Appendix A

A Metalanguage

In this book, we will explore many aspects of programming languages. Along the way, we will need to discuss both their form and their meaning. But we need some language in which we can carry out these discussions! A language used for describing other languages is called a **metalanguage**. The purpose of this chapter is to introduce the metalanguage that we will be using throughout the rest of the book.

The most obvious choice for a metalanguage is a natural language, such as English, that we use in our everyday lives. When it comes to talking about programming languages, natural language is certainly useful for describing features, explaining concepts at a high level, expressing intuitions, and conveying the big picture. But natural language is too bulky and imprecise to adequately treat the details and subtleties that characterize programming languages. For these we require the precision and conciseness of a mathematical language.

We present our metalanguage as follows. We begin by reviewing the basic mathematics upon which the metalanguage is founded. Next, we explore two concepts at the core of the metalanguage: **functions** and **domains**. We conclude with a summary of the metalanguage notation.

A.1 The Basics

The metalanguage we will use is based on set theory. Since set theory serves as the foundation for much of popular mathematics, you are probably already familiar with many of the basics described in this section. However, since some of our notation is nonstandard, we recommend that you at least skim this section in order to familiarize yourself with our conventions.

A.1.1 Sets

A set is an unordered collection of elements. Sets with a finite number of elements are written by enclosing the written representations of the elements within braces and separating them by commas. So $\{2, 3, 5\}$ denotes the set of the first three primes. Order and duplication don't matter within set notation, so $\{3, 5, 2\}$ and $\{3, 2, 5, 5, 2, 2\}$ also denote the set of the first three primes. A set containing one element, such as $\{19\}$, is called a **singleton**. The set containing no elements is called the **empty set** and is written $\{\}$.

We will assume the existence of certain sets:

$$
\begin{array}{lll}
\mathit{Unit} = \{\mathit{unit}\} & & \textit{;The standard singleton} \\
\mathit{Bool} = \{\mathit{true}, \mathit{false}\} & & \textit{;Truth values} \\
\mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} & & \textit{;Integers} \\
\mathit{Pos} = \{1, 2, 3, \ldots\} & & \textit{;Positive integers} \\
\mathit{Neg} = \{-1, -2, -3, \ldots\} & & \textit{;Negative integers} \\
\mathit{Nat} = \{0, 1, 2, \ldots\} & & \textit{;Natural numbers} \\
\mathit{Rat} = \{0, 1, \frac{1}{2}, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{3}, \frac{2}{3}, -\frac{2}{3}, \ldots\} & & \textit{;Rationals} \\
\mathit{String} = \{'', '\mathit{a}', '\mathit{b}', \ldots, '\mathit{foo}', \ldots, '\mathit{a\ string}', \ldots\} & & \textit{;All text strings}
\end{array}
$$

(The text in *slanted font* following the semi-colon is just a comment and is not a part of the definition. We will use this commenting convention throughout the book.) *Unit* (the canonical singleton set) and *Bool* (the set of boolean truth values) are finite sets, but the other examples are infinite. Since it is impossible to write down all elements of an infinite set, we use ellipses, "...", to stand for the missing elements, and depend on the reader's intuition to fill them out. Note that our definition of *Nat* includes 0.

We consider numbers, truth values, and the unit value to be **primitive** elements that cannot be broken down into subparts. Set elements are not constrained to be primitive; sets can contain any structure, including other sets. For example,

$$\{\mathit{Int}, \mathit{Nat}, \{2, 3, \{4, 5\}, 6\}\}$$

is a set with three elements: the set of integers, the set of natural numbers, and a set of four elements (one of which is itself a set of two numbers). Here the names *Int* and *Nat* are used as synonyms for the set structure they denote.

Membership is specified by the symbol $\in$ (pronounced "element of" or "in"). The notation $e \in S$ asserts that e is an element of the set S, while $e \notin S$ asserts that e is not an element of S. (In general, a slash through a symbol indicates the negation of the property denoted by that symbol.) For example,

$$\begin{array}{rcl} 0 & \in & \mathit{Nat} \\ 0 & \notin & \mathit{Neg} \\ \mathit{Int} & \in & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \\ \mathit{Neg} & \notin & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \\ 2 & \notin & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \end{array}$$

In the last example, 2 is not an element of the given set even though it is an element of one of that set's elements.

Two sets A and B are **equal** (written $A = B$) if they contain the same elements, i.e., if every element of one is an element of the other. A set A is a **subset** of a set B (written $A \subseteq B$) if every element of A is also an element of B. Every set is a subset of itself, and the empty set is trivally a subset of every set. E.g.,

$$\begin{array}{rcl} \{\} & \subseteq & \{1,\ 2,\ 3\} \subseteq \mathit{Pos} \subseteq \mathit{Nat} \subseteq \mathit{Int} \subseteq \mathit{Rat} \\ \mathit{Nat} & \subseteq & \mathit{Nat} \\ \mathit{Nat} & \not\subseteq & \mathit{Pos} \end{array}$$

Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$. A is said to be a **proper subset** of B (written $A \subset B$) if $A \subseteq B$ and $A \neq B$.

Sets are often specified by describing a defining property of their elements. The **set builder** notation $\{x \mid P_x\}$ (pronounced "the set of all x such that P_x") designates the set of all elements x such that the property P_x is true of x. For example, *Nat* could be defined as $\{n \mid n \in \mathit{Int} \text{ and } n \geq 0\}$. The sets described by set builder notation are not always well-defined. For example, $\{s \mid s \notin s\}$, (the set of all sets that are not elements of themselves) is a famous nonsensical description known as Russell's paradox.

Some common binary operations on sets are defined below using set builder notation:

$$\begin{array}{lcll} A \cup B & = & \{x \mid x \in A \ \text{ or } \ x \in B\} & ; \textit{union} \\ A \cap B & = & \{x \mid x \in A \ \text{ and } \ x \in B\} & ; \textit{intersection} \\ A - B & = & \{x \mid x \in A \ \text{ and } \ x \notin B\} & ; \textit{difference} \end{array}$$

The notions of union and intersection can be extended to (potentially infinite) collections of sets. If A is a set of sets, then $\bigcup A$ denotes the union of all of the component sets of A. That is,

$$\bigcup A = \{x \mid \text{ there exists an } a \in A \text{ such that } x \in a\}$$

If A_i is a family of sets indexed by elements i of some given index set I, then

$$\bigcup_{i \in I} A_i = \bigcup \{A_i \mid i \in I\}$$

denotes the union of all the sets A_i as i ranges over I. Intersections of collections of sets are defined in a similar fashion.

The **cardinality** of a set A (written $|A|$) is the number of elements in A. The cardinality of an infinite set is said to be infinite. Thus $|\mathit{Int}|$ is infinite, but

$$|\{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\}| = 3$$

Still, there are distinctions between infinities. Informally, two sets are said to be in a **one-to-one correspondence** if it is possible to pair every element of one set with a unique and distinct element in the other set without having any elements left over. Any set that is in a one-to-one correspondence with *Int* is said to be **countable**. For instance, the set *Even* of even integers is countable because every element $2n$ in *Even* can be paired with n in *Int*. Similarly, *Nat* is obviously countable, and a little thought shows that *Rat* is countable as well. Informally, all countable sets "have the same size". On the other hand, any infinite set that is not in a one-to-one correspondence with *Int* is said to be

uncountable. Cantor's celebrated diagonalization proof shows that the real numbers are uncountable.[1] Informally, the size of the reals is a much "bigger" infinity than the size of the integers.

The **powerset** of a set A (written $\mathcal{P}(A)$) is the set of all subsets of A. For example,

$$\mathcal{P}(\{1,\ 2,\ 3\}) = \{\{\},\ \{1\},\ \{2\},\ \{3\},\ \{1,\ 2\},\ \{1,\ 3\},\ \{2,\ 3\},\ \{1,\ 2,\ 3\}\}$$

The cardinality of the powerset of a finite set is given by:

$$|\mathcal{P}(A)| = 2^{|A|}$$

In the above example, the powerset has size $2^3 = 8$. The set of all subsets of the integers, $\mathcal{P}(\mathit{Int})$, turns out to be an uncountable set.

A.1.2 Tuples

A tuple is an ordered collection of elements. A tuple of length n, called an **n-tuple**, can be envisioned as a structure with n slots arranged in a row, each of which is filled by an element. Tuples with a finite length are written by writing the slot values down in order, separated by commas, and enclosing the result in angle brackets. Thus $\langle 2,\ 3,\ 5\rangle$ is a tuple of the first three primes. Length and order of elements in a tuple matter, so $\langle 2,\ 3,\ 5\rangle$, $\langle 3,\ 2,\ 5\rangle$, and $\langle 3,\ 2,\ 5,\ 5,\ 2,\ 2\rangle$ denote three distinct tuples. Tuples of size 2 through 5 are called, respectively, **pairs**, **triples**, **quadruples**, and **quintuples**. The 0-tuple, $\langle\rangle$, and 1-tuples also exist.

The element of the ith slot of a tuple t can be obtained by **projection**, written $t \downarrow i$. For example, if s is the triple $\langle 2, 3, 5\rangle$, then $s \downarrow 1 = 2$, $s \downarrow 2 = 3$, and $s \downarrow 3 = 5$. If t is an n-tuple, then $t \downarrow i$ is only well-defined when $1 \leq i \leq n$. Two tuples s and t are **equal** if they have the same length n and $s \downarrow i = t \downarrow i$ for all $1 \leq i \leq n$.

As with sets, tuples may contain other tuples; e.g. $\langle\langle 2,\ 3,\ 5,\ 7\rangle,\ 11,\ \langle 13,\ 17\rangle\rangle$ is a tuple of three elements: a quadruple, an integer, and a pair. Moreover, tuples may contain sets and sets may contain tuples. For instance, the following is a well-defined mathematical structure:

[1] A description of Cantor's method can be found in many books on mathematical analysis and computability. We particularly recommend [Hof80].

$$\langle\langle 2,\ 3,\ 5\rangle,\ \mathit{Int},\ \{\{2,\ 3,\ 5\},\ \langle 7,\ 11\rangle\}\rangle$$

If A and B are sets, then their **Cartesian product** (written $A \times B$) is the set of all pairs whose first slot holds an element from A and whose second slot holds an element from B. This can be expressed using set builder notation as:

$$A \times B = \{\langle a,\ b\rangle \mid a \in A \ \text{ and } \ b \in B\}$$

For example,

$$\{2,\ 3,\ 5\} \times \{7,\ 11\} = \{\langle 2,\ 7\rangle,\ \langle 2,\ 11\rangle,\ \langle 3,\ 7\rangle,\ \langle 3,\ 11\rangle,\ \langle 5,\ 7\rangle,\ \langle 5,\ 11\rangle\}$$

$$\mathit{Nat} \times \mathit{Bool} = \{\langle 0,\ \mathit{false}\rangle,\ \langle 1,\ \mathit{false}\rangle,\ \langle 2,\ \mathit{false}\rangle,\ \ldots,\ \langle 0,\ \mathit{true}\rangle,\ \langle 1,\ \mathit{true}\rangle,\ \langle 2,\ \mathit{true}\rangle,\ \ldots\}$$

If A and B are finite, then $|A \times B| = |A| \cdot |B|$

The product notion extends to families of sets. If $A_1,\ \ldots,\ A_n$ is a family of sets, then their product (written $A_1 \times A_2 \times \ldots \times A_n$ or $\prod_{i=1}^{n} A_i$) is the set of all n tuples $\langle a_1,\ a_2,\ \ldots, a_n\rangle$ such that $a_i \in A_i$. The notation A^n ($= \prod_{i=1}^{n} A$) stands for the n-fold product of the set A.

A.1.3 Relations

A **binary relation on** A is a subset of $A \times A$.[2] For example, the less-than relation, $<_{Nat}$, on natural numbers is the subset of $\mathit{Nat} \times \mathit{Nat}$ consisting of all pairs of numbers $\langle n,\ m\rangle$ such that n is less than m:

$$< \ = \{\langle 0,\ 1\rangle,\ \langle 0,\ 2\rangle,\ \langle 0,\ 3\rangle,\ \ldots,\ \langle 1,\ 2\rangle,\ \langle 1,\ 3\rangle,\ \ldots,\ \langle 2,\ 3\rangle,\ \ldots\}$$

For a binary relation R on A, the notation $a_1\ R\ a_2$ is shorthand for $\langle a_1,\ a_2\rangle \in R$. Similarly, the notation $a_1\ \not R\ a_2$ means that $\langle a_1,\ a_2\rangle \notin R$. Thus, the assertion $1 < 2$ is really just another way of saying $\langle 1,\ 2\rangle \in\ <$, and $3 \not< 2$ is another way of saying $\langle 3,\ 2\rangle \notin\ <$.

Binary relations are often classified by certain properties. Let R be a binary relation on a set A. Then:

[2] The notion of a relation can be generalized to arbitrary products, but binary relations will be sufficient for our purposes.

- R is **reflexive** if, for all $a \in A$, $a \; R \; a$.
- R is **symmetric** if, for all $a_1, a_2 \in A$, $a_1 \; R \; a_2$ implies $a_2 \; R \; a_1$.
- R is **transitive** if, for all $a_1, a_2, a_3 \in A$, $a_1 \; R \; a_2$ and $a_2 \; R \; a_3$ imply $a_1 \; R \; a_3$.
- R is **anti-symmetric** if, for all $a_1, a_2 \in A$, $a_1 \; R \; a_2$ and $a_2 \; R \; a_1$ implies $a_1 = a_2$. (This assumes the existence of a reflexive, symmetric, and transitive equality relation $=$ on A.)

For example, the $<$ relation on integers is anti-symmetric and transitive, the "is a divisor of" relation on natural numbers is reflexive and transitive, and the $=$ relation on integers is reflexive, symmetric, and transitive.

A binary relation that is reflexive, symmetric and transitive is called an **equivalence relation**. An equivalence relation R on A uniquely **partitions** the elements of A into disjoint **equivalence classes** A_i whose union is A and which satisfy the following: $a_1 \; R \; a_2$ if and only if a_1 and a_2 are elements of the same A_i. For example, let $=_{mod3}$ be the "has the same remainder modulo 3" relation on natural numbers. Then it's easy to show that $=_{mod3}$ satisfies the criteria for an equivalence relation. It partitions *Nat* into three equivalence classes:

$$
\begin{aligned}
Nat_0 &= \{0,\ 3,\ 6,\ 9,\ \ldots\} \\
Nat_1 &= \{1,\ 4,\ 7,\ 10,\ \ldots\} \\
Nat_2 &= \{2,\ 5,\ 8,\ 11,\ \ldots\}
\end{aligned}
$$

The **quotient** of a set A by an equivalence relation R (written A/R) is the set of equivalence classes into which R partitions A. Thus,

$$(Nat/ =_{mod3}) = \{Nat_0,\ Nat_1,\ Nat_2\}$$

There are a number of operations on binary relations that produce new relations. The **n-fold composition** of a binary relation R, written R^n, is the unique relation such that $a_{\text{left}} \; R^n \; a_{\text{right}}$ if and only if there exist a_i, $1 \leq i \leq n+1$, such that $a_1 = a_{\text{left}}$, $a_{n+1} = a_{\text{right}}$, and for each i, $a_i \; R \; a_{i+1}$. The **closure** of a binary relation R on A over a specified property P is the smallest relation R_P

such that $R \subseteq R_P$ and R_P satisfies the property P. The most important kind of closure we will consider is the **reflexive, transitive closure** of a relation R, written R^*: $a_{\text{left}}\ R^*\ a_{\text{right}}$ if and only if $a_{\text{left}}\ R^n\ a_{\text{right}}$ for some natural number n. For example, the reflexive, transitive closure of the "is one less than" relation on integers is the "is less than" relation on integers.

A.2 Functions

Functions are a crucial component of our metalanguage. We will devote a fair amount of care to explaining what they are and developing notation to express them.

A.2.1 Definition

Informally, a function is a mapping from an argument to a result. More formally, a function is a triple of three components:[3]

1. The **source** S of the function — the set from which the argument is taken.
2. The **target** T of the function — the set from which the result is taken.
3. The **graph** of a function — a subset G of $S \times T$ such that each $s \in S$ appears as the first component in no more than one pair $\langle s,\ t\rangle \in G$.

For example, the increment function inc_{Int} on the integers can be defined as

$$inc_{Int} = \langle Int,\ Int,\ G_{\text{inc}}\rangle$$

where G_{inc} is the set of all pairs $\langle i,\ i+1\rangle$ such that $i \in Int$. That is,

$$G_{\text{inc}} = \{\ldots,\ \langle -3,\ -2\rangle,\ \langle -2,\ -1\rangle,\ \langle -1,\ 0\rangle,\ \langle 0,\ 1\rangle,\ \langle 1,\ 2\rangle,\ \langle 2,\ 3\rangle,\ \ldots\}$$

The **type**[4] of a function specifies its source and target. The type of a function with source S and target T is written $S \to T$. For example, the type of inc_{Int} is $Int \to Int$. The notation $f : S \to T$ means that f has type $S \to T$.

[3] What we call source and target are commonly called **domain** and **codomain**, respectively. We use different names so as not to cause confusion with the meaning of the term **domain** introduced in Section A.3.

[4] The type of a function is sometimes called its **signature**.

Two functions are **equal** if they are equal as triples — i.e., if their sources, targets, and graphs are respectively equal. In particular, it is not sufficient for their graphs to be equal — they must have the same type as well. For example, consider the following two functions

$$abs_1 = \langle Int,\ Int,\ G_{\text{abs}} \rangle$$

$$abs_2 = \langle Int,\ Nat,\ G_{\text{abs}} \rangle$$

where G_{abs} is the set of all pairs $\langle i,\ i_{\text{abs}} \rangle$ such that i is an integer and i_{abs} is the absolute value of i. Then even though abs_1 and abs_2 have the same graph, they are not equal as functions because the type of abs_1, $Int \to Int$, is different from the type of abs_2, $Int \to Nat$.

Many programming languages use the term "function" to refer to a kind of subroutine. To avoid confusion, we will use the term **procedure** for a programming language subroutine, and will reserve the term **function** for the mathematical notion. We wish to carefully distinguish them because they differ in some important respects:

- We often think of procedures as methods, or sometimes even agents, for computing an output from an input. A function doesn't use any method or perform any computation; it doesn't *do* anything. It simply *is* a structure that contains the source, the target, and all input/output pairs.

- We typically view procedures as taking multiple arguments or returning multiple results. But a function always has exactly one argument and exactly one result. However, we will see shortly how these procedural notions can be simulated with functions.

- In addition to returning a value, procedures often have a side-effect — e.g., changing the state of the memory or the status of a screen. There is no equivalent notion of side-effect for a function. However, we will see in Chapter 10 how to use functions to model side-effects in a programming language.

- When viewed in terms of their input/output behavior, procedures can only specify a subset of functions known as the **computable functions**. The most famous example of a non-computable function is the halting function, which maps the text of a program to a boolean that indicates whether or not the program will halt when executed.

The above points do not necessarily apply to the procedural entities in all languages. In particular, the subroutines in so-called **functional programming languages** are very close in spirit to mathematical functions.

A.2.2 Application

The primary operation involving a function is the **application** of the function to an **argument**, an element in its source. The function is called the **operator** of the application, while the argument is called the **operand** of the application. The result of applying an operator f to an operand s is the unique element t in the target of f such that $\langle s,\ t\rangle$ is in the graph of f. If there is no pair $\langle s,\ t\rangle$ in the graph of f, then the application of f to s is said to be **undefined**.

A **total function** is one for which application is defined for all elements of its source. If there are source elements for which the function is undefined, the function is said to be **partial**. Most familiar numerical functions are total, but some are partial. The reciprocal function on rationals is partial because it is not defined at 0. And a square root function defined as

$$\langle Int,\ Int,\ \{\langle i^2,\ i\rangle \mid i \in Int\}\rangle$$

is partial because it is defined only at perfect squares. We will use the type notation $S \to T$ to designate the class of total functions and the special notation $S \rightharpoonup T$ to designate the class of partial functions.

It is always possible to turn a partial function into a total function by adding a distinguished element to the target that represents "undefined". By convention, this element is called **bottom** and is written $\perp$. Using this element, we can define a total reciprocal function whose type is $Rat \to (Rat \cup \{\perp\})$ and whose graph is:

$$\{\langle 0,\ \perp\rangle\} \cup \{\langle q,\ 1/q\rangle \mid q \in Rat, q \neq 0\}$$

Bottom plays a crucial role in the explanation of fixed points in Chapter 7.

We will use the juxtaposition $f\ s$ to denote the application of a function f to an element s.[5] For instance, the increment of 3 is written $inc_{Int}\ 3$. Parentheses are used to structure nested applications. Thus,

$$inc_{Int}\ (inc_{Int}\ 3)$$

expresses the increment of the increment of 3. In the metalanguage, parentheses that don't affect the application structure can always be added without changing the meaning of an expression[6] the following is equivalent to the above:

$$((inc_{Int})\ (inc_{Int}\ (3)))$$

By default, function application associates to the left, so that the expression $a\ b\ c\ d$ parses as $(((a\ b)\ c)\ d)$.

The **type** of an application is the type of the target of the operator of the application. For example, if $sqr : Int \rightarrow Nat$, then $(sqr\ -3) : Nat$ (pronounced "$(sqr\ -3)$ has type Nat"). An application is **well-typed** only when the type of the operand is the same as the source of the operator type. (The type of a number like 3 depends on context: it can be considered a natural, an integer, a rational, etc.) For example, if f is a function with type $Nat \rightarrow Int$, then the application $(f\ -3)$ is not well-typed because $-3 \not\in Nat$. However, the application $(f\ (sqr\ -3))$ is well-typed. In our metalanguage, an application is only legal if it is well-typed.

A.2.3 More Function Terminology

For any set A, there is an **identity** function id_{A} that maps every element of A to itself:

[5] The reader may find it strange that we depart from the more traditional notation for application, which is written $f(s)$ for single arguments, and $f(s_1, s_2, \ldots, s_n)$ for multiple arguments. The reason is that in the traditional notation, f is usually restricted to be a *function name*, whereas we will want to allow the function position of an application to be any metalanguage expression that stands for a function. Application by juxtaposition is a superior notation for handling this more general case because it visually distinguishes less between the function position and the argument position.

[6] This contrasts with SCHEME+ and s-expression grammars, in which no parentheses are optional.

$$id_{\mathrm{A}} = \langle A,\ A,\ \{\langle a,\ a\rangle \mid a \in A\}\rangle$$

For each element a of a set A, there is a **constant** function $const_a$ that maps every element of A to a:

$$const_a = \langle A,\ A,\ \{\langle a',\ a\rangle \mid a' \in A\}\rangle$$

For any set B such that $B \subseteq A$, there is an **inclusion** function $B \hookrightarrow A$ that maps every element of B to the same element in the larger set:

$$\mathrm{B} \hookrightarrow \mathrm{A} = \langle B,\ A,\ \{\langle b,\ b\rangle \mid b \in B\}\rangle$$

Inclusion functions are handy for making a metalanguage expression the "right type". For example, if *sqr* has type $Int \rightarrow Nat$, then the expression

$$(sqr\ \ (sqr\ \ {-3}))$$

is not well-typed, but the expression

$$(sqr\ \ (Nat \hookrightarrow Int\ \ (sqr\ \ {-3})))$$

is well-typed.

If $f : A \rightarrow B$ and $g : B \rightarrow C$, then the **composition** of g and f, written $g \circ f$, is a function of type $A \rightarrow C$ defined as follows:

$$(g \circ f)\ \ a = (g\ \ (f\ \ a)),\ \text{for all } a \in A$$

The composition function[7] is associative, so that

$$f \circ g \circ h = (f \circ g) \circ h = f \circ (g \circ h)$$

If $f : A \rightarrow A$ then the **n-fold composition** of f, written f^n, is

$$\underbrace{f \circ f \circ \ldots \circ f}_{n \text{ times}}$$

[7]There is not a single composition function, but really a family of composition functions indexed by the types of their operands.

f^0 is defined to be the identity function on A. Because of the associativity of composition, $f^n \circ f^m = f^{n+m}$.

The **image** of a function is that subset of the target that the function actually maps to. That is, for $f : S \to T$, the image of f is

$$\{t \mid \text{ there exists an } s \text{ such that } (f \;\; s) = t\}$$

A function is **injective** when no two elements of the source map to the same target element, i.e., when $(f \;\; d_1) \; = (f \;\; d_2)$ implies $d_1 \; = d_2$. A function is **surjective** when every element in the target is the result of some application, i.e., when the image is equal to the target. A function is **bijective** if it is both injective and surjective. Two sets A and B are said to be in a **one-to-one correspondence** if there exists a bijective function with type $A \to B$.

A.2.4 Higher-Order Functions

The sources and targets of functions are not limited to familiar sets like numbers, but may be sets of sets, sets of tuples, or even sets of functions. Functions whose sources or targets themselves include functions are called **higher-order functions**.

As a natural example of a function that returns a function, consider a function *make-expt* that, given a power, returns a function that raises numbers to that power. The type of *make-expt* is

$$Nat \to (Nat \to Nat)$$

That is, the source of *make-expt* is *Nat*, and the target of *make-expt* is the set of all functions with type $Nat \to Nat$. The graph of *make-expt* is:

$$\begin{aligned}
&\{\langle 0,\ \langle Nat,\ Nat,\ \{\langle 0,\ 1\rangle,\ \langle 1,\ 1\rangle,\ \langle 2,\ 1\rangle,\ \langle 3,\ 1\rangle,\ \langle 4,\ 1\rangle,\ \ldots\}\rangle\rangle,\\
&\ \langle 1,\ \langle Nat,\ Nat,\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 1\rangle,\ \langle 2,\ 2\rangle,\ \langle 3,\ 3\rangle,\ \langle 4,\ 4\rangle,\ \ldots\}\rangle\rangle,\\
&\ \langle 2,\ \langle Nat,\ Nat,\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 1\rangle,\ \langle 2,\ 4\rangle,\ \langle 3,\ 9\rangle,\ \langle 4,\ 16\rangle,\ \ldots\}\rangle\rangle,\\
&\ \langle 3,\ \langle Nat,\ Nat,\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 1\rangle,\ \langle 2,\ 8\rangle,\ \langle 3,\ 27\rangle,\ \langle 4,\ 64\rangle,\ \ldots\}\rangle\rangle,\\
&\ \ldots\}
\end{aligned}$$

That is, (*make-expt* 0) denotes a function that maps every number to 1, (*make-expt* 1) denotes the identity function on natural numbers, (*make-expt* 2) denotes the squaring function, (*make-expt* 3) denotes the cubing function, and so on.

As an example of a function that takes functions as arguments, consider the function *apply-to-five* that takes a function between natural numbers and returns the value of this function applied to 5. The type of *apply-to-five* is

$$(Nat \to Nat) \to Nat$$

and its graph is

$$\{\langle id_{Nat}, 5\rangle,\ \langle inc_{Nat}, 6\rangle,\ \langle dec_{Nat}, 4\rangle,\ \langle square_{Nat}, 25\rangle,\ \langle cube_{Nat}, 125\rangle,\\ \ldots,\ \langle\langle Nat,\ Nat,\ \{\ldots, \langle 5,\ n\rangle, \ldots\}\rangle, n\rangle,\ \ldots\}$$

where inc_{Nat}, dec_{Nat}, $square_{Nat}$, and $cube_{Nat}$ denote, respectively, the incrementing function, decrementing function, squaring function, and cubing function on natural numbers.

We make extensive use of higher order functions throughout this book.

A.2.5 Multiple Arguments and Results

We noted before that every mathematical function has a single argument and a single result. Yet, as programmers, we are used to thinking that many familiar procedures, like addition and multiplication, have multiple arguments. Sometimes we think of procedures as returning multiple results; for instance, a division procedure can profitably be viewed as returning a both a quotient and a remainder. How can we translate these programming language notions into the world of mathematical functions?

Multiple Arguments

There are two common approaches for handling multiple arguments:

1. The multiple arguments can be boxed up into a single argument tuple. For instance, under this approach, the binary addition function $+_{Nat}$ on natural numbers would have type

   $$(Nat \times Nat) \to Nat$$

   and would have the following graph:

$$\begin{array}{l}\{\langle\langle 0,\ 0\rangle, 0\rangle,\ \langle\langle 0,\ 1\rangle, 1\rangle,\ \langle\langle 0,\ 2\rangle, 2\rangle,\ \langle\langle 0,\ 3\rangle, 3\rangle,\ \ldots,\\ \ \langle\langle 1,\ 0\rangle, 1\rangle,\ \langle\langle 1,\ 1\rangle, 2\rangle,\ \langle\langle 1,\ 2\rangle, 3\rangle,\ \langle\langle 1,\ 3\rangle, 4\rangle,\ \ldots,\\ \ \langle\langle 2,\ 0\rangle, 2\rangle,\ \langle\langle 2,\ 1\rangle, 3\rangle,\ \langle\langle 2,\ 2\rangle, 4\rangle,\ \langle\langle 2,\ 3\rangle, 5\rangle,\ \ldots,\\ \ \ldots\}\end{array}$$

Then an application of the addition function to 3 and 5, say, would be written as $(+_{Nat}\ \langle 3,\ 5\rangle)$.

2. A function of multiple arguments can be represented as a higher-order function that takes the first argument and returns a function that takes the rest of the arguments. This approach is named **currying**, after its inventor, Haskell Curry. Under this approach, the binary addition function $+_{Nat}$ on natural numbers would have type

   $$Nat \to (Nat \to Nat)$$

   and would have the following graph:

   $$\begin{array}{l}\langle\langle 0,\ \langle Nat,\ Nat,\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 1\rangle,\ \langle 2,\ 2\rangle,\ \langle 3,\ 3\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 1,\ \langle Nat,\ Nat,\ \{\langle 0,\ 1\rangle,\ \langle 1,\ 2\rangle,\ \langle 2,\ 3\rangle,\ \langle 3,\ 4\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 2,\ \langle Nat,\ Nat,\ \{\langle 0,\ 2\rangle,\ \langle 1,\ 3\rangle,\ \langle 2,\ 4\rangle,\ \langle 3,\ 5\rangle,\ \ldots\}\rangle\rangle,\\ \ \ldots\rangle\end{array}$$

   When $+_{Nat}$ is applied to n, the resulting value is the increment-by-n function. So, given 0, it returns the identity function on natural numbers; given 1, it returns the increment-by-one function; given 2, it returns the increment-by-two function; and so on. With currying, the application of $+_{Nat}$ to 3 and 5 is written as $((+_{Nat}\ 3)\ 5)$ or as $(+_{Nat}\ 3\ 5)$. (relying on the left-associativity of application).

   In the currying approach, functions like $+_{Nat}$ or *make-expt* can be viewed differently according to the context in which they are used. Sometimes, we may like to think of them as functions that "take two arguments". Other times, it is helpful to view them as functions that take a single argument and return a function. Of course, they are exactly the same function in both cases; the only difference is the glasses through which we're viewing them.

Throughout this book, we will use the second approach, currying, as our standard method of handling multiple arguments. We will assume that standard binary numerical function and predicate names, such as $+$, $-$, $\times$, $/$, $<$, $=$, $>$, denote curried functions with type $N \rightarrow (N \rightarrow N)$ or $N \rightarrow (N \rightarrow Bool)$, where N is a numerical set like the naturals, integers, or rationals. When disambiguation is necessary, the name of the function or predicate will be subscripted with an indication of what numerical source is intended. So, $+_{Nat}$ is addition on the naturals, while $+_{Int}$ is addition on the integers, etc. For example, $(\times_{Int}\ 2)$ denotes a doubling function on integers.

Since infix notation for standard binary functions is so much more familiar than the curried prefix form, we will typically use infix notation when both arguments are present. Thus, the expression $(3\ +_{Int}\ 4)$ is synonymous with $(+_{Int}\ 3\ 4)$.

We will also assume the existence of a curried three-argument conditional function if_S with type

$$Bool \rightarrow (S \rightarrow (S \rightarrow S))$$

that returns the second argument if the first argument is *true*, and returns the third argument if the first argument is *false*. E.g.,

$$(if_{Nat}\ (1\ =_{Nat}\ 1)\ \ 3\ \ 4) = 3$$

$$(if_{Nat}\ (1\ =_{Nat}\ 2)\ \ 3\ \ 4) = 4$$

Multiple Results

The handling of multiple return values parallels the handling of multiple arguments. Again, there are two common approaches:

1. Return a tuple of the results. Under this approach, a quotient-and-remainder function *quot&rem* on natural numbers would have type

   $$Nat \rightarrow (Nat \rightarrow (Nat \times Nat)).$$

   Some sample applications using this approach:

$$(quot\&rem\ 14\ 4) = \langle 3,\ 2 \rangle$$

$$(quot\&rem\ 21\ 5) = \langle 4,\ 1 \rangle$$

2. Suppose the goal is to define a function f of k arguments that "returns" n results. What we will do instead is define a function f' that accepts the k arguments that f would, but in addition also takes a special extra argument called a **receiver**. The value returned by f' is the result of applying the receiver to the n values we want f to return. The receiver indicates how the two returned values can be combined into a single value. For example:

$$(quot\&rem\ 14\ 4\ -_{Int}) = (3\ -_{Int}\ 2) = 1$$

$$(quot\&rem\ 14\ 4\ \times_{Int}) = (3\ \times_{Int}\ 2) = 6$$

In these examples the type of *quot&rem* is

$$Int \rightarrow (Int \rightarrow ((Int \rightarrow Int) \rightarrow Int))$$

In general, the notation

$$(f'\ a_1\ \ldots\ a_m\ r)$$

can be pronounced "Apply r to the n results of the application of f to $a_1 \ldots a_m$." Note how this pronunciation mentions the f upon which f' is based.

We will use both of these approaches for returning multiple values. The second approach probably seems mighty obscure and bizarre at first reading, but it will prove to be a surprisingly useful technique in many situations. In fact, it is just a special case of a more general technique called **continuation-passing style** that is studied in Chapters 11 and 16.

A.2.6 Lambda Notation

Up to this point, the only notation we've had to express new functions is a combination of tuple notation and set builder notation. For example, the squaring function on natural numbers can be expressed by the notation:

$$square = \langle Nat,\ Nat,\ \{\langle n,\ n^2\rangle \ |\ n \in Nat\}\rangle$$

This notation is cumbersome for all but the simplest of functions.

For our metalanguage, we will instead adopt **lambda notation** as a more compact and direct notation for expressing functions. The lambda notation version of the above *square* function is:

$$square : Nat \rightarrow Nat = \lambda n\ .\ (n\ \times_{Nat}\ n)$$

Here, the source and target of the function are encoded in the type that is attached to the function name. The Greek lambda symbol, proc, introduces an **abstraction** that specifies the graph of the function, i.e., how the function maps its argument to a result. An abstraction has the form

$$\lambda\ formal\,.\ body$$

where *formal* is a **formal parameter** variable that ranges over the source of the function, and *body* is a metalanguage expression, possibly refering to the formal parameter, that specifies a result in the target of the function. The abstraction λ *formal*. *body* is pronounced "A function that maps *formal* to *body*."

For a function with type $A \rightarrow B$, an abstraction defines the graph of the function to be the following subset of $A \times B$:

$$\{\langle formal,\ body\rangle \ |\ formal \in A \text{ and } body \text{ is defined}\}$$

Thus, the abstraction λn . $(n\ \times_{Nat}\ n)$ specifies the graph:

$$\{\langle n,\ (n\ \times_{Nat}\ n)\rangle\}$$

The condition that *body* be defined (i.e., is not undefined) handles the fact that the function defined by the abstraction may be partial. For example, consider a reciprocal function defined as:

$$recip : Rat \to Rat = \lambda q\ .\ (1\ /_{Rat}\ q)$$

The graph of *recip* defined by the abstraction contains no pair of the form $\langle 0, i_0 \rangle$ because $(1\ /_{Rat}\ 0)$ is undefined.

An important advantage of lambda notation is that it facilitates the expression of higher-order procedures. For example, suppose that *expt* is a binary exponentiation function on natural numbers. Then the *make-expt* function from Section A.2.4 can be expressed succinctly as:

$$make\text{-}expt : Nat \to (Nat \to Nat) = \lambda n_1\ .\ (\lambda n_2\ .\ (expt\ \ n_2\ \ n_1))$$

The abstraction $\lambda n_1\ .\ \ldots$ can be read as "The function that maps n_1 to an exponentiating function that raises its argument to the n_1 power." Similarly, the *apply-to-five* function can be concisely written as:

$$apply\text{-}to\text{-}five : (Nat \to Nat) \to Nat = \lambda f\ .\ (f\ \ 5)$$

By the type of *apply-to-five*, the argument f is constrained to range over functions with the type $Nat \to Nat$. The lambda notation says that such a function f should map to the result of applying f to 5.

Like applications, all abstractions in our metalanguage must be **well-typed**. An abstraction is well-typed if there is a unique way to assign a type to its formal parameter such that its body expression is well-typed. If the type of body is T when the formal parameter is assumed to have type S, then the abstraction has type $S \to T$.

The type of an abstraction is often explicitly specified, as in the above definitions of *square*, *make-expt*, and *apply-to-five*. If the type of an abstraction has been explicitly specified to be $S \to T$, then the type of the formal parameter must be S. For example, in the definition

$$square : Nat \to Nat = \lambda n\ .\ (n\ \times_{Nat}\ n)$$

the formal parameter n has type *Nat* within the body expression $(n\ \times_{Nat}\ n)$. This body expression has type *Nat* and so is well-typed. On the other hand, the definition

$$dec : Nat \rightarrow Nat = \lambda n \,.\, (-1 \;\; +_{Nat} \;\; n)$$

is not well-typed because in the body of the abstraction $+_{Nat}$ is applied to an argument, -1, that is not of type *Nat*.

The type of a formal parameter can always be extracted from a type explicitly specified for an abstraction. However, even when the type of the abstraction is not supplied, it is often possible to determine the type of a formal parameter based on constraints implied by the body expression. For example, in the abstraction

$$\lambda x \,.\, (1 \;\; +_{Int} \;\; x)$$

the formal parameter x must be of type *Int* because the application involving $+_{Int}$ is only legal when both arguments are elements of type *Int*. However, sometimes there aren't enough constraints to unambiguously reconstruct the types of formal parameters. For example, in

$$\lambda f \,.\, (f \;\; 5)$$

the type of the formal parameter f must be of the form $N \rightarrow T$, where N is a numeric type; but there are many choices for N, and T is totally unconstrained. This is a case where an explicit type must be given to the abstraction.

An abstraction of type $S \rightarrow T$ can appear anywhere that an expression of type $S \rightarrow T$ is allowed. For example, an application of the squaring function to the result of adding 2 and 3 is written:

$$((\lambda n \,.\, (n \;\; \times_{Nat} \;\; n)) \;\; (2 \;\; +_{Nat} \;\; 3))$$

Such an application can be simplified by any manipulation that maintains the meaning of the expression. For instance:

$$\begin{aligned}
&((\lambda n \,.\, (n \;\; \times_{Nat} \;\; n)) \;\; (2 \;\; +_{Nat} \;\; 3)) \\
&= ((\lambda n \,.\, (n \;\; \times_{Nat} \;\; n)) \;\; 5) \\
&= (5 \;\; \times_{Nat} \;\; 5) \\
&= 25
\end{aligned}$$

In the next to last step above, the number 5 was substituted for the formal n in the body expression $(n \times_{Nat} n)$. This step is justified by the meaning of application in conjunction with the function graph specified by the abstraction. As another sample application, consider:

$$\begin{aligned}&(\mathit{make\text{-}expt}\ 3)\\ &= ((\lambda n_1 \,.\, (\lambda n_2 \,.\, (\mathit{expt}\ \ n_2\ \ n_1)))\ \ 3)\\ &= \lambda n_2 \,.\ \ (\mathit{expt}\ \ n_2\ \ 3)\end{aligned}$$

In this case, the result of the application is a cubing function.

Often the same abstraction can be used to define many different functions. For example, the expression $\lambda a \,.\, a$ can be used to define the graph of any identity or inclusion function. Because the variable a ranges over the source, though, the resulting graphs are different for each source. A family of functions defined by the same abstraction is said to be a **polymorphic function**. We will often parameterize such functions over a type or types to take advantage of their common structure. Thus, we can define the polymorphic identity function as

$$\mathit{identity}_A : A \rightarrow A = \lambda a \,.\, a$$

where the subscript A means that $\mathit{identity}_A$ defines a family of functions indexed by the type A. We specify a particular member of the family by fixing the subscript to be a known type. So $\mathit{identity}_{Int}$ is the identity function on integers, and $\mathit{identity}_{Bool}$ is the identity function on booleans.

There are several conventions that are used to make lambda notation more compact:

- It is common to abbreviate nested abstractions by collecting all the formal parameters and putting them between a single λ and dot. Thus,

  $$\lambda a_1 \,.\, \lambda a_2 \,.\, \ldots . \lambda a_n \,.\, \mathit{body}$$

  can also be written as

  $$\lambda a_1 a_2 . . . a_n \,.\, \mathit{body}$$

This abbreviation promotes the view that curried functions accepts "multiple arguments"; $\lambda a_1 a_2 \ldots a_n \,.\, body$ can be considered a specification for a function that "takes n arguments".

- Formal parameter names are almost always single characters, perhaps annotated with a subscript or prime. This means that whitespace separating such names can be removed without resulting in any ambiguity. In combination with the left-associativity of application, these conventions allow $\lambda a\ b\ c \,.\, ((b\ c)\ a)$ to be written as $\lambda abc \,.\, bca$.

- Nested abstractions are potentially ambiguous since it's not always apparent where the body of each abstraction ends. For example, the abstraction $\lambda x \,.\, \lambda y \,.\, yx$ could be parsed either as $\lambda x \,.\, \lambda y \,.\, (yx)$ or as $\lambda x \,.\, (\lambda y \,.\, y)x$. The following disambiguating convention is used in such cases: the body of an abstraction is assumed to extend as far right as explicit parentheses allow. By this convention, $\lambda x \,.\, \lambda y \,.\, yx$ means $\lambda x \,.\, (\lambda y \,.\, (yx))$.

A.2.7 Recursion

Using lambda notation, it is possible to write **recursive** function specifications: functions that are directly or indirectly defined in terms of themselves. For example, the factorial function *fact* on natural numbers can be defined as:

$$fact : Nat \rightarrow Nat = \lambda n \,.\, (if_{Nat}\ (n\ =_{Nat}\ 0)\ 1\ (n\ \times_{Nat}\ (fact\ (n\ -_{Nat}\ 1))))$$

We can argue that *fact* is defined on all natural numbers based on the principle of mathematical induction. That is, for the base case of an argument equal to 0, the definition clearly specifies the value of *fact* to be 1. For the inductive case, assume that *fact* is defined for the argument m. Then, according to the definition, the value of $(fact\ (m+1))$ is $((m+1)\ \times_{Nat}\ (fact\ m))$. But by the assumption that $(fact\ m)$ is defined, this expression has a clear meaning. So $(fact\ (m+1))$ is also defined. By induction, *fact* is defined on every element of *Nat*, so the above definition determines a unique total function.

There are many recursive definitions for which the above kind of inductive argument fails. Consider the definition of the *strange* function given below:

$$strange : Nat \rightarrow Nat = \lambda n \,.\, (if_{Nat}\ (even?_{Nat}\ n)\ 0\ (strange\ (n\ +_{Nat}\ 2)))$$

(Assume that $even?_{Nat}$ is a predicate that tests whether its argument is even.) Clearly the function *strange* maps every even number to 0. But what does it map odd numbers to? Induction does not help us because the argument never gets smaller. If we think in terms of function graphs, then we see that for any natural number c, the above definition is consistent with a graph of the form

$$\{\langle 2n,\ 0\rangle \mid n \in Nat\} \cup \{\langle 2n+1,\ c\rangle \mid n \in Nat\}$$

So the specification for *strange* is ambiguous; it designates any of an infinite number of function graphs!

The *strange* example illustrates that recursive definitions need to be handled with extreme care. For now, we will assume that the only case in which a recursive definition has a well-defined meaning is one for which it is possible to construct an inductive argument of the sort used for *fact*. Chapter 7 presents a technique for determining the meaning of a broad class of recursive definitions that includes functions like *strange*.

A.2.8 Lambda Notation is not Lisp!

Those familiar with a dialect of the Lisp programming language may notice a variety of similarities between lambda notation and Lisp. (Those unfamiliar with Lisp may safely skip this section.) Although Lisp is in many ways related to our metalanguage, we emphasize that there are some crucial differences:

- Our metalanguage requires all expressions to be well-typed. In particular, source and target types must be provided for every abstraction. Most dialects of Lisp, on the other hand, have no notion of a well-typed expression, and they provide no mechanism for specifying argument and result types for procedures.[8]

- Most Lisp-like languages support procedures that handle multiple arguments. Because abstractions specify mathematical functions, they always take a single argument. However, the notion of multiple arguments can be simulated by currying or tupling.

[8]The FX language [GJSO92] is a notable exception.

- Every parenthesis in a Lisp expression is required, but parentheses are only strictly necessary in our lambda notation to override the default way in which an expression is parsed. Of course, extra parenthesis may be added to clarify a metalanguage expression.

- Lisp dialects are characterized by evaluation strategies that determine details like which subexpressions of a conditional are evaluated and when argument expressions are evaluated relative to the evaluation of a procedure body. Our metalanguage, on the other hand, is not associated with any notion of a dynamic evaluation strategy. Rather, it is just a notation to describe the graph of a function, i.e., a set of argument/result pairs. Any reasoning about an abstracion is based on the structure of the graph it denotes.

  For example, compare the metalanguage abstraction

  $$\lambda a \,.\, (\mathit{if}_{Nat}\ (\mathit{even?}_{Nat}\ a)\ (a\ +_{Nat}\ 1)\ (a\ \times_{Nat}\ 2))$$

  with the similar Lisp expression

  ```
  (lambda (a) (if (even? a) (+ a 1) (* a 2)))
  ```

  In the case of Lisp, only one branch of the conditional is evaluated for any given argument `a`; if `a` is even, then `(+ a 1)` is evaluated, and if it's odd, `(* a 2)` is evaluated. In the case of the metalanguage, the value of the function for any argument a is the result of applying the *if* function to the three arguments $(\mathit{even?}_{Nat}\ a)$, $(a\ +_{Nat}\ 1)$, and $(a\ \times_{Nat}\ 2)$. Here there is no notion of evaluation, no notion that some event does or does not happen, and no notion of time. The expression simply designates the mathematical function:

  $$\langle \mathit{Nat},\ \mathit{Nat},\ \{\langle 0,\ 1\rangle,\ \langle 1,\ 2\rangle,\ \langle 2,\ 3\rangle,\ \langle 3,\ 6\rangle,\ \langle 4,\ 5\rangle,\ \langle 5,\ 10\rangle,\ \ldots\}\rangle$$

  In fact, a metalanguage abstraction can be viewed as simply a structured name for a particular function.

Although there are many differences between Lisp and lambda notation, the two obviously share some important similarities. Some **functional programming languages** have features that are even more closely patterned after lambda notation. (The FL language presented in Chapter 8 is an example.) However, our purpose for introducing lambda notation here is to have a convenient notation for expressing mathematical functions, not for writing programs. The relationship between mathematical functions and programs is the essence of semantics, which is studied in the main text of the book.

A.3 Domains

A.3.1 Motivation

It turns out that sets and set-theoretic functions have too simple a structure to model some important aspects of the semantics of programming languages. Yet, we would like to proceed with the simplifying assumption that sets are adequate for our purposes until the need for more structure arises. And when we do augment sets with more structure (see Chapter 7), we would prefer not to throw away all of the concepts and notations developed up to that point and start from scratch.

To protect against such a disaster, we will use the same technique that good programmers use to guarantee that their code can be easily modified: **abstraction**.[9] The essence of abstraction is constructing an **abstraction barrier** or **interface** that clearly separates behavior from implementation. In programming, an interface usually consists of a collection of procedures that manipulate elements of an abstract data type. The data type is abstract in the sense that it can only be manipulated by the procedures in the interface; its internal representation details are hidden. The power of abstraction is that changes to the representation of a data type are limited to the implementation side of the barrier; as long as the interface specification is maintained, no client of the abstraction needs to be modified.

We introduce an abstract structure called a **domain** that will serve as our basic entity for modelling programming languages. Domains are set-like struc-

[9]Note that this use of the term "abstraction" is different from that used in the previous section, where it meant a metalanguage expression that begins with a proc.

tures that have constituent elements, but may have other structure as well. The interface to domains is specified by a collection of **domain constructors** introduced below. In our initial naïve implementation, domains will just be sets. In Chapter 7, however, we will change this implementation by extending the sets with additional structure.

Together, domains and domain constructors define a simple domain language. The language comes equipped with a collection of fundamental building blocks called **primitive domains**. These cannot be decomposed into simpler domains. Domain constructors build more complex domains from simpler ones. The resulting **compound domains** can be decomposed into the parts out of which they were made.

In the naïve implementation of domains, primitive domains are sets whose elements have no structure. That is, the elements of primitive domains may be items like numbers, truth values, or the unit value; but they may not be tuples, sets, or functions. Examples of primitive domains include *Unit*, *Bool*, *Int*, *Nat*, and *Rat*.

Compound domains are built by four domain contructors: $\times$, $+$, *, and $\rightarrow$. We shall study these in turn.

A.3.2 Product Domains

The product of two domains, written $D_1 \times D_2$, is the domain version of a Cartesian product. Elements of a compound domain are created by an appropriate **constructor** function. In the case of products, the constructor *tuple* creates elements of the product domain, which are called **tuples**. We will extend the type notation, $d : D$ to indicate that d is an element of the domain D. If $d_1 : D_1$ and $d_2 : D_2$ then

$$(\mathit{tuple}_{D_1, D_2}\ d_1\ d_2) : D_1 \times D_2$$

The subscripts on *tuple* emphasize that it is really a family of functions indexed by the component domains. For example, $\mathit{tuple}_{Nat, Bool}$ and $\mathit{tuple}_{Int, Int}$ both serve to pair elements, but the fact that they have different sources, targets, and graphs makes them different functions.

We will abbreviate $(\mathit{tuple}_{D_1, D_2}\ d_1\ d_2)$ as $\langle d_1,\ d_2\rangle_{D_1, D_2}$, and will drop the subscripts when they are clear from context. For example, the product of *Nat* and *Bool* technically is

$$\begin{array}{l} Nat \times Bool = \\ \quad \{(tuple_{Nat,\,Bool}\ 0\ \ false),\ (tuple_{Nat,\,Bool}\ 1\ \ false),\ (tuple_{Nat,\,Bool}\ 2\ \ false),\ \ldots, \\ \quad\ (tuple_{Nat,\,Bool}\ 0\ \ true),\ (tuple_{Nat,\,Bool}\ 1\ \ true),\ (tuple_{Nat,\,Bool}\ 2\ \ true),\ \ldots\} \end{array}$$

but we will usually write it as

$$\begin{array}{ll} Nat \times Bool = & \{\langle 0,\ false\rangle,\ \langle 1,\ false\rangle,\ \langle 2,\ false\rangle,\ \ldots, \\ & \ \langle 0,\ true\rangle,\ \langle 1,\ true\rangle,\ \langle 2,\ true\rangle,\ \ldots\} \end{array}$$

Domains of n-tuples (known as n-ary products) are written

$$\prod_{i=1}^{n} D_i = D_1 \times D_2 \times \cdots \times D_n = \{\langle d_1, d_2, \ldots d_n\rangle_{D_1,\,D_2,\ldots,\,D_n}\ \mid\ d_i : D_i\}$$

The notation D^n stands for the product of n copies of D.

Every product domain $\prod_{i=1}^{n} D_i$ comes equipped with n **projection functions**

$$Proj\,i_{D_1,\,D_2,\ldots,\,D_n} : (D_1 \times D_2 \times \ldots \times D_n) \rightarrow D_i$$

to extract the ith element from an n-tuple:

$$Proj\,i_{D_1,\ldots,\,D_n}\ \langle d_1, \ldots, d_n\rangle_{D_1,\ldots,\,D_n} = d_i,\ \ 1 \leq i \leq n$$

For example,

$$\begin{array}{rcl} Proj\,1_{Nat,\,Bool}\ \langle 19,\ true\rangle & = & 19 \\ Proj\,2_{Nat,\,Bool}\ \langle 19,\ true\rangle & = & true \end{array}$$

Again, the subscripts indicate that for each i, $Proj\,i$ is a family of functions indexed by the component domains of the tuple being operated on. They will be omitted when they are clear from context.

Notice that we have overloaded the notation $\langle \ldots \rangle$, which may now denote either a set-theoretic tuple or a domain-theoretic one. We have done this because in the simple implementation of domains as sets, product domains simply *are* set-theoretic Cartesian products, and set-theoretic tuples *are* tuples. However, thinking in terms of a concrete implementation for domains can be somewhat dangerous. Product domains are really defined only by the behavior of *tuple* and *Proj i*, which must satisfy the following two properties:

1. $Proj\,i_{D_1,\ldots,D_n}\ (tuple_{D_1,\ldots,D_n}\ d_1\ \ldots\ d_n) = d_i,\ 1 \leq i \leq n$
2. $tuple_{D_1,\ldots,D_n}\ (Proj\,1_{D_1,\ldots,D_n}\ d)\ \ldots\ (Proj\,n_{D_1,\ldots,D_n}\ d) = d,$
   $d : \prod_{i=1}^{n} D_i$

Any implementation of *tuple* and *Proj i* that satisfies these two properties is a valid implementation of products for domains. For example, it's perfectly legitimate to define $tuple_{Nat,\,Bool}$ by

$$tuple_{Nat,\,Bool}\ n\ \ b = \langle b,\ n\rangle,$$

where the order of elements in the concrete (set-theoretic) representation is reversed, as long as $Proj\,i_{Nat,\,Bool}$ is defined consistently:

$$\begin{array}{rcl} Proj\,1_{Nat,\,Bool}\ \langle b,\ n\rangle & = & n \\ Proj\,2_{Nat,\,Bool}\ \langle b,\ n\rangle & = & b \end{array}$$

From here on, the $\langle\ldots\rangle$ notation will by default denote domain-theoretic tuples rather than set-theoretic tuples.

Since writing out compound domains in full can be cumbersome, it is common to introduce synonyms for them via a **domain definition** of the form

$$name = compound\text{-}domain$$

For example, the domain definitions

$$\begin{array}{rcl} Vector & = & Int \times Int \\ Circle & = & Vector\ \times Int \times Bool \end{array}$$

introduces the name *Vector* as a synonym for a domain of pairs of integers and the name *Circle* as a synonym for a domain of triples whose components represent the state of a graphical circle object: the position of its center (a pair of integers), its radius (an integer), and a flag indicated whether or not it is filled (a boolean). Domain definitions are often used merely to introduce more mnemonic names for domains. The following set of domain definitions is equivalent to the set above:

$$\begin{aligned}
\mathit{Vector} &= \mathit{X\text{-}coord} \times \mathit{Y\text{-}coord} \\
\mathit{X\text{-}coord} &= \mathit{Int} \\
\mathit{Y\text{-}coord} &= \mathit{Int} \\
\mathit{Circle} &= \mathit{Position} \times \mathit{Radius} \times \mathit{Filled?} \\
\mathit{Position} &= \mathit{Vector} \\
\mathit{Radius} &= \mathit{Int} \\
\mathit{Filled?} &= \mathit{Bool}
\end{aligned}$$

Domain equality is purely structural and has nothing to do with names. Thus, the assertion $\mathit{Position} = (\mathit{Int} \times \mathit{Int})$ is true because both descriptions designate the domain of pairs of integers.[10]

A.3.3 Sum Domains

Sum domains are analogous to variant records and unions in programming languages. The sum of two domains, written $D_1 + D_2$, is a domain that is the **disjoint union** of the two domains. A disjoint union differs from the usual set union in that each element of the union is effectively "tagged" to indicate which component set it comes from. An element of a sum domain, which we will call a **oneof**, is built by an **injection function**

$$\mathit{Inj}\,i_{D_1,D_2} : D_i \rightarrow (D_1 + D_2)$$

Here, i, which can be either 1 or 2, indicates which component domain the element is from.

A sum domain contains all oneofs that can be constructed from its component domains. E.g.,

$$\begin{aligned}
\mathit{Nat} + \mathit{Int} = \{&(\mathit{Inj}\,1_{\mathit{Nat},\,\mathit{Int}}\ 0),\ (\mathit{Inj}\,1_{\mathit{Nat},\,\mathit{Int}}\ 1),\ (\mathit{Inj}\,1_{\mathit{Nat},\,\mathit{Int}}\ 2),\ \ldots, \\
&(\mathit{Inj}\,2_{\mathit{Nat},\,\mathit{Int}}\ {-2}),\ (\mathit{Inj}\,2_{\mathit{Nat},\,\mathit{Int}}\ {-1}),\ (\mathit{Inj}\,2_{\mathit{Nat},\,\mathit{Int}}\ 0),\ (\mathit{Inj}\,2_{\mathit{Nat},\,\mathit{Int}}\ 1),\ \ldots\}
\end{aligned}$$

[10] It may seem confusing that the equality symbol, $=$, is used both to test domains for equality and to define new domain names. But this confusion is standard in mathematics. In the first case, it is assumed that the meaning of all names is known, and $=$ asserts that the left and right hand sides are equal. In the second case, it is assumed that the meaning of the left hand names are unknown, and the equations are solved to make the $=$ assertions true. In the examples above, the equations are trivial to solve, but domain equations with recursion can be difficult to solve (see Chapter 7).

Note that if the familiar set-theoretic union were performed on the domains *Nat* and *Int*, it would be impossible to determine the source domain for any $n \geq 0$ in the union.

The notion of sum naturally extends to n-ary sums, which are constructed by the notation:

$$\sum_{i=1}^{n} D_i = D_1 + D_2 + \ldots + D_n = \{(Inj\,i_{D_1,\ldots,D_n}\ d_i) \mid d_i : D_i,\}$$

When $S = D_1 + \ldots + D_n$ and all the domain names D_i are distinct, we will write $D_i \mapsto S$ as a synonym for $Inj\,i_{D_1,\ldots,D_n}$. For example, since the *Bool* domain contains only two elements, we can represent it as the sum of two *Unit* domains:

$$\begin{aligned} Bool &= True + False \\ True &= Unit \\ False &= Unit \end{aligned}$$

Then the value *true* would be a synonym for $(True \mapsto Bool\ \mathit{unit})$ and the value *false* would be a synonym for $(False \mapsto Bool\ \mathit{unit})$. If *Bool* were instead described as the sum *Unit* + *Unit*, the mnemonic injection functions could not be used because the name $Unit \mapsto Bool$ would be ambiguous.[11]

Elements of a sum domain are detagged by a **matching** $_{S,\,D}$ construct that maps an element of the domain S into an element of the domain D. A **matching** $_{S,\,D}$ construct has one clause for each possible summand domain in S. If $S = \sum_{i=1}^{n} D_i$ and $s : S$, then the form of this construct is:

$$\begin{array}{l} \textbf{matching}_{S,\,D}\ s \\ \triangleright\ (D_1 \mapsto S\ \ I_1)\ \rrbracket\ E_1 \\ \triangleright\ (D_2 \mapsto S\ \ I_2)\ \rrbracket\ E_2 \\ \quad \ldots \\ \triangleright\ (D_n \mapsto S\ \ I_n)\ \rrbracket\ E_n \\ \textbf{endmatching} \end{array}$$

where each E_i is a metalanguage expression of type D. The subscripts on **matching** will be omitted when they are clear from context.

[11] Note that the alternate injection notation is one place where the name, not the structure of a domain, matters. So even though *True* = *Unit*, the injection function $True \mapsto Bool$ is *not* the same as the $Unit \mapsto Bool$.

In this notation, s is called the **discriminant**, and lines of the form $\triangleright\,(D_i \mapsto S\ I_i)$ $[\!]\ E_i$ are called **clauses**. The part of the clause between the $\triangleright$ and the $[\!]$ is called the **head** of the clause, and the part of the clause after the $[\!]$ is called the **body** of the clause. This notation is pronounced "If the discriminant is the oneof $(\mathit{Inj}\,1_{D_1,\ldots,D_n}\ d_i)$, then the value of the **matching** $_{S,\,D}$ expression is the value of E_i in a context where the identifier I_i stands for d_i."

The head of a clause is treated as a pattern that can potentially be matched by the discriminant. That is, if the discriminant could have been injected into the sum domain by the expression $(D_i \mapsto S\ I_i)$, then in this expression I_i must denote a value from D_i. When such a match is successful, the body is evaluated assuming I_i has this value.

For example, the value of

$$\begin{array}{l}
\textbf{matching}\ (\mathit{Inj}\,1_{\mathit{Nat},\ \mathit{Int}}\ 3) \\
\triangleright\,(\mathit{Nat} \mapsto \mathit{Nat} + \mathit{Int}\ I_{\text{nat}})\ [\!]\ (\mathit{Nat} \hookrightarrow \mathit{Int}\ (I_{\text{nat}}\ +_{\mathit{Nat}}\ 1)) \\
\triangleright\,(\mathit{Int} \mapsto \mathit{Nat} + \mathit{Int}\ I_{\text{int}})\ [\!]\ (I_{\text{int}}\ \times_{\mathit{Int}}\ I_{\text{int}}) \\
\textbf{endmatching}
\end{array}$$

is 4, because the element $(\mathit{Inj}\,1_{\mathit{Nat},\ \mathit{Int}}\ 3)$ matches the head of the first clause, and when I_{nat} is 3, the value of $(I_{\text{nat}}\ +_{\mathit{Nat}}\ 1)$ is 4. Similarly, the value of

$$\begin{array}{l}
\textbf{matching}\ (\mathit{Inj}\,2_{\mathit{Nat},\ \mathit{Int}}\ 3) \\
\triangleright\,(\mathit{Nat} \mapsto \mathit{Nat} + \mathit{Int}\ I_{\text{nat}})\ [\!]\ (\mathit{Nat} \hookrightarrow \mathit{Int}\ (I_{\text{nat}}\ +_{\mathit{Nat}}\ 1)) \\
\triangleright\,(\mathit{Int} \mapsto \mathit{Nat} + \mathit{Int}\ I_{\text{int}})\ [\!]\ (I_{\text{int}}\ \times_{\mathit{Nat}}\ I_{\text{int}}) \\
\textbf{endmatching}
\end{array}$$

is 9. Note that the inclusion function $\mathit{Nat} \hookrightarrow \mathit{Int}$ is necessary to guarantee that body expresion of the first clause has type *Int*.

Since a **matching** construct has one clause for each summand, there is exactly one clause that matches the discriminant. However, for convenience, the distinguished clause head $\triangleright$ **else** may be used as a catch-all to handle all tags unmatched by previous clauses.

When the expression E_{test} denotes a boolean truth value, the notation

$$\textbf{if}\ _D E_{\text{test}}\ \textbf{then}\ E_{\text{if}-\text{true}}\ \textbf{else}\ E_{\text{if}-\text{false}}\ \textbf{fi}$$

is an abbreviation for the case expression

$$\begin{array}{l}\textbf{matching}_{\,Bool,\,D}\ E_{\text{test}}\\ \triangleright\,(True \mapsto Bool\ I_{\text{ignore}})\ [\!]\ E_{\text{if}-\text{true}}\\ \triangleright\,(False \mapsto Bool\ I_{\text{ignore}})\ [\!]\ E_{\text{if}-\text{false}}\\ \textbf{endmatching}\,.\end{array}$$

This abbreviation treats the *Bool* domain as the sum of two *Unit* domains (see page A-30). The **if** is subscripted with the domain D of the result, but we will omit it when it is clear from context. Here, the identifier I_{ignore} should be an identifier that does not appear in either $E_{\text{if}-\text{true}}$ or $E_{\text{if}-\text{false}}$.[12]

Like products, sums are an abstraction that are defined only by the behavior of injection functions and the **matching** construct. In particular, these must satisfy the following two properties:

1. $$\boxed{\begin{array}{l}\textbf{matching}\,(D_i \mapsto S\ \ d_i)\\ \triangleright\,(D_1 \mapsto S\ \ I_1)\ [\!]\ I_1\\ \ \ \vdots\\ \triangleright\,(D_n \mapsto S\ \ I_n)\ [\!]\ I_n\\ \textbf{endmatching}\end{array}} = d_i,\ \ 1 \leq i \leq n$$

2. $$\boxed{\begin{array}{l}\textbf{matching}\,d\\ \triangleright\,(D_1 \mapsto S\ \ I_1)\ [\!]\ (D_1 \mapsto S\ \ I_1)\\ \ \ \vdots\\ \triangleright\,(D_n \mapsto S\ \ I_n)\ [\!]\ (D_n \mapsto S\ \ I_n)\\ \textbf{endmatching}\end{array}} = d,\ \ 1 \leq i \leq n$$

Any implementation of sums in which the injection functions and **matching** satisfy these two properties is a legal implementation of sums.

A.3.4 Sequence Domains

Sequence domains model finite sequences of elements all taken from the same domain. They are built by the * domain constructor; a sequence domain whose sequences contain elements from domain D is written $D*$. An element of a sequence domain is simply called a **sequence**. A sequence is characterized by its length n and its ordered elements, which are indexed from 1 to n.

A length-n sequence over the domain D is constructed by the function

[12] This restriction prevents the variable capture problems discussed in Section 8.3.

$$sequence_{n,D} : D^n \to D^*.$$

Thus $sequence_{3,Int}\langle -5,\ 7,\ -3\rangle$ is a sequence of length three with -5 at index 1, 7 at index 2, and -3 at index 3. We will abbreviate $(sequence_{n,D}\ \ d_1, \ldots, d_n)$. as $[d_1, \ldots, d_n]_D$. So the sample sequence above could also be written $[-5, 7, -3]_{Int}$, and the empty sequence of integers would be written $[\,]_{Int}$. As elsewhere, we will omit the subscripts when they can be inferred from context.

Every sequence domain D^* is equipped with the following constructor, predicate, and selectors:

- $cons_D : D \to D^* \to D^*$
  If $d : D$ and s is a length-n sequence over D^*, then $(cons_D\ \ d\ \ s)$ is a length-$n{+}1$ sequence whose first elements is d and whose ith element is the $i-1$th element of s, $2 \leq i \leq n+1$.

- $empty?_D : D^* \to Bool$
  $(empty?_D\ \ s)$ is *true* if $s\ = [\,]_D$ and *false*otherwise.

- $head_D : D^* \to D$
  If $s : D^*$ is nonempty, $(head_D\ \ s)$ is the first element of s. Defining the *head* of an empty sequence is somewhat problematic. One approach is to treat $(head_D\ \ [\,]_D)$ as undefined, in which case *head*is only a partial function. An alternate approach that treats *head* as a total function is to define $(head_D\ \ [\,]_D)$ as a particular element of D.

- $tail_D : D^* \to D^*$
  If $s : D^*$ is nonempty, $(tail_D\ \ s)$ is the subsequence of the sequence s that consists of all elements but the first element. If s is empty, $(tail_D\ \ s)$ is defined as $[\,]_D$.

Other useful functions can be defined in terms of the above functions:

$length_D : D^* \to Nat$
$= \lambda d^* .\ \ \mathbf{if}\,(empty?_D\ \ d^*)\ \ \mathbf{then}\ 0\ \ \mathbf{else}\ (1\ \ +_{Nat}\ \ (length_D\ \ (tail_D\ \ d^*)))\ \ \mathbf{fi}$

$$nth_D : Pos \rightarrow D* \rightarrow D$$
$$= \lambda p d* \,.\ \textbf{if}\,(p\ =_{Pos}\ 1)\ \ \textbf{then}\,(head_D\ \ d*)\ \ \textbf{else}\,(nth_D\ \ (p\ -_{Pos}\ 1)\ \ (tail_D\ \ d*))\ \ \textbf{fi}$$

$$append_D : D* \rightarrow D* \rightarrow D*$$
$$\begin{aligned} = \lambda d_1{*}d_2{*} \,.\ & \textbf{if}\,(empty?_D\ \ d_1{*})\ \ \textbf{then}\ d_2{*} \\ & \quad \textbf{else}\,(cons_D\ \ (head_D\ \ d_1{*})\ \ (append_D\ \ (tail_D\ \ d_1{*})\ \ d_2{*}))\ \ \textbf{fi} \end{aligned}$$

$length_D$ returns the length of a sequence. nth_D returns the element of the given sequence at the given index. $append_D$ concatenates a length-m sequence and a length-n sequence to form a length-$m+n$ sequence.

In the above definitions, we use the convention that if d is a variable ranging over the domain D, we will typically use $d*$ as a variable to range over the domain $D*$. All of the above function definitions exhibit a simple form of recursion in which the size of first argument is reduced at every recursive call; by the principle of mathematical induction, all of the functions are therefore well-defined.

The *cons* and *append* functions are common enough to warrant some convenient abbreviations:

- $d \,.\, d*$ is an abbreviation of $(cons\ \ d\ \ d*)$. The dot, ".", is an infix binary function that naturally associates to the right. Thus, $d_1 \,.\, d_2 \,.\, d*$ is parsed as $d_1 \,.\, (d_2 \,.\, d*)$.
- $d_1{*} \,@\, d_2{*}$ is an abbreviation of $(append\ \ d_1{*}\ \ d_2{*})$. The at sign, @, is an associative infix binary operator.

As with products and sums, sequences are defined purely in terms of their abstract behavior. A legal implementation of sequence domains is one which satisfies the following properties for all domains D and all $d : D$ and $d* : D*$

1. $(empty?_D\ \ [\,]_D)\ = true$
2. $(empty?_D\ \ d \,.\, d*)\ = false$
3. $(head_D\ \ d \,.\, d*)\ = d$
4. $(head_D\ \ [\,]_D)\ = d_{empty-head}$, where $d_{empty-head}$ is a particular element of D chosen for this purpose.

5. $(tail_D\ \ d\,.\,d*)\ = d*$

6. $(tail_D\ \ [\,]_D)\ = [\,]_D$

7. $(cons_D\ \ (head_D\ \ d*)\ \ (tail_D\ \ d*))\ = d*$

A.3.5 Function Domains

The final constructor we will consider is the binary infix function domain constructor, $\rightarrow$. In the naïve implementation of domains as sets, $D_1 \rightarrow D_2$ is the domain of all total functions with D_1 as their source and D_2 as their target. Elements of a function domain are called **functions**. As with tuples, there is the possibility for confusion between set-theoretic functions and domain-theoretic functions. These are the same in the naïve implementation, but will differ when we change the implementation of domains. Henceforth, function will mean domain-theoretic function by default; we will explicitly use the term "set-theoretic function" to override the default. The same holds for arrow notation, which from now on will always refer to the function domain constructor unless otherwise specified.

The arrow notation meshes nicely with the use of arrows already familiar from set-theoretic function types. Thus, the notation $f : \mathit{Int} \rightarrow \mathit{Bool}$ can now be intepreted as "f is an element of the function domain $\mathit{Int} \rightarrow \mathit{Bool}$." Elements of this domain are predicates on the integers, such as functions testing whether an integer is even or odd, or for testing whether an integer is positive or negative. Similarly, the domain $\mathit{Int} \rightarrow (\mathit{Int} \rightarrow \mathit{Int})$ is the domain of all partial functions on two (curried) integer arguments that return an integer. The (curried) binary integer addition, multiplication, etc., functions are all elements of this domain.

The $\rightarrow$ constructor is right-associative:

$$D_1 \rightarrow D_2 \rightarrow \cdots \rightarrow D_{n-1} \rightarrow D_n \ \text{ means } \ (D_1 \rightarrow (D_2 \rightarrow \cdots (D_{n-1} \rightarrow D_n) \cdots))$$

The right-associativity of $\rightarrow$ interacts nicely with the left associativity of application in lambda notation. That is, if $a : A$, $b : B$, and $f : (A \rightarrow B \rightarrow C)$, then $(f a) : B \rightarrow C$, so that $(f\ \ a\ \ b) : C\ = ((f\ \ a)\ \ b) : C$, just as we'd like.

We write particular elements of a function domain using lambda notation. Thus

$$(\lambda n \,.\, (n \ \times_{Nat} \ n)) : Nat \rightarrow Nat$$

is the squaring function on natural numbers, and

$$(\lambda i \,.\, (i \ >_{Int} \ 0)) : Int \rightarrow Bool$$

is a predicate for testing whether an integer is positive.

As before, we require all abstractions to be well-typed. We can always specify the type of an abstraction by giving it an explicit type. So

$$procx.x : Int \rightarrow Int$$

specifies the identity function on integers, while

$$procx.x : Bool \rightarrow Bool$$

specifies the identity function on booleans.

However, to enhance the readability of abstractions, we will use a convention in which each domain of interest has associated with it a **domain variable** that ranges over elements of the domain. For example, consider the following domain definitions:

$$\begin{array}{lcl} b & \in & Bool \\ n & \in & Nat \\ p & \in & Nat\text{-}Pred \ = \ Nat \rightarrow Bool \end{array}$$

The domain variable b ranges over the $Bool$ domain, the domain variable n ranges over the Nat domain, and the domain variable p ranges over the function domain $Nat \rightarrow Bool$.

Domain variables, possibly in subscripted or primed form, are used in metalanguage expressions to indicate that they denote only entities from their associated domain. Thus $(\lambda b \,.\, b)$ and $(\lambda b_1 \,.\, b_1)$ unambiguously denote the identity function in the domain $Bool \rightarrow Bool$, $(\lambda n \,.\, n)$ and $(\lambda n' \,.\, n')$ both denote the identity function in the domain $Nat \rightarrow Nat$, and $(\lambda p \,.\, p)$ denotes the identity function in the domain

$$Nat\text{-}Pred \rightarrow Nat\text{-}Pred = (Nat \rightarrow Bool) \rightarrow (Nat \rightarrow Bool) \ .$$

As another example, the expression $(\lambda n \,.\, \lambda p \,.\, p\, n)$ is an element of the function domain

$$Nat \rightarrow Nat\text{-}Pred \rightarrow Bool = Nat \rightarrow (Nat \rightarrow Bool) \rightarrow Bool\ .$$

In practice, we will use both explicit and implicit typing of domain elements. When we define a value named v from a domain D, we will first write a type of the form $v : D$ that specifies that v names an element from D. Then we will give a definition for the name that uses domain variables where appropriate. So an integer identity function is written

$$integer\text{-}identity : Int \rightarrow Int = proci.i$$

and the notation for an identity parameterized over a domain D is:

$$identity_D : D \rightarrow D = procd.d$$

In fact, we have already used this notation to describe the operations on a sequence domain in Section A.3.4.

Our description of function domains in this section has a different flavor than the description of product, sum, and sequence domains. With the other domains, elements of the compound domain were abstractly defined by assembly functions that had to satisfy certain properties with respect to disassembly functions. But with function domains, we concretely specify the elements as set-theoretic functions designated by lambda notation. Is there a more abstract approach to defining function domains? Yes, but it is rather abstract and not important to our current line of development. See Exercise A.3.

▷ **Exercise A.1** It is natural to represent a oneof in $\sum_{i=1}^{n} D_i$ as a set-theoretic pair containing the tag i and an element d_i of D_i:

$$(Inj\, i_{D_1,\ldots,D_n}\ d_i) = \langle i,\ d_i \rangle$$

Assuming that oneofs are represented as pairs, use lambda notation to construct a set-theoretic function of a oneof argument $s \in D_1 + D_2$ that has the same meaning as the following **matching** expression:

matching s
$\triangleright\, (D_1 \mapsto S\ I_1)\ \rrbracket\ E_1$
$\triangleright\, (D_2 \mapsto S\ I_2)\ \rrbracket\ E_2$
endmatching

(Use the three argument if_T function on page A-16 rather than the **if** abbreviation on page A-31, which itself is implemented in terms of a **matching** expression. Assume E_1 and E_2 are of type T.)

◁

▷ **Exercise A.2** This exercise explores some further properties of products. Suppose that A, B, and C are any domains. Extend the notation $\times$ so that it defines a binary infix operator on functions with the following signature:

$$((A \to B) \times (A \to C)) \to (A \to (B \times C))$$

If $f : A \to B$, $g : A \to C$, and $a : A$, then $f \times g : A \to (B \times C)$ is defined by:

$$(f \times g)\ a = \langle (f\ a),\ (g\ a) \rangle$$

a. Suppose that $f : A \to B$ and $g : A \to C$ are set-theoretic functions. Show the following two equivalences:

$$\mathit{Proj}\,1 \circ (f \times g) = f$$

$$\mathit{Proj}\,2 \circ (f \times g) = g.$$

(Recall that two set-theoretic functions are equal when their sources, targets, and graphs are all equal).

b. Suppose that $h : A \to (B \times C)$ is a set-theoretic function. Show that

$$h = (\mathit{Proj}\,1\ \ h) \times (\mathit{Proj}\,2\ \ h)$$

The above two properties are an alternate way of describing the abstract properties that a product must satisfy. ◁

▷ **Exercise A.3** This exercise explores some further properties of function domains. Consider the following two functions:

- $\mathit{apply}_{A,B} : ((A \to B) \times A) \to B$ If $f : A \to B$ and $a : A$, then $(\mathit{apply}_{A,B}\ \langle f,\ a \rangle)$ denotes the result of applying f to a.
- $\mathit{curry}_{A,B,C} : ((A \times B) \to C) \to (A \to (B \to C))$ If $f : (A \times B) \to C$, then $(\mathit{curry}_{A,B,C}\ \ f)$ denotes a curried version of f — i.e., it denotes a function g such that $(g\ a\ b) = (f\ \langle a,\ b \rangle)$ for all $a \in A$ and $b \in B$.

a. Use lambda notation to define set-theoretic versions of *apply* and *curry*.

b. Using your definitions from above, show that if $f : (A \times B) \to C$, then

$$f = apply_{(A \to B \to C),B} \circ ((curry_{A,B,C}\ f) \times id_B)$$

The meaning of $\times$ on functions is defined in Exercise A.2. Recall that id_D is the identity function on domain D.

c. Using your definitions from above, show that if $g : A \to (B \to C)$, then

$$g = (curry_{A,B,C}\ (apply_{B,C} \circ (g \times id_B)))$$

It turns out that any domain implementation with an *apply* and a *curry* function that satisfy the above properties is a legal implementation of a function domain. This is the abstract view of function domains alluded to earlier. ◁

A.4 Metalanguage Summary

So far we've introduced many pieces of the metalanguage. The goal of this section is to put all of the pieces together. We'll summarize the metalanguage notation introduced so far, and introduce a few more handy notations.

In the study of programming languages, it is often useful to break up the description of a language into two parts: the core of the language, called the **kernel**, and various extensions that can be expressed in terms of the core, called the **syntactic sugar**. We shall use this approach to summarize the metalanguage. (See Section 8.2 for an example of using this approach to specify a programming language.)

A.4.1 The Metalanguage Kernel

The entities manipulated by the metalanguage are domains and their elements. Domains are either primitive, in which case they can be viewed sets of unstructured elements, or compound, in which case they are built out of component domains. Domains are denoted by **domain expressions**. Domain expressions are either domain names (such as *Bool*, *Nat*, etc.) or are the application of the domain operators $\times$, $+$, *, and $\to$ to other domain expressions. New names can be given to domains via **domain definitions**. Domain definitions can also introduce domain variables that range over elements of the domains.

Domain elements are denoted by **element expressions**. The kernel element expressions are summarized in Figure A.1.

- **constants**: e.g., *0*, *true*, $+_{Nat}$, $tuple_{Nat,\,Bool}$, $Proj\,i_{Nat,\,Bool}$
- **variables**: e.g., a, b', c_2, *fact*
- **applications**: e.g, (*fact* *5*), $((+_{Nat}\ 2)\ 3)$, $((\lambda a\,.\,a)\ 1)$
- **abstractions**: e.g., $(\lambda a\,.\,a)$, $(\lambda b\,.\,1)$, $(\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,(ca)b)$
- **case analysis**:

matching s	or	matching s
$\triangleright\ (D_1 \mapsto S\ \ I_1)\ \rrbracket\ E_1$		$\triangleright\ (D_1 \mapsto S\ \ I_1)\ \rrbracket\ E_1$
$\triangleright\ (D_2 \mapsto S\ \ I_2)\ \rrbracket\ E_2$		$\triangleright\ (D_2 \mapsto S\ \ I_2)\ \rrbracket\ E_2$
...		...
$\triangleright\ (D_n \mapsto S\ \ I_n)\ \rrbracket\ E_n$		$\triangleright$ **else** $\rrbracket\ E_{\text{else}}$
endmatching		**endmatching**

Figure A.1: The kernel element expressions.

Constants are names for primitive domain elements and functions; these include numbers, booleans, and functions. We will assume that the domain of every constant is evident from context. Variables are names introduced as formal parameters in abstractions or as the defined name of a definition. Every variable ranges over a particular domain. If a variable is the domain variable introduced by some domain definition, it is assumed to range over the specified domain; otherwise, the type of the variable should be explicitly provided to indicate what domain it ranges over.

Applications are compound expressions in which an operator is applied to an operand. The operator expression must denote an element of a function domain $S \to T$, and the operand expression must denote an element of the domain S; in this case, the application denotes an element of type T. Applications with multiple operands are usually expressed by currying. Elements of primitive domains are often the operands to functions (such as arithmetic and logical functions) associated with the domain. Elements of product, sum, and sequence domains can be built by the application of constructor functions ($tuple_{D_1,\ldots,D_n}$, $D_i \mapsto S$, or $sequence_{n,D}$, respectively) to the appropriate arguments. Compound domains are equipped with many other useful functions that operate on elements of the domain.

Abstractions are compound expressions that denote the elements of function domains. Structurally, an abstraction consists of a formal parameter variable and a body element expression. An abstraction $(\lambda I \,.\, E_{\text{body}})$ specifies the function graph containing all pairs $\langle I,\, t\rangle$ where I ranges over the source domain of the function and t is the target domain element that is the value of the body expression E_{body} for the given I. The type of the abstraction should either be given explicitly or should be inferrable from the structure of the parts of the abstraction.

While the parts of products and sequences are extracted by function application, elements of a sum domain are disassembled by the **matching** construct. A **matching** construct consists of a discriminant and a set of clauses, each of which has a pattern and a body. There must be one clause to handle each summand in the domain of the discriminant. All body expressions must denote elements of the same domain so that the domain of the value denoted by the **matching** expression is clear.

The element expressions in Figure A.1 are often used in conjunction with definitions to specify a domain element. A definition has the form

$$name : type = expression$$

where *name* is name of the element being defined, *type* is a domain expression that denotes the domain to which the defined element belongs, and *expresion* is metalanguage expression. that specifies the element. Definitions may only be recursive in the case where it can be shown by induction that they define a unique element in the domain specified by the type. One way to do this is to use induction; another way is to use the iterative fixed point technique developed in Chapter 7.

A.4.2 The Metalanguage Sugar

It is *possible* to write all element expressions using the kernel element expressions, but it is not always *convenient* to do so. We have introduced various notational conventions to make the metalanguage more readable and concise. We review those notations here, and introduce a few more.

Figure A.2 summarizes the syntactic sugar for element expressions. Applications and abstractions are simplified by various conventions. The default left-associativity of application simplifies the expression of multi-argument applications; thus, $(\mathit{expt}\ \mathit{2}\ \mathit{5})$ is an abbreviation for $((\mathit{expt}\ \mathit{2})\ \mathit{5})$. This default can be overidden by explicit parenthesization. Applications of familiar functions like $+_{Nat}$ are often written in infix style to enhance readability. For example, $(\mathit{2} +_{Nat} \mathit{3})$ is an abbreviation for $((+_{Nat}\ \mathit{2})\ \mathit{3})$. The formal parameters of nested abstractions are often coalesced into a single abstraction. For instance, $\lambda abc\,.\,ca$ is shorthand for $\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,ca$.

- **applications**: e.g., $(\mathit{expt}\ \mathit{2}\ \mathit{5})$, $(\mathit{2} +_{Nat} \mathit{3})$
- **abstractions**: e.g., $(\lambda abc\,.\,ca)$
- **tuples**: e.g., $\langle \mathit{1}, \mathit{true}\rangle$
- **oneofs**: e.g., $(\mathit{Nat} \mapsto \mathit{Nat} \times \mathit{Int}\ \mathit{3})$
- **sequences**: e.g., $[\mathit{5}, \mathit{3}, \mathit{2}, \mathit{7}]$, $\mathit{5}\,.\,[\mathit{3}, \mathit{2}, \mathit{7}]$, $[\mathit{5}, \mathit{3}]\,@\,[\mathit{2}, \mathit{7}]$
- **if**: **if** E_{bool} **then** $E_{\text{if}-\text{true}}$ **else** $E_{\text{if}-\text{false}}$ **fi**
- **let**: **let** I_1 **be** E_1 **and**
I_2 **be** E_2 **and**
$\vdots$
I_{n} **be** E_{n}
in E_{body}
- **matching**: **matching** E_{disc} or **matching** E_{disc}

matching E_{disc}	**matching** E_{disc}
$\triangleright\ p_1\ [\!]\ E_1$	$\triangleright\ p_1\ [\!]\ E_1$
$\triangleright\ p_2\ [\!]\ E_2$	$\triangleright\ p_2\ [\!]\ E_2$
$\vdots$	$\vdots$
$\triangleright\ p_n\ [\!]\ E_n$	$\triangleright$ **else** $[\!]\ E_n$
endmatching	**endmatching**

Figure A.2: Sugar for element expressions.

The construction of elements in product, sum, and sequence domains is aided by special notation. Thus,

$$\begin{array}{rll} \langle \mathit{1},\ \mathit{true}\rangle & \text{is shorthand for} & (\mathit{tuple}_{\mathit{Nat},\,\mathit{Bool}}\ \mathit{1}\ \mathit{true}) \\ (\mathit{Nat} \mapsto \mathit{Nat} + \mathit{Int}\ \mathit{3}) & \text{is shorthand for} & (\mathit{Inj}\,\mathit{1}_{\mathit{Nat},\mathit{Int}}\ \mathit{3}) \\ [\mathit{5},\mathit{3},\mathit{2},\mathit{7}] & \text{is shorthand for} & \mathit{sequence}_{4,\mathit{Nat}}\ \langle \mathit{5},\ \mathit{3},\ \mathit{2},\ \mathit{7}\rangle \end{array}$$

(We have assumed in all these examples that the numbers are elements of *Nat* rather than of some other numerical domain.) The notations $d\,.\,d^*$ and $d_1{}^*$ @ $d_2{}^*$ are abbreviations for *cons* and *append* respectively, so that the following notations all denote the same sequence of natural numbers:

$$[\mathit{5},\mathit{3},\mathit{2},\mathit{7}] = \mathit{5}\,.\,[\mathit{3},\mathit{2},\mathit{7}] = [\mathit{5},\mathit{3}]\ @\ [\mathit{2},\mathit{7}]$$

The **if** conditional expression

$$\textbf{if } E_{\text{bool}} \textbf{ then } E_{\text{if}-\text{true}} \textbf{ else } E_{\text{if}-\text{false}} \textbf{ fi}$$

is an abbreviation for the following case analysis:

$$\begin{array}{l} \textbf{matching } E_{\text{bool}} \\ \triangleright\ (\mathit{True} \mapsto \mathit{Bool}\ I_{\text{ignore}})\ []\ E_{\text{if}-\text{true}} \\ \triangleright\ (\mathit{False} \mapsto \mathit{Bool}\ I_{\text{ignore}})\ []\ E_{\text{if}-\text{false}} \\ \textbf{endmatching} \end{array}$$

where E_{bool} is an expression that denotes an element of the domain *Bool* and $E_{\text{if}-\text{true}}$ and $E_{\text{if}-\text{false}}$ denote elements from the same domain. The variable I_{ignore} can be any variable that does not appear in $E_{\text{if}-\text{true}}$ or $E_{\text{if}-\text{false}}$. This notation assumes that the *Bool* domain is represented as a sum of two unit domains.

The **let** expression is new:

$$\begin{array}{l} \textbf{let } I_1 \textbf{ be } E_1 \textbf{ and} \\ \quad I_2 \textbf{ be } E_2 \textbf{ and} \\ \quad \vdots \\ \quad I_{\text{n}} \textbf{ be } E_{\text{n}} \\ \textbf{in } E_{\text{body}} \end{array}$$

is pronounced "Let I_1 be the value of E_1 and I_2 be the value of E_2 ... and I_n be the value of E_n in the expression E_body." The **let** expression is used to name intermediate results that can then be referenced by name in the body expression. The value of a **let** expression is the value of its body in a context where the specified bindings are in effect. The **let** expression is just a more readable form of an application of a manifest abstraction:

$$((\lambda I_1 I_2 \ldots I_\mathrm{n} \,.\, E_\mathrm{body})\ E_1\ E_2\ \ldots\ E_\mathrm{n})$$

The **matching** expression is extended to simplify the extraction of tuple and sequence components:

$$\begin{array}{l}
\textbf{matching}\ E_\mathrm{disc} \\
\triangleright\ p_1\ \|\ E_1 \\
\triangleright\ p_2\ \|\ E_2 \\
\vdots \\
\triangleright\ p_n\ \|\ E_n \\
\textbf{endmatching}
\end{array}$$

As before, a **matching** expression consists of a discriminant and a number of clauses. The two parts of a **matching** clause are called the **pattern** and the **body**. A pattern is composed out of constants, variables, and tuple and sequence constructors; for example, the following are typical patterns:

$$\begin{array}{l}
\langle n,\ 1\rangle, \\
\langle\langle w,\ x\rangle,\ y,\ z\rangle, \\
[i_1, -3, i_2], \\
n \,.\, n^*, \\
n_1{}^* \,@\, n_2{}^*
\end{array}$$

A pattern is said to **match** a value v if it is possible to assign values to the variables such that the pattern would denote v if it were interpreted as an element expression with the assignments in effect. Thus, the pattern $\langle n,\ 1\rangle$ matches the value $\langle 2,\ 1\rangle$ with $n = 2$, but it does not match the values $\langle 3,\ 4\rangle$ or $\langle 2,\ 1,\ 3\rangle$. Similarly, $n \,.\, n^*$ matches $[3, 7, 4]$ with $n = 3$ and $n^* = [7, 4]$, but it does not match $[\,]$.

The value of the **matching** expression is determined by the first clause (reading top down) whose pattern matches the discriminant. In this case, the value of the **matching** expression is the value of the clause body in a context where all the variables introduced by the pattern are assumed to denote the value determined by the match. For example, consider the following expression, where $d : Nat \times Nat$:

$$\begin{array}{l} \textbf{matching}\ d \\ \triangleright \langle n,\ 1\rangle \ [\!]\ (n\ -_{Nat}\ 1) \\ \triangleright \langle n,\ 2\rangle \ [\!]\ (n\ \times_{Nat}\ n) \\ \triangleright \langle n_1,\ n_2\rangle \ [\!]\ (n_1\ +_{Nat}\ n_2) \\ \textbf{endmatching} \end{array}$$

If the second component of d is 1, then the value of the **matching** expression is one less than the first component; if the second component is 2, then the value of the **matching** expression is the square of the first component; otherwise, the value of the **matching** expression is the sum of the two components. As before, the last clause of the **matching** expression can have an $\triangleright$ **else** pattern that handles any discriminant that did not successfully match the preceding patterns. A **matching** expression is ill-formed if no pattern matches the discriminant.

A **matching** expression can always be rewritten in terms of conditional expressions and explicit component extraction functions. Thus, the above **matching** clause is equivalent to:

$$\begin{array}{l} \textbf{if}\ (Proj2_{Nat,\,Nat}\ d)\ =_{Nat}\ 1 \\ \ \textbf{then}\ (Proj1_{Nat,\,Nat}\ d)\ -_{Nat}\ 1 \\ \ \textbf{else if}\ (Proj2_{Nat,\,Nat}\ d)\ =_{Nat}\ 2 \\ \qquad \textbf{then}\ (Proj1_{Nat,\,Nat}\ d)\ \times_{Nat}\ (Proj1_{Nat,\,Nat}\ d) \\ \qquad \textbf{else}\ (Proj1_{Nat,\,Nat}\ d)\ +_{Nat}\ (Proj2_{Nat,\,Nat}\ d) \\ \qquad \textbf{fi} \\ \ \textbf{fi} \end{array}$$

In this case, the **matching** expression is more concise and more readable than the desugared form.

In fact, the pattern matching approach is such a powerful notational tool that we shall extend many of our other notations to use implicit pattern matching. For example, we shall allow formal parameters to an abstraction to be patterns rather than just variables. Thus, the abstraction

$$\lambda\langle n_1,\ n_2\rangle\ .\ (n_1\ +_{Nat}\ n_2)$$

specifies a function with type $(Nat \times Nat) \rightarrow Nat$ that is shorthand for

$$\begin{array}{ll} \lambda d\ . & \textbf{matching}\ d \\ & \triangleright \langle n_1,\ n_2 \rangle\ [\!]\ (n_1\ +_{Nat}\ n_2) \\ & \textbf{endmatching} \end{array}$$

where d is assumed to range over $Nat \times Nat$. Similarly, we will allow the variable positions of **let** expressions to be filled by general pattern expressions.

Reading

The concept of domains introduced in this appendix are refined in Chapter 7. See the references there for reading on domain theory.

The defining products, sums, and functions in an abstract way is at the heart of **category theory**. [Pie91] and [BW90] are accessible introductions to category theory aimed at computer scientists.

For coverage of computability issues, we recommend [HU79], [Min67], and[Hof80].