© Rajeev Alur and Thomas A. Henzinger

Chapter 99

Prerequisite Notions and Notations

In this appendix we define some of the mathematical concepts that are used throughout the book. The appendix is not intended as a tutorial on the pertinent topics in discrete mathematics, but only as a concise reference that formalizes our use of background terminology in order to avoid ambiguities. We make no attempt at completeness: many common and (we hope) unambiguous notions, such as the standard operators of naive set theory and boolean logic, are assumed to be familiar to the reader.

Types

A *type* is a set of values together with a set of operations on these values. A type is *finite* if the corresponding set of values is finite. When no confusion arises, we use the same symbol for a type and its set of values. We distinguish between primitive types and composite types. Our primitive types are the booleans $\mathbb{B} = \{true, false\}$ (finite), the nonnegative integers $\mathbb{N} = \{0, 1, 2, \ldots\}$ (infinite), and the positive integers $\mathbb{N}^{>0} = \mathbb{N}\backslash\{0\}$ (infinite), all with the usual operations. Given a type $\mathbb{T}$, the composite type $\mathbb{T}_\perp$ has the set $\mathbb{T} \cup \{\perp\}$ of values, which includes the "undefined" value $\perp \notin \mathbb{T}$, and the operations of $\mathbb{T}$, which behave strictly on $\perp$; that is, every operation that is performed on one or more undefined arguments returns an undefined result. Other composite types are **queue of** $\mathbb{T}$ and **stack of** $\mathbb{T}$. For these two composite types, the values are the finite (possibly empty) sequences of values of type $\mathbb{T}$; thus they are infinite types. The queue and stack types differ in their operations.

The type **queue of** $\mathbb{T}$ represents "first-in-first-out" sequences, called *queues*, of values in $\mathbb{T}$ and supports five operations. In the following, let a be a value in $\mathbb{T}$, let B be a queue of values in $\mathbb{T}$, and let x be a variable of type $\mathbb{T}$. The operation *EmptyQueue* returns the empty queue. The operation $\mathit{Enqueue}(a, B)$ returns the queue that results from adding the value a to the end of the queue B. The operation $\mathit{IsEmpty}(B)$ returns *true* if the queue B is empty, and otherwise returns *false*. If B is a nonempty queue, then the operation $\mathit{Front}(B)$ returns the first element of B and the operation $\mathit{Dequeue}(B)$ returns the queue that results from removing the first element from B. The implementation of queues as linked lists supports all five operations in constant time. We use the notation **foreach** x **in** B **do** to describe a loop whose body is executed once for each element of the queue B, and during successive executions of the loop body the variable x is bound to the successive elements of B, beginning with the first element.

The type **stack of** $\mathbb{T}$ represents "last-in-first-out" sequences, called *stacks*, of values in $\mathbb{T}$ and supports six operations. In the following, let a be a value in $\mathbb{T}$ and let C be a stack of values in $\mathbb{T}$. The operation *EmptyStack* returns the empty stack. The operation $\mathit{Push}(a, C)$ returns the stack that results from adding the value a to the beginning of the stack C. The operation $\mathit{IsEmpty}(C)$ returns *true* if the stack C is empty, and otherwise returns *false*. If C is a nonempty stack, then the operation $\mathit{Top}(C)$ returns the first element of C and the operation $\mathit{Pop}(C)$ returns the stack that results from removing the first element from C. The operation $\mathit{Reverse}(C)$ returns the stack that contains the elements of C in reverse order. The implementation of stacks as linked lists supports all six operations in constant time (doubly linked lists are necessary for stack reversal).

Functions and Relations

Functions and relations can be viewed as special kinds of sets. We take this view only for relations, and consider functions as primitives.

Functions

A *function* f from a set A to a set B maps each element $a \in A$ to a unique element $f(a) \in B$. The set A is called the *domain* of f, and B is the *range* of f. We write $[A \to B]$ for the set of functions with domain A and range B. The function f is *one-to-one* if for all $a, b \in A$, if $a \neq b$, then $f(a) \neq f(b)$, and f is *onto* if for all $b \in B$, there is an element $a \in A$ such that $f(a) = b$. A *bijection* between A and B is a function that is both one-to-one and onto. The bijections between A and A are called the *permutations* on A. A *partial function* g from A to B is a function from A to $B \cup \{\perp\}$, whose range includes the undefined value $\perp \notin B$; the partial function g is *undefined* on $a \in A$ iff $g(a) = \perp$. To emphasize that a function is not partial, it may be called *total*.

