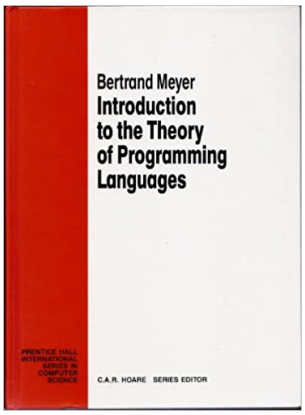

2 Mathematical background

Mathematicians are like Frenchmen: whatever you say to them they translate into their own language and forthwith it is something entirely different…

Goethe

This chapter introduces the essential mathematical concepts needed for the rest of the presentation, and some of the basic conventions used in this book to express formal specifications.

These are mostly conventions borrowed from standard mathematics, augmented by some notations specifically designed for the modeling of programming language concepts. Since it is convenient to have a name, the resulting metalanguage will be called **Metanot**.

2.1 CONVENTIONS AND NOTATIONS

2.1.1 Typographical and naming conventions

Any notation that belongs to either a programming language or the Metanot metalanguage will be in *italics*, with some keywords in **boldface**.

Names of sets (for example syntactic domains, as introduced in the next chapter) will normally start with an upper-case letter, for example *State*; names of members of these sets are in lower-case, for example *initial_state*. Function names will also be in lower case (although functions are formally defined as sets below).

When set names are words borrowed from ordinary English, they will as a rule be in the singular; that is, when a set of "states" is needed (chapters 4, 6, 7, 10), it will be called *State* rather than *States*; this is in part because it is appropriate to introduce a member of this set, say *s*, by writing

s: *State*

which reads naturally as "*s* is a *State*".

2.1.2 Meta-comments

In a formal definition expressed in Metanot, any sequence of characters beginning with two dashes (--) and extending to the end of the line is a comment, which is not part of the definition (this is another notation borrowed from programming languages, in this case Ada and Eiffel).

Comments may need to refer to identifiers (from Metanot or from programming languages) that, as indicated above, are written in italics. The names used as identifiers may be words of everyday language. To avoid any confusion, plain comment text is in roman font, as in the following (somewhat extreme) example:

```
-- The value of value is equal to the value of equal
value = equal
```

2.1.3 Definition *vs.* equality

The equal sign = is commonly used with two different meanings. Consider the following mathematical statement:

> Let a, b, c be real numbers. Let $D = b^2 - 4ac$. If $D = 0$, then the second-degree polynomial $ax^2 + bx + c$ has two equal roots.

The two equal signs in this extract play quite different roles. The first denotes a definition: it introduces a new object, D, in terms of previously introduced ones. The second is a relational operator, applied to two operands D and 0, and yielding a result that may be true or false depending on the value of D.

In precise formal specifications, we cannot tolerate such possible confusions, and need distinct symbols for distinct operators. In Metanot the equality symbol = will serve exclusively as relational operator (the second case). For "is defined as", Metanot will follow a widely established convention by using the symbol $\triangleq$. With this notation, the above mathematical statement, properly rewritten, becomes

> Let a, b, c be real numbers. Let $D \triangleq b^2 - 4ac$. If $D = 0$, then the second-degree polynomial $ax^2 + bx + c$ has two equal roots.

The definition symbol $\triangleq$ is the appropriate one for function definitions, such as the definition of $f(p)$, using a **case** expression, in 1.4.3. Since we will define many functions in the rest of this book, $\triangleq$ will be a common sight.

If you ever hesitate on whether to use = or $\triangleq$ in a particular case, the following guidelines should provide an immediate answer:

- If you may replace the symbol by some other relational operator (such as $\leq$ on numbers, the "and" operator $\wedge$ on booleans, or the subset operator $\subseteq$ which will be introduced shortly), and still yield a formula that makes sense — perhaps a *wrong* formula, but one that means something! —, then you need the relational operator =; otherwise you should use the definition operator $\triangleq$. For example, the first equal sign above serves to define D, so replacing it by $>$ or $\leq$ would be absurd; but the second expresses a condition that could be replaced with $D > 0$ and still yield a meaningful sentence.
- If reversing the order of the two operands yields a meaningless sentence, then you have a definition, not an equality. This applies to the first operator in our example, since if we change it to read "Let $b^2 - 4ac = D$" it stops making any sense.
- Expressions such as "if...", "such that..." and the like call for the relational operator =; immediately after "Let x...", you want the definition operator $\triangleq$.

The distinction between the = and $\triangleq$ operators is not unlike that between equality and assignment in programming languages. The Algol-Pascal-Ada-Eiffel line of languages reserves = for equality and uses := for assignment; this is closer to standard mathematical practice than the Fortran-C-C++-Java convention of writing = for assignment and having some special symbol (*.EQ.* or ==) for equality. The designers of PL/I thought they would satisfy everybody by using = in both cases and succeeded at least in providing generations of instructors with ready-made quizzes (guess what the instruction $A = B = C$ does).

Although such decisions only affect the "concrete syntax" of a language (contrasted in the next chapter with the deeper properties captured by "abstract syntax"), they contribute significantly to its elegance, or lack thereof.

2.1.4 Auxiliary variables in expressions

We can make a complex expression more readable by introducing local names to denote subexpressions. In Metanot, the notation **given** ... **then** ... **end** allows this. Following **given** are the definitions of one or more local names introduced as abbreviations for subexpressions; following the **then** is some expression e that may involve the local names. The value of the expression as a whole is the value of e, determined after substitution of the sub-expressions for each of the corresponding local names. For example, the expression

given

$$D \triangleq b^2 - 4 * a * c\ ;$$

$$denom \triangleq 2 * a$$

then

$$\frac{-b + \sqrt{D}}{denom}$$

end

is an expression denoting exactly the same thing as

$$\frac{-b + \sqrt{b^2 - 4 * a * c}}{2 * a}$$

The $\triangleq$ symbol is the appropriate one in the **given** clause, which says "let … be defined as ...".

The **given** ... construct is a mere notational device that adds nothing to the intrinsic power of the metalanguage. The "local names" are not variables in the sense in which this word is used in programming; a "definition" such as $x \triangleq x + 1$ would be just as meaningless in a **given** ... clause as it is in mathematics. More generally, if the clause contains a definition

$$x \triangleq e$$

then expression e may only involve independently defined objects, or objects introduced by a previous definition of the same clause. This is the principle of **non-creativity** of definitions: a definition introduces new names for existing objects or properties; it does not introduce new objects or properties. Chapter 8, devoted to recursion, will examine the non-creativity principle in more detail, and reveal that you may actually use certain kinds of recursive definitions (definitions in which e seems to involve x) without violating the principle. Until then we have to be quite strict, limiting what may appear in the right-hand side of a definition to:

- Well-known mathematical objects, such as integers.
- Items introduced as the left-hand side of *previous* definitions observing the same rules.

The extensions of chapter 8 do not actually changes these rules; they simply allow, for convenience, certain kinds of definition that seem to be recursive, but can easily be reformulated in non-recursive terms.

2.2 PROPOSITIONAL AND PREDICATE CALCULUS

Metanot uses standard boolean operators. If a and b have boolean values (true or false), then:

- $a \wedge b$, read as "a and b", is true if and only if both a and b are true.
- $a \vee b$, read as "a or b", is false if and only if both a and b are false.
- $\neg a$, read as "not a", is true if and only if a is false.
- Boolean implication uses the symbol $\Longrightarrow$; the implication $a \Longrightarrow b$, read as "a implies b", is defined as $\neg a \vee b$. In other words, $a \Longrightarrow b$ is true except if a is true and b false. Note in particular that $a \Longrightarrow b$ is true whenever a is false.

This property that false implies anything takes some getting used to when you start the study of logic, but it makes sense: "a implies b" means that if a is true, b is true as well; the only way to defeat this property (make it false) is to have a true and b false. If a is false the whole thing is true. For example, it is true that

"If I am the pope, two plus two equals 5"

because the only way you could invalidate this property would be to show that all these years I have been hiding my secret life for all these years I am indeed the pope, and (the easier part of the proof) that two plus two is not equal to 5. But since (personal disclosure) I am in fact not the pope, you can't invalidate this property, and it is indeed true, even though the right-hand side of the implication (2 + 2 = 5) is not. This is why we define $a \Longrightarrow b$ as $\neg a \vee b$: true if a is false or b is true.

Instead of the $\Longrightarrow$ symbol, most logic texts denote implication by the symbol $\supset$. But most people other than logicians are more comfortable with $\Longrightarrow$, which doesn't cause any particular problem. In addition, $\supset$ is also the symbol for the "superset" relation between sets, whereas implication has a natural interpretation — especially in theories developed in this book, such as denotational semantics — as a *subset* relation, so it could in fact be quite confusing.

We will also need the quantifiers of predicate calculus: $\forall$ (For all) and $\exists$ (There exists). If E is a set and P is a boolean-valued property (true or false), then:

- $\forall x : E \bullet P(x)$ means "All members of E, if any, satisfy P".
- $\exists x : E \bullet P(x)$ means "There exists at least one member of E that satisfies P".

These definitions of the quantifiers are informal. Here are more precise ones:

- $\forall x : E \bullet P(x)$ is false if and only if there is a member x of E such that $P(x)$ is false.
- $\exists x : E \bullet P(x)$ is true if and only if there is a member x of E such that $P(x)$ is true.

As an important consequence, any $\forall$ expression for which the set E is empty is true — regardless of what the property P is. In this case, any $\exists$ expression is false.

The notations introduced above for quantified expressions reflects an important constraint, enforced by Metanot to avoid certain theoretical difficulties. The dummy identifier, x in the above examples, is always constrained to range over a specific set, here E. In other words: you cannot express a property of the form "All x satisfy P" or "Some x satisfies P"; you must be more precise and express that all members x of a clearly specified set E, or some x in this set, satisfy P.

2.3 SETS

Elementary set theory provides much of the basis for formal semantic specifications. This section introduces the basic concepts and notations.

2.3.1 Basic sets

The specifications will use a few predefined sets.

The set of integers (positive, zero or negative) is written **Z**. The subset of **Z** containing only natural (non-negative) integers is written **N**. **R** denotes the set of real numbers.

The set of boolean values is written **B**; this is a set with only two members, *true* and *false*.

The set of character strings is written **S**; character strings are written in double quotes, as in "*A CHARACTER STRING*".

2.3.2 Defining sets

You may define a finite set by enumerating its members in braces, as in

$$\mathbf{B} \triangleq \{true, false\}$$

As a special case, { } is the empty set, more commonly written ∅.

A set definition of the above form is known as a definition by **extension**. Another way of defining a set (finite or infinite) is by **comprehension**, that is to say, by a characteristic property. For example, we may define the set of even integers as

$$Even \triangleq \{n : \mathbf{Z} \mid \exists\, k : \mathbf{Z} \bullet n = 2 * k\}$$

As with quantifiers, we always restrict the domain of the dummy variable (here n) to a set assumed to be well-defined, such as a basic set (**Z** in the case of *Even*) or a set that we have previously defined by extension or comprehension. This means that when we apply definition by comprehension it's always to define **subsets** of known sets.

Metanot borrows from Pascal and other programming languages (Ada, Eiffel) its notation for subsets of **Z** that are contiguous intervals: for a, b in **Z**, $a..b$ is the set of integers (if any) between a and b included. Formally:

$$a..b \triangleq \{i : \mathbf{Z} \mid a \leq i \wedge i \leq b\}$$

As implied by this definition, $a..b$ is an empty set if $b < a$.

For any set E, **P** (E) denotes the powerset of E, that is to say the set whose members are all subsets of E. For example, **P** (**Z**) is the set of all sets of integers; any interval $a..b$ as defined above belongs to that set.

2.3.3 Operators on sets and subsets

If E is a set and x is some mathematical object, x may or may not belong to E, also stated as "x is a **member** of E". The boolean expression

$$x \in E$$

has value true if and only if x is a member of E. For example, it is true that $-2 \in \mathbf{Z}$, but not that $-2 \in \mathbf{N}$. The notation $x \notin E$ is a synonym for $\neg (x \in E)$.

We may say "E **contains** x" to mean the same as "x is a member of E" (formally, $x \in e$).

The only information that a set E carries about a certain object x is whether or not x is a member of E. In particular, "how many times" x appears in E and the "order" of E's members are both meaningless notions. The situation is different with sequences, as will be seen below.

If A and B are two subsets of a given set E, then their union and intersection are defined respectively as

$$A \cup B \triangleq \{x : E \mid x \in A \vee x \in B\}$$

$$A \cap B \triangleq \{x : E \mid x \in A \wedge x \in B\}$$

The members of $A \cup B$ include any object that is a member of A or a member of B (or both); those of $A \cap B$ include every object that is a member of both A and B.

We will also need the generalization to infinite unions and intersections. If a subset E_i of a certain set E is given for every $i \in \mathbf{N}$, then we can define

$$\bigcup_{i : \mathbf{N}} E_i \triangleq \{x : E \mid \exists\, i \in \mathbf{N} \bullet x \in E_i\}$$

$$\bigcap_{i : \mathbf{N}} E_i \triangleq \{x : E \mid \forall\, i \in \mathbf{N} \bullet x \in E_i\}$$

The first of these sets contains every object that is a member of **at least one** of the E_i; the second contains every object that is a member of **every** E_i.

The operator $\subseteq$, **subset**, takes two subsets as arguments and yields a boolean value. It may be defined by

$$A \subseteq B \triangleq \{\forall x : A \bullet x \in B\}$$

that is to say, $A \subseteq B$ (read: "A is a subset of B") is true if and only if every member of A is also a member of B. Its variant $\subset$, **proper subset**, is defined as

$$A \subset B \triangleq (A \subseteq B) \wedge (\exists x : B \bullet x \notin A)$$

that is to say, all members of A are also members of B, but at least one member B is not a member of A. (With $\subseteq$ the two sets may be equal, but not with $\subset$.)

In line with the conventions introduced above, the operators $\cup$, $\cap$, $\subseteq$ and $\subset$ may not be applied to pairs of arbitrary set operands: in each case, both operands A and B must be subsets of a common set E.

A set may be finite or infinite. The expression **finite** E is true if and only if set A is a finite set; for example, **finite B** is true but **finite Z** is false. If E is finite, then **card** E (the cardinal of E) is the number of its members; for example, **card B** is 2.

2.4 SEQUENCES, PAIRS, CARTESIAN PRODUCT

Let X be a set. A **sequence** over X is an ordered list of members of X. We will write sequences in angle brackets, as in

<*monday, tuesday, wednesday, thursday, friday, saturday, sunday, monday*>

In particular, <> is the empty sequence. The terms "sequence" and "list" will be used interchangeably.

The value appearing at the i-th position in the sequence for some i is called the i-th **element** of the sequence.[1]

Sequences, unlike sets, are ordered: the sequence <*a, b*> is distinct from the sequence <*b, a*>. As the above example indicates, the same object can appear more than once; <*a, a*> is distinct from <*a*>.

The set of finite sequences over X is written X^*. The next chapter introduces a few operators on sequences.

---

[1] To avoid confusions, this book is careful in its use of a few words often employed interchangeably when less precision is required: a set has *members* and a sequence has *elements*; the word *object* is more general and applies to any well defined mathematical entity such as a set, a relation, a function or a member of some set. The next chapter will add the concept of the *specimens* of a syntactic type.

A generalization of the notion of sequence is the **tuple**.[2] Let X_1, X_2, .. .X_n be sets ($n \geq 0$) and X be their union. A tuple built from the X_i is a member of X^* – a sequence of elements in X — of length n, such that the i-th element of the sequence belongs to X_i ($1 \leq i \leq n$). For example, possible tuples built from **N**, **S** and **N** are

<3, "*Text*", 2>
<7, "*Other text*", 0>

The set of all tuples built from given sets is called the **cartesian product** of these sets and written

$$X_1 \times X_2 \times \ldots \times X_n$$

A tuple with two elements is called a **pair**. Being sequences, tuples and pairs are ordered: the pair $<a, b>$ is not the same as the pair $<b, a>$.

If we use the same set X for all the X_i, the resulting tuples are just sequences. In other words, there is a one-to-one correspondence between X^* and the set

$$\bigcup_{i\,:\,\mathbf{N}} X^i$$

where X^0 is {<>} (the set with one element, the empty sequence), X^1 is X, and X^{i+1} is $X \times X^i$ (for $i \geq 1$).

2.5 RELATIONS

Relations describe associations between objects. As used in this book, the word "relation" is a shorthand for "binary relation"; more general multiary relations (used for example for relational databases) will not be needed.

2.5.1 Definition

Consider two sets X and Y. A **relation** r between X and Y is a set of pairs

$$\{<a_1, b_1>, <a_2, b_2>, \cdots\}$$

where every a_i is a member of X and every b_i is a member of Y. X is called the source set of r and Y its target set. The following figure illustrates the finite relation

$$\mathit{finrel} \triangleq \{<x_1, y_1>, <x_1, y_2>, <x_3, y_2>, <x_4, y_2>, <x_4, y_5>\}$$

[2] Although some people pronounce the "u" of "tuple" as in "ruble", it seems that the correct model is "rubble".

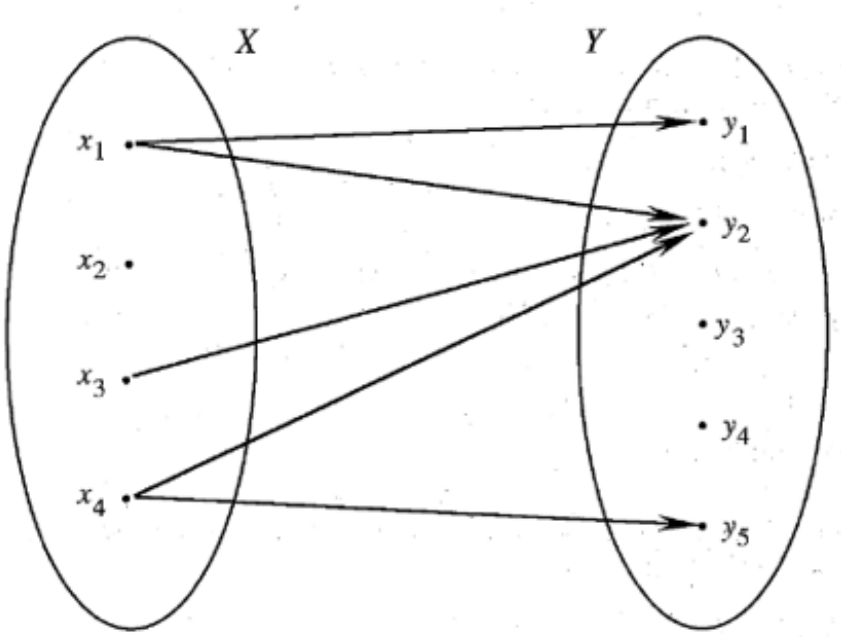

Figure 2.1: A relation

This way of viewing relations as sets may be new for you, since you may think of a relation as a property that links objects, such as the relation "is the mother of" between people, which links certain pairs of objects: Jill is the mother of Jane, Jane is the mother of John and so on. But it is not difficult to go from one viewpoint to the other: simply consider the pairs of elements linked by the relation, such as the pairs $<jill, jane>$ and $<jane, john>$. Now consider the set *Is_mother_of* of all such pairs,

$$Is\_mother\_of \triangleq \{<jill, jane>, <jane, john>, ...\}$$

that is to say, the set of all $<mother, child>$ pairs in our set of persons; it completely determines the relation, in the sense that m is the mother of c if and only if the pair $<m, c>$ is a member of the set *Is_mother_of*. So for us that's what a relation is: a set of pairs.

The definition of relation *finrel* was by extension. You may also define a relation by comprehension, as with the following relation between integers:

$$pm\_double \triangleq \{<m, n>: \mathbf{Z} \times \mathbf{N} \mid (m = 2 * n) \vee (m = -2 * n)\}$$

Relation *pm_double* is so named because the first element of each pair is "plus or minus the double" of the second. Elements include <0, 0>, <–6, 3>, <6, 3> and so on. Unlike *finrel*, relation *pm_double* is an infinite relation.

The set of binary relations between X and Y is written $X \leftrightarrow Y$ and defined as

$$X \leftrightarrow Y \triangleq \mathbf{P}(X \times Y)$$

In words: $X \leftrightarrow Y$ is the set whose members are all subsets of $X \times Y$, that is to say all sets of pairs whose first element is in X and whose second element is in Y.

2.5.2 Domain and range

The **domain** and **range** of a relation r, written **dom** r and **ran** r, are the sets of objects that appear as first and second elements, respectively, of at least one pair that is a member of r:

$$\textbf{dom}\ r \triangleq \{x: X \mid \exists\ y: Y \bullet \langle x, y\rangle \in r\}$$
$$\textbf{ran}\ r \triangleq \{y: Y \mid \exists\ x: X \bullet \langle x, y\rangle \in r\}$$

The domain of a relation is a subset of its source set and its range is a subset of its target set. For the above two relations:

$$\textbf{dom}\ \mathit{finrel} = \{x_1, x_3, x_4\}$$
$$\textbf{ran}\ \mathit{finrel} = \{y_1, y_2, y_5\}$$

$$\textbf{dom}\ \mathit{pm\_double} = \mathit{Even}$$ -- The set *Even* was defined on page 20

$$\textbf{ran}\ \mathit{pm\_double} = \mathbf{N}$$

2.5.3 Inverse and image

Let r be a relation in $X \leftrightarrow Y$. Its **inverse**, written r^{-1}, is another relation, a member of $Y \leftrightarrow X$, defined as follows:

$$r^{-1} \triangleq \{\langle y, x\rangle : Y \times X \mid \langle x, y\rangle \in r\}$$

In other words, r^{-1} contains a pair $\langle y, x\rangle$ if and only if r contains the pair $\langle x, y\rangle$. In the first example above, finrel^{-1} is

$$\{\langle y_1, x_1\rangle, \langle y_2, x_1\rangle, \langle y_2, x_3\rangle, \langle y_2, x_4\rangle, \langle y_5, x_4\rangle\}$$

The inverse of relation *pm_double* is *abs_half*; you can easily see what it is.

Let A be a subset of X, that is to say $A \in \mathbf{P}(X)$. The **image** of A through r is the subset of Y containing the objects related by r to at least one member of A. This image will be written $r(\bullet A\bullet)$; its precise definition is:

$$r(\bullet A\bullet) \triangleq \{y: Y \mid \exists\ x: A \bullet \langle x, y\rangle \in r\}$$

Taking the above relations as examples:

finrel (• {x_1, x_2, x_3}•) = {y_1, y_2}

pm_double (• {1, –1, 6, –6, 0, 14}•) = {0, 3, 7}

-- Here 1 and –1 are not in the domain of the relation
-- and so do not contribute to the image. Both 6 and –6
-- contribute the same object, –3.

abs_half (• **N**•) = *Even*

The following figure illustrates the first of these examples.

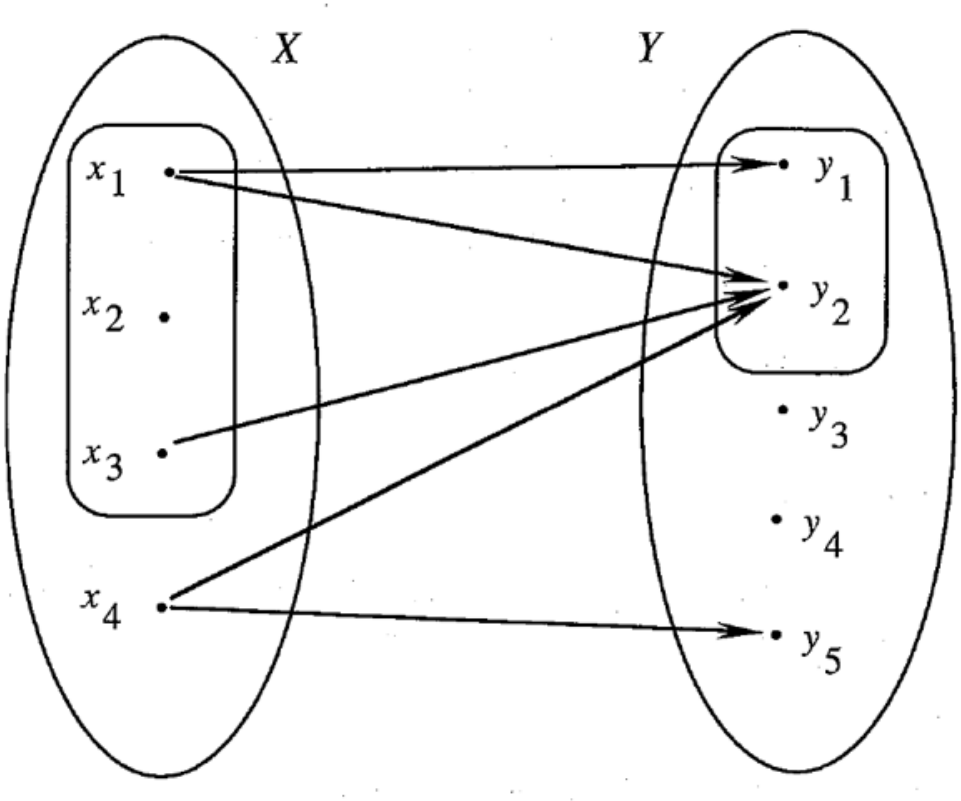

Figure 2.2: Image of a subset by a relation

Exercise 2.6 includes important properties of the image operator, which will be useful in the proof of consistency between axiomatic and denotational semantics (chapter 10).

2.6 FUNCTIONS

2.6.1 Definition

In general, given a relation r and an object $x \in X$, there may be zero, one or more objects $y \in Y$ such that the pair $<x, y>$ belongs to r (there may even be infinitely many such y). In relation *finrel*, for example, x_1 has two "buddies", x_3 has one and x_2 has none.

A relation such that there is **at most one** such y for every x is said to be **functional**. Relation *finrel* is not functional; it would be if we removed $<x_1, y_2>$ and $<x_4, y_2>$. Relation *pm_double* is functional, but its inverse *abs_half* is not (since, among other violations, it associates both –6 and 6 to the value *3*).

The following notation will denote the set of functional relations between X and Y (the reason for using this particular symbol will be explained shortly):

$$X \pfun Y$$

The set of functional relations is a subset of $X \rel Y$. We may define it precisely as

$$X \pfun Y \triangleq \{r : X \rel Y \mid \forall x : X \bullet \textbf{given } \mathit{Image\_of\_x} \triangleq r(\!|\{x\}|\!) \textbf{ then } \textbf{finite } \mathit{Image\_of\_x} \land (\textbf{card } \mathit{Image\_of\_x} \leq 1) \textbf{ end}\}$$

A functional relation is called a **function**. A function is an interesting kind of relation since we have the guarantee that, for any member of its domain, it will give us just exactly one member of the range. For example the relation *is_child_of_mother*, which holds between two persons if only if the second is the mother of the first, is a function.

2.6.2 Partial and total functions

If f is a function ($f \in X \pfun Y$) and x is a member of X, there may be zero or one y such that the pair $<x, y>$ belongs to f. If there is one, that is to say if

$$x \in \textbf{dom } f,$$

then you may refer to y as $f(x)$. A function $f \in X \pfun Y$ defined for all $x \in X$, in other words such that

$$\textbf{dom } f = X$$

is said to be a **total** function. The set of total functions from X to Y will be written

$$X \rightarrow Y$$

An equivalent definition is to say that a function is total if and only if its domain is equal to its source set. (In the general case, as we have seen, the domain is a subset of the source set.)

A function that is not total — in other words, some members of its source set are not in its domain — is said to be **partial**. Note that whenever the discussion uses the word "function" without further qualification, it refers to both total and partial functions. The bar across the arrow in the symbol $\nrightarrow$ serves as a reminder that f may be partial, in which case $f\ (x)$ is not defined for some members x of X.

2.6.3 Finite functions

Because in practice computers deal with finite information — such as the content of their memory — we will often encounter functions guaranteed to have a finite domain. They are called finite functions. (Some authors prefer the term "finite mapping".) The set of finite functions from X to Y will be written $X \nrightarrow\!\!\!\!\!\!+\ Y$, defined as

$$X \nrightarrow\!\!\!\!\!\!+\ Y \quad \triangleq \quad \{f : X \nrightarrow Y \mid \textbf{finite dom}\ \ f\}$$

To summarize the three conventions just seen:

- $\rightarrow$ is the symbol used for sets of total functions. If f is a total function you may use $f\ (\mathrm{x})$ without fear of writing something be meaningless, writing something meaningless, since this always denotes a value for any x of the source set.
- $\nrightarrow$ indicates possibly partial functions; the bar reminds you to exercise care in using such a function, since $f\ (\mathrm{x})$ is only defined for $x \in \textbf{dom}\ f$.
- $\nrightarrow\!\!\!\!\!\!+$ indicates finite functions, the double bar reminding you that they can be even "more" partial, since $f\ (\mathrm{x})$ only makes sense for a finite set of possible x.

Finite functions are particularly important for the modeling of programming concepts because they are the only functions that can be entirely represented in the memory of a computer: if a function has an infinite domain, there is no way you can ever hope to see the result of a complete computation of the function.

2.6.4 Defining functions by extension

One way to define a finite function is to indicate the value it takes for every possible argument in its domain. This is similar to defining a finite set by extension (page 20).

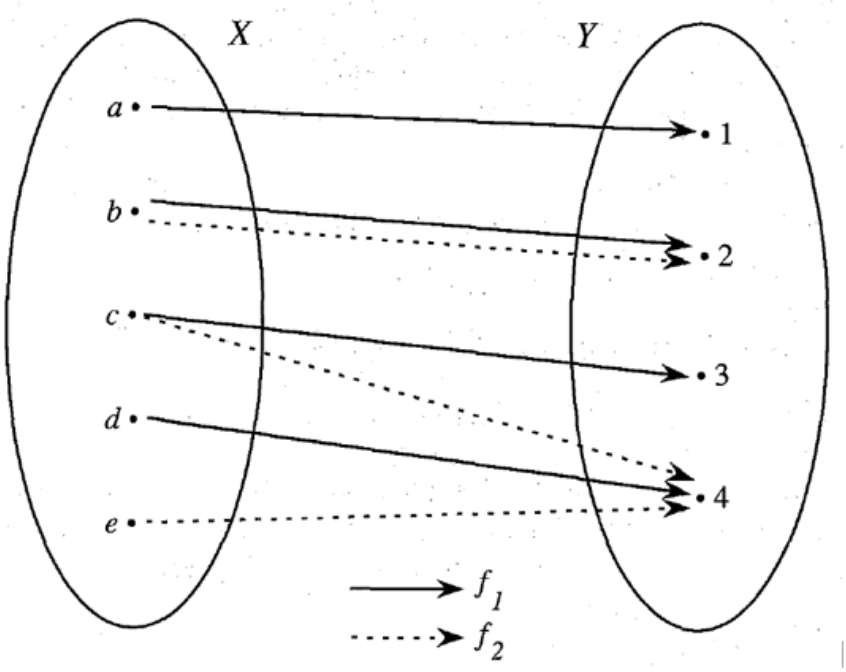

Figure 2.3: Two functions

To define a finite function by extension, we will simply list its constituent pairs. The preceding figure shows two functions both in $\{a, b, c, d, e\} \pfun \mathbf{N}$; we may define them as

$$f_1 \triangleq \{\langle a, 1\rangle, \langle b, 2\rangle, \langle c, 3\rangle, \langle d, 4\rangle\}$$

$$f_2 \triangleq \{\langle b, 2\rangle, \langle c, 4\rangle, \langle e, 4\rangle\}$$

The order in which you list the <*argument, result*> pairs doesn't matter, but all the *argument*s must be different if the definition is to yield a function rather than just a relation.

2.6.5 Defining functions by expressions

We saw (page 20) that it is possible to define a set not only by extension — by listing all its members — but also by comprehension, through a characteristic property of its members. This was also applicable to relations. Definitions by comprehension is particularly interesting for functions. This is the well-known technique of defining a function by an expression involving its formal arguments; to obtain the function's value for any actual argument values, it then suffices to substitute these values for the formal arguments in the expression. A typical example is

[2.1]

$$square: \mathbf{R} \longrightarrow \mathbf{R}$$

$$square\ (x) \triangleq x * x$$

This definition means that to get the value of *square* (*a*) for any real number a, you substitute a for x on the right-hand side of the $\triangleq$ sign, getting $a * a$. In contrast with definition by extension, such a definition by comprehension applies to infinite as well as finite functions.

The formal basis for this technique is known as **lambda calculus**; we will study it in detail in chapter 5. For the time being, definitions of the above form [2.1] will be considered clear enough. To make it absolutely obvious what the source and target sets and the domain of any function are, every function definition will consist of the following two or three steps. First you must give the source and target sets of the function under one of the forms

$f: X \longrightarrow Y$ -- For a total function

f: $X \nrightarrow$ Y -- For a possibly partial function

$f: X \pfun Y$ -- For a finite function

Then (in the second and third cases only) you must specify the domain:

dom $f \triangleq \{x \in X \mid$ … Some boolean-valued expression on x ...$\}$

These two indications (one for a total function) constitute the function's **signature**.

Finally, you must in all cases specify the value that the function yields for an arbitrary member of its domain, as was done above for *square*:

$f\ (x) \triangleq$ … Some value in Y …

using the "is defined as" symbol. For clarity it is almost always desirable, in this last part of the definition, to repeat the sets to which the arguments belong, as in

$f\ (x: X) \triangleq$ … Some value in Y …

which imitates argument type declarations in the routine headings of programming languages (such as Pascal, Ada, Eiffel).

Be careful not to confuse

$f: X \longrightarrow Y$

which gives the signature of *f*, declaring *f* to be some total function from *X* to *Y*, with

$S \triangleq X \longrightarrow Y$

which defines S as the set of all total functions from X to Y.

2.7 OPERATIONS ON FUNCTIONS

Several operations will prove useful on functions: we may define the intersection, "overriding union" and composition of two functions, the restriction of a function to a subset of its source set, the quotient of a set by a predicate (a function with a boolean result). We must also study what becomes of the notions of inverse and image, originally defined for relations, when we apply them to functions.

2.7.1 Intersection

We have seen that functions are a special kind of relations, themselves a special case of sets. Since functions are sets, we may define the *intersection* of two functions, itself a function; it is the set of [argument, result] pairs that belong to both functions.

For example, the intersection $f_1 \cap f_2$ of the two functions defined earlier is the function pictured next, whose domain has only one member:

$h \triangleq \{<b, 2>\}$

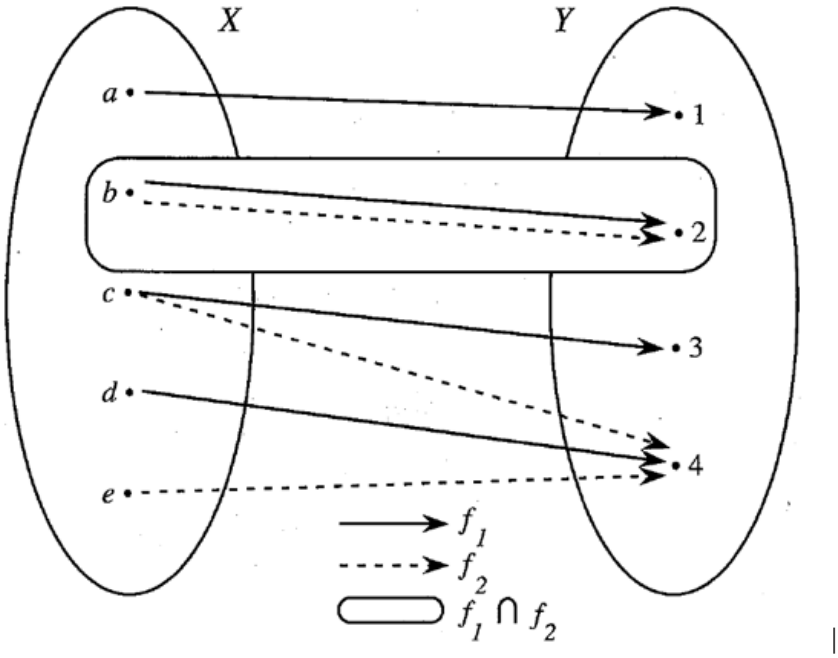

Figure 2.4: Function intersection

Here is the precise definition of function intersection in the general case:

Let $f, g : X \nrightarrow Y$;

Let $h \triangleq f \cap g$; then:

- **dom** $h = \{x : \textbf{dom}\ f \cap \textbf{dom}\ g \mid f(x) = g(x)\}$
- $h(x) = f(x) = g(x)$ for $x \in \textbf{dom}\ h$

That is to say: h is the function that yields the common value of f and g wherever the two functions "agree" (yield the same value).

2.7.2 Overriding union

After intersection, it is natural to turn our attention to the *union* of functions. Here, however, we must be a bit more careful: although the union of two functions is always defined as a set (since the functions themselves are sets, of which we can take the union), that set is not necessarily a function.

The problem arises wherever the two functions are both defined but disagree.

For example, f_1 and f_2 as defined above have conflicting values for the argument c, so that $f_1 \cup f_2$ is not a function. (The values for b do not cause a problem since the two functions coincide on that element.)

In the general case, $f \cup g$ is a relation, but not always a function.

We could define a $\cup$ operation on functions by restricting it to pairs of functions that agree on any common argument value, but that operation would not be very useful. Instead, we may define a union operation that is applicable to any pair of functions, and always yields a function, by making it **non-commutative**; in other words, it will not treat its operands symmetrically.

That non-commutative operation is the **overriding union**, for which Metanot uses the symbol $\cup\!\!\!\!\!-$. The convention in the functional expression $f \cup\!\!\!\!\!- g$ is that g overrides f wherever they disagree. As a reminder of this convention, the bar in the symbol $\cup\!\!\!\!\!-$ makes the union "lean" towards the second operand.

The operator " $\cup\!\!\!\!\!-$ " is defined more precisely as follows for f, g in $X \nrightarrow Y$. Calling

$h \triangleq f \cup\!\!\!\!\!- g$

then:

dom h = **dom** $f \cup$ **dom** g;

$h(x) = f(x)$ if $x \in \textbf{dom}\ f$ and $x \notin \textbf{dom}\ g$;

$h(x) = g(x)$ if $x \in \textbf{dom}\ g$.

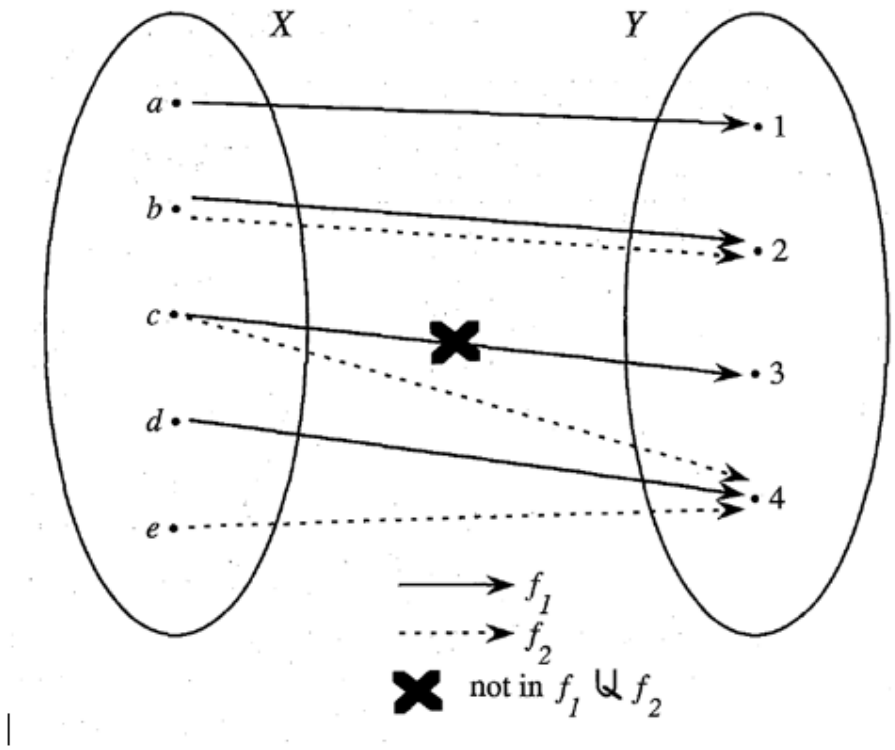

Figure 2.5: Overriding union

As an illustration of overriding union, consider again our two example functions. As shown on the figure above:

$$f_1 \cup f_2 = \{<a, 1>, <b, 2>, <c, 4>, <d, 4>, <e, 4>\}$$

2.7.3 Restriction

Another important operator on functions is restriction. The restriction of a function f to a subset A of its source set, written $f \setminus A$, is the same function as f, but with its domain restricted to A.

Taking one of the earlier functions as an example again:

$$f_1 \setminus \{a, b, e\} = \{<a, 1>, <b, 2>\}$$

The precise definition of restriction is as follows:

Let $f : X \nrightarrow Y$.

Let A: **P** (X) (in other words, A is a subset of X).

Then $h \triangleq f \setminus A$, the restriction of f to A, is the function

$h : A \nrightarrow Y$

such that

dom $h = A \cap$ **dom** f

and $h\ (a) = f\ (a)$ for $a \in$ **dom** h

Remembering that functions, being relations, are sets of pairs, we may also define $f \setminus A$ more concisely as

$f \cap (A \times Y)$

Using this technique we could define restriction for arbitrary relations, not just functions. But in this book we only need it for functions.

2.7.4 Composition

Composition is another operation defined for relations but used only for functions in this book. The following figure illustrates function composition:

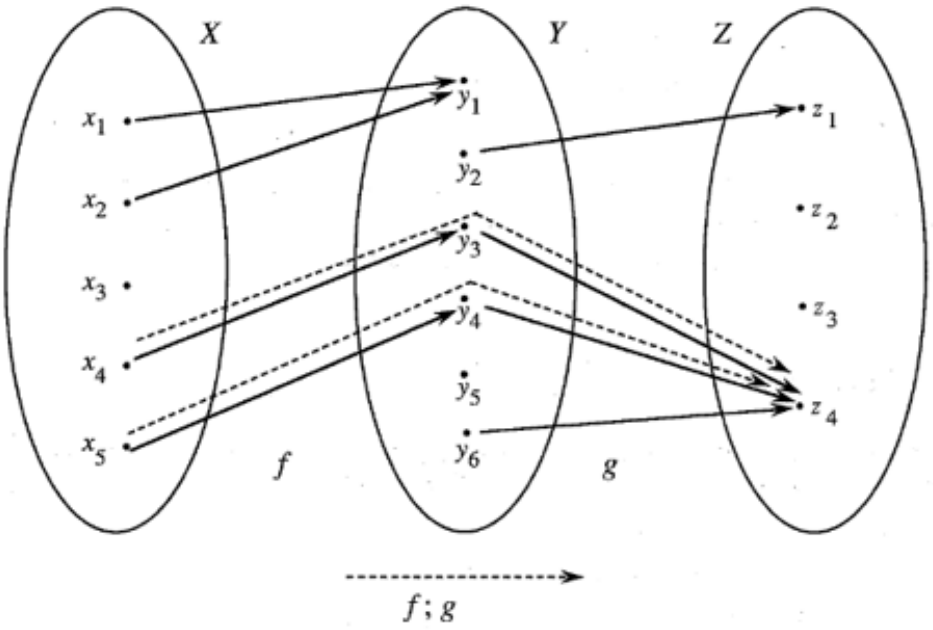

Figure 2.6: Function composition

For any two functions f and g of signatures

$f: X \nrightarrow Y$

$g: Y \nrightarrow Z$

their composition, written $f ; g$ (a notation borrowed from VDM), is the function

$h: X \nrightarrow Z$

such that

$$\textbf{dom}\ h = \{x : \textbf{dom}\ f \mid f(x) \in \textbf{dom}\ g\}$$

and, wherever h is defined, then

$$h(x) = g(f(x))$$

Rather than $f ; g$, the more common notation for composition is $g \circ f$. The semicolon suggests (in accordance with its use in ordinary written language) that f and g are applied in the order in which they are listed.

This use of the semicolon may be seen as an homage to the concrete syntax of the Algol family of languages: as will be seen in the next chapters, composition is the mathematical equivalent of statement sequencing. No confusion will result since programs extracts, when embodied in a Metanot description, are always written with abstract, not concrete syntax.

2.7.5 Infix operators as functions

One more convention will be useful for handling functions. In common mathematical (and programming) practice, many binary functions — functions of two arguments — use a so-called **infix** notation, with an operator between the two arguments rather than in front, as in $a + b$ rather than *plus* (a, b). Examples of such infix operators are +, –, ∗, /, or the just introduced composition operator ";".

It is often convenient to be able to talk about the functions associated with these operators without having to introduce special names and definitions (as in "let *plus* be the function such that, for any a and b, *plus* $(a, b) \triangleq a + b$").

The Metanot convention is borrowed from the Eiffel and Ada programming languages. If § is an infix binary operator, you may refer to the associated function through the notation

infix "§"

which you may also abbreviate to just "§" in expressions involving the function. Applications of the function to actual arguments a and b will use the usual infix notation: $a \S b$.

For example, the function

$$\textbf{infix}\ \text{"+"}: \quad \mathbf{N} \times \mathbf{N} \longrightarrow \mathbf{N}$$

is the addition on natural numbers, and for any sets X, Y, Z:

$$\textbf{infix}\ \text{";"}: \quad ((X \nrightarrow Y) \times (Y \nrightarrow Z)) \longrightarrow (X \nrightarrow Z)$$

is function composition over X, Y, Z. As the definition indicates, **infix** ";" is a function of two arguments, one a function from X to Y and the other a function from Y to Z; its result is a function from X to Z.

Defined in this way, function composition is a typical example of a **functional**, or function that admits other functions as arguments or results. Functionals are discussed further below.

2.7.6 Predicates and the quotient operator

A **predicate** on a set X is a total function

$$pred: X \rightarrow \mathbf{B}$$

from X to the set **B** of boolean values (*true* and *false*).

There is a natural connection between predicates on X and subsets of X. If A is a subset of X, then we may associate with A the predicate

$$characteristic_A : X \longrightarrow \mathbf{B}$$

such that

$characteristic_A\ (x)$ = *true* if $x \in A$, and
$characteristic_A\ (x)$ = *false* if $x \notin A$.

Function $characteristic_A$ is called the **characteristic function** of the subset A; it's the predicate that yields true for arguments within A, and false outside. The following figure illustrates it.

Conversely, if *pred* is a predicate on X, we may define the **quotient** of X by *pred*, written X / *pred*, as the subset of X containing only the objects that satisfy *pred*:

$$X / pred \triangleq \{x: X \mid pred\ (x)\}$$

For example, if X is a set of persons and *female* (x) is *true* if and only if x is a female, then X / *female* is the set of female members of X.

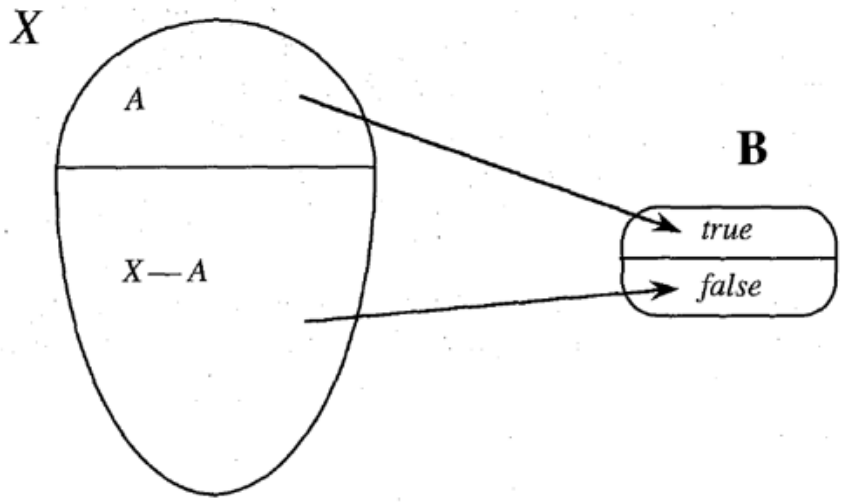

Figure 2.8: Characteristic function

The notions of characteristic function and predicate are inverse of each other, in the sense that for any subset A of X (that is, $A : \mathbf{P}\ (X)$):

$$X \ / \ characteristic_A = A$$

and for any predicate *pred* on X (that is, $pred : X \longrightarrow \mathbf{B}$):

$$characteristic_{X \ / \ pred} = pred$$

2.7.7 Inverse and image

The inverse and image operators, introduced earlier (page 25) for relations, apply to the special case of functions. If f is a function of signature $X \leftrightarrow Y$ and A is a subset of X, then:

- $f\ (\cdot A \cdot)$ is a subset of Y.
- f^{-1} is a relation (a member of $Y \leftrightarrow X$). It is not necessarily a function.

Exercise 2.6 covers several properties of images of subsets through functions.

2.8 FUNCTIONALS

A functional, also called a higher-order function, is a function that admits functions among its arguments, results, or both. The previous section already introduced several important examples, such as the composition operator, a function that takes two functions f and g and yields as a result another function, their composition $f \ ; g$.

The word "functional" as used in this book will only apply to **total** higher-order functions.[3]

It is important to familiarize yourself with the use of functionals, which play a major role in denotational semantics. Exercises 2.1 to 2.4 will help you master them.

2.8.1 Dispatching and parallel application

The discussion of recursion in chapter 8 uses two typical functionals, "dispatching" and "parallel application". They are **generic**, meaning that you can apply them to arbitrary sets U, V, X and Y (dispatching doesn't need V). We will express both of them through infix operators: **infix** "&" for dispatching and **infix** "#" for parallel application.

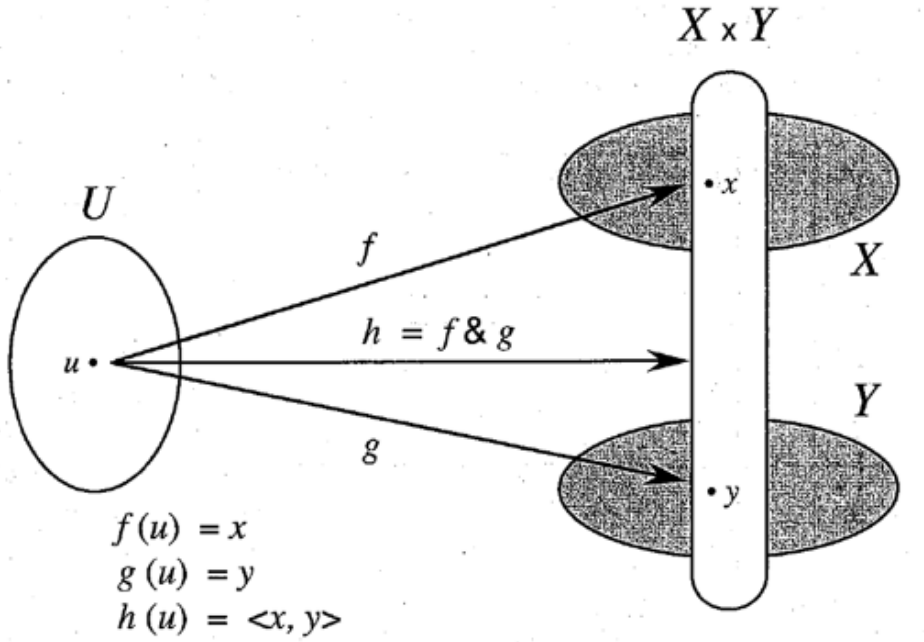

Figure 2.9: Dispatching

Dispatching, illustrated by the figure above, is the functional of signature

infix "&": $((U \nrightarrow X) \times (U \nrightarrow Y)) \rightarrow (U \nrightarrow (X \times Y))$

such that for any functions

f: $U \nrightarrow X$

g: $U \nrightarrow Y$

[3] Do not confuse "a functional", used here as a noun, with the adjective "functional" as used to characterize a relation (page 27).

f & g is the function

h: $U \rightarrow\!\!\!\!\!+\ X \times Y$

dom h = **dom** $f \cap$ **dom** g

$h\ (u) \triangleq \langle f\ (u),\ g\ (u)\rangle$

As shown on the preceding figure, f & g "dispatches" an argument u to X through f and at the same time to Y through g.

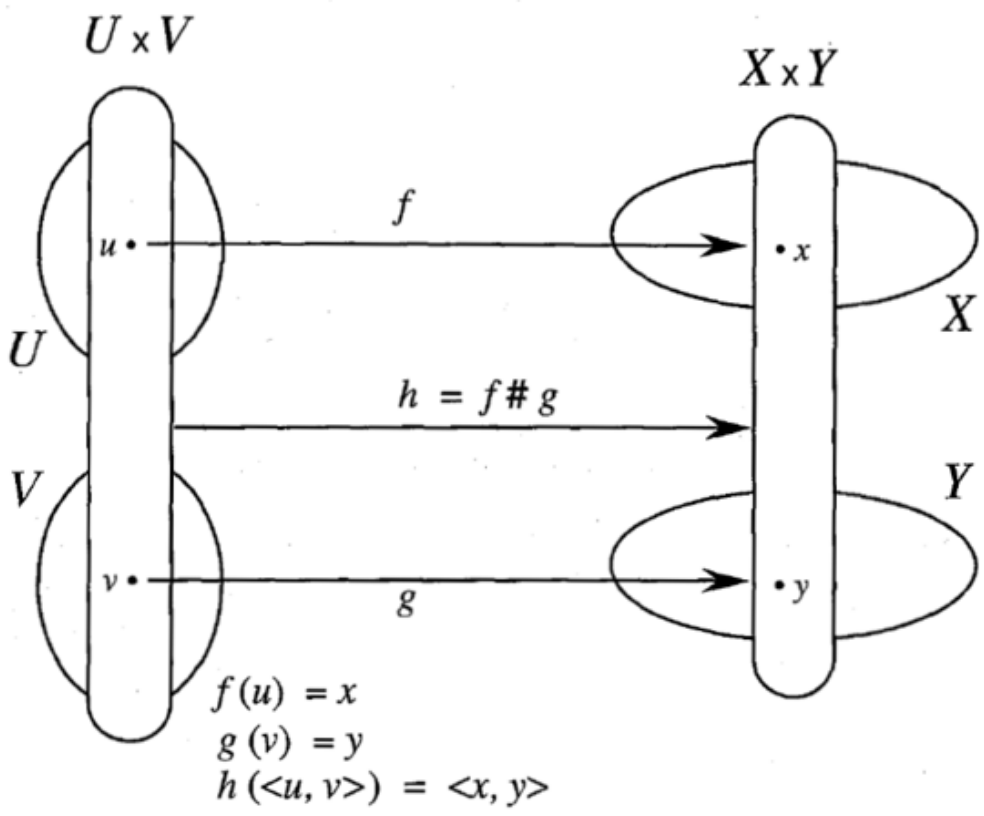

Figure 2.10: Parallel application

Parallel application, illustrated by the figure above, is the functional of signature

infix "#": $((U \rightarrow\!\!\!\!\!+\ V) \times (X \rightarrow\!\!\!\!\!+\ Y)) \rightarrow ((U \times V) \rightarrow\!\!\!\!\!+\ (X \times Y))$

such that for any functions

f: $U \rightarrow\!\!\!\!\!+\ X$

g: $V \rightarrow\!\!\!\!\!+\ Y$

f # g is the function

$h: U \times V \nrightarrow X \times Y$

dom h = **dom** $f \times$ **dom** g

$h\ (u, v) \triangleq \langle f\ (u), g\ (v) \rangle$

As shown on the figure, $f \# g$ applies f and g "in parallel" to an argument from U and an argument from V, yielding a pair in $X \times Y$.

Dispatching and parallel applications allow us to apply two functions together, either to the same argument or to two separate ones.

2.8.2 Currying

Another important functional is *curry* (named after the mathematician H.B. Curry, one of the major contributors to combinatory logic). This functional transforms any two-argument function into a one-argument function. For a number of discussions, it is convenient to consider that all functions take exactly one argument. So what if we are given a two-argument function? The trick is to consider it as a one-argument function, whose result is itself a *function* of one argument.

> This will extend to more than two arguments: we may view a three-argument function as a one-argument function returning a one argument function, itself taking a one-argument function as argument.

Here's how the transformation from two-argument to one-argument functions works. Considering total functions only, we define *curry* as follows. For any sets X, Y, Z, if f is a total function of signature

$f: X \times Y \rightarrow Z$

then *curry* (f) is a total function g of signature

$g: X \rightarrow (Y \rightarrow Z)$

such that for any $x : X$ and $y : Y$

$g\ (x)\ (y) = f\ (x, y)$

If this is the first time you see currying you may find this a bit strange, but it's really very simple. *curry* (f), called g above, really represents the same function as f — in the sense that it eventually yields the same value — but initially restricts its attention to only one argument. So whereas the result of applying f to two arguments x and y is a value $z : Z$, the result of applying g to one argument x is still a (total) function from Y to Z, whose result, for any y, is precisely $f\ (x, y)$, that is to say z.

Currying achieves **partial evaluation** of a function: in our example $g\ (x)$ is like f evaluated on one argument, x, and hence (since f takes two arguments) still needing one argument to yield a final value. That's why g, its curried version, is still a function.

The operator *curry* is itself a total function — what we have called a functional since it manipulates functions. Its signature is

$$curry\text{: } ((X \times Y) \rightarrow Z) \rightarrow (X \rightarrow (Y \rightarrow Z))$$

Exercise 2.4 explores properties of the *curry* functional and its generalization to arguments that are not necessarily total. You are particularly encouraged to do exercise 2.5, which applies currying to programming language processing tools such as compilers and interpreters.

2.9 STRUCTURAL INDUCTION

The last mathematical technique at which we need to take a look serves to define sets of complex objects and to prove properties of such objects. Known as structural induction, it is particularly useful for defining the concrete structure of languages.

2.9.1 An example: S-expressions in Lisp

The S-expressions of Lisp (serving as the basis for data and program structures in that language) provide a typical example of objects obtained by structural induction.

Here is a possible definition of S-expressions. It assumes a separately specified notion of *atom*; to keep the discussion simple, atoms will be identifiers built as sequences of letters and digits, beginning with a letter. (Actual Lisp atoms also include more possibilities, such as numbers.) We may then define an S-expression by structural induction as being one of the following:

- An atom.
- Of the form $(s_1 \ . \ s_2)$, where s_1 and s_2 are S-expressions.

The intuitive meaning of such a definition is clear: S-expressions cover atoms such as *atom1*, and more complex expressions written with parentheses and a dot, such as

(*atom1* **.** *atom2*)

((*atom1* **.** *atom2*) **.** ((*atom3* **.** (*atom4* **.** *atom5*) **.** *atom6*) **.** (*atom7* **.** *atom8*) **.** *atom9*))

2.9.2 General form of definitions by structural induction

More generally, a definition by structural induction defines a certain set S as being made of members that are either:

A • Members of one or more predefined sets (such as the set of atoms above), which we will call the **base sets**.

B • Deduced from previous elements of S through one or more well defined mechanisms (such as the above form with parentheses and a dot).

Such a definition has a clear mathematical interpretation: it means defining the set S as the union of an infinite family of sets

$$S \triangleq \bigcup_{i : \mathbf{N}} S_i$$

where we may define the S_i by induction of the ordinary, familiar kind, using integers:

- S_0 is the union of the base sets (the set of all objects obtainable under A).
- Each S_{i+1} is the set of all objects obtainable from one or more members of S_j, for one or more $j \in 1 .. i$, through the mechanisms introduced under B.

In the S-expression example, S_0 is the set of atoms; S_1 is the set of objects of the form $(a \ . \ b)$, where a and b are atoms; S_2 is the set of objects of the form $(a \ . \ b)$, where a and b are either atoms or in S_1; and so on.

Some objects may be in S_i for more than one i (actually, in the S-expression example, every S_i is a subset of S_{i+1} for positive i); this is fine since S is defined as the union of all S_i, so that it does not matter that a member of S may belong to several S_i.

Viewed in this way, structural induction is a straightforward application of the usual induction on integers, applied here to define inductively the sequence of sets S_i.

2.9.3 Proofs by structural induction

Sets defined by structural induction lend themselves to proofs organized along the same line. To prove by structural induction that all members of S satisfy a certain property p, you may successively prove that:

A • All members of the base sets satisfy p. (Base step.)

B • If a set of objects in various S_j satisfy p, any object built from them by any of the construction mechanisms that define S_{i+1} also satisfies p. (Induction step.)

The validity of this technique follows immediately from the validity of proofs by ordinary integer induction: a proof by structural induction simply amounts to proving by integer induction the property

$$p\ (i : \mathbf{N}) \ \triangleq \ \text{“All elements of } S_i \text{ satisfy } p\text{”}$$

As an example, let us prove by structural induction that every S-expression has an equal number of opening and closing parentheses. The proof contains two parts:

A • An atom has no parentheses, and so satisfies the property.

B • Consider two S-expressions e_1 and e_2, each satisfying the property. Let p_1 and p_2 be their respective numbers of opening parentheses; by assumption, these are also their numbers of closing parentheses. The construction mechanism given yields only one new S-expression from e_1 and e_2:

$$n \triangleq (e_1 \cdot e_2)$$

Counting parentheses in n, we find $p_1 + p_2 + 1$ left parentheses, and the same number of right parentheses. □

Definitions by structural induction are a simple case of *recursive* definitions, whose significance and mathematical properties will be explored in a much more general context in chapter 8. Structural induction proofs will also find a generalization there through the notion of *stable predicate*. Justifying structural induction within the more general theory is the subject of exercise 8.4.

2.10 BIBLIOGRAPHICAL NOTES

Many of the notations introduced in this chapter have their equivalents in the work on the VDM denotational specification method [Bjørner 1982] [Jones 1986]. Some come from an early version of Z [Abrial 1980].

The article by John Backus on functional programming [Backus 1978] describes high-level functional operators not unlike some of those used in this chapter and in the exercises below. However Backus' language, FP, includes only a fixed set of higher-level functional operators; new ones may not be defined in the language proper, but in a supporting notation called FFP. Languages that do permit definition of functions of an arbitrary level are Miranda [Turner 1985] and Haskell [Haskell Web].

EXERCISES

You may use lambda notation (chapter 5) to simplify some of the answers, but it is not indispensable.

Exercises 2.1 to 2.4 use **R**, the set of real numbers, and some also need the set $\mathbf{R}^\bullet$ of non-zero real numbers.

2.1 Properties of simple functions

Consider the following functions on real numbers :

square, the square function

inverse, the inverse function (*inverse* (*x*) ≙ *1/x*)

"+", "–", "∗", "/" (addition, subtraction, multiplication, division)

Id (the identity function)

add1, such that *add1* (*x*) ≙ *x+1* for all *x* in **R**.

1 What are the signatures of these functions?
2 What function is *inverse* ; *inverse*?
3 What function is "+" ; *square*?

2.2 Dispatching

For arbitrary sets *U, X, Y,* the "dispatching" functional **infix** "&" was introduced in 2.8.1. Take *U, X* and *Y* to be all $\mathbf{R}^{\bullet}$. Prove the following (referring to the functions of the previous exercise):

1 (*square* & *Id*) ; "/" = $Id \setminus R^{\bullet}$

2 (*square* & *inverse*) ; "∗" = *Id*

3 *add1* ; *square* = (*square* & ((*Id* & *Id*) ; "+")) ; "+" ; *add1*

2.3 Parallel application

For arbitrary sets *U, V, X, Y,* the "parallel application" functional **infix** "#" was introduced in 2.8.1. Let *proj1* and *proj2,* be defined as the two projections from $U \times V$:

proj1 (<*u, v*>) $\triangleq$ *u*

proj2 (<*u, v*>) $\triangleq$ *v*

1 What are the signatures of functions *proj1* and *proj2*?
2 Take *U, V, X* and *Y* to be all $\mathbf{R}^{\bullet}$. Prove the following:

2.1 "/" = (*Id* # *inverse*) ; "∗"

2.2 "∗" ; *square* = (*square* # *square*) ; "∗"

2.3 "/" ; *square* = (*square* # *square*) ; "/"

2.4 "+" ; *square* = (((*square* # *square*) ; "+") & ("∗" ; (*Id* & *Id*) ; "+")) ; "+"

3 Express the following properties in the style of the equalities 2.1 to 2.4, that is to say without any reference to members of **R**, using only the functions and functionals defined so far.

3.1 $(a - b) * (a + b) = a^2 - b^2$ for all a, b in $\mathbf{R}$

3.2 $b * (a/b) = a$ for all a in $\mathbf{R}$, b in $\mathbf{R}^\bullet$

3.3 $a * (a + b) = a^2 + a * b$

3.4 f is commutative
(where f is a total function of signature $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$).

2.4 Iterate, apply and curry

All the functions considered in this exercise are total except in question 5.

Let A, X, Y, Z be arbitrary sets. For any function

$$f: A \rightarrow A$$

and any non-negative integer n, define *iterate* (f, n) to be the n-th iterate of f, in other words the function h such that

$$h = f ; f ; f \ldots ; f \qquad (n \text{ times})$$

Also, define *apply* to be the function such that, for any member a of A,

$$apply\ (f, a) \triangleq f\,(a).$$

In other words, *apply* takes two arguments, the first of which is a function. Its result is the application of its first argument to its second.

Finally, *curry* is the function defined on page 40, which takes any two-argument function as argument and yields a one-argument function as result. Its signature (if it is applied to sets X, Y and Z) is

$$((X \times Y) \rightarrow Z) \rightarrow (X \rightarrow (Y \rightarrow Z))$$

1 What are the signatures of *iterate* and *apply*?

2 Show that if the set A is given, it is possible to choose sets X, Y, Z so that *curry* (*apply*) is a valid expression, denoting a function. What then is the signature of this function? What is the function itself?

3 Show that if the set A is given, it is possible to choose sets X, Y, Z so that *curry* (*iterate*) is a valid expression, denoting a function. What then is the signature of this function? Explain informally what this function "does".

4 For each of the following functional expressions, give set assignments for A, X, Y, Z such that the expression has a value. Then give that value (or, in the case the value is a function, explain what that function is, for example "the square root function on $\mathbf{R}$", or "the function $f : \mathbf{N} \rightarrow \mathbf{N}$ such that, for any $n : \mathbf{N}$, $f\,(n) = n + 2$)".

4.1 *curry* ("/") (1)

4.2 *curry* ("+")

4.3 *curry* (*iterate*) (*add1*) (1)

4.4 (*curry* ("+") # *Id*) ; *iterate*

5 As defined on page 40, the functional *curry* applies to arguments that are total functions. Extend the definition so that its arguments and result are possibly partial functions. (**Hint**: the new definition must specify precisely the domain of the functional's result, and of its result's result, as was done for "#" and "&" in 2.8.1.)

2.5 Compilers and interpreters

Let M be a simple computer whose machine programs are assumed to compute functions of signature

$$I \rightarrow O$$

where I is the set of possible inputs and O the set of possible outputs. Machine programs for M may thus be viewed as implementations of functions from I to O.

Let L be a high-level language; let *COMP* be a compiler for L, generating M machine code and *INT* be an interpreter for L running on M.

Let *fcomp* and *fint* be the functions performed by *COMP* and *INT*, respectively.

1 What are the signatures of *fcomp* and *fint*?

2 Express a mathematical relationship between *fcomp* and *fint*? (**Hint**: look at *curry*.)

2.6 Properties of images

Let X, Y and Z be arbitrary sets, r and s relations in $X \leftrightarrow Y$, t a relation in $Y \leftrightarrow Z$, f and g functions in $X \nrightarrow Y$ with disjoint domains, A and A' subsets of X, B and B' subsets of Y. Prove the following properties of the image operation $r(\cdot A \cdot)$ introduced page 25:

1 $r(\cdot A \cup A' \cdot) = r(\cdot A \cdot) \cup r(\cdot A' \cdot)$

2 $f^{-1}(\cdot B \cap B' \cdot) = f^{-1}(\cdot B \cdot) \cap f^{-1}(\cdot B' \cdot)$

3 $(s ; t)(\cdot A \cdot) = t(\cdot s(\cdot A \cdot) \cdot)$

4 $(r \cup s)(\cdot A \cdot) = r(\cdot A \cdot) \cup s(\cdot A \cdot)$

5 $(r \cap s)(\cdot A \cdot) \subseteq r(\cdot A \cdot) \cap s(\cdot A \cdot)$

6 $(A \subseteq A') \Rightarrow (r(\cdot A \cdot) \subseteq r(\cdot A' \cdot))$

7 $(f \setminus A')(\bullet A \bullet) = f(\bullet A \cap A' \bullet)$

8 $(f \Cup g)(\bullet A \bullet) = f(\bullet A \bullet) \cup g(\bullet A \bullet)$

9 $(f \Cup g)^{-1}(\bullet B \bullet) = f^{-1}(\bullet B \bullet) \cup g^{-1}(\bullet B \bullet)$

10 $(f \setminus A)^{-1}(\bullet B \bullet) = f^{-1}(\bullet B \bullet) \cap A$

(**Hint**: to prove that two functions are equal, you must prove that they have the same domains and yield the same value for any member of that domain. Note that not all relations appearing above are functions.)

11 Dropping the hypothesis that f and g have disjoint domains, update properties 8 and 9 accordingly.

2.7 Expressing properties of relations and functions

The aim of this exercise is to learn to characterize various properties of relations and functions by higher-level functional predicates.

The generic identity function *Id*, written *Id* [X] if the set X to which it applies must be made explicit, is such that *Id* (x) = x for any $x : X$.

1 Prove that a relation $r \in X \leftrightarrow Y$ is functional if and only if $r^{-1} ; r \subseteq Id\ [Y]$.

The next question uses the notion of total relation. A relation $r \in X \leftrightarrow Y$ is said to be total if and only if

$$\forall x : X \bullet \exists y : Y \bullet <x, y> \in r$$

In other words, r is total if and only if it associates at least one member of the target set with every member of the source set. (This is compatible with the definition of "total" when applied to functions.)

2 Prove that a relation $r : X \leftrightarrow Y$ is total if and only if $Id\ [X] \subseteq r ; r^{-1}$.

The next questions require that you express formally various properties of relations and functions in the same style as in questions 1 and 2, that is to say using relational operators such as composition (";") and inverse, with no explicit references to members of the source or target sets (such as x and y above).

3 Express that r is a surjective (or "onto") relation, that is to say such that

$$\forall y : Y \bullet \exists x : X \bullet <x, y> \in r$$

4 Express that r is an injective relation, that is to say such that

$$\forall x_1, x_2 : X, y : Y \bullet (<x_1, y> \in r \land <x_2, y> \in r) \implies (x_1 = x_2)$$

5 Express that r is a one-to-one function (injective, surjective, total).

6 A relation $r \in X \leftrightarrow X$ is said to be

- Reflexive iff $\forall x : X \bullet \langle x, x \rangle \in r$
- Irreflexive iff $\forall x : X \bullet \langle x, x \rangle \notin r$
- Symmetric iff $\forall x, y : X \bullet (\langle x, y \rangle \in r) \Rightarrow (\langle y, x \rangle \in r)$
- Asymmetric iff $\forall x, y : X \bullet (\langle x, y \rangle \in r) \Rightarrow (\langle y, x \rangle \notin r)$
- Antisymmetric iff $\forall x, y : X \bullet (\langle x, y \rangle \in r \land \langle y, x \rangle \in r) \Rightarrow (x = y)$
- Transitive iff $\forall x, y, z : X \bullet (\langle x, y \rangle \in r \land \langle y, z \rangle \in r) \Rightarrow (\langle x, z \rangle \in r)$

Express each of these properties in the above style.