\)

(vii) \((I \to \text{next})\mu_1\) where

\[
\text{next} : \text{Char} \to \text{Char} \\
\text{next}"a" \equiv "b" \\
\text{next}"b" \equiv "c" \\
\vdots \\
\text{next}"z" \equiv "a"
\]

36
Q5.2 \( \text{inv-} \text{Dict}_2 \) is a homomorphism. What monoids are the source and target?

Q5.3 What can you conclude about

\[
\text{inv-} \text{Dict}_2(\mu_1 \uparrow \mu_2), \quad \text{inv-} \text{Dict}_2 \mu_1, \quad \text{inv-} \text{Dict}_2 \mu_2
\]

Q5.4 Using \( \theta \) as base case, and \( \mu \sqcup \{ a \mapsto b \} \) as the step case, give a recursive definition of \( \text{dom} \).

Q5.5 Give a recursive definition of map application.

10 Class 10

We now have three levels of dictionary:

\[
\begin{align*}
\text{Dict}_0 & \quad \overset{\text{P Word}}{\xrightarrow{\text{elems}}} \quad \text{Dict}_1 \\
& \quad \overset{\text{dom}}{\xrightarrow{\text{Word}}} \quad \text{Dict}_2 \\
& \quad \overset{\text{Word}^+}{\xrightarrow{\text{Word}}} \quad \text{Word} \rightarrow \text{WordDef}
\end{align*}
\]

We have kept the operations at each level as similar as possible: similar preconditions; similar arguments; and similar outcome.

For example, at all levels, the insert operation only inserts a new word (and definition, if relevant).

\[
\begin{align*}
\delta \in \text{P Word} & \quad \overset{\text{pre-Ins}_0(w)}{\rightarrow} \quad B \quad w \notin \delta \\
\sigma \in \text{Word}^+ & \quad \overset{\text{pre-Ins}_1(w)}{\rightarrow} \quad B \quad w \notin \text{elems} \sigma \\
\mu \in \text{Word} \rightarrow \text{WordDef} & \quad \overset{\text{pre-Ins}_2(w)}{\rightarrow} \quad B \quad w \notin \text{dom} \mu
\end{align*}
\]

If we want to replace an existing word, or modify a definition, then we define a new operation.

We observe that \( \text{Dict}_2 \) is an elaboration of \( \text{Dict}_0 \), because it allows new operations to be defined for the former model which have no counterpart in the latter.

10.1 Map Range

So far our main concern has been spell-checking, which is achieved by asking whether or not a word is in the dictionary. Now we turn our attention to the word definitions, i.e. the target elements of our map. We have seen map application which allows to extract the target of a given source value, provided
it is present in the map. Now we ask about getting a set of all the target values — a function that is to those values as \( \text{dom} \) is to the source values.

The function \( \text{rng} \) returns a set of all the target values in a given map:

\[
\text{rng} : (A \to B) \to \mathcal{P}B \\
\text{rng} \theta \overset{\equiv}{=} \emptyset \\
\text{rng} (\mu \cup \{a \mapsto b\}) \overset{\equiv}{=} \text{rng} \mu \cup \{b\}
\]

This is a recursive definition. Can we define it the same way we defined \( \text{dom} \)? As a homomorphism? Consider the following example

\[
\begin{align*}
\mu_1 &= \{a \mapsto b_1\} & \text{rng} \mu_1 &= \{b_1\} \\
\mu_2 &= \{a \mapsto b_2\} & \text{rng} \mu_2 &= \{b_2\} \\
\mu_1 \downarrow \mu_2 &= \{a \mapsto b_2\} & \text{rng}(\mu_1 \downarrow \mu_2) &= \{b_2\}
\end{align*}
\]

We see that, in general,

\[
\text{rng}(\mu_1 \downarrow \mu_2) \neq \text{rng} \mu_1 \cup \text{rng} \mu_2
\]

So \( \text{rng} \) is not a homomorphism!

### 10.2 Update Operation

We can introduce the notion of updating a word definition, by replacing it with a new one:

\[
\text{Update}_2 : \text{Word} \times \text{WordDef} \to \text{Dict}_2 \to \text{Dict}_2 \\
\text{Update}_2(w, d)\mu \overset{\equiv}{=} \mu \upharpoonright \{w \mapsto d\}
\]

This works whether or not the word is already present, so no pre-condition is required. It is interesting to note that, as override covers both cases, that we can in fact define override in terms of map removal and map extend:

\[
\mu_1 \upharpoonright \mu_2 = \mathsf{d}(\text{dom} \mu_2)\mu_1 \cup \mu_2
\]

Consider the example where

\[
\mu_1 = \{a_1 \mapsto b_1, a_2 \mapsto b_2, a_3 \mapsto b_3, a_4 \mapsto b_4\} \quad \mu_2 = \{a_2 \mapsto b'_2, a_4 \mapsto b'_4, a_5 \mapsto b_5\}
\]

We have \( \text{dom} \mu_2 = \{a_2, a_4, a_5\} \), and can visualise the overriding of \( \mu_1 \) by \( \mu_2 \) as:

\[
\begin{array}{cccccc}
\mu_1 & \mu'_1 & \mu_2 & \mu'_2 \\
\{a_1 \mapsto b_1\} & \{a_1 \mapsto b_1\} & \{a_2 \mapsto b_2\} & \{a_2 \mapsto b'_2\} \\
\{a_2 \mapsto b_2\} & \{a_2 \mapsto b'_2\} & \{a_3 \mapsto b_3\} & \{a_3 \mapsto b_3\} \\
\{a_3 \mapsto b_3\} & \{a_3 \mapsto b_3\} & \{a_1 \mapsto b_4\} & \{a_1 \mapsto b'_4\} \\
\{a_4 \mapsto b_4\} & \{a_4 \mapsto b'_4\} & \{a_5 \mapsto b_5\} & \{a_5 \mapsto b_5\}
\end{array}
\]

where \( \mu'_1 = \mathsf{d}\{a_2, a_4, a_5\}\mu_1 \).
10.3 Extend Operation

Another way to combine maps might be to extend dictionary definitions, by adding more detail to existing definitions. We shall also allow for adding a new definition if the key word is not present. So given a dictionary

\[ \{ w_1 \mapsto d_1, w_2 \mapsto d_2 \} \]

we can extend the definition associated with \( w_1 \) by adding material \( d' \) (say), by combining our dictionary with the maplet \( \{ w_1 \mapsto d' \} \) to get

\[ \{ w_1 \mapsto d_1 \setminus d', w_2 \mapsto d_2 \} \]

Basically we have concatenated the previous definition with the new material and updated the dictionary map to reflect the change. We can capture this process with the following definition:

\[
\text{Ext}_2 : \text{Word} \times \text{WordDef} \rightarrow \text{Dict}_2 \rightarrow \text{Dict}_2 \\
\text{Ext}_2(w, d)\mu \triangleq w \notin \text{dom } \mu \rightarrow \mu \cup \{ w \mapsto d \} \\
\text{for } w \in \text{dom } \mu \rightarrow \mu \upharpoonright \{ w \mapsto \mu(w) \setminus d \}
\]

We have two cases — the first handles new words, the second handles extending an already existent definition. The second case has a complex expression, which is worth examining in more detail. To extend a definition of word \( w \) with new material \( d \), when \( w \) is already in dictionary \( \mu \),

\[ \text{Ext}_2(w, d)\mu \text{ where } w \in \text{dom } \mu \]

we:

(i) lookup dictionary to get current definition \( (d_c) \) for word:

\[ d_c = \mu(w) \]

(ii) combine current and new definitions to give \( d' \):

\[ d = d_c \setminus d \]

(iii) update dictionary to refer to revised definition:

\[ \mu \upharpoonright \{ w \mapsto d' \} \]

By substituting for \( d' \) and \( d_c \) in the last expression we obtain

\[ \mu \upharpoonright \{ w \mapsto \mu(w) \setminus d \} \]

— the form in the official definition of the operator.

Note, we could have defined this case using the lookup operator

\[ \mu \upharpoonright \{ w \mapsto \text{Lkp}_2(w)\mu \setminus d \} \]

In general however, in almost all proofs involving this, we would replace the operator by its definition i.e. map application. It is sometimes useful however, when trying to structure large and complex models
11 Class 11

We shall now introduce a completely new model of a new system, in order to
- explore maps even further.
- develop a better feel for the method and the modelling

11.1 Health Informatics

We are going to model aspects of a health informatics system that might include

- Hospital Management
- Medical Records
- Doctor Patient Relationship
- Laboratory Test Results

to name but a few.

What can we model, and why ?

We could model the entire system (or interesting parts) in order to understand the system more clearly. This is an activity often referred to as “Domain Modelling”. We could model Requirements, in order to understand what the User wants from the “system”. We could model a Specification in order to understand and describe how the “system” should behave.

As far as requirements and specification are concerned, the term “system” usually refers to the computing system which we are planning to construct.

We are going to an exercise in Domain Modelling in the area of Health Informatics. In order to do this we need to identify:

- key entities, things (nouns).
- key relationships between entities.
- key events changes, operations, actions in system (verbs).

11.2 Patients and Doctors

So to begin we might consider two possible entities in health informatics to be

\[
\begin{align*}
\text{Patients} & \quad p \in \text{Patient} \\
\text{Doctors} & \quad d \in \text{Doctor}
\end{align*}
\]

Some questions about the entities and their relationships immediately arise:

Can a doctor be a patient ? Clearly the answer to this is yes. We might express this as:

\[ \text{Doctor} \subseteq \text{Patient} \]

However, this implies that every doctor is a patient ? Is this true ?
What has happened here is that we have raised an issue of interpretation here. Does the type \textit{Patient} include all people, including actual patients as well as healthy (non-patient) people. \textit{There is no correct answer! We need to decide!} For this exercise, we shall decide that \textit{Patients} can be any person, whether of not they are ill. The type \textit{Patient} is the set of actual and potential patients. So, any doctor is a potential patient \textit{Doctor} \subseteq \textit{Patient}

A doctor may have registered patients. If someone is ill, they will go to a doctor, usually a specific one, known as a \textit{General Practitioner} (GP). We can capture this with a predicate

\[
\text{isGP : Doctor} \rightarrow \mathbb{B} \quad \text{returns true if doctor is GP}
\]

or by introducing special type

\[
g \in \text{GP} \subset \text{Doctor}
\]

We then have \text{isGP}(g) = \text{TRUE} and \text{isGP}(d) = d \in \text{GP}

We shall assume that a patient registers with one GP. We will associate with each GP a set of registered patients. We record the registration information in a map:

\[
r \in \text{Registry} = \text{GP} \rightarrow \mathcal{P}\text{Patient}
\]

So, if GP \text{g}_1 \text{ has patients } \text{p}_1, \text{p}_2; \text{GP g}_2 \text{ has patients } \text{p}_3, \text{p}_4; \text{and GP g}_3 \text{ has no patients, we could record this as:}

\[
\rho_1 = \left\{ \begin{array}{c}
g_1 \mapsto \{p_1, p_2\} \\
g_2 \mapsto \{p_3, p_4\} \\
g_3 \mapsto \emptyset
\end{array} \right\}
\]

Who are all the GPs in \rho_1? Answer:

\[
\text{dom } \rho_1 = \{g_1, g_2, g_3\}
\]

Who are all the registered patients? Answer:

\[
\text{rng } \rho_1 = \left\{ \begin{array}{c}
\{p_1, p_2\} \\
\{p_3, p_4\} \\
\emptyset
\end{array} \right\}
\]

\[
(\cup \circ \text{rng})\rho_1 = \{p_1, p_2, p_3, p_4\}
\]

11.3 \textbf{Patient-GP Actions}

What actions can we think of?

- A patient registering with a GP
- A new GP entering the system
- A patient changing their GP.
- A GP retiring (or being struck off!)

What happens to existing patients of that GP? What happens in reality?

What would we prefer to happen?
The last item above raises an important issue. Are we modelling reality as it is, or how we would like things to be, or the result of changes which we intend to bring about? Again, we need to decide on the scope of our model. This model here is being used to illustrate mathematics, method and modelling, so we will not concern ourselves too much with the answer to these questions here. In a “real-world” situation, these issues need to be clearly addressed.

11.3.1 Registering with a GP

We can define an operation that takes a GP and patient and updates a GP registry:

\[
\text{Reg}_0 : \text{GP} \times \text{Patient} \rightarrow \text{Registry} \rightarrow \text{Registry}
\]

\[
\text{Reg}_0(g, p) \rho = \rho \upharpoonright \{g \mapsto \rho(g) \cup \{p\}\}
\]

This operator is not defined if \( g \notin \text{dom} \rho \). Apart from this, we have a choice for what this operator is meant to do, if the GP is not already registered. Do we add the GP in as well as the patient, or do we require that the GP must already be present?

We shall decide that the GP must be already present in the system before a patient can register:

\[
\text{pre-Reg}_0(g, p) \rho \equiv g \in \text{dom} \rho
\]

One reason is that we might expect the actions of adding in a new GP to the system, and registering a patient with a GP really ought to be distinct actions. The fact that we could come up with a mathematical expression that does both if necessary is beside the point. The mixing of the two actions would not occur in real-life so we want to keep them conceptually apart in the model.

Another observation to make is that, given the definition above of \( \text{Reg}_0 \), we are forced to that particular pre-condition in order to ensure that the definition is well-defined.

\( g \in \text{dom} \rho \) because of sub-expression \( \rho(g) \)

In some sense it is possible for us to compute a precondition for a given expression, being the minimal conditions under which that expression is well-defined. Often however, we may strengthen the precondition for conceptual or modelling reasons. In this example both technical and conceptual considerations give the same result.

11.3.2 Adding a New GP

So, we need to have an operation explicitly for adding in new GPs to our system. Again, we might simply go ahead and posit:

\[
\text{New}_0 : \text{GP} \rightarrow \text{Registry} \rightarrow \text{Registry}
\]

\[
\text{New}_0(g) \rho = \rho \cup \{g \mapsto \emptyset\}
\]

This requires us to stipulate that \( g \notin \text{dom} \rho \).

Alternatively we might have chosen:

\[
\text{New}_0(g) \rho \equiv \rho \upharpoonright \{g \mapsto \emptyset\}
\]

42
This has no technical requirement for a pre-condition. However, what happens in this case if \( g \) was in \( \rho \) before we did the new operation?

\[
\text{New}_0(g) \{ \ldots, g \mapsto P, \ldots \} = \{ \ldots, g \mapsto P, \ldots \} \uplus \{ g \mapsto \emptyset \} = \{ \ldots, g \mapsto \emptyset, \ldots \}
\]

Effectively by doing “\text{New}” for an existing GP we have unregistered all their patients! For our model to match the real world, we shall insist that the new operation is only used for GPs not already in the system:

\[
\text{pre-New}_0(g) \rho \equiv g \notin \text{dom } \rho
\]

We shall leave the operations of changing ones GP, and a GPs retirement until later. All of these operations have involved changes to the state of the system. What about some inquiries on an given system?

### 11.3.3 Patient Count

How many patients are in the system? We have seen that all the patients in the system are given by

\[
(\cup / \circ \text{rng}) \rho
\]

We simply need to count them:

\[
\#(\cup / \circ \text{rng}) \rho
\]

This leads us to the following query:

\[
\begin{align*}
\text{PatientCount}_0 : \text{Registry} & \to \mathbb{N} \\
\text{PatientCount}_0 \rho & \equiv \#(\cup / \circ \text{rng}) \rho
\end{align*}
\]

### 11.3.4 Patients per GP

A more useful result for planning and allocation purposes might be how many patients does each GP have?

\[
\text{PatientsPerGP}_0 : \text{Registry} \to (\text{GP} \to \mathbb{N})
\]

We want to know for each GP how many patients are involved, hence the mapping from GPs to numbers. Basically we want to replace the set of patients in the registry by the size of that set. We want to transform

\[
\rho_2 = \left\{ \begin{array}{ll}
g_1 & \mapsto \{ p_1, p_2 \} \\
g_2 & \mapsto \{ p_3 \} \\
g_3 & \mapsto \{ p_4, p_5 \} \\
\end{array} \right\}
\]

\[
\kappa_2 = \left\{ \begin{array}{ll}
g_1 & \mapsto 2 \\
g_2 & \mapsto 1 \\
g_3 & \mapsto 2 \\
\end{array} \right\}
\]

We can do this by mapping set cardinality across the range:

\[
\kappa_2 = (\text{Id} \to \#) \rho_2
\]

So we define

\[
\text{PatientsPerGP}_0(\rho) \equiv (\text{Id} \to \#) \rho
\]

43
In general, a map $A \rightarrow N$ is called a "bag" or "multi-set". The bag
\[ \{a_1 \mapsto 2, a_2 \mapsto 1, a_3 \mapsto 3\} \]
can be viewed as the multi-set:
\[ \{a_1, a_2, a_3, a_3\} \]

We need to consider some bag operations.

12 Class 12

12.1 Registry Merge

Imagine bringing two different health administrative systems together — combining the corresponding registries. Assume that some doctors are registered separately in both zones, so we cannot simply extend or glue maps together.

We need a registry merging operator:

$$\text{Mr}_0 : \text{Registry} \times \text{Registry} \rightarrow \text{Registry}$$

Doctors may be in both registries, with the same or different patients:

\[
\begin{align*}
\text{Mr}_0 : & \quad \{ g_1 \mapsto \{p_1\}, \}
\quad \{ g_2 \mapsto \{p_2\} \} \\
\text{Mr}_0 : & \quad \{ g_1 \mapsto \{p_3\}, \}
\quad \{ g_2 \mapsto \{p_4, p_2\} \}
\end{align*}
\]

We can give a recursive definition. Merging a registry 1 with an empty one gives registry one unchanged

$$\text{Mr}_0(p_1) \mid \emptyset \cong p_1$$

Merging registry 1 with registry 2 extended by a doctor patient maplet is merging and updated form for registry 1 with registry 2:

$$\text{Mr}_0(p_1) \mid (p_2 \cup \{g \mapsto P\}) \cong \text{Mr}_0(p')((p_2)$$

The updated form of registry 1 is either an extension by the maplet if not present, or an override, with the relevant range sets joined by set union:

\[
\rho' = \begin{cases}
\forall g \notin \text{dom} \rho_1 \rightarrow \rho_1 \cup \{g \mapsto P\} \\
\forall g \in \text{dom} \rho_1 \rightarrow \rho_1 \uparrow \{g \mapsto \rho_1(g) \cup P\}
\end{cases}
\]
12.2 Registry Add

At this point, it is instructive to ask what happens to the number of patients per doctor? In other words, how is

\[ \text{PatientsPerGP}(\text{Mrg}_0(p_1) \cup (p_2)) \]

related to

\[ \text{PatientsPerGP}(p_1) \quad \text{and} \quad \text{PatientsPerGP}(p_2) \quad ? \]

If we take an example:

\[
\begin{align*}
\{ g_1 &\mapsto \{ p_1 \}, \\
g_2 &\mapsto \{ p_2 \} \}
\end{align*}
\]

\[
\begin{align*}
\{ g_2 &\mapsto \{ p_3 \}, \\
g_3 &\mapsto \{ p_4 \} \}
\end{align*}
\]

The corresponding counts are represented as “bags” (maps from some set into the natural numbers).

\[ \beta \in \text{Bag}_A = A \rightarrow \mathbb{N} \]

(Often the type \( A \) is clear from context, so we omit it as a suffix of Bag.) Bags can be combined using bag addition (BagAdd or \( \oplus \)). Our example becomes:

\[
\begin{align*}
\{ g_1 &\mapsto 1, \\
g_2 &\mapsto 1 \}
\end{align*}
\]

\[
\begin{align*}
\{ g_2 &\mapsto 1, \\
g_3 &\mapsto 1 \}
\end{align*}
\]

We can give a recursive definition of bag addition as:

\[
\begin{align*}
\text{Bag} \times \text{Bag} &\rightarrow \text{Bag} \\
\beta_1 \oplus \theta &\equiv \beta_1 \\
\beta_1 \oplus (\beta_2 \cup \{ a \mapsto n \}) &\equiv \beta \oplus \beta_2 \\
\text{where} \\
\beta &\equiv a \nexists \beta_1 \rightarrow \beta_1 \cup \{ a \mapsto n \} \\
a \in \beta_1 \rightarrow \beta_1 \uparrow \{ a \mapsto \beta_1(a) + n \}
\end{align*}
\]
12.3 A Common Theme: Resolution

In both previous cases, we are combining two maps, and resolving conflicts by using a binary operator to combine the conflicting target values.

\[ \{a \mapsto n_1\} \oplus \{a \mapsto n_2\} = \{a \mapsto n_1 + n_2\} \]

\[ \text{Mrg}_0(\{g \mapsto P_1\})(\{g \mapsto P_2\}) = \{g \mapsto P_1 \cup P_2\} \]

We might be tempted to introduce the following notation: \( \text{Mrg}_0 = \Theta \).

12.4 Indexing a Binary Operator

Let \( * \) be a binary operation \( * : A \times A \rightarrow A \).

Let us build maps with \( A \) as target. We call the source set (here called \( Y \)), the “index set”.

\[ Y \rightarrow A \quad \text{“elements of } A \text{ indexed by elements of } Y \text{”} \]

We can define a binary operator combining such maps in terms of \( * \) — we call it \( \odot \), “circled-\( * \)” or “indexed-\( * \)”.

\[ \{y \mapsto a_1\} \odot \{y \mapsto a_2\} = \{y \mapsto a_1 * a_2\} \]

i.e. \( \odot \) applies \( * \) to the target or “backend” of the map. Sometimes, \( \odot \) is referred to as “pointwise extension of \( * : A \times A \rightarrow A \) to \( (Y \rightarrow A) \times (Y \rightarrow A) \rightarrow (Y \rightarrow A) \)”.

Many textbooks use the same symbol to denote an operator and its pointwise extension.

We can now define the registry merge operation as

\[ \text{Mrg}_0(p_1)(p_2) = p_1 \odot p_2 \]

12.5 Properties of Indexing

Indexing preserves important properties:

- if \( * \) is commutative, then so is \( \odot \).
- if \( * \) is associative, then so is \( \odot \).

The null map \( \theta \) is the identity element for \( \odot \), no matter what \( * \) is.

We can summarise some of this by saying that \((Y \rightarrow A, \odot, \theta)\) is a monoid, if \( * \) is associative, or alternatively, if \((A, *)\) is a semi-group.

If \( h : (A, *, e) \rightarrow (B, \times, e') \) is a monoid homomorphism, then

\[ (\mathcal{I} \rightarrow h) : (Y \rightarrow A, \odot, \theta) \rightarrow (Y \rightarrow B, \otimes, \theta) \]

is also a monoid homomorphism.

An interesting question arises concerning groups. Given group \((A, *, e, e^{-1})\) is

\[ (Y \rightarrow A, \odot, \theta, \tau) \]
a group — if so what is the inverse function?

For a concrete example consider the integer group

\[(\mathbb{Z}, +, 0, -)\]

Is the following a group?

\[(Y \to \mathbb{Z}, \oplus, \theta, \Box)\]

13 Class 13

We consider trying to index the group \((\mathbb{Z}, +, 0, -)\), using mapping of unary minus over the range as a putative inverse:

\[(\mathbb{Z}, +, 0, -) \quad \sim \quad (Y \to \mathbb{Z}, \oplus, \theta, (I \to (-)) \]

We can try a simple example, by first taking a possible inverse:

\[(I \to (-)) \{y \mapsto 3\} = \{y \mapsto -3\}\]

and then indexed-adding it to the original value:

\[\{y \mapsto 3\} \oplus \{y \mapsto -3\} = \{y \mapsto 0\} \neq \theta\]

We don’t obtain the null map — instead we end up with a map with the identity element in the range. Unless we somehow view these as “null” maplets, we do not have proper inverses.

Rather than define an equivalence class of null maps

\[\text{nullmap} = \{\theta, \{y \mapsto 0\}, \{y_i \mapsto 0, y_j \mapsto 0\}, \ldots\}\]

we prefer to provide an alternative form of indexing, on which strips out identity elements from the range.

Given \(A, \star\) and \(e\), we come up with a form of indexing, which uses maps of the form

\[Y \to A' \quad \text{where} \quad A' = A \setminus \{e\}\]

and an indexed operator \(\Box'\) modified to discard any maplets whose range is \(e\).

This form of “identity-forgetting” indexing can be used to transform monoids to monoids:

\[(A, \star, e) \quad \sim \quad (Y \to A', \Box', \theta)\]

and groups to groups:

\[(A, \star, e, \star^{-1}) \text{ quad } \sim \quad (Y \to A', \Box', \theta, (I \to \star^{-1}))\]

While not necessary for indexing monoids, it is often useful. For example, if we index sets under union this way:

\[(\mathcal{P}A, \cup, \emptyset) \quad \sim \quad (Y \to \mathcal{P}A, \cup, \theta)\]

where \(\mathcal{P}A = \mathcal{P}A \setminus \{\emptyset\}\), we get a structure which is isomorphic to the usual model of relations:

\[Y \to \mathcal{P}A \quad \cong \quad \mathcal{P}(Y \times A)\]
13.1 Healthcare Elaboration: Consultants

...missing material to introduce the model...

[BEGIN Reconstruction...

We introduce consultants \( c \in \text{Consult} \subset \text{Doctor} \) who have patients referred to them by GPs. A doctor cannot be both a GP and consultant:

\[
GP \cap \text{Consult} = \emptyset
\]

We have a corresponding mapping \( \kappa \in \text{Clients} = \text{Consult} \to \text{Patient} \).

Our health registry now has two maps --- the original GP registry, and a new consultant clients register:

\[
(\rho, \kappa), \Sigma \in S_{\text{sys}_0} = \text{Registry} \times \text{Clients}
\]

There is an invariant on systems, which is that every client of a consultant must also be registered with a GP:

\[
\text{inv}_{S_{\text{sys}_0}}(\rho, \kappa) \equiv \left( \left( \rho \cap \text{rng} \right) \cup \left( \rho \cap \text{rng} \right) \right) \subseteq \left( \left( \rho \cap \text{rng} \right) \cup \left( \rho \cap \text{rng} \right) \right)
\]

...Reconstruction END]

13.2 Initial State of System

The initial healthcare system has no GPs or consultants registered:

\[
\begin{align*}
\Sigma_0 & : \quad S_{\text{sys}_0} \\
\Sigma_0 & \equiv (\emptyset, \emptyset)
\end{align*}
\]

We need to show that the initial state satisfies the invariant:

Proof Obligation: \( \text{inv}_{S_{\text{sys}_0}}(\Sigma_0) = \text{TRUE} \)

13.3 Incorporating Registry Operations

We can extend all our registry operation to the new system in an obvious way. Consider the operation to register a new GP:

\[
\begin{align*}
\text{New}_0 & : \quad GP \to \text{Registry} \to \text{Registry} \\
\text{New}_0(\rho) & \equiv \rho \cup \{ g \mapsto \emptyset \}
\end{align*}
\]

with a precondition stating that the GP is not already registered. The corresponding system version is:

\[
\begin{align*}
\text{New}_0 & : \quad GP \to S_{\text{sys}_0} \to S_{\text{sys}_0} \\
\text{New}_0(\rho, \kappa) & \equiv (\rho \cup \{ g \mapsto \emptyset \}, \kappa)
\end{align*}
\]

We have to show that this operation preserves the invariant:

Proof Obligation: \( \text{inv}_{S_{\text{sys}_0}} \land \text{pre-New}_0(\rho) \Rightarrow \text{inv}_{S_{\text{sys}_0}}(\text{New}_0(\rho)) \)

We can extend operations \( \text{Reg}_0, \text{Mrg}_0 \) to systems in a similar manner, and provide a definition for operation \( \text{New}_C \) to add a consultant.
13.4 Consultant Referrals

A GP may refer a patient to a consultant, an operation whose outcome is to update and extend the Clients mapping:

\[
\text{Ref}_0 : \text{Patient} \times \text{Consult} \rightarrow \text{Sys}_0 \rightarrow \text{Sys}_0
\]

\[
\text{Ref}_0(p, c)(\rho, \kappa) \triangleq (\rho, \kappa \downarrow \{c \mapsto \{p\}\})
\]

At the very least, we expect a pre-condition that the consultant already is present:

\[
\text{pre-Ref}_0(p, c)(\rho, \kappa) \triangleq c \in \text{dom} \kappa
\]

Note that this pre-condition is not required for technical reasons, as the \( \downarrow \) operator is total. However, we might wish to use it for conceptual reasons as already discussed. (Is there more? Hint: consider the system invariant!)

Proof Obligation: \( \text{inv-Sys}_0 \Sigma \land \text{pre-Ref}_0(p, c) \Sigma \Rightarrow \text{inv-Sys}_0(\text{Ref}_0(p, c) \Sigma) \)

We shall attempt to prove this obligation. Because the operation is well-defined regardless of whether or not \( c \in \text{dom} \kappa \), we shall ignore the pre-condition for now. We are going to assume

\[
\text{inv-Sys}_0(p, \kappa)
\]

\[
= (\downarrow / \circ \text{rng}|\kappa \subseteq (\downarrow / \circ \text{rng}|\rho)
\]

and try to prove that

\[
\text{inv-Sys}_0(\text{Ref}_0(p, c)(\rho, \kappa))
\]

\[
= \text{inv-Sys}_0(\text{Ref}_0(p, c)(\rho, \kappa \downarrow \{c \mapsto \{p\}\}))
\]

\[
= (\downarrow / \circ \text{rng}|(\kappa \downarrow \{c \mapsto \{p\}\}) \subseteq (\downarrow / \circ \text{rng}|\rho)
\]

A technical problem is that, whereas \( \downarrow \) is a monoid operator, \( \text{rng} \) is not a monoid homomorphism, so we cannot simply apply simply homomorphism laws.

We need to do a detailed analysis, and the obvious is to do a case-split, between \( c \notin \text{dom} \kappa \) and \( c \in \text{dom} \kappa \). We shall prove it case separately, using the cases as assumptions. If true in both cases, then it must be true overall.

13.5 Proof Principle: Case Analysis

**Case Analysis** is used when a proof is easier to handle by considering a distinct sets of possibilities. The cases must be exhaustive, covering all possibilities. If we have \( n \) cases, described by predicates \( \text{case}_1, \text{case}_2 \ldots \text{case}_n \), then we require that

\[
\text{case}_1 \lor \text{case}_2 \lor \ldots \lor \text{case}_n
\]

The proof principle we use is the following:

\[
\begin{align*}
\text{case}_1 \lor \text{case}_2 \lor \ldots \lor \text{case}_n & \land (\text{case}_1 \Rightarrow P = \text{TRUE}) \\
& \land (\text{case}_2 \Rightarrow P = \text{TRUE}) \\
& \ldots \\
& \land (\text{case}_n \Rightarrow P = \text{TRUE}) \\
\Rightarrow P = \text{TRUE}
\end{align*}
\]

49
where $P$ is the overall property we are trying to prove.
Case analysis is most often used to reason in the presence of conditional expressions. The cases correspond to the condition being \textit{FALSE} and \textit{TRUE}, for a simply binary condition.

13.6 Proof Attempt

Assuming

\[(\mathcal{L} \circ \text{rng}) \kappa \subseteq (\mathcal{L} \circ \text{rng}) \rho\]

we are trying to show

\[(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho\]

13.6.1 Case 1 — \textit{c not in} $\kappa$

\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{given } c \notin \text{dom } \kappa \text{ and defn. of } \cup)
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{defn. of } \circ)
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{defn. of defn. of } \circ)
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{is a homomorphism})
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{c \mapsto \{p\}\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{defn. of } \mathcal{L})
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{p\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (X \cup Y \subseteq Z = X \subseteq Z \wedge Y \subseteq Z)
\]
\[
(\mathcal{L} \circ \text{rng})(\kappa \cup \{p\}) \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{Hypothesis})
\]
\[
\text{TRUE} \wedge \{p\} \subseteq (\mathcal{L} \circ \text{rng}) \rho
\]
\[
= (\text{Boolean Algebra, defn. of } \subseteq)
\]
\[
p \in (\mathcal{L} \circ \text{rng}) \rho
\]

We find that we can only complete our proof if we can show that

\[
p \in (\mathcal{L} \circ \text{rng}) \rho
\]

What is this saying? \((\mathcal{L} \circ \text{rng}) \rho\) is the set of all patients registered with GPs. In effect we have discovered another precondition, namely that in order to refer a patient to a consultant, that patient must be registered with a GP (presumably the one doing the referral).

This is an example of using a proof to deduce the need for extra material in the invariant.

13.6.2 Case 2 — \textit{c in} $\kappa$

This case is very similar and gives exactly the same outcome.
13.6.3 A Conceptual Note

Conceptually speaking, the referral operation as presented is somewhat flawed, as the discovery of the extra precondition pointed out. Perhaps the operator should have the referring GP as a parameter, even though its value is not used in the definition of the operator, it would useful for a conceptual precondition which states that not only must a patient be registered with a GP in order to be referred, but they must also be referred by the actual GP with whom they are registered.

13.7 Exercises 6

Q6.1 Prove $\Sigma_0$ satisfies the invariant $(\Sigma_0 = (\theta, \theta))$, i.e. that

$$\text{inv-S}_{\Sigma_0}(\Sigma_0) = \text{true}$$

Q6.2 Write down the precondition for New$_0$ on $S_{\Sigma_0}$, stating that the new GP cannot already be present in the system

Q6.3 Prove that

$$\text{New}_0 : GP \rightarrow S_{\Sigma_0} \rightarrow S_{\Sigma_0}$$

preserves the invariant

A Solutions to Exercises

A.1 Exercises 1

Q1.1

- \{borrow\}
- \{borrow, curry\}
- no change as precondition not satisfied (isUK(color = false))
- \{borrow, curry, colour\}
- no change as precondition not satisfied $\neg(borrow \notin \{borrow, curry, colour\})$

Q1.2

- \{curry, colour\}
- no change as word ask not present, \{curry,colour\})

Q1.3

$$\text{pre-Rem}_0(w) \delta \equiv w \in \delta$$

(Q1.4 Optional) To be proved:

$$\text{inv-Dict}_0(\delta) \wedge \text{pre-Ins}_0(w) \delta \Rightarrow \text{inv-Dict}_0(\text{Ins}_0(w) \delta)$$

51
Expanding all definitions requires us to prove
\[ \forall (\text{isUK})(\delta \cup \{w\}) \]
given that we can assume
\[ \forall (\text{isUK})\delta \text{ and } \text{isUK}(w) \text{ and } w \notin \delta \]
The proof:
\[
\begin{align*}
& \forall (\text{isUK})(\delta \cup \{w\}) \\
& = \text{(obvious property of } \forall \text{)} \\
& \forall (\text{isUK})\delta \land \text{isUK}(w) \\
& = \text{(precondition assumption)} \\
& \forall (\text{isUK})\delta \land \text{TRUE} \\
& = \text{(boolean algebra)} \\
& \forall (\text{isUK})\delta \\
& = \text{(invariant assumption)} \\
& \text{TRUE}
\end{align*}
\]

A.2 Exercises 2
Q2.1
\[
\forall \text{s} \in \{-2, 2, 4\}
= \forall \text{s} \in \{2, 4\} \cup \{\text{sgn}(-2)\}
= \forall \text{s} \in \{2, 4\} \cup \{-1\}
= \forall \text{s} \in \{4\} \cup \{\text{sgn}(2)\} \cup \{-1\}
= \forall \text{s} \in \{4\} \cup \{+1\} \cup \{-1\}
= \forall \text{s} \in \emptyset \cup \{+1\} \cup \{+1\} \cup \{-1\}
= \{+1, -1\}
\]
Q2.2 \(\forall S\text{ computes the product of all the numbers in the set } S\)
Q2.3
\[
\forall (\forall \text{Odd})\{2, 3, 1, 4\}
= \forall (\forall \text{Odd}(\{2, 3, 1, 4\}))
= \forall (\{\text{FALSE}\} \cup \{\text{TRUE}\} \cup \{\text{TRUE}\} \cup \{\text{FALSE}\})
= \forall \{\text{FALSE}, \text{TRUE}\}
= \text{FALSE} \lor \text{TRUE}
= \text{TRUE}
\]
Q2.4 (i) \text{TRUE} (i) \text{FALSE}
Q2.5
\[
\begin{align*}
\forall \{a\} & \equiv a \\
\forall (S \cup \{a\}) & \equiv (\forall S) \ast a
\end{align*}
\]
Q2.6

\[
\begin{align*}
\neg \{1, 3, 7\}
&= 1 - (3 - 7) = +5 \\
&= 1 - (7 - 3) = -3 \\
&= 3 - (1 - 7) = +9 \\
&= 3 - (7 - 1) = -3 \\
&= 7 - (1 - 3) = +9 \\
&= 7 - (3 - 1) = +5 \\
&= (1 - 3) - 7 = -9 \\
&= (1 - 7) - 3 = -9 \\
&= (3 - 1) - 7 = -5 \\
&= (3 - 7) - 1 = -5 \\
&= (7 - 1) - 3 = +3 \\
&= (7 - 3) - 1 = +3
\end{align*}
\]

The problem is that subtraction is neither associative or commutative, so the order in which reduction occurs is important. However, in set reduction, we cannot assume that reduction occurs in any given order.

Q2.7 0 is the identity element for +:

\[
0 + a = a = a + 0
\]

and 1 is the identity element for \( \times \):

\[
1 \times a = a = a \times 1
\]

A.3 Exercises 3

Q3.1 By induction over \( S \):

\[
p(S) \equiv \#(S \uplus T) = \#S + \#T
\]

Base Case \( p(\emptyset) \):

\[
\#(\emptyset \uplus T) = \#\emptyset + \#T
\]

Simplify lhs:

\[
\#(\emptyset \uplus T) = \#T
\]

Simplify rhs:

\[
\#(\emptyset \uplus T) = \#T
\]

53
Inductive Step Case \( p(S \cup \{x\}) \)

\[ \#((S \cup \{x\}) \cup T) = \#(S \cup \{x\}) + \#T \]

Assuming, for fixed \( S \), arbitrary \( T \), that:

\[ \#(S \cup T) = \#S + \#T \]

Simplify lhs:

\[ \#((S \cup \{x\}) \cup T) \]

= (extra law)

\[ \#(\{x\} \cup (S \cup T)) \]

= (defn. of \#)

\[ 1 + \#(S \cup T) \]

Simplify rhs:

\[ \#(S \cup \{x\}) + \#T \]

= (defn of \#, associativity of +)

\[ 1 + \#S + \#T \]

= (assumption, rhs \( \mapsto \) lhs)

\[ 1 + \#(S \cup T) \]

Q3.2 Prove

\[ \text{elems}(\sigma \leftarrow \tau) = \text{elems} \sigma \cup \text{elems} \tau \]

By induction over \( \sigma \):

\[ p(\sigma) \equiv \text{elems}(\sigma \leftarrow \tau) = \text{elems} \sigma \cup \text{elems} \tau \]

Base Case \( p(\Lambda) \)

\[ \text{elems}(\Lambda \leftarrow \tau) = \text{elems} \Lambda \cup \text{elems} \tau \]

Simplify lhs:

\[ \text{elems}(\Lambda \leftarrow \tau) \]

= (defn. of \( \leftarrow \))

\[ \text{elems} \tau \]

Simplify rhs:

\[ \text{elems} \Lambda \cup \text{elems} \tau \]

= (defn. of \text{elems})

\[ \emptyset \cup \text{elems} \tau \]

= (set property)

\[ \text{elems} \tau \]

Inductive Step Case \( p(x : \sigma) \)

\[ \text{elems}(x : \sigma \leftarrow \tau) = \text{elems}(x : \sigma) \cup \text{elems} \tau \]

Assuming, for fixed \( \sigma \), arbitrary \( \tau \), that:

\[ \text{elems}(\sigma \leftarrow \tau) = \text{elems} \sigma \cup \text{elems} \tau \]

Simplify lhs:
\[ \text{elems}(x : \sigma \setminus \tau) \]
\[ = \text{(defn. of } \setminus \text{)} \]
\[ \text{elems}(x : (\sigma \setminus \tau)) \]
\[ = \text{(defn. of } \text{elems)} \]
\[ \{x\} \cup \text{elems}(\sigma \setminus \tau) \]

Simplify rhs:

\[ \text{elems}(x : \sigma) \cup \text{elems}\tau \]
\[ = \text{(defn of } \text{elems, associativity of } \cup \text{)} \]
\[ \{x\} \cup \text{elems } \sigma \cup \text{elems}\tau \]
\[ = \text{(assumption, rhs } \Rightarrow \text{ lhs)} \]
\[ \{x\} \cup \text{elems}(\sigma \setminus \tau) \]

Q.E.D.

A.4 Exercises 4

Q4.1 The function \( h : A \rightarrow B \) is a Group Homomorphism from \( (A, \ast, e, \ast^{-1}) \) to \( (B, \odot, e', \odot^{-1}) \) iff

- \( h(a_1 \ast a_2) = h(a_1) \odot h(a_2) \), for all \( a_1, a_2 \in A \).
- \( h(e) = e' \)
- \( h(a^{-1}) = -h(a) \), for all \( a \in A \).

Q4.2 No, because \( (i) \) the source structure is not a group \( (0^{-1} \) is not defined), and \( (ii) \), \( \log \) is not defined on zero or negative numbers. To get a homomorphism, we must restrict the carrier set of the first structure to all positive, non-zero reals:

\( \log : (\mathbb{R}_{\geq 0}, \times, 1, \times^{-1}) \rightarrow (\mathbb{R}, +, 0, -) \)

Q4.3 No

\( 4 \{a\} \{\{a, b\} \cup \{a, c\}\} = 4 \{a\} \{a, b, c\} \)
\[ = 4 \{a\} \{a, b, c\} \]
\[ = \{b, c\} \neq \emptyset = \]

Q4.4 No

\( \varphi \{a\} \{a \cap \{b\}\} = \varphi \{a\} \{a\} \cup \varphi \{a\} \{b\} \)
\[ = \varphi \{a\} \emptyset \]
\[ = \emptyset \neq \{a\} \]

Q4.5 The removal isn’t because the same counterexample as in Q4.3 will apply. The restriction isn’t because:

\( \varphi \emptyset = \emptyset \neq A \)

(Q4.6 Optional) No. Structure \( (P.A, \cap, \emptyset) \) is not a monoid, as \( \emptyset \) is not an identity for \( \cap \).
A.5 Exercises 5

Q5.1  
(i) \{1, 2, 3\}  
(ii) \{1 \mapsto \text{"a"}, 2 \mapsto \text{"x"}, 3 \mapsto \text{"a"}, 4 \mapsto \text{"z"}, 5 \mapsto \text{"e"}, \}  
(iii) same as (ii)  
(iv) \{1 \mapsto \text{"a"}, 2 \mapsto \text{"x"}, 3 \mapsto \text{"a"}, 5 \mapsto \text{"e"} \}  
(v) \text{"z"}  
(vi) not defined  
(vii) \{3 \mapsto \text{"b"}, 4 \mapsto \text{"a"}, 5 \mapsto \text{"f"} \}

Q5.2  
\text{inv-Def}_2 : (\text{Word} \rightarrow \text{WordDef}, \dagger, \theta) \rightarrow (\mathbb{B}, \land, \text{TRUE})

Q5.3 What can you conclude about 

\text{inv-Def}_2(\mu_1 \dagger \mu_2) = \text{inv-Def}_2\mu_1 \land \text{inv-Def}_2\mu_2

Q5.4  
\text{dom} : (A \rightarrow B) \rightarrow \mathcal{P}A  
\text{dom} \theta \subseteq \emptyset  
\text{dom}(\mu \cup \{a \mapsto b\}) \equiv \text{dom} \mu \cup \{a\}

Q5.5  
\text{dom} : (A \rightarrow B) \rightarrow A \rightarrow B  
\theta(a) \equiv \bot  
(\mu \cup \{a \mapsto b\})(a') \equiv a = a' \rightarrow b, \ \mu(a)

A.6 Exercises 6

Q6.1  
\text{inv-Sys}_0(\Sigma_0) = \text{TRUE}  
= (\text{defn. of } \Sigma_0)  
\text{inv-Sys}_0(\theta, \theta) = \text{TRUE}  
= (\text{defn. of } \text{inv-Sys}_0)  
(\text{}\circ \text{rang})\theta \subseteq (\text{}\circ \text{rang})\theta = \text{TRUE}  
= (X \subseteq X \text{ is always true})  
\text{TRUE}

Q6.2  
\text{pre-New}_0 : GP \rightarrow \text{Sys}_0 \rightarrow \mathbb{B}  
\text{pre-New}_0(\mu, \kappa) \equiv g \notin \text{dom} \rho

Q6.3 Assuming  
\text{inv-Sys}_0(\rho, \kappa)  
and  
\text{pre-New}_0(\mu, \kappa)

56
show

\[ \text{inv-Sys}_0(\text{New}_0(g)(\rho, \kappa)) \]

We first expand out assumption definitions, so they become

\[ (\cup/ \circ \text{rng}) \rho \subseteq (\cup/ \circ \text{rng}) \kappa \]

and

\[ g \notin \text{dom } \rho \]

We then proceed

\[
\text{inv-Sys}_0(\text{New}_0(g)(\rho, \kappa)) \\
= \quad (\text{defn. of } \text{New}_0) \\
\text{inv-Sys}_0(\rho \cup \{g \mapsto \emptyset\}, \kappa) \\
= \quad (\text{defn. of } \text{inv-Sys}_0) \\
(\cup/ \circ \text{rng})(\rho \cup \{g \mapsto \emptyset\}) \subseteq (\cup/ \circ \text{rng}) \kappa \\
= \quad (\text{2nd. assumption and defn. of } \cup \text{ backwards}) \\
(\cup/ \circ \text{rng})(\rho \cup \{g \mapsto \emptyset\}) \subseteq (\cup/ \circ \text{rng}) \kappa \\
= \quad (\cup/ \circ \text{rng}) \text{ is a homomorphism} \\
(\cup/ \circ \text{rng}) \rho \cup (\cup/ \circ \text{rng}) \{g \mapsto \emptyset\} \subseteq (\cup/ \circ \text{rng}) \kappa \\
= \quad (\cup/ \circ \text{rng}) \text{ on a maplet} \\
(\cup/ \circ \text{rng}) \rho \cup \emptyset \subseteq (\cup/ \circ \text{rng}) \kappa \\
= \quad (\text{Basic Set Theory}) \\
\text{True}
\]
B Definitions (Recursive)

B.1 Set Definitions

Membership \( \in \) : \( A \times \mathcal{P}A \rightarrow \mathbb{B} \)
\[ x \in \emptyset \equiv \text{FALSE} \]
\[ x \in (S \cup \{y\}) \equiv x = y \lor x \in S \]
\[ \chi : A \rightarrow \mathcal{P}A \rightarrow \mathbb{B} \]
\[ \chi(x)S \equiv x \in S \]

Union \( \cup \) : \( \mathcal{P}A \times \mathcal{P}A \rightarrow \mathcal{P}A \)
\[ S \cup \emptyset \equiv S \]
\[ S \cup (T \cup \{x\}) \equiv x \in S \rightarrow S \cup T \]
\[ x \notin S \rightarrow (S \cup \{x\}) \cup T \]

Intersection \( \cap \) : \( \mathcal{P}A \times \mathcal{P}A \rightarrow \mathcal{P}A \)
\[ S \cap \emptyset \equiv \emptyset \]
\[ S \cap (T \cup \{x\}) \equiv x \in S \rightarrow S \cap T \]
\[ x \notin S \rightarrow (S \setminus \{x\}) \cap T \]

Restriction \( \downarrow \) : \( \mathcal{P}A \rightarrow \mathcal{P}A \rightarrow \mathcal{P}A \)
\[ \downarrow(S)T \equiv S \cap T \]

Difference \( \setminus \) : \( \mathcal{P}A \times \mathcal{P}A \rightarrow \mathcal{P}A \)
\[ 0 \setminus T \equiv \emptyset \]
\[ (S \cup \{x\}) \setminus T \equiv x \in T \rightarrow S \setminus T \]
\[ x \notin S \rightarrow (S \setminus T) \cup \{x\} \]

Removal \( \downarrow \) : \( \mathcal{P}A \rightarrow \mathcal{P}A \rightarrow \mathcal{P}A \)
\[ \downarrow(S)T \equiv T \setminus S \]

Cardinality \# : \( \mathcal{P}A \rightarrow \mathbb{N} \)
\[ \#(0) \equiv 0 \]
\[ \#(S \cup \{x\}) \equiv \# S + 1 \]

Mapping \( \mathcal{P} \) : \( (A \rightarrow B) \rightarrow \mathcal{P}A \rightarrow \mathcal{P}B \)
\[ \mathcal{P}f0 \equiv \emptyset \]
\[ \mathcal{P}f(S \cup \{x\}) \equiv \mathcal{P}fS \cup \{f(x)\} \]

Reduction \( */ \) : \( (A \times A \rightarrow A) \rightarrow \mathcal{P}A \rightarrow \mathcal{A} \)
\[ */A \equiv a \]
\[ */(S \cup \{a\}) \equiv a */S \]
\[ */\emptyset \equiv e, \text{where } e \text{ is identity for *} \]

For-All \( \forall \) : \( A \rightarrow \mathbb{B} \rightarrow \mathcal{P}A \rightarrow \mathbb{B} \)
\[ \forall(P)S \equiv (\text{*/} \circ \mathcal{P}p)S \]

There-Exists \( \exists \) : \( A \rightarrow \mathbb{B} \rightarrow \mathcal{P}A \rightarrow \mathbb{B} \)
\[ \exists(P)S \equiv (\text{*/} \circ \mathcal{P}p)S \]
B.2 Sequence Definitions

\begin{align*}
\text{Head} & \quad \text{hd} : \quad A^* \rightarrow A \\
\quad \text{pre-hd} \sigma & \triangleq \quad \sigma \neq \Lambda \\
\quad \text{hd}(a : \sigma) & \triangleq \quad a \\
\text{Tail} & \quad t1 : \quad A^* \rightarrow A^* \\
\quad \text{pre-t1} \sigma & \triangleq \quad \sigma \neq \Lambda \\
\quad t1(a : \sigma) & \triangleq \quad \sigma \\
\text{Elements} & \quad \text{elems} : \quad A^* \rightarrow \mathcal{P}A \\
\quad \text{elems} \Lambda & \triangleq \quad \emptyset \\
\quad \text{elems}(x : \sigma) & \triangleq \quad \{x\} \cup \text{elems} \sigma \\
\text{Concatenation} & \quad \cdot \quad : \quad A^* \times A^* \rightarrow A^* \\
\quad \Lambda \cdot \tau & \triangleq \quad \tau \\
\quad (x : \sigma) \cdot \tau & \triangleq \quad x : (\sigma \cdot \tau) \\
\text{Indices} & \quad \text{inds} : \quad A^* \rightarrow \mathbb{N}^* \\
\quad \text{inds} \Lambda & \triangleq \quad 0 \\
\quad \text{inds} \sigma & \triangleq \quad \{1, \ldots, \text{len} \sigma\} \\
\text{Indexing} & \quad [\_] : \quad A^* \rightarrow \mathbb{N}_1 \rightarrow A \\
\quad \text{pre-}[i] & \triangleq \quad i \in \text{inds} \sigma \\
\quad (a : \sigma)[1] & \triangleq \quad a \\
\quad (a : \sigma)[i + 1] & \triangleq \quad \sigma[i] \\
\text{Slicing} & \quad \ldots : \quad A^* \rightarrow \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow A^* \\
\quad (a : \sigma)[1 \ldots r + 1] & \triangleq \quad a : (\sigma[1 \ldots r]) \\
\quad (a : \sigma)[l + 1 \ldots r + 1] & \triangleq \quad \sigma[l \ldots r] \\
\quad \sigma[l \ldots r] & \triangleq \quad \Lambda \\
\text{Restriction} & \quad \lhd : \quad \mathcal{P}A \rightarrow A^* \rightarrow A^* \\
\quad \lhd(S) \Lambda & \triangleq \quad \Lambda \\
\quad \lhd(S)(x : \sigma) & \triangleq \quad x \notin S \rightarrow \lhd(S)\sigma \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow x : (\lhd(S)\sigma) \\
\text{Removal} & \quad \rhd : \quad \mathcal{P}A \rightarrow A^* \rightarrow A^* \\
\quad \rhd(S) \Lambda & \triangleq \quad \Lambda \\
\quad \rhd(S)(x : \sigma) & \triangleq \quad x \in S \rightarrow \rhd(S)\sigma \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow x : (\rhd(S)\sigma) \\
\text{Length} & \quad \text{len} : \quad A^* \rightarrow \mathbb{N} \\
\quad \text{len}(\Lambda) & \triangleq \quad 0 \\
\quad \text{len}(x : \sigma) & \triangleq \quad \text{len} S + 1 
\end{align*}
Mapping \( \star : (A \to B) \to A^\star \to B^\star \)
\[ f^\star \Delta \equiv \Delta \]
\[ f^\star (x : \sigma) \equiv f(x) : f^\star \sigma \]

Reduction \( \gamma : (A \times A \to A) \to A^\star \to A \)
\[ \gamma / (a) \equiv a \]
\[ \gamma / (a : \sigma) \equiv a \star \gamma / \sigma \]
\[ \gamma / \Delta \equiv e, \text{where } e \text{ is identity for } \star \]

For-All \( \forall : A \to B \to A^\star \to B \)
\[ \forall(P)\sigma \equiv (\gamma / \circ P^\star)\sigma \]

There-Exists \( \exists : A \to B \to A^\star \to B \)
\[ \exists(P)\sigma \equiv (\gamma / \circ P^\star)\sigma \]
B.3  Set & Sequence Properties

\[ R \cup (S \cup T) = (R \cup S) \cup T \]
\[ R \cap (S \cap T) = (R \cap S) \cap T \]

\[ x \in (S \cup T) = x \in S \vee x \in T \]
\[ x \in (S \cap T) = x \in S \wedge x \in T \]
\[ \chi(x)(S \cup T) = \chi(x)S \lor \chi(x)T \]
\[ \chi(x)(S \cap T) = \chi(x)S \land \chi(x)T \]

\[ \downarrow(R)(S \cup T) = \downarrow(R)S \cup \downarrow(R)T \]
\[ \downarrow(R)(S \cap T) = \downarrow(R)S \cap \downarrow(R)T \]
\[ \downarrow(R)(S \cup T) = \downarrow(R)S \cup \downarrow(R)T \]
\[ \downarrow(R)(S \cap T) = \downarrow(R)S \cap \downarrow(R)T \]
\[ \downarrow(S)T \cup \downarrow(S)T = T \]

\[ #(S \cup T) = #S + #T \]
\[ #(S \cup T) + #(S \cap T) = #S + #T \]

\[ \mathcal{P}f(S \cup T) = \mathcal{P}fS \cup \mathcal{P}fT \]
\[ ^*/(S \cup T) = (^*/S) \ast (^*/T) \]

\[ \text{hd}(\sigma) : (\text{tl}\sigma) = \sigma, \quad \sigma \neq \Lambda \]
\[ \rho \setminus (\sigma \setminus \tau) = (\rho \setminus \sigma) \setminus \tau \]

\[ \text{elems } (\sigma \setminus \tau) \supseteq \text{elems } \sigma \cup \text{elems } \tau \]
\[ \downarrow(R)(\sigma \setminus \tau) = \downarrow(R)\sigma \setminus \downarrow(R)\tau \]
\[ \downarrow(R)(\sigma \setminus \tau) = \downarrow(R)\sigma \setminus \downarrow(R)\tau \]

\[ \text{len}(\sigma \setminus \tau) \supseteq \text{len } \sigma + \text{len } \tau \]
\[ f^\ast(\sigma \setminus \tau) = f^\ast\sigma \setminus f^\ast\tau \]
\[ ^*/(\sigma \setminus \tau) = (^*/\sigma) \ast (^*/\tau) \]

if \(*\) is associative
B.4 Set & Sequence Examples

\[ x \in \{x\} = \text{TRUE} \]
\[ \{1, 3, 5\} \cup \{3, 4, 5, 6\} = \{1, 3, 4, 5, 6\} \]
\[ \{1, 3, 5\} \cap \{3, 4, 5, 6\} = \{3, 5\} \]
\[ \{1, 3, 5\} \setminus \{3, 4, 5, 6\} = \{1\} \]
\[ \{3, 4, 5, 6\} \setminus \{1, 3, 5\} = \{4, 6\} \]

\[ \#\{1, 3, 4, 5, 6\} = 5 \]
\[ P_{\text{sq}r}\{1, 3, 5\} = \{1, 9, 25\} \]
\[ \times/\{1, 2, 3, 4, 5, 6\} = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 6! \]
\[ (+/\circ P_{\text{sq}r})\{1, 2, 3, 4\} = +/\{1, 4, 9, 16\} = 30 \]
\[ \text{hd}(4, 3, 5, 6) = 4 \]
\[ \text{tl}(4, 3, 5, 6) = (3, 5, 6) \]
\[ \text{elems}(2, 2, 2) = \{2\} \]
\[ \langle 1, 3, 2, 1 \rangle \ominus \langle 1, 2, 3, 1 \rangle = \langle 1, 3, 2, 1, 1, 2, 3, 1 \rangle \]
\[ \text{a}((\{1, 3, 5\})\{6, 5, 4, 3\}) = (5, 3) \]
\[ \text{a}((\{1, 3, 5\})\{6, 5, 4, 3\}) = (6, 4) \]
\[ \text{l}en(2, 2, 2) = 3 \]
\[ \text{sq}r^\times(1, 3, 5) = \{1, 9, 25\} \]
\[ \times/\{1, 2, 3, 4, 5, 6\} = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 6! \]
\[ (+/\circ \text{sq}r^\times)(\{1, 2, 3, 4\}) = +/\{1, 4, 9, 16\} = 30 \]

C Structures and their Morphisms

C.1 Definition of Structures

C.1.1 Semigroup

Given:

- set \( A \),
- binary operation \( \ast : A \times A \to A \),
- property \( a_1 \ast (a_2 \ast a_3) = (a_1 \ast a_2) \ast a_3 \) (associativity)
we say that \((A, \ast)\) is a **Semigroup**, sometimes written

\[
\text{SGr}(A, \ast)
\]

**Given:**

- two semigroups \((A, \oplus)\) and \((B, \otimes)\)
- function \(h : A \rightarrow B\)
- property \(h(a_1 \oplus a_2) = h(a_1) \otimes h(a_2)\)

we say that \(h\) is a **Semigroup Homomorphism** from \((A, \oplus)\) to \((B, \otimes)\), sometimes written

\[
h : (A, \oplus) \rightarrow (B, \otimes)
\]

**C.1.2 Monoid**

**Given:**

- set \(A\),
- binary operation \(\ast : A \times A \rightarrow A\),
- specific element \(e : A\),
- property \(a_1 \ast (a_2 \ast a_3) = (a_1 \ast a_2) \ast a_3\) (associativity)
- property \(e \ast a = a = a \ast e\) (identity element)

we say that \((A, \ast, e)\) is a **Monoid**, sometimes written

\[
\text{Mon}(A, \ast, e)
\]

**Given:**

- two Monoids \((A, \oplus, e)\) and \((B, \otimes, e')\)
- function \(h : A \rightarrow B\)
- property \(h(a_1 \oplus a_2) = h(a_1) \otimes h(a_2)\)
- property \(h(e) = e'\)

we say that \(h\) is a **Monoid Homomorphism** from \((A, \oplus, e)\) to \((B, \otimes, e')\), sometimes written

\[
h : (A, \oplus, e) \rightarrow (B, \otimes, e')
\]
C.1.3 Group

Given:

- set $A$,
- binary operation $*: A \times A \rightarrow A$,
- specific element $e : A$,
- function $^{-1}: A \rightarrow A$
- property $a_1 * (a_2 * a_3) = (a_1 * a_2) * a_3$ (associativity)
- property $e * a = a = a * e$ (identity element)
- property $a * a^{-1} = e = a^{-1} * a$ (inverse)

we say that $(A, *, e, ^{-1})$ is a Group, sometimes written

$$\text{Grp}(A, *, e, ^{-1})$$

Given:

- two Groups $(A, \oplus, e, ^{-1})$ and $(B, \otimes, e', ^{-1'})$
- function $h : A \rightarrow B$
- property $h(a_1 \oplus a_2) = h(a_1) \otimes h(a_2)$
- property $h(e) = e'$
- property $h(a^{-1}) = (h(a))^{-1'}$

we say that $h$ is a Group Homomorphism from $(A, \oplus, e, ^{-1})$ to $(B, \otimes, e', ^{-1'})$, sometimes written

$$h : (A, \oplus, e, ^{-1}) \rightarrow (B, \otimes, e', ^{-1'})$$

C.1.4 Morphism Composition

Given two composable homomorphisms $h_1$ and $h_2$

- $h_1 : (A, \oplus, \ldots) \rightarrow (B, \otimes, \ldots)$
- $h_2 : (B, \otimes, \ldots) \rightarrow (D, *, \ldots)$

then their composition is also a homomorphism:

$$h_2 \circ h_1 : (A, \oplus, \ldots) \rightarrow (C, *, \ldots)$$
C.1.5 Morphism Definition

To completely define a homomorphism it is necessary to:

- Describe the source and target structures.
- Define the result returned by the homomorphism for all the generating elements for the target structure (with the exception of the identity element if structure is monoidal).

C.2 Booleans

C.2.1 Boolean Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B, ∧, True)</td>
<td>{False, True}</td>
</tr>
<tr>
<td>(B, ∨, False)</td>
<td>{False, True}</td>
</tr>
</tbody>
</table>

C.2.2 Boolean Morphisms

\[ \neg : (B, \land, \text{True}) \rightarrow (B, \lor, \text{False}) \]
\[ \neg : (B, \lor, \text{False}) \rightarrow (B, \land, \text{True}) \]

C.3 Numbers

C.3.1 Number Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N, +, 0)</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>(N, ×, 1)</td>
<td>{0} ∪ {p</td>
</tr>
<tr>
<td>(Z, +, 0, -)</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>(Z, ×, 1)</td>
<td>{0} ∪ {±p</td>
</tr>
<tr>
<td>(Q, ×, 1, -1)</td>
<td>{p</td>
</tr>
<tr>
<td>(R, +, 0, -)</td>
<td></td>
</tr>
<tr>
<td>(R_{&gt;0}, ×, 1, -1)</td>
<td></td>
</tr>
</tbody>
</table>

C.3.2 Number Morphisms

\[ \log_b : (R_{>0}, \times, 1, -1) \rightarrow (R, +, 0, -) \]
\[ \log_b (b) \cong 1.0 \]
### C.4 Sets

#### C.4.1 Set Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\mathcal{P}A \cup \emptyset) )</td>
<td>( {\emptyset} \cup {{a} \mid a \in A} )</td>
</tr>
<tr>
<td>( (\mathcal{P}A \cap A) )</td>
<td>( {A} \cup {A \setminus {a} \mid a \in A} )</td>
</tr>
</tbody>
</table>

#### C.4.2 Set Morphisms

\[
\begin{align*}
\mathcal{P} f & : (\mathcal{P}A \cup \emptyset) \to (\mathcal{P}B \cup \emptyset) \\
\mathcal{P} f \{a\} & \doteq \{f(a)\} \\
\mathcal{A} & : (\mathcal{P}A \cup \emptyset) \to (\mathcal{P}A \cup \emptyset) \\
\mathcal{A}\{a\} & \doteq a \in S \to \{a\}, \emptyset \\
\mathcal{A} : (\mathcal{P}A \cap A) \to (\mathcal{P}A \cap A) \\
\mathcal{A}(A \setminus \{a\}) & \doteq a \in S \to S \setminus \{a\}, S \\
\mathcal{A} : (\mathcal{P}A \cup \emptyset) \to (\mathcal{P}A \cup \emptyset) \\
\mathcal{A}\{a\} & \doteq a \in S \to \emptyset, \{a\} \\
\mathcal{A} : (\mathcal{P}A \cap A) \to (\mathcal{P}A \cap A) \\
\mathcal{A}(A \setminus \{a\}) & \doteq a \in S \to A \setminus S, A \setminus (A \cup \{a\})
\end{align*}
\]

where \( f : A \to B \) and \( S : \mathcal{P}A \).

### C.5 Sequences

#### C.5.1 Sequence Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (A^*, \sqsubseteq, \Lambda) )</td>
<td>( {\Lambda} \cup {\langle a\rangle \mid a \in A} )</td>
</tr>
</tbody>
</table>

#### C.5.2 Sequence Morphisms

\[
\begin{align*}
f^* & : (A^*, \sqsubseteq, \Lambda) \to (B^*, \sqsubseteq, \Lambda) \\
f^* \langle a\rangle & \doteq \langle f(a)\rangle \\
\mathcal{A} & : (A^*, \sqsubseteq, \Lambda) \to (A^*, \sqsubseteq, \Lambda) \\
\mathcal{A}\langle a\rangle & \doteq a \in S \to \langle a\rangle, \Lambda \\
\mathcal{A} & : (A^*, \sqsubseteq, \Lambda) \to (A^*, \sqsubseteq, \Lambda) \\
\mathcal{A}\langle a\rangle & \doteq a \in S \to \Lambda, \langle a\rangle
\end{align*}
\]

where \( f : A \to B \) and \( S : \mathcal{P}A \).
C.6 Maps

C.6.1 Map Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>((A \to B, \oplus, \theta))</td>
<td>({\theta} \cup \{{a \mapsto b} \mid a \in A, b \in B})</td>
</tr>
<tr>
<td>((A \to B, \odot, \theta))</td>
<td>where ((B, \ast)) is a semi-group</td>
</tr>
</tbody>
</table>

C.6.2 Map Morphisms

\[
(f \to g) \quad : \quad (A \to B, \oplus, \theta) \to (C \to D, \oplus, \theta)
\]

\[
(f \to g) \{a \mapsto b\} \equiv \{f(a) \mapsto g(b)\}
\]

\[
\triangledown(S) \quad : \quad (A \to B, \oplus, \theta) \to (A \to B, \oplus, \theta)
\]

\[
\triangledown(S) \{a \mapsto b\} \equiv \{a \in S \to \{a \mapsto b\}, \theta\}
\]

\[
\triangledown(S) \{a \mapsto b\} \quad \equiv \quad a \in S \to \theta, \{a \mapsto b\}
\]

where \(f : A \to C, g : B \to D, S : \mathcal{P}A,\) and \(f\) is injective \((f(a_1) = f(a_2) \Rightarrow a_1 = a_2)\).

C.7 Mixed Morphisms

\[
\text{len} \quad : \quad (A^*, \sqcap, \Lambda) \to (\mathbb{N}, +, 0)
\]

\[
\text{len}(a) \quad \equiv \quad 1
\]

\[
\text{sum}_f \quad : \quad (A^*, \sqcap, \Lambda) \to (\mathbb{N}, +, 0)
\]

\[
\text{sum}_f(a) \quad \equiv \quad f(a)
\]

where \(f : A \to \mathbb{N}\)

\[
\text{dom} \quad : \quad (A \to B, \oplus, \theta) \to (\mathcal{P}A, \sqcup, \emptyset)
\]

\[
\text{dom} \{a \mapsto b\} \quad \equiv \quad \{a\}
\]

\[
\sqcup/ \quad : \quad (\mathcal{P}A, \sqcup, \emptyset) \to (A, \sqcup, e)
\]

\[
\sqcup/\{a\} \quad \equiv \quad a
\]

where \(\sqcup : A \times A \to A\) with identity \(e\)

and \(\sqcup\) is associative and commutative.

\[
\sqcup/ \quad : \quad (A^*, \sqcap, \Lambda) \to (A, \sqcup, e)
\]

\[
\sqcup/\{a\} \quad \equiv \quad a
\]

where \(\sqcup : A \times A \to A\) with identity \(e\)

and \(\sqcup\) is associative.

67