The *identity function* on a set A is the function $id \in [A \to A]$ such that $id(a) = a$ for all $a \in A$. The identity function is a bijection. The *inverse function* f^{-1} of a one-to-one function f from A to B is the partial function g from B to A such that for all $b \in B$, we have $g(b) = a$ if $f(a) = b$, and $g(b) = \perp$ if $f(a) \neq b$ for all $a \in A$. The inverse function of a bijection is again a bijection. Given a function $f \in [A \to B]$ and a function $g \in [B \to C]$, the *compound function* $g \circ f$ is the function h from A to C such that $h(a) = g(f(a))$ for all $a \in A$. The composition of two one-to-one (or onto) functions is again one-to-one (or onto). A function f on a set A is extended to subsets of A and to finite and infinite sequences over A in the natural way. Let $B \subseteq A$, and let $\overline{a} = a_0 a_1 \cdots$ be a sequence of elements a_i from A. Then $f(B) = \{f(b) \mid b \in B\}$, and $f(\overline{a}) = f(a_0) f(a_1) \cdots$. A function from A^n to A is called an *n-ary function on A*. Given a unary function f on A, by f^0 we denote the identity function on A, and for all $i \in \mathbb{N}$, by f^{i+1} we denote the compound function $f^i \circ f$, which is again a unary function on A.

Given a binary function $\star$ on A, and given $a, b \in A$, we usually write $a \star b$ instead of $\star(a, b)$. The function $\star$ is *commutative* if $a \star b = b \star a$ for all $a, b \in A$, and $\star$ is *associative* if $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in A$. The element $a \in A$ is an *identity element* with respect to $\star$ if $a \star b = b \star a = b$ for all $b \in A$. An associative binary function on A with an identity element is a *monoid*, and A is called the *carrier* of the monoid. For example, the composition $\circ$ of the unary functions on a set B is a monoid (with carrier $[B \to B]$) whose identity element is the identity function on B. If a is an identity element with respect to $\star$, and $b \star c = c \star b = a$ for $b, c \in A$, then b is an *inverse element* of c with respect to $\star$. If each carrier element of a monoid has an inverse element, then the monoid is a *group*. For example, the composition $\circ$ of the permutations on B is a group (the inverse function of a permutation is an inverse element with respect to composition). For associative binary functions $\star$ we use the following notations. In expressions such as $a \star b \star c$, we may omit parentheses. If the arguments a, b, and c are themselves large expressions, we may use a vertical arrangement

$$\begin{array}{l} \star\ a \\ \star\ b \\ \star\ c \end{array}$$

of the arguments, each preceded by the function symbol. Provided that $\star$ has an identity, if the arguments a, b, and c can be parameterized —say, $a = f(0)$, $b = f(1)$, and $c = f(2)$— we may use the function symbol as a quantifier and write $(\star\ 0 \leq i \leq 2 \mid f(i))$. If the variable that is bound by the quantifier $\star$ ranges over an empty set, then the quantified expression denotes the identity of $\star$; for example, $(+\ i \in \emptyset \mid \ldots) = 0$ and $(\circ\ i \in \emptyset \mid \ldots) = id$.

Binary relations

A (*binary*) *relation* $\sim$ between two sets A and B is a subset of $A \times B$. The set $A \times B$ itself is the *universal relation* between A and B. For $a \in A$ and $b \in B$, we usually write $a \sim b$ instead of $(a, b) \in \sim$. Let $post_\sim(a) = \{b \in B \mid a \sim b\}$. The relation $\sim$ is *serial* if $post_\sim(a)$ is nonempty for all $a \in A$,[1] and $\sim$ is *finitely branching*, if $post_\sim(a)$ is finite for all $a \in A$. In the following, we assume that $A = B$ —in this case we refer to $\sim$ as a binary relation on A. Given $B \subseteq A$, we write $\sim[B]$ for the restriction $\{(a, b) \in B^2 \mid a \sim b\}$ of $\sim$ to B. The relation $\sim$ is *reflexive* if $a \sim a$ for all $a \in A$; *irreflexive* if $a \not\sim a$ for all $a \in A$; *transitive*, if for all $a, b, c \in A$, if $a \sim b$ and $b \sim c$, then $a \sim c$; *symmetric*, if for all $a, b \in A$, if $a \sim b$, then $b \sim a$; *asymmetric*, if for all $a, b \in A$, if $a \sim b$, then $b \not\sim a$; *antisymmetric*, if for all $a, b \in A$, if $a \sim b$ and $b \sim a$, then $a = b$; and *total*, if for all $a, b \in A$, if $a \neq b$, then $a \sim b$ or $b \sim a$. A reflexive and transitive relation is a *preorder*; a symmetric preorder is an *equivalence* (*relation*); an antisymmetric preorder is a *weak partial order*; an irreflexive, asymmetric, and transitive relation is a *strict partial order*; a total (weak or strict) partial order is a (*weak* or *strict*) *linear order*. For a partial order $\sim$, a linear order that is a superset of $\sim$ is called a *linearization* of $\sim$. Every partial order has at least one linearization, and possibly several.

The *identity relation* on A, written $=$, is the smallest reflexive relation on A. The *inverse relation* $\sim^{-1}$ of the relation $\sim$ is the binary relation $\approx$ on A such that for all $a, b \in A$, we have $a \approx b$ iff $b \sim a$. Given two binary relations $\sim_1$ and $\sim_2$ on A, the *compound relation* $\sim_1 \circ \sim_2$ is the binary relation $\approx$ on A such that for all $a, b \in A$, we have $a \approx b$ iff there is an element $c \in A$ such that $a \sim_1 c$

[1] A serial binary relation $\sim$ between A and B can be thought of as a *nondeterministic function* from A to B which maps each domain element $a \in A$ to the set $post_\sim(a) \subseteq B$ of range elements.

and $c \sim_2 b$. By $\sim^0$ we denote the identity relation on A, and for all $i \in \mathbb{N}$, by $\sim^{i+1}$ we denote the compound relation $\sim^i \circ \sim$. The *reflexive closure* $\sim^{\mathit{refl}}$ of the relation $\sim$ is the smallest reflexive superset of $\sim$; that is, $\sim^{\mathit{refl}} = (\sim^0 \cup \sim^1)$. Reflexive closure is a bijection between the strict partial (or linear) orders and the weak partial (or linear) orders, and therefore, we often do not distinguish between the weak and strict varieties of orders. The *transitive closure* $\sim^+$ is the smallest transitive superset of $\sim$; that is, $\sim^+ = (\cup\ i \in \mathbb{N}^{>0} \mid \sim^i)$. The *reflexive-transitive closure* $\sim^*$ is the smallest preorder that is a superset of $\sim$; that is, $\sim^* = (\cup\ i \in \mathbb{N} \mid \sim^i)$. The *symmetric closure* $\sim^{\mathit{symm}}$ is the smallest symmetric superset of $\sim$; that is, $\sim^{\mathit{symm}} = (\sim \cup \sim^{-1})$.

Syntactic Objects

We assume a global universe of typed variables in which each variable has a unique type. This universe is not fixed, but may change from one example to the next. For instance, in one example, the variable x may have the type $\mathbb{N}$, and in another example, x may have the type $\mathbb{B}$. However, we never combine or relate two syntactic objects (such as two reactive modules) from two different universes. In every universe we assume that each variable x has a primed twin x' of the same type. If X is a set of variables, we denote by $X' = \{x' \mid x \in X\}$ the set of all primed variables whose unprimed twins are contained in X.

Expressions and valuations

Let X be a finite set of typed variables. An *expression over* X is a typed expression e whose free variables are from X. We write $\mathit{free}(e)$ for the set of variables that occur freely in the expression e; then $\mathit{free}(e) \subseteq X$. If the variable x and the expression d are type-compatible, we write $e[x := d]$ for the expression over $X \cup \mathit{free}(d)$ which results from safely substituting d for all free occurrences of x in e.[2] A *valuation* for X is a function s that maps each variable $x \in X$ to a value $s(x)$ of the appropriate type. By Σ_X we denote the set of valuations for X.[3] The function $s \in \Sigma_X$ is extended to expressions over X in the standard way. If y is a variable that may or may not be contained in X, and a is a value in the type of y, then $s[y \mapsto a]$ is the valuation in $\Sigma_{X \cup \{y\}}$ which maps y to a, and maps each variable $x \in X$ different from y to the value $s(x)$. For a valuation $s \in \Sigma_X$ and a set $Y \subseteq X$ of variables, the valuation $s[Y] \in \Sigma_Y$ is the restriction of s to the domain of variables in Y. For two disjoint sets X and Y of variables, and two valuations $s \in \Sigma_X$ and $t \in \Sigma_Y$, the valuation $(s \cup t) \in \Sigma_{X \cup Y}$ maps

[2] Expressions may contain quantifiers. Safe substitution requires that the bound variables of e that occur freely in d are suitably renamed before the free occurrences of x are replaced with d. For example, if e is the boolean expression $(\exists y \mid y = x + 1)$ and d is the integer expression $2y$, then $e[x := d]$ denotes, up to renaming of the bound variable y', the boolean expression $(\exists y' \mid y' = 2y + 1)$.

[3] There is precisely one valuation for the empty set of variables.

each variable $x \in X$ to the value $s(x)$, and maps each variable $y \in Y$ to the value $t(y)$.

Let p be a boolean expression over X, and let s be a valuation for X. The valuation s *satisfies* the expression p, written $s \models p$, if $s(p) = \mathit{true}$; otherwise s *violates* p. If s satisfies p, then s is called a *model* of p. We write $[\![p]\!]$ for the set of models of p; that is, $[\![p]\!] = \{s \in \Sigma_X \mid s \models p\}$. The boolean expression p is *satisfiable* if there is a valuation for X that satisfies p; that is, $[\![p]\!] \neq \emptyset$. The expression p is *valid*, written $\models p$, if all valuations for X satisfy p; that is, $[\![p]\!] = \Sigma_X$. Let q be a second boolean expression over X. The boolean expression p *implies* the boolean expression q if every model of p is a model of q; that is, $[\![p]\!] \subseteq [\![q]\!]$. The two expressions p and q are *equivalent* if they have the same models; that is, $[\![p]\!] = [\![q]\!]$.

Guarded commands

Let X and Y be two finite sets of typed variables. A *guarded assignment* γ *from* X *to* Y consists of a *guard* p_γ and for each variable $y \in Y$, an *assignment* e^y_γ. The guard p_γ is a boolean expression over X. Each assignment e^y_γ is an expression over X that is type-compatible with y. Informally, the guarded assignment γ can be executed if the guard p_γ evaluates to true, and then each variable $y \in Y$ is updated to the value of the assignment e^y_γ. Formally, the guarded assignment γ defines a partial function $[\![\gamma]\!]$ from the valuations for X to the valuations for Y: given $s \in \Sigma_X$, if $s \models p_\gamma$, then $[\![\gamma]\!](s)$ maps each variable $y \in Y$ to the value $s(e^y_\gamma)$; otherwise $[\![\gamma]\!](s)$ is undefined. When writing guarded assignments, we may suppress assignments that leave the value of a variable unchanged: we specify the guarded assignment γ using the notation

$$p_\gamma \;\to\; y_1 := e^{y_1}_\gamma;\; \ldots;\; y_m := e^{y_m}_\gamma,$$

where $y_1, \ldots, y_m$ are pairwise distinct variables from Y (possibly $m = 0$) such that $e^y_\gamma = y$ for all variables $y \in Y$ that do not appear in the list $y_1, \ldots, y_m$.

A *guarded command* Γ *from* X *to* Y is a finite set $\{\gamma_i \mid 1 \leq i \leq n\}$ of guarded assignments from X to Y such that the disjunction $(\vee\, 1 \leq i \leq n \mid p_i)$ of the guards is valid[4] (this implies that $n > 0$). Informally, a guarded command nondeterministically chooses one of the guards that evaluates to true, and then executes the corresponding guarded assignment. Formally, the guarded command Γ defines a serial binary relation $[\![\Gamma]\!]$ between Σ_X and Σ_Y, namely, $(s,t) \in [\![\Gamma]\!]$ iff $[\![\gamma_i]\!](s) = t$ for some $1 \leq i \leq n$. The guarded assignment $\gamma_n \in \Gamma$ is called a *default assignment* if (1) $p_n = (\wedge\, 1 \leq i < n \mid \neg p_i)$ and (2) $e^y_n = y$ for all variables $y \in Y$; that is, if none of the other guards evaluates to true, then the values of all variables stay unchanged. When writing guarded commands, we

[4] Given a guarded assignment $\gamma_i \in \Gamma$, we write p_i short for the guard p_{γ_i}, and similarly for the assignments of γ_i.

may suppress default assignments: we specify the guarded command Γ using the notation

$$\gamma_1 \ [\!] \ \cdots \ [\!] \ \gamma_{n-1}$$

if γ_n is a default assignment; otherwise we write $\gamma_1 [\!] \cdots [\!] \gamma_n$. The guarded command Γ is *deterministic* if for all guarded assignments $\gamma_i, \gamma_j \in \Gamma$ and all valuations $s \in \Sigma_X$, if both $s \models p_i$ and $s \models p_j$, then $[\![\gamma_i]\!](s) = [\![\gamma_j]\!](s)$. If the guarded command Γ is deterministic, then $[\![\Gamma]\!]$ is a (total) function from Σ_X to Σ_Y. For finite sets $\mathbb{T} = \{a_1, \ldots, a_n\}$, we freely use abbreviations such as $p \rightarrow y := \mathbb{T}$ for the nondeterministic guarded command

$$p \rightarrow y := a_1 \ [\!] \ \cdots \ [\!] \ p \rightarrow y := a_n.$$

Variable renamings

Let X be a finite set of typed variables. A *renaming* ρ for X is a one-to-one function that maps each variable $x \in X$ to a type-compatible variable $x[\rho]$. Given a set $Y \subseteq X$ of variables, we write $Y[\rho]$ for the set $\{y[\rho] \mid y \in Y\}$ of renamed variables. Given an expression e over X, we write $e[\rho]$ for the expression over $X[\rho]$ that results from e by replacing all free occurrences of each variable $x \in X$ with the variable $x[\rho]$ using safe substitution. Renaming extends to guarded assignments and guarded commands in the natural way, by applying the renaming to all subexpressions: given a guarded command Γ from X to Y, and a renaming ρ for $X \cup Y$, we write $\Gamma[\rho]$ for the renamed guarded command from $X[\rho]$ to $Y[\rho]$. We specify the renaming ρ using the notation

$$x_1, \ldots, x_m \ := \ x_1[\rho], \ldots, x_m[\rho],$$

where $x_1, \ldots, x_m$ are pairwise distinct variables from X (possibly $m = 0$) such that $x[\rho] = x$ for all variables $x \in X$ that do not appear in the list $x_1, \ldots, x_m$.

Words and Languages

Let A be a nonempty set of letters. A *word* $\overline{a} = a_1 \cdots a_m$ *over the alphabet* A is a finite sequence of letters a_i from A. We write $|\overline{a}| = m$ for the *length* of $\overline{a}$; that is, $|\overline{a}|$ denotes the number of letters in $\overline{a}$. By ϵ we denote the *empty word* (then $|\epsilon| = 0$). By $\overline{a} \cdot \overline{b}$ we denote the word that results from *concatenating* the two words $\overline{a}$ and $\overline{b}$ (then $|\overline{a} \cdot \overline{b}| = |\overline{a}| + |\overline{b}|$). By $\overline{a}_{i..j}$, for $1 \leq i \leq j \leq m$, we denote the word $a_i \cdots a_j$ that results from $\overline{a}$ by removing $i - 1$ initial and $m - j$ final letters (then $|\overline{a}_{i..j}| = j - i + 1$). We write A^* for the set of words over A, and A^+ for the set of nonempty words. A *language* L *over the alphabet* A is a set of nonempty words over A; that is, $L \subseteq A^+$. The word $\overline{a}$ is a *prefix* of the word $\overline{b}$ if there exists a word $\overline{c}$ such that $\overline{b} = \overline{a} \cdot \overline{c}$; and $\overline{a}$ is a *suffix* of $\overline{b}$ if there exists a word $\overline{c}$ such that $\overline{b} = \overline{c} \cdot \overline{a}$. The language L is *prefix-closed* if for every word $\overline{a}$ in L, all prefixes of $\overline{a}$ are also in L; and L is *suffix-closed* if for every word $\overline{a}$

in L, all suffixes of $\overline{a}$ are also in L. The language L is *fusion-closed* if for all letters a, if $\overline{b} \cdot a \cdot \overline{c}$ and $\overline{b}' \cdot a \cdot \overline{c}'$ are in L, then so is $\overline{b} \cdot a \cdot \overline{c}'$. The language L is *upward stutter-closed* if for all letters a, if $\overline{b} \cdot a \cdot \overline{c}$ is in L, then so is $\overline{b} \cdot a \cdot a \cdot \overline{c}$. The language L is *downward stutter-closed* if for all letters a, if $\overline{b} \cdot a \cdot a \cdot \overline{c}$ is in L, then so is $\overline{b} \cdot a \cdot a \cdot \overline{c}$. The language L is *stutter-closed* if L is both upward stutter-closed and downward stutter-closed.