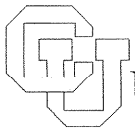
A POWERDOMAIN PRIMER

A Tutorial for The Bulletin of the EATCS

Michael G. Main

CU-CS-375-87



University of Colorado at Boulder

DEPARTMENT OF COMPUTER SCIENCE

* Supported by NSF NYI #CCR-9357740, ONR #N00014-96-1-0720, and a Packard Fellowship in Science and Engineering from the David and Lucile Packard Foundation.

A POWERDOMAIN PRIMER*

A Tutorial for The Bulletin of the EATCS

Michael G. Main

CU-CS-375-87 September 1987

*This research has been supported in part by National Science Foundation grant DCR-8402341.

A POWERDOMAIN PRIMER*

Michael G. Main
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
Phone: 303-492-7579

1. Motivation

The order-theoretic approach to programming semantics uses certain partially-ordered sets, called *domains*. Typically, the elements of a domain D are the "machine states" in which a computation may proceed, and a program is represented by a state-transformation function $f : D \to D$. The meaning of such a function is this: when the program is started in a state $x \in D$, then it will end in the state $f(x)$. This "end-state" might be a special element of D which indicates that the program never terminated. This special element is usually considered to be just another "state" — one that we frequently want to avoid.

Of course, this is not the entire story of order-theoretic semantics: for example, I have not even mentioned what kind of partial-order a domain possesses, or the reason for the order. But this is enough of the story to motivate *powerdomains*. The motivation comes from a problem with the "typical" situation described above. We assumed that the state-transition relationship was a function, so that given a start-state $x \in D$, there is a single end-state $f(x) \in D$ which will be reached by the program. But, some programs are *nondeterministic* — meaning that a given start-state does not uniquely determine an end-state. We may also be uncertain about precisely which state a nondeterministic program starts in.

Powerdomains are the solution to this problem. Intuitively, a powerdomain P is a special kind of domain whose elements are various "nondeterministic combinations of elements" from another domain. In this setting, a nondeterministic program represents a function $f : P \to P$. The meaning of such a function is this: when the program is started in one of the states indicated by the nondeterministic combination $x \in P$, then it will end in one of the states of $f(x)$. In general, different notions of powerdomains are based on different intuitions about what constitutes a "nondeterministic combination of elements".

*This research has been supported in part by National Science Foundation grant DCR-8402341.

This paper is a tutorial to explain these different intuitions and the resulting powerdomains. The tutorial begins with a review of domain theory and its use in the order-theoretic semantics of deterministic programs. This is followed in Section 3 by an introduction to nondeterministic programs, and their order-theoretic semantics using Gordon Plotkin's powerdomain. Section 4 provides an algebraic justification for Plotkin's choice of a powerdomain. Alternatives to Plotkin's powerdomain are explained in Sections 5 through 7, each with a similar algebraic justification. As a reference, the symbols used in the paper are collected together in the following box.

b: a Boolean function.
e: an arithmetic function.
f through h: domain morphisms.
i through k: integers.
m and n: natural numbers.
x and y: elements of a domain.

B: a Boolean expression in your favorite programming language.
C: a set of states for a computation.
D: a domain.
E: an arithmetic expression in your favorite programming language.
P and Q: nondeterministic domains.
R: a program.
S through V: subsets of a domain.
X and Y: variable names in a program.
Z: the set of integers.

$\perp$: the least element in a domain.
$BASE_D$: the set of BASE elements of a domain D.
$C_\perp$: the flat domain whose non-$\perp$ elements are the set C.
$[D \to D]$: the set of domain morphisms from D to itself.
$\mathbf{P}(D)$: the free powerdomain generated by D.
$\mathbf{P}_{ANGEL}(D)$: the free angelic powerdomain generated by D.
$\mathbf{P}_{DEMON}(D)$: the free demonic powerdomain generated by D.
$S^{\uparrow}$: the upward-closure of a subset S of a domain.
η: an insertion morphism from a domain to a powerdomain.

