# Chapter 3

# Data types

Components of reactive systems often exchange messages containing data items among themselves and with the environment. For example, recall the alarm clock with multiple alarms from chapter 2, of which the labelled transition system is reproduced in figure 3.1. In this figure, the data parameter of action $set$ is to be used by the environment to communicate the number of times the alarm should sound. We need a language to describe data types and their elements to be used in specifying behaviour.

In this section we first describe a basic equational data type specification mechanism. One may define data types (called sorts), by defining their elements (constructors), their operations (maps) and the rules (equations) defining the operations. It is also allowed to use function types in this setting.

As there are a number of data types that are commonly used in behavioural specifications, these have been predefined. These are, for example, common data types such as natural numbers, booleans and real numbers. These data types are designed to be as close as possible to their mathematical counterpart. So, there are an unbounded number of natural and real numbers. The full specification of all built-in data types is given in appendix A. There are also mechanisms to compactly define more elaborate data types without giving a full equational data type specification. Typical examples are structured data types (specified by enumerating their members), lists and sets. This chapter is dedicated to the data type specification language and in the next chapter we study how data types can be attached to actions and how data expressions can influence the behaviour of processes.

## 3.1 Data type definition mechanism

In mCRL2, we have a straightforward data definition mechanism using which all data sorts are built. One can declare arbitrary sorts using the keyword **sort**. Sorts are non-empty, possibly infinite sets with data elements. For a sort one can define constructor functions using the keyword **cons**. These are functions by which exactly all elements in the sort can be denoted. For instance

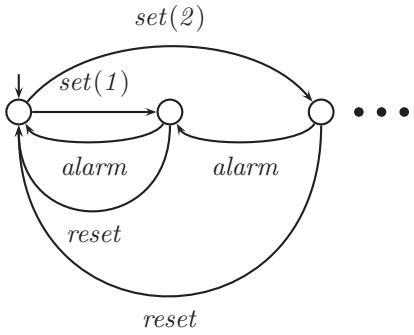


Figure 3.1: An alarm clock of which the number of alarms can be set

**sort** $D$;
**cons** $c, d : D$;

declares sort $D$ in which all elements can be denoted by either $c$ or $d$. So, $D$ has either two elements, or in case $c$ and $d$ are the same, it has one element.

Using constructor functions, it is possible to declare a sort $Nat$ representing the natural numbers. This is not the actual built-in sort $\mathbb{N}$ in mCRL2, which for efficiency purposes has a different internal structure. The definition of the built-in data type can be found in appendix A.

**sort** $Nat$;
**cons** $zero : Nat$;
$successor : Nat \to Nat$;

In this case we have a domain $Nat$ of which all elements can be denoted by $zero$, or an expression of the form:

$$successor(successor(\ldots successor(zero)\ldots)).$$

Without explicitly indicating so, these elements are not necessarily different. Below, it is shown how it can be guaranteed that all elements of the sort *Nat* must differ.

Similarly to the definition of sort *Nat*, a sort $\mathbb{B}$ of booleans can be defined. The standard definition for $\mathbb{B}$ is as follows:

**sort** $\mathbb{B}$;
**cons** $true, false : \mathbb{B}$;

The sort $\mathbb{B}$ plays a special role. In the first place the semantics of the language prescribes that the constructors $true$ and $false$ must be different. This is the only exception to the rule that constructors are not presumed to be different. Using this exception one can build sorts with necessarily different elements. So, there are exactly two booleans. In the second place, booleans are used to connect data specification with behaviour, namely, conditions in processes and conditional equations must be of sort $\mathbb{B}$. In appendix A additional operators for $\mathbb{B}$ are defined.

The following example does not define a proper sort, because it can only be empty. All data elements in $D$ must be denoted as a term consisting of applications of the function $f$ only. But as there is no constant, i.e., a function symbol with no argument, such a term cannot be constructed, and hence the sort $D$ must be empty. However, empty sorts are not permitted and therefore there is no data type satisfying this specification. We say that such a data type is inconsistent.

**sort** $D$;
**cons** $f : D \to D$;

It is possible to declare sorts without constructor functions. In this case the sort can contain an arbitrary number of elements. In particular, the sort can contain elements that cannot be denoted by a term. As an example it is possible to declare a sort $Message$ without constructors. In this way, one can model for instance data transfer protocols without assuming anything about messages being transferred.

**sort** $Message$;

Auxiliary functions can be declared using the keyword **map**. For instance, the equality, addition and multiplication operators on natural numbers are not necessary to construct all the numbers, but they are just useful operations. They can be declared as follows:

**map** $eq : Nat \times Nat \to \mathbb{B}$;
$plus, times : Nat \times Nat \to Nat$;

Here the notation $eq : Nat \times Nat \to \mathbb{B}$ says that $eq$ is a function with two arguments of sort $Nat$ yielding an element of sort $\mathbb{B}$. The symbol $\times$ is called the Cartesian product operator.

But it is not sufficient to only declare the type of an operator. It must also be defined how the operator calculates a value. This can be done by introducing equations using the keyword **eqn**. Two terms are equal if they can be transformed into each other using the equations. It does not matter if the equations are applied from left to right, or from right to left. However, for efficiency reasons, most tools strictly apply the equations from left to right, which is generally called term rewriting. This directed view of equations is an informal convention, but it has a profound effect on the equations.

For addition and multiplication the equations are given below. Using the keyword **var** the variables needed in the next equation section are declared.

**var** $n, m : Nat$;
**eqn** $eq(n, n) = true$;
$eq(zero, successor(n)) = false$;
$eq(successor(n), zero) = false$;
$eq(successor(n), successor(m)) = eq(n, m)$;
$plus(n, zero) = n$;
$plus(n, successor(m)) = successor(plus(n, m))$;
$times(n, zero) = zero$;
$times(n, successor(m)) = plus(n, times(n, m))$;

By applying these equations, one can show which terms are equal to others. For instance showing that $2{\cdot}2 = 4$ goes as follows (where the numbers 2 and 4 represent $successor(successor(zero))$ and $successor(successor(successor(successor(zero))))$, respectively):

$$
\begin{aligned}
&times(successor(successor(zero)), successor(successor(zero))) = \\
&plus(successor(successor(zero)), \\
&\qquad times(successor(successor(zero)), successor(zero))) = \\
&plus(successor(successor(zero)), plus(successor(successor(zero)), \\
&\qquad times(successor(successor(zero)), zero))) = \\
&plus(successor(successor(zero)), plus(successor(successor(zero)), zero)) = \\
&plus(successor(successor(zero)), successor(successor(zero))) = \\
&successor(plus(successor(successor(zero)), successor(zero))) = \\
&successor(successor(plus(successor(successor(zero)), zero))) = \\
&successor(successor(successor(successor(zero))))
\end{aligned}
$$

There is much to say about whether these equations suffice or whether their symmetric variants (e.g., $plus(zero, n)$) should be included, too. These equations are essentially sufficient to prove properties, but for the tools adding more equations can make a huge difference in performance.

When defining functions, it is a good strategy to define them on terms consisting of the constructors applied to variables. In the case above these constructors are $zero$ and $successor(n)$ and the constructor patterns are only used in one argument. But sometimes such patterns are required in more arguments, and it can even be necessary that the patterns are more complex. As an example consider the definition of the function $even$ below which in this form requires patterns $zero$, $successor(zero)$ and $successor(successor(n))$.

**map** $even : Nat \to \mathbb{B}$;
**var** $n : Nat$;
**eqn** $even(zero) = true$;
$even(successor(zero)) = false$;
$even(successor(successor(n))) = even(n)$;

It is very well possible to only partly define a function. Suppose the function $even$ should yield $true$ for even numbers, and it is immaterial whether it should deliver $true$ or $false$ for odd numbers. Then the second equation can be omitted. This does not mean that $even(successor(zero))$ is undefined. It has a fixed value (either $true$ or $false$), except that we do not know what it is.

The equations can have conditions that must be valid before they can be applied. The definition above can be rephrased as:

**var** $n : Nat$;
**eqn** $eq(n, zero) \to even(n) = true$;
$eq(n, successor(zero)) \to even(n) = false$;
$even(successor(successor(n))) = even(n)$;

Besides equational reasoning there are two major proof principles to be used in the context of equational data types, namely proof by contradiction and induction, which are explained below.

The equations in an equation section can only be used to show that certain data elements are equal. In order to show that $zero$ and $successor(zero)$ are not equal another mechanism is required. We know that $true$ and $false$ are different. Within this basic data definition mechanism, this is the only assumption about terms not being equal. In order to show that other data elements are not equal, we assume that they are equal and then derive $true = false$, from which we conclude that the assumption is not true. This is called proof by contradiction or reductio ad absurdum. Note that such a reduction always requires an auxiliary function from such a data sort to booleans. In case of the sort $Nat$ we can define the function $less$ to do the job:

**map** $less : Nat \times Nat \to \mathbb{B}$;
**var** $n, m : Nat$;
**eqn** $less(n, zero) = false$;
$less(zero, successor(n)) = true$;
$less(successor(n), successor(m)) = less(n, m)$;

Now assume that $zero$ and $successor(zero)$ are equal, more precisely,

$$zero = successor(zero).$$

Then, we can derive:

$$true = less(zero, successor(zero)) \overset{\text{assumption}}{=} less(zero, zero) = false.$$

So, under this assumption, $true$ and $false$ coincide, leading to the conclusion that $zero$ and $successor(zero)$ must be different. In a similar way it can be proven that any pair of different natural numbers are indeed different when the function $less$ is present.

When constructors are present, we know that all terms have a particular shape and we can use that to prove properties about all terms. For instance, to prove a property $\phi(b)$ for a boolean variable $b$, it suffices to prove that $\phi(false)$ and $\phi(true)$ hold. This proof principle is called *induction* on the structure of booleans.

When there is a constructor $f$ of a data type $D$ that depends on $D$, we can assume that a property $\phi(x)$ holds for a variable of sort $D$ to prove $\phi(f(x))$. Here $\phi(x)$ is called the induction hypothesis. In this way, we have shown that if property $\phi$ holds for smaller terms, it also holds for larger terms, and by simply repeating this, $\phi$ holds for all terms. A precise formulation of induction on the basis of constructors can be found in section 9.5.

**Example 3.1.1.** Consider the data type $Nat$ as defined above. There are constructors $zero$ and $successor$. Therefore, in order to prove that a property $\phi(n)$ holds for $n$ a variable of sort $Nat$, it must be shown that $\phi(zero)$ holds, and $\phi(n)$ implies $\phi(successor(n))$. This form of induction is sometimes referred to as mathematical induction.

Concretely, we can prove that $plus(zero, n) = n$. So, we must show:

$$plus(zero, zero) = zero, and$$
$$plus(zero, n) = n \text{ implies } plus(zero, successor(n)) = successor(n).$$

The first equation is easy to prove with the equations for $plus$. The second equations follows by

$$\begin{aligned} & plus(zero, successor(n)) = \\ & successor(plus(zero, n)) \overset{\text{induction hypothesis}}{=} \\ & successor(n). \end{aligned}$$

So, we can conclude by induction on $Nat$ that $plus(zero, n) = n$.

**Exercise 3.1.2.** Give equational specifications of 'greater than or equal' $\geq$, 'smaller than' $<$ and 'greater than' $>$ on the natural numbers.

**Exercise 3.1.3.** Give specifications of $max{:}Nat \times Nat \to Nat$, $minus{:}\ Nat \times Nat \to Nat$ and $power{:}Nat \times Nat \to Nat$ where $power(m, n)$ denotes $m^n$.

**Exercise 3.1.4.** Explain why any of the following equations are not wrong, but unpleasant or problematic, when using term rewriting tools.

**var** $n : Nat$;
**eqn** $true = even(zero)$;
$false = even(successor(zero))$;
$even(n) = even(successor(successor(n)))$;

**Exercise 3.1.5.** Prove using the mapping $eq$ that $zero$ and $successor(zero)$ represent different data elements.

**Exercise 3.1.6.** Prove using induction that

$$plus(successor(n), m) = successor(plus(n, m)).$$

Use this result to prove, again with induction on $Nat$, that $plus(n, m) = plus(m, n)$.

**Exercise 3.1.7.** Prove for variables $n, m$ of sort $Nat$ that if $less(n, m) = true$, then $n \neq m$.

**Exercise 3.1.8.** Define a sort *List* on an arbitrary non-empty domain $D$, with as constructors the empty list $[]{:}List$ and $in{:}D \times List \to List$ to insert an element of $D$ at the beginning of a list. Extend this with the following non-constructor functions: $append{:}D \times List \to List$ to insert an element of $D$ at the end of a list; $top{:}List \to D$ and $toe{:}List \to D$ to obtain the first and the last element of a list; $tail{:}List \to List$ and $untoe{:}List \to List$ to remove the first and the last element from a list, respectively; $nonempty{:}List \to \mathbb{B}$ to check whether a list is empty, $length{:}List \to Nat$ to compute the length of a list, and ${+}{+}{:}List \times List \to List$ to concatenate two lists.

| Sort | *Notation* |
|---|---|
| Booleans | $\mathbb{B}$ |
| Positive numbers | $\mathbb{N}^+$ |
| Natural numbers | $\mathbb{N}$ |
| Integers | $\mathbb{Z}$ |
| Real numbers | $\mathbb{R}$ |
| Structured types | $\mathbf{struct} \ldots \mid \ldots \mid \ldots .$ |
| Functions | $D_1 \times \cdots \times D_n \to E$ |
| Lists | $List(D)$ |
| Sets | $Set(D)$ |
| Bags | $Bag(D)$ |

Table 3.1: The predefined sorts ($D$, $D_i$ and $E$ are sorts)

## 3.2 Standard data types

When modelling communicating systems, often the same data types are used, namely booleans, numbers, structured types, lists, functions and sets. Therefore, these are predefined. For these data types we use common mathematical notation. The sorts are summarised in table 3.1.

Each sort (both user-defined and built-in) has automatically generated built-in functions for if-then-else ($if(\_,\_,\_)$), equality ($\approx$), inequality ($\not\approx$) and ordering operators ($\leq, <, >, \geq$). These functions are functions from their respective data domains to booleans. Most predefined functions are denoted using infix notation, instead of the somewhat cumbersome prefix notation used in the previous section.

The equality function should not be confused with equality among terms ($=$), which indicates which terms are equal. It is the strength of the defining equations for data equality, that allows us to use term equality and data equality interchangeably.

The defining equations for $\approx$, $\not\approx$ and $if(\_,\_,\_)$ are as follows (the equations for missing operators can be found in appendix A). Note that for a specific sort $D$, the user may specify more equations for calculating functions such as $\approx$ by exploiting the structure of $D$.

**map** $\approx, \not\approx, \leq, <, >, \geq: D \times D \to \mathbb{B}$;
$\quad if : \mathbb{B} \times D \times D \to D$;
**var** $x, y : D$;
$\quad b : \mathbb{B}$;
**eqn** $x \approx x = true$; $\qquad x < x = false$;
$\quad x \not\approx y = \neg(x \approx y)$; $\qquad x \leq x = true$;
$\quad if(true, x, y) = x$; $\qquad x > y = y < x$;
$\quad if(false, x, y) = y$; $\qquad x \geq y = y \leq x$;
$\quad if(b, x, x) = x$;
$\quad if(x \approx y, x, y) = y$;

The last equation is called *Bergstra's*[1] *axiom*. As said above equality on terms is

[1] Jan Bergstra (1951-. . .) is the founding father of process algebra in the Amsterdam style (ACP). He

strongly related to data equality $\approx$. More precisely, the following lemma holds:

**Lemma 3.2.1.** For any data sort $D$ for which the equations above are defined, it holds that:

$$x \approx y = true \quad \text{iff} \quad x = y.$$

**Proof.** For the direction from left to right, we derive:

$$x = if(true, x, y) = if(x \approx y, x, y) = y.$$

For the direction from right to left, we derive:

$$x \approx y = x \approx x = true.$$

□

Bergstra's axiom is generally not used by tools since the shape of the axiom is not very convenient for term rewriting.

**Exercise 3.2.2.** Consider the specification

**sort** $D$;
**map** $d_1, d_2{:}D$;

Which of the following expressions are equal to $true$ or $false$: $d_1 \approx d_1$, $d_1 \approx d_2$, $d_1 > d_2$, $d_2 \not\approx d_1$ and $d_2 \geq d_2$.

### 3.2.1 Booleans

The sort for boolean has already been introduced as $\mathbb{B}$. It consists of exactly two different constructors $true$ and $false$. For this sort, the operations are listed in table 3.2. The syntax used by the tools in the mCRL2 language can be found in appendix B. The equations with which the operators on booleans are defined are found in appendix A.

Most functions on $\mathbb{B}$ are standard and do not need an explanation. In boolean expressions, it is possible to use quantifiers $\forall$ and $\exists$. They add a substantial amount of expressivity to the language. This is important because compact, insightful behavioural specifications reduce the number of errors, increase the comprehensibility and in general lead to better balanced behaviour of designs.

We illustrate the expressiveness of the specification language with quantifiers with an example.

**Example 3.2.3.** Fermat's Last Theorem was an open problem for more than 3 centuries. It states that there is no positive number $n > 2$ such that $a^n + b^n = c^n$ for natural number $a$, $b$ and $c$.

**map** $fermat : \mathbb{B}$;
**eqn** $fermat = \forall a, b, c, n{:}\mathbb{N}^+.(n \leq 2 \vee a^n + b^n \not\approx c^n)$;

contributed to many other topics, such as the theory of abstract data types and modules and program algebras.

| Operator | *Notation* |
|---|---|
| true | $true$ |
| false | $false$ |
| negation | $\neg\_$ |
| conjunction | $\_\wedge\_$ |
| disjunction | $\_\vee\_$ |
| implication | $\_\Rightarrow\_$ |
| equality | $\_\approx\_$ |
| inequality | $\_\not\approx\_$ |
| conditional | $if(\_,\_,\_)$ |
| universal quantification | $\forall\_:\_.\_$ |
| existential quantification | $\exists\_:\_.\_$ |

Table 3.2: Operators on booleans

As the conjecture holds, $fermat$ is equal to $true$ but any tool that wants to figure this out, must be sufficiently strong to prove Fermat's Last Theorem.

As demonstrated by the example given above the downside of the expressiveness of quantifiers is that tools can have difficulties to handle them. This may mean that a specification with quantifiers cannot even be simulated. It is the subject of continuous research to make the tools more effective in dealing with these primitives. It requires experience to know which expressions can and which cannot be handled effectively by the tools in their current state.

### 3.2.2 Numbers

Positive numbers, natural numbers, integers and reals are represented by the sorts $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$, respectively. Numbers are denoted in the common way, e.g., $1$, $2$, $45978$, $0$, $-1$. There is no special notation for reals, but these can be constructed using, for instance, the division operator.

The common notation for numbers is there just for convenience. Each number is immediately translated to a number as an element of an abstract data type as defined in appendix A. All data types in this appendix have been designed such that each constructor term uniquely defines a data element and has an efficient representation. For instance there is only one constructor term representing $0$ in the integers (and not for instance $-0$ and $+0$) and the internal representation uses a binary encoding (and not zero and successor).

For positive numbers there are two constructors, namely $@c1{:}\mathbb{N}^+$ and $@cDub$ : $\mathbb{B} \times \mathbb{N}^+ \to \mathbb{N}^+$. The constructor $@c1$ represents $1$ and $@cDub(b, p)$ equals $2p$ if $b$ is false, and $2p + 1$ if $b$ is true. So, $2$ is represented by $@cDub(false, p)$ and $3$ is represented by $@cDub(true, p)$. A natural number is either $0$ or a positive number. An integer is either a natural number representing $0$, $1$, etc., or a positive number, representing $-1$, $-2$, ....

Generally, we will not refer to the representation of numbers in their internal format and use their common denotation. The numbers satisfy all properties that we expect from ordinary mathematical numbers. In particular, the numbers are unbounded, which means that there is no largest natural number, and there are no smallest and largest integers. Real numbers are exact, so there is no issue regarding precision.

But the internal representation and the equations in appendix A can become relevant if a more detailed scrutiny of the data types is required. It can be that a certain equality between data terms cannot be proven, and in order to understand why, the precise defining equations are required. Also, the induction on numbers that is defined by the constructors, is not ordinary mathematical induction, which uses zero and successor. In order to understand that mathematical induction on natural numbers is sound the precise definitions are also required. The relation between constructors and induction is an issue that we address in chapter 9.

There is an implicit type conversion between numbers. Any positive number can become a natural number, which in turn can become an integer, which can become a real number. These automatic conversions apply to any object, not only to constants but also to variables and terms.

The operators on numbers are given in table 3.3. They are all well-known. Most operators are defined for all possible types of numbers. So, there are additional operators for $\mathbb{N}^+ \times \mathbb{N}^+$, $\mathbb{N} \times \mathbb{N}$, $\mathbb{Z} \times \mathbb{Z}$ and for $\mathbb{R} \times \mathbb{R}$. The resulting type is the most restrictive sort possible. Addition on $\mathbb{N}^+ \times \mathbb{N}^+$ has as resulting sort $\mathbb{N}^+$, but subtraction on $\mathbb{N}^+ \times \mathbb{N}^+$ has as result sort $\mathbb{Z}$, as the second number can be larger than the first. Some operators have restricted sorts. For instance, for the modulo operator the sort of the second operator must be $\mathbb{N}^+$ as $x|_0$ is generally not defined. In accordance with common usage, we write multiplication as a dot, or leave it out completely, although in the tools we write a '$*$'.

In some cases the sort of the result must be upgraded. For numbers, we have the explicit type conversion operations $A2B$ (pronounce $A$ to $B$) where $A, B \in \{Pos, Nat, Int, Real\}$. For instance, the expression $n-1$ has sort $\mathbb{Z}$, because $n$ can be zero. However, if it is known that $n$ is larger than 0, it can be retyped to $\mathbb{N}$ by writing $Int2Nat(n-1)$. These operators are generally only written in specifications intended for tools. In textual specifications we typically leave them out.

The reals $\mathbb{R}$ are a complex data type, because the number of reals that exist is uncountably infinite. Without going into detail, this means that we cannot describe reals with constructors, and that means that we do not have an induction principle on reals. The definition of reals that we give allow to denote the rational numbers only, although it would be possible to extend this data type to include operators such as the square root or the logarithm. Reals are important for timed behaviour as the moment an action can take place is expressed by a real.

**Exercise 3.2.4.** What are the sorts of the successor function $succ(\_)$, and what are the sorts of the predecessor function $pred(\_)$.

**Exercise 3.2.5.** Prove using the equations in appendix A that the numbers 0 and 1 are different.

**Exercise 3.2.6.** Calculate using the axioms in appendix A that $2|_1 = 0$.

| Operator | *Rich* |
|---|---|
| positive numbers | $\mathbb{N}^+$ $(1, 2, 3, \ldots)$ |
| natural numbers | $\mathbb{N}$ $(0, 1, 2, \ldots)$ |
| integers | $\mathbb{Z}$ $(\ldots, -2, -1, 0, 1, 2, \ldots)$ |
| reals | $\mathbb{R}$ |
| equality | $\_ \approx \_$ |
| inequality | $\_ \not\approx \_$ |
| conditional | $if(\_, \_, \_)$ |
| conversion | $A2B(\_)$ |
| less than or equal | $\_ \leq \_$ |
| less than | $\_ < \_$ |
| greater than or equal | $\_ \geq \_$ |
| greater than | $\_ > \_$ |
| maximum | $\max(\_, \_)$ |
| minimum | $\min(\_, \_)$ |
| absolute value | $\mathrm{abs}(\_)$ |
| negation | $-\_$ |
| successor | $succ(\_)$ |
| predecessor | $pred(\_)$ |
| addition | $\_ + \_$ |
| subtraction | $\_ - \_$ |
| multiplication | $\_ \cdot \_$ |
| integer div | $\_ \,\mathrm{div}\, \_$ |
| integer mod | $\_ \vert \_$ |
| exponentiation | $\_^{\_}$ |
| real division | $\_ / \_$ |
| rounding operators | $floor(\_), ceil(\_), round(\_)$ |

Table 3.3: Operations on numbers

| Operator | *Rich* |
|---|---|
| function application | $\_(\_, \ldots, \_)$ |
| lambda abstraction | $\lambda \_{:}D_0, \ldots, \_{:}D_n.\_$ |
| equality | $\_ \approx \_$ |
| inequality | $\_ \not\approx \_$ |
| less than or equal | $\_ \leq \_$ |
| less than | $\_ < \_$ |
| greater than or equal | $\_ \geq \_$ |
| greater than | $\_ > \_$ |
| conditional | $if(\_, \_, \_)$ |
| function update | $\_[\_ \rightarrow \_]$ |

Table 3.4: Lambda abstraction and function application

## 3.3 Function data types

Functions are objects in common mathematical use and they are very convenient for abstract modelling of data in behaviour. Therefore, it is possible to use function sorts in specifications. For example,

**sort** $F = \mathbb{N} \rightarrow \mathbb{N}$;
$G = \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$;
$H = \mathbb{R} \rightarrow F \rightarrow G$;

declares that $F$ is the sort of functions from natural numbers to natural numbers and $G$ is the sort of functions from $\mathbb{R}$ and $\mathbb{N}$ to $\mathbb{R}$. Functions of the complex sort $H$ map reals to functions from $F$ to $G$. Function types associate to the right. Hence, the sort $H$ equals $\mathbb{R} \rightarrow (F \rightarrow G)$. If the sort $(\mathbb{R} \rightarrow F) \rightarrow G$ were required, explicit bracketing is needed.

Functions can be made using lambda abstraction and application (see table 3.4). Lambda abstraction is used to denote functions. E.g.,

$$\lambda n{:}\mathbb{N}.n^2$$

represents a function of sort $\mathbb{N}$ to $\mathbb{N}$ that yields for each argument $n$ its square. The variable $n$ is said to be *bound* by lambda abstraction. A variable that is bound does not occur freely in a term. E.g., the variable $n$ is free in $n^2$, bound in $\lambda n{:}\mathbb{N}.n^2$ and both free and bound in $n^2 + \lambda n{:}\mathbb{N}.n^2$. In this last case the bound and free variables $n$ are different.

The function defined above can be applied to an argument by putting it directly behind the function. For instance

$$(\lambda n{:}\mathbb{N}.n^2)(4)$$

equals 16. It is common to drop the brackets around the argument of such a function as follows

$$(\lambda n{:}\mathbb{N}.n^2)4.$$

However, the syntax of mCRL2 requires these brackets. When functions require more arguments, brackets associate to the left. E.g., consider a function $f$ and terms $t$ and $u$, then the application $f(t)(u)$ is parsed as $(f(t))(u)$, and not as $f(t(u))$.

Lambda terms can have more variables. For example the function $f$ defined as $\lambda x, y{:}\mathbb{N}, b{:}\mathbb{B}.\ \mathit{if}(b, x, x{+}y)$ is a function of sort $\mathbb{N} \times \mathbb{N} \times \mathbb{B} \to \mathbb{N}$. The function $f$ must always be applied to three arguments simultaneously. It is not allowed to feed it with only a partial number of arguments, as in $f(3, 4)$. When it should be possible to provide the arguments one at a time, three lambda's are necessary, as in $g = \lambda x{:}\mathbb{N}.\lambda y{:}\mathbb{N}.\lambda b{:}\mathbb{B}.\mathit{if}(b, x, x{+}y)$. The type of this function is $\mathbb{N}{\to}\mathbb{N}{\to}\mathbb{B}{\to}\mathbb{N}$. In this case $g(3)$ is a function from $\mathbb{N}{\to}\mathbb{B}{\to}\mathbb{N}$, and $g(3)(4)(\mathit{true})$ is an expression of sort $\mathbb{N}$.

There are two conversion rules for lambda expressions. A lambda term $\lambda x{:}\mathbb{N}.t$ binds the variable $x$ in $t$ and the variable $y$ has exactly the same role in $\lambda y{:}\mathbb{N}.t[x{:=}y]$, if $y$ does not appear freely in $t$. Here, $t[x{:=}y]$ is the substitution operator that replaces each free occurrence of $x$ in $t$ by $y$. Renaming a bound variable is called $\alpha$-conversion, and it is generally denoted as

$$\lambda x{:}\mathbb{N}.t =_\alpha \lambda y{:}\mathbb{N}.t[x{:=}y].$$

If a lambda expression has more bound variables, then renaming any subset of these is also called $\alpha$-conversion.

When applying substitutions to lambda terms care must be taken that substituted variables do not accidentally get bound. Consider the term $\lambda x{:}D.y$ to which the substitution $[y{:=}x]$ is applied. We see $(\lambda x{:}D.y)[y{:=}x] = (\lambda x{:}D.x)$. The unbound $y$ is replaced by $x$ and becomes bound. This is wrong. Therefore, it is necessary when applying a substitution to a lambda term to first $\alpha$-convert the term such that there is no conflict between the bound variables and the variables in the substitution. In the case above the correct sequence should be $(\lambda x{:}D.y)[y{:=}x] =_\alpha (\lambda z{:}D.y)[y{:=}x] = \lambda z{:}D.x$.

The second conversion is called $\beta$-conversion. It corresponds to function application. A term $\lambda x{:}D.t$ applied to a term $u$ is equal to $t[x{:=}u]$, namely $t$ where $u$ is substituted for all occurrences of $x$. It is generally denoted by

$$(\lambda x{:}D.t)(u) =_\beta t[x{:=}u].$$

If there are more bound variables in a lambda expression, then $\beta$-conversion has the following shape requiring multiple substitutions of variables.

$$(\lambda x_1{:}D_1, \ldots, x_n{:}D_n.t)(u_1, \ldots, u_n) =_\beta t[x_1{:=}u_1, \ldots, x_n{:=}u_n].$$

**Example 3.3.1.** ★ Lambda abstraction and application are far more powerful than is apparent on first sight. They can be used to specify fundamental mathematical and computational concepts. Alonzo Church[2] represented numbers by lambda terms. The number $n$ is represented by a lambda term that applies a function $n$ times to an argu-

[2] Alonzo Church (1903-1995) developed the lambda calculus as a basic mechanism for calculations and showed the existence of undecidable problems.

ment.

$$\begin{array}{ll} 0 & \lambda f{:}D{\to}D.\lambda x{:}D.x \\ 1 & \lambda f{:}D{\to}D.\lambda x{:}D.f(x) \\ 2 & \lambda f{:}D{\to}D.\lambda x{:}D.f(f(x)) \\ 3 & \lambda f{:}D{\to}D.\lambda x{:}D.f(f(f(x))) \\ & \ldots \\ n & \lambda f{:}D{\to}D.\lambda x{:}D.f^n(x) \end{array}$$

Adding two Church numerals can be done by the lambda term

$$\lambda g_1, g_2{:}D{\to}D.\lambda f{:}D{\to}D.\lambda x{:}D.g_1(f)(g_2(f)(x)).$$

Evaluation is represented by $\beta$-conversion. So, adding one and two looks like:

$$\begin{array}{l} \lambda g_1, g_2{:}D{\to}D.\lambda f{:}D{\to}D.\lambda x{:}D. \\ \qquad g_1(f)(g_2(f)(x))(\lambda f{:}D{\to}D.\lambda x{:}D.f(x), \lambda f{:}D{\to}D.\lambda x{:}D.f(f(x))) =_\beta \\ \lambda f{:}D{\to}D.\lambda x{:}D. \\ \qquad g_1(f)(g_2(f)(x))[g_1{:=}\lambda f{:}D{\to}D.\lambda x{:}D.f(x), g_2{:=}\lambda f{:}D{\to}D.\lambda x{:}D.f(f(x))] = \\ \lambda f'{:}D{\to}D.\lambda x'{:}D. \\ \qquad g_1(f')(g_2(f')(x'))[g_1{:=}\lambda f{:}D{\to}D.\lambda x{:}D.f(x), g_2{:=}\lambda f{:}D{\to}D.\lambda x{:}D.f(f(x))] = \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.(\lambda f{:}D{\to}D.\lambda x{:}D.f(x))(f')((\lambda f{:}D{\to}D.\lambda x{:}D.f(f(x)))(f')(x')) =_\beta \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.(\lambda x{:}D.f(x)[f{:=}f'])((\lambda x{:}D.f(f(x))[f{:=}f'])(x')) = \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.(\lambda x{:}D.f'(x))((\lambda x{:}D.f'(f'(x)))(x')) =_\beta \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.f'(x)[x{:=}(\lambda x{:}D.f'(f'(x)))(x')] = \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.f'((\lambda x{:}D.f'(f'(x)))(x')) =_\beta \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.(f'(f'(f'(x)))[x{:=}x']) = \\ \lambda f'{:}D{\to}D.\lambda x'{:}D.f'(f'(f'(x'))). \end{array}$$

Function sorts are sorts like any other. This means that equality, its negation, comparison operators and an if-then-else function are also available for function sorts.

There is one specific operator for unary function sorts, namely the *function update operator*. For a function $f : D \to E$ it is written as $f[d{\to}e]$. It represents the function $f$ except if applied to value $d$ it must yield $e$. Using lambda notation the update operator can be defined by

$$f[d{\to}e] = \lambda x{:}D.\mathit{if}(x{\approx}d, e, f(x)).$$

The advantage of using a function update over explicit lambda notation is that the equations for function updates allow to effectively determine whether two functions after a number of updates represent the same function. This is not always possible when using the explicit lambda notation. Besides that the function update notation is easier to read.

**Example 3.3.2.** Suppose we want to specify a possibly infinite buffer containing elements of some sort $D$ where the elements are stored at specific positions. This can be done by declaring the buffer to be a function $\mathbb{N} \to D$. The definitions of the empty buffer, an insert and get function are straightforward.

**sort** $Buffer = \mathbb{N} \to D$;
**map** $default\_buffer : Buffer$;
$insert : \mathbb{N} \times D \times Buffer \to Buffer$;
$get : \mathbb{N} \times Buffer \to D$;
**var** $n{:}\mathbb{N}; d{:}D; B{:}Buffer$;
**eqn** $default\_buffer(n) = d_0$;
$insert(n, d, B) = B[n \to d]$;
$get(n, B) = B(n)$;

**Exercise 3.3.3.** Consider arrays containing booleans as a function from $\mathbb{N}$ to $\mathbb{B}$. Specify a function $get$ to get the boolean at index $i$ and another function $assign$ that assigns a boolean value to a certain position $i$ in the array.

**Exercise 3.3.4.★** The booleans true and false can be represented by the lambda expressions $\lambda x, y{:}D.x$ and $\lambda x, y{:}D.y$. Give a lambda term for an if-then-else function on Church numerals. Also provide a function $is\_zero$ that tests whether a Church numeral is equal to zero. Show that $is\_zero$ applied to zero is true, and $is\_zero$ applied to one is false.

## 3.4 Structured data types

Structured types, also called functional or recursive types, find their origin in functional programming. The idea is that the elements of a data type are explicitly characterised. For instance, an enumerated type $Direction$ with elements $up$, $down$, $left$ and $right$ can be characterised as follows:

**sort** $Direction = \mathbf{struct}\ up?isUp \mid down?isDown \mid left?isLeft \mid right?isRight$;

This says the sort $Direction$ has four constructors characterising different elements. The optional recognisers such as $isUp$ are functions from $Direction$ to $\mathbb{B}$ and yield true iff they are applied to the constructor to which they belong. E.g., $isUp(up) = true$ and $isUp(down) = false$.

It is possible to let the constructors in a structured sort depend on other sorts. Hence, pairs of elements of fixed sorts $A$ and $B$ can be declared as follows:

**sort** $Pair = \mathbf{struct}\ pair(fst{:}A, snd{:}B)$;

This says that any term of sort $Pair$ can be denoted as $pair(a, b)$ where $a$ and $b$ are data elements of sort $A$ and $B$. The functions $fst$ and $snd$ are so called projection functions. They allow to extract the first and second element out of a pair. They satisfy the equations:

$$fst(pair(a, b)) = a;$$
$$snd(pair(a, b)) = b;$$

Projection functions are optional, and can be omitted.

In structured sorts it is even possible to let sorts depend on itself. Using this, well-known recursive data types such as lists and trees can be constructed. A sort *Tree* for binary trees has the following minimal definition:

| Operator | *Rich* |
|---|---|
| constructor $i$ | $c_i(\_, \ldots, \_)$ |
| recogniser for $c_i$ | $is\_c_i(\_)$ |
| projection $(i, j)$, if declared | $pr_{i,j}(\_)$ |
| equality | $\_\approx\_$ |
| inequality | $\_\not\approx\_$ |
| less than or equal | $\_ \leq \_$ |
| less than | $\_ < \_$ |
| greater than or equal | $\_ \geq \_$ |
| greater than | $\_ > \_$ |
| conditional | $if(\_, \_, \_)$ |

Table 3.5: Operators for structured types

**sort** $Tree = \mathbf{struct}\ leaf(A) \mid node(Tree, Tree)$;

By adding projection and recogniser functions it looks like:

**sort** $Tree = \mathbf{struct}\ leaf(val{:}A)?isLeaf \mid node(left{:}Tree, right{:}Tree)?isNode$;

As an example we define a function $HE$, short for 'holds everywhere', that gets a function of sort $A \to \mathbb{B}$ and checks whether the function yields true in every leaf of the tree.

**map** $HE : (A \to \mathbb{B}) \times Tree \to \mathbb{B}$;
**var** $f : A \to \mathbb{B}$;
$t, u : Tree$;
$a : A$;
**eqn** $HE(f, leaf(a)) = f(a)$;
$HE(f, node(t, u)) = HE(f, t) \wedge HE(f, u)$;

The following definition of sort $Tree$ allows the definition of operation $HE$ without pattern matching.

**var** $f : A \to \mathbb{B}$;
$t : Tree$;
**eqn** $HE(f, t) = if(isLeaf(t), f(val(t)), HE(f, left(t)) \wedge HE(f, right(t)))$;

This last definition has as disadvantage that the equation is not a terminating rewrite rule. Under certain circumstances, tools will have difficulties dealing with such an equation.

The general form of a structured type is the following, where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$ with $1 \leq i \leq n$:

$$\begin{aligned}&\mathbf{struct}\ c_1(pr_{1,1} : A_{1,1},\ \ldots, pr_{1,k_1} : A_{1,k_1})?isC_1\\ &\quad \mid c_2(pr_{2,1} : A_{2,1},\ \ldots, pr_{2,k_2} : A_{2,k_2})?isC_2\\ &\qquad\quad \vdots\\ &\quad \mid c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?isC_n;\end{aligned}$$

| Operator | *Rich* |
|---|---|
| construction | $[\_, \ldots, \_]$ |
| element test | $\_ \in \_$ |
| length | $\#\_$ |
| cons | $\_ \triangleright \_$ |
| snoc | $\_ \triangleleft \_$ |
| concatenation | $\_ +\!\!+ \_$ |
| element at position | $\_ \cdot \_$ |
| the first element of a list | $head(\_)$ |
| list without its first element | $tail(\_)$ |
| the last element of a list | $rhead(\_)$ |
| list without its last element | $rtail(\_)$ |
| equality | $\_ \approx \_$ |
| inequality | $\_ \not\approx \_$ |
| less than or equal | $\_ \leq \_$ |
| less than | $\_ < \_$ |
| greater than or equal | $\_ \geq \_$ |
| greater than | $\_ > \_$ |
| conditional | $if(\_, \_, \_)$ |

Table 3.6: Operations on lists

This declares $n$ constructors $c_i$, projection functions $pr_{i,j}$ and recognisers $isC_i$. All names have to be chosen such that no ambiguity can arise. The operations in table 3.5 are available after declaring the sort above. For the comparison operators the first function in a struct is the smallest and the arguments are compared from left to right. The precise definition of structured sorts is given in section A.10.

**Exercise 3.4.1.** Define the sort $Message$ that contains message frames with a header containing the type of the message ($ack$, $ctrl$, $mes$), a checksum field, and optionally a data field. Leave the data and checksums unspecified.

## 3.5 Lists

Lists, where all elements are of sort $A$, are declared by the sort expression $List(A)$. The operations in table 3.6 are predefined for this sort. Lists consist of constructors $[\,]$, the empty list, and $\triangleright$, putting an element in front of a list. All other functions on lists are internally declared as mappings.

Lists can also be denoted explicitly, by putting the elements between square brackets. For instance for lists of natural numbers $[1, 5, 0, 234, 2]$ is a valid list. Using the . operator, an element at a certain position can be obtained where the first element has index 0 (e.g., $[2, 4, 1].1$ equals $4$). The concatenation operator $+\!\!+$ can be used to append one list to the end of another. The $\triangleleft$ operator can be used to add an element to the end of a list. So, the lists $[a, b]$, $a \triangleright [b]$, $[a] \triangleleft b$ and $[a] +\!\!+ [] \triangleleft b$ are all equivalent.

| Operator | *Rich* | `Plain` |
|---|---|---|
| set enumeration | $\{\_,\ldots,\_\}$ | `{ _,...,_ }` |
| bag enumeration | $\{\_:\_,\ldots,\_:\_\}$ | `{ _:_,...,_:_}` |
| comprehension | $\{\_:\_ \mid \_\}$ | `{ _:_ | _ }` |
| element test | $\_ \in \_$ | `_ in _` |
| bag multiplicity | $count(\_,\_)$ | `count(_,_)` |
| subset/subbag | $\_ \subseteq \_$ | `_ <= _` |
| proper subset/subbag | $\_ \subset \_$ | `_ < _` |
| union | $\_ \cup \_$ | `_ + _` |
| difference | $\_ - \_$ | `_ - _` |
| intersection | $\_ \cap \_$ | `_ * _` |
| set complement | $\overline{\_}$ | `!_` |
| convert set to bag | $Set2Bag(\_)$ | `Set2Bag(_)` |
| convert bag to set | $Bag2Set(\_)$ | `Bag2Set(_)` |
| equality | $\_ \approx \_$ | `_ == _` |
| inequality | $\_ \not\approx \_$ | `_ != _` |
| less than or equal | $\_ \leq \_$ | `_ <= _` |
| less than | $\_ < \_$ | `_ < _` |
| greater than or equal | $\_ \geq \_$ | `_ >= _` |
| greater than | $\_ > \_$ | `_ > _` |
| conditional | $if(\_,\_,\_)$ | `if(_,_,_)` |

Table 3.7: Operations on sets and bags

The precise equations for lists are given in appendix A.

**Exercise 3.5.1.** Specify a function $map$ that gets a function and applies it to all elements of a given list.

**Exercise 3.5.2.** Specify a function $stretch$ that given a list of lists of some sort $D$, concatenates all these lists to one single list.

**Exercise 3.5.3.** Define an $insert$ operator on lists of some sort $D$ such that the elements in the list occur at most once. Give a proof that the insert operation is indeed correct.

## 3.6 Sets and bags

Mathematical specifications often use sets or bags. These are declared as follows (for an arbitrary sort $A$):

**sort** $D = Set(A)$;
$B = Bag(A)$;

An important difference between lists and sets (or bags) is that lists are inherently finite structures. It is impossible to build a list of all natural numbers, whereas the set of all

natural numbers can easily be denoted as $\{n{:}\mathbb{N} \mid true\}$. Similarly, the infinite set of all even numbers is easily denoted as $\{n{:}\mathbb{N} \mid n|_2{\approx}0\}$. The difference between bags and sets is that elements can occur at most once in a set whereas they can occur with any multiplicity in a bag.

The empty set is represented by an empty set enumeration $\{\}$. A set enumeration declares a set where each element can at most occur once. So, $\{a, b, c\}$ declares the same set as $\{a, b, c, c, a, c\}$. In a bag enumeration the number of times an element occurs has to be declared explicitly. So e.g., $\{a{:}2, b{:}1\}$ declares a bag consisting of two $a$'s and one $b$. Also $\{a{:}1, b{:}1, a{:}1\}$ declares the same bag. The empty bag is represented by the empty bag enumeration $\{:\}$.

A set comprehension $\{\, x{:}A \mid P(x) \,\}$ declares the set consisting of all elements $x$ of sort $A$ for which predicate $P(x)$ holds, i.e., $P(x)$ is an expression of sort $\mathbb{B}$. A bag comprehension $\{\, x{:}A \mid f(x) \,\}$ declares the bag in which each element $x$ occurs $f(x)$ times, i.e., $f(x)$ is an expression of sort $\mathbb{N}$.

**Exercise 3.6.1.** Specify the set of all prime numbers.

**Exercise 3.6.2.** Specify the set of all lists of natural numbers that only contain the number $0$. This is the set $\{[], [0], [0, 0], \ldots\}$. Also specify the set of all lists with length $2$.

## 3.7 Where expressions and priorities

*Where expressions* are an abbreviation mechanism in data expressions. They have the form $e$ **whr** $a_1{=}e_1, \ldots, a_n{=}e_n$ **end**, with $n \in \mathbb{N}$. Here, $e$ is a data expression and, for all $1 \leq i \leq n$, $a_i$ is an identifier and $e_i$ is a data expression. Expression $e$ is called the body and each equation $a_i = e_i$ is called a *definition*. Each identifier $a_i$ is used as an abbreviation for $e_i$ in $e$, even if $a_i$ is already defined in the context. An identifier $a_i$ is not used in any of the expressions $e_j$, $1 \leq j \leq n$. As a consequence, the order in which the definitions in a where expression occur is irrelevant.

**Example 3.7.1.** The expression $(n$ **whr** $n{=}m, m{=}3$ **end**$)$ **whr** $m{=}255$ **end** is equal to $255$. The expression $(f(n, n)$ **whr** $n{=}g(m, m)$ **end**$)$ **whr** $m{=}h(p, p)$ **end** is equal to $f(g(h(p, p), h(p, p)), g(h(p, p), h(p, p)))$.

For the construction of data terms the following priority rules apply. The prefix operators have the highest priority, followed by the infix operators, followed by the lambda operator together with universal and existential quantification, followed by the where clause. The precise priority rules can be found in appendix C where the syntax of constructs in mcrl2 are given, including data expressions, together with priority and associativity rules.

## 3.8 Historical notes

Our approach to data type specification has its root in the field of universal algebra, to which [40] gives a good introduction. Universal algebra has been adopted extensively

for formalisation of abstract data types. A standard textbook for abstract data types is [118]. The data type specification language developed for mCRL2 is rather complex and contains ingredients (e.g., multi-sorted signature with sub-sorting and higher-order functions) that do not occur simultaneously in other formalisms for abstract data types.

Earlier data type specification languages in the context of process calculi, such as those of PSF [127], $\mu$CRL [33], and LOTOS [106], lacked the practicality of built-in standard data types as in mCRL2. A modern data type specification language that has some of these features is CASL [140], which is incorporated in the process-algebraic language CSP-CASL [158]. Other examples are the data type languages of E-LOTOS [52] and LOTOS-NT [165]. We defer the discussion on the historical developments in the semantics of abstract data types to chapter 15, where the semantics of all formalisms used in this book, including the data specification language, are defined precisely.