Table of Symbols

2. Domains and Deterministic Semantics

In *denotational semantics*, a program denotes a function $f : D \rightarrow D$, where the elements of D represent "machine states" for the computation of the program. One goal of denotational semantics is to assign such a function to each possible program in a programming language. The method of making this assignment for recursive or iterative programs is a prominent part of *order-theoretic semantics* — which is a particular kind of denotational semantics. This method of order-theoretic semantics is based on the assumption that D forms a certain kind of partially-ordered set called a *domain*. This section of the tutorial explains what domains are, and how they are used in order-theoretic semantics.

2.1 Domains

Intuitively, the elements of a domain can be viewed as "partial descriptions" or "approximations" of objects. A good place to begin is an example domain called $P\omega$. The elements of this domain are finite and infinite sets of natural numbers. Any set in $P\omega$ can be viewed as an approximation of its supersets. For example, the set $\{0,2\}$ can be viewed as an approximation of the infinite set $\{0,2,4,\cdots\}$ of even natural numbers.

Of course, there are better and worse approximations to the even numbers. For example, $\{2\}$ is a worse approximation than $\{0,2\}$, while $\{0,2,8\}$ is somewhat better. The best approximation to the even numbers is the infinite set $\{0,2,4,\cdots\}$ itself. Every domain possesses a partial order which indicates when one approximation is better than another. The symbol $\sqsubseteq$ indicates this relationship, so that $\{2\} \sqsubseteq \{0,2\}$ and $\{0,2\} \sqsubseteq \{0,2,8\}$. In $P\omega$ the relationship $\sqsubseteq$ is completely defined by $x \sqsubseteq y$ if and only if x is a subset of y. In a domain, this relationship is always a partial-order — *i.e.*, a reflexive, transitive and antisymmetric relation.

In our use of domains we will often see sequences of better-and-better approximations. The notation $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$ indicates such a sequence of elements $x_0, x_1, x_2 \cdots$ with $x_0 \sqsubseteq x_1$ and $x_1 \sqsubseteq x_2$ and so on. Such a sequence is called a *chain*. Here's an example chain in $P\omega$:

$$\{0\} \sqsubseteq \{0,2\} \sqsubseteq \{0,2,4\} \sqsubseteq \{0,2,4,6\} \sqsubseteq \cdots$$

This particular chain has the set of even natural numbers as an *upper-bound* — *i.e.* for any element x_n in the chain $x_n \sqsubseteq \{m \mid m \text{ is even}\}$. In fact, the set of even numbers is the *least* upper-bound of

this chain, since whenever y is another upper bound of this chain, then $\{m|\ m$ is even $\} \sqsubseteq y$. The least upper-bound of a chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$ is denoted by $\bigsqcup_{n=0}^{\infty} x_n$.

Another important concept in domain theory is the notion of an *isolated* element of a domain. Informally, isolated elements are elements which contain only a finite amount of information. In fact, isolated elements are sometimes called *finite* elements, but the use of this term can sometimes cause confusion. Formally, an element x of a domain is isolated provided that whenever $x \sqsubseteq \bigsqcup_{n=0}^{\infty} x_n$ then there exists some n such that $x \sqsubseteq x_n$.

Exercise 1: Show that every chain in $P\omega$ has a least upper bound. Show that the isolated elements of $P\omega$ are precisely the finite sets. Show that every element of $P\omega$ is the least upper-bound of a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$, where each x_n is an isolated element.

With this background, domains can be defined, which is done in Figure 2.1. The domains defined there are sometimes called ω-algebraic, complete partial orders.

Definition: A *domain* is a set D with a partial order $\sqsubseteq$ such that:

(1) There is an element $\bot \in D$ such that for all $x \in D$: $\bot \sqsubseteq x$. (This is called the *bottom* of D).

(2) Every chain has a least upper-bound.

(3) There is a countable number of isolated elements, and every element of D is the least upper-bound of a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$, where each x_n is an isolated element.

The set $\{x \in D|\ x$ is isolated $\}$ is called the *base* of D, denoted by $BASE_D$. A partially-ordered set which meets the first two conditions, but perhaps not the third, is called an ω-CPO (complete partial order).

Figure 2.1 Definition of a Domain

Examples: The set N^{∞} of natural numbers plus an infinity-element is partially-ordered by the usual $\leq$ relation. This is a domain, and infinity is the only non-isolated element. The set $N^{\infty} \times N^{\infty}$ of pairs is partially-ordered by the relation $(i_1, j_1) \sqsubseteq (i_2, j_2)$ if and only if $i_1 \leq i_2$ and $j_1 \leq j_2$. This is also a domain, where (i, j) is isolated only if both i and j are finite. The set of non-negative rational numbers plus an infinity-element is partially-ordered by the usual $\leq$ relation. This is an ω-CPO, but not a domain (zero is the only isolated element). If C is any countable set, then the set

$C_\perp = C \cup \{\perp\}$ is a domain with the partial-order: $x \sqsubseteq y$ if and only if $x = \perp$ or $x = y$. This is called a *flat* domain — an apt name for the picture of $C_\perp$ in Figure 2.2. Most of the domains used to provide semantics for simple programming languages are flat domains.

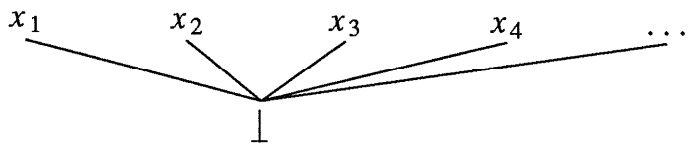


Figure 2.2. A Flat Domain $C_\perp$ where $C = \{x_1, x_2, x_3, \cdots\}$.

2.2 Domain Morphisms

In order-theoretic semantics, elements of domains represent different levels of information about computation states. A program denotes a function $f : D \rightarrow D$ on such a domain. But, not just any sort of function! The functions denoted by programs are *domain morphisms*, as defined in Figure 2.3.

Definition: Let D_1 and D_2 be domains. A function $f : D_1 \rightarrow D_2$ is a *domain morphism* provided that it meets these conditions:

Monotonicity: Whenever $x \sqsubseteq y$ in D_1 then $f(x) \sqsubseteq f(y)$ in D_2.

Strictness: $f(\perp) = \perp$.

Continuity: For any chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$ in D_1: $f(\bigsqcup_{n=0}^{\infty} x_n) = \bigsqcup_{n=0}^{\infty} (f(x_n))$.

Figure 2.3. Definition of a Domain Morphism

The restriction to domain morphisms reflects our intuition about how programs work. For example, the monotonicity requirement corresponds to the intuition that better information about an input

state results in better information about an output state.

Some of the mathematics of domains carries over to domain morphisms. For example, the set of domain morphisms from D to D (written $[D \rightarrow D]$) is partially-ordered by the relation $f \sqsubseteq g$ if and only if for all $x \in D: f(x) \sqsubseteq g(x)$. The set $[D \rightarrow D]$ is an ω-CPO, but not always a domain. This means that any chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \cdots$ of domain morphisms has a least upper-bound $f = \bigsqcup_{n=0}^{\infty} f_n$, defined by $f(x) = \bigsqcup_{n=0}^{\infty} (f_n(x))$. In order-theoretic semantics, this least upper-bound of a chain of domain morphisms is used to define the function denoted by an iterative or recursive program.

2.3 Semantics of a Toy Language

Now that the mathematics of domains and domain morphisms has been established, the ideas of order-theoretic semantics can be demonstrated on a little language called XY. The programs in this language compute in a state space of two integer variables named X and Y. The programs may also enter unending "loops". The domain for these computations is the flat domain $(Z \times Z)_\perp$, where Z is the set of integers and $\perp$ represents an "unending computation". A pair $(i,j) \in Z \times Z$ represents a computation state where the variable X has value i and Y has value j. As an example, consider a program which computes the factorial of X and stores it in Y. This program denotes a function $f: (Z \times Z)_\perp \rightarrow (Z \times Z)_\perp$ with $f(i,j) = (i, i!)$ for any $i \geq 0$ and any j. A program with $f(i,j) = \perp$ means that an input of (i,j) will result in an unending loop in the computation. All programs have $f(\perp) = \perp$; intuitively this means that if the program which provided input to f had an unending loop, then there is no way for f to correct this.

The legal XY-programs are defined recursively on the left side of Figure 2.4. Each time a new program **R** is defined, then a function $[\![\mathbf{R}]\!]: (Z \times Z)_\perp \rightarrow (Z \times Z)_\perp$ is also defined. This is the function which the program **R** denotes. The machinery of order-theoretic semantics comes into effect at only one point: the definition of the function denoted by an iterative program. Throughout Figure 2.4, D is the flat domain $(Z \times Z)_\perp$.

The XY-language is merely a toy, but it's syntax and semantics in Figure 2.4 illustrate the most important features of order-theoretic semantics for simple languages. The domain for the computations in such a language is typically a flat domain of the form $C_\perp$, where C is the state space of the

SYNTAX	SEMANTICS
1. *Simple Commands:* *SKIP* and *FAIL* are *XY*-programs. Intuitively, *SKIP* is a program that does nothing, and *FAIL* is a program that always enters an unending loop.	1. $[\![SKIP]\!]: D \to D$ is the identity function. $[\![FAIL]\!]: D \to D$ is the constant function which maps everything to $\perp$.
2. *Assignment Statements:* Let E be any totally defined integer arithmetic expression with at most two integer variables X and Y in your favorite programming language. Then these are *XY*-programs: $X := E$ $Y := E$	2. Let $e: Z \times Z \to Z$ be the function which maps a pair (i, j) to the value of the arithmetic expression E when $X = i$ and $Y = j$. Then $[\![X := E]\!]: D \to D$ maps a pair (i, j) to $(e(i, j), j)$, and $[\![Y := E]\!]: D \to D$ maps a pair (i, j) to $(i, e(i, j))$. Both of these functions map $\perp$ to $\perp$.
3. *Composition:* Let $R_1, R_2, \cdots R_k$ be *XY*-programs. Then this is an *XY*-program: *BEGIN* $R_1; R_2; \cdots R_k$ *END*	3. Let R be the composition program. Then $[\![R]\!]: D \to D$ is the composition function $[\![R_k]\!] \circ \cdots \circ [\![R_2]\!] \circ [\![R_1]\!]$.
4. *Conditional Statement:* Let R_1 and R_2 be *XY*-programs, and let B be any totally defined Boolean expression with at most two integer variables X and Y in your favorite programming language. Then this is an *XY* program: *IF B THEN* R_1 *ELSE* R_2	4. Let $b: Z \times Z \to \{TRUE, FALSE\}$ be the function which maps a pair (i, j) to the value of the Boolean expression B when $X = i$ and $Y = j$. If $b(i, j)$ is *TRUE* then $[\![IF\ B\ THEN\ R_1\ ELSE\ R_2]\!](i, j) = [\![R_1]\!](i, j)$, otherwise $[\![IF\ B\ THEN\ R_1\ ELSE\ R_2]\!](i, j) = [\![R_2]\!](i, j)$. It always maps $\perp$ to $\perp$.
5. *Iterative Statement:* Let R be an *XY*-program, and let B be any totally defined Boolean expression with at most two integer variables X and Y in your favorite programming language. Then this is an *XY* program: *WHILE B DO* R	5. Let R_0 be the program *FAIL*; for any integer $n > 0$, let R_n be this program: *IF B THEN BEGIN* R; R_{n-1} *END* *ELSE SKIP* Then the sequence of functions $[\![R_0]\!] \sqsubseteq [\![R_1]\!] \sqsubseteq [\![R_2]\!] \cdots$ is a chain. The function $[\![WHILE\ B\ DO\ R]\!]$ is the least upper-bound of this chain.

Figure 2.3. Syntax and Semantics of the *XY*-language

computations and $\perp$ is an extra element representing unending computations.

The function denoted by an iterative program is the least upper-bound of a sequence of better-and-better approximations to the program. In part 5 of Figure 2.4, the programs $\mathbf{R}_n$ are these approximations. Intuitively, the program $\mathbf{R}_n$ is the program *WHILE B DO* **R** — with a restriction that the body of the loop cannot be executed more than $n-1$ times. The n^{th} attempt to execute the loop's body results in the *FAIL* program. So, the intuition embodied by the order-theoretic semantics is this:

WHILE B DO **R**

is

$$\lim_{n \to \infty} \Big[\text{Execute } \textit{WHILE } B \textit{ DO } \mathbf{R} \text{ — But } \textit{FAIL} \text{ if the loop needs more than } n \text{ iterations.} \Big]$$

Exercise 2: Prove that the sequence of functions in part 5 of Figure 2.4 is indeed a chain.

2.4 A Toy Program

It is traditional to provide the semantics of a small language, followed by an application of the semantics to a small program which calculates the factorial function. I shall not violate this tradition, so here's the factorial function written in the *XY*-language (the factorial of i is written $i!$):

```
(*******************************************
  Program to calculate the factorial of a number i.
  At the start of the program, the number i must be stored
  in X, and some number j is stored in the variable Y.
  At the end of the program, X will be 0 and Y will
  be i! * j. If i is negative, then the program never
  terminates.
*******************************************)
  WHILE (X ≠ 0) DO BEGIN
    Y := X * Y;
    X := X - 1;
  END
```

Let **R** be the portion of the *WHILE*-loop from the *BEGIN* to the *END*. Note that the function $[\![\mathbf{R}]\!]: D \to D$ is defined by $[\![\mathbf{R}]\!](i,j) = (i-1, i*j)$ and $[\![\mathbf{R}]\!](\perp) = \perp$. We can use this to calculate the function denoted by the *WHILE*-statement. This function is the least upper-bound of a chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \cdots$, where f_0 is the constant function which maps everything to $\perp$, and

$$f_n(i,j) = \begin{cases} \text{if } (i=0) \text{ then } (i,j) \\ \text{else } f_{n-1}([\![\mathbf{R}]\!](i,j)) \end{cases}$$

What is the least upper-bound of the f_n? We will show that it is the strict function $f : (Z \times Z)_\perp \to (Z \times Z)_\perp$ defined by

$$f(i,j) = \begin{cases} \text{if } (i<0) \text{ then } \perp \\ \text{else}(0, i! * j) \end{cases}$$

First we show that f is an upper-bound of the f_n — *i.e.* that $f_n \sqsubseteq f$ for any n. The proof is by induction on n, with the base step ($f_0 \sqsubseteq f$) being trivial since f_0 is the least function. For the induction step, assume that $f_k \sqsubseteq f$ whenever $k < n$, for some fixed $n > 0$. We must show that this implies $f_n \sqsubseteq f$. Toward this end, let i and j be any integers and note the following relations:

<u>If $i = 0$:</u>

$$f_n(i,j) = f_n(0,j) = (0,j) = (0, 0! * j) = f(0,j) = f(i,j)$$

<u>If $i < 0$:</u>

$$f_n(i,j) = f_{n-1}([\![\mathbf{R}]\!](i,j)) = f_{n-1}(i-1, i*j) \sqsubseteq f(i-1, i*j) = \perp = f(i,j)$$

<u>If $i > 0$:</u>

$$f_n(i,j) = f_{n-1}([\![\mathbf{R}]\!](i,j)) = f_{n-1}(i-1, i*j) \sqsubseteq f(i-1, i*j) = (0, (i-1)! * i * j) = (0, i! * j) = f(i,j)$$

The $\sqsubseteq$ in the second and third lines follow from the induction hypothesis. Thus, we have shown that for all $n \geq 0$: $f_n \sqsubseteq f$ — so f is an upper-bound of the sequence. To show that it is the *least* upper-bound, suppose that h is another upper-bound of the chain. Whenever $i < 0$ then $f(i,j) = \perp \sqsubseteq h(i,j)$. And whenever $i \geq 0$ then:

$$f(i,j) = (0, i! * j) = f_{i+1}(i,j) \sqsubseteq h(i,j)$$

Therefore, $f \sqsubseteq h$, and f is the least upper-bound of the sequence — hence f is the function denoted by the *WHILE*-statement.

3. A Domain for Nondeterministic Programs

3.1 Nondeterministic *XY*-programs and their Semantics

Programs in the *XY*-language are *deterministic* — meaning that any fixed input yields a unique output. But there may be other situations where determinism does not hold. A situation which is potentially nondeterministic is when two or more processes are running in parallel. It may be impossible to calculate the precise relative speeds of the processes. In this case, different outcomes may result from different relative speeds of the parallel processes. From a program designer's standpoint, it may also be desirable to explicitly introduce nondeterminism to a language — since a designer may be willing to accept any one of several correct outputs. In this case, a designer may specify several possibilities that he is willing to accept, and allow some considerations beyond his control to dictate which of these possibilities is actually realized.

In order to study the semantics of nondeterminism in a simple setting, an explicit nondeterministic construction to will be added to the *XY*-language. specifically, whenever $\mathbf{R}_1$ and $\mathbf{R}_2$ are *XY*-programs, then $(\mathbf{R}_1 \textbf{ or } \mathbf{R}_2)$ is a new program which is a nondeterministic choice between $\mathbf{R}_1$ and $\mathbf{R}_2$. For example, the program $(X := 1 \textbf{ or } X := 2)$ maps a pair (i,j) to one of two possible places: $(1,j)$ or $(2,j)$. Obviously, such a program is not represented by a *function* on the domain $(Z \times Z)_\perp$. Instead, we will create a new domain P, whose elements are various nondeterministic combinations of elements from $(Z \times Z)_\perp$. A nondeterministic program $\mathbf{R}$ denotes a function $\mathbf{[R]}: P \rightarrow P$. (Notice the use of "bold" brackets **[R]** to distinguish this function from the function [[**R**]] which a deterministic program **R** denotes.)

Elements of P will be collections of states such as $\{(1,0),(2,0)\}$, which is an appropriate output for a program which can finish with $Y = 0$ and either $X = 1$ or $X = 2$. From the direction of this

discussion, you might think that we can take the elements of P to be all possible subsets of $(Z \times Z)_\perp$ (*i.e.*, P is the power*set* of $(Z \times Z)_\perp$). But, this does not work, for at least two reasons:

(1) From a practical standpoint, there is no easy way to make the powerset of a domain into a domain itself.

(2) There are subsets of $(Z \times Z)_\perp$ which can never be the output of a nondeterministic XY-program. Specifically, if a nondeterministic XY-program starts in a state (i, j), and if the program is guaranteed to terminate regardless of the nondeterministic choices made, then there are only a *finite* number of possible output states for the program. This suggests that we should omit from P any infinite set that does not contain $\perp$. Also, each nondeterministic XY-program has at least one output state for each input state (although this output may be $\perp$).

The second point listed above suggests that we define P to be:

$$P = \{S \subseteq (Z \times Z)_\perp \mid S \text{ is non-empty and finite or } \perp \in S\}$$

This is a subset of the powerset of $(Z \times Z)_\perp$, and it also forms a domain using a partial-order first suggested by Egli and Milner Here's the Egli-Milner order:

$$S \sqsubseteq T \text{ iff } \begin{cases} \text{For all } x \in S \text{ there exists } y \in T \text{ such that } x \sqsubseteq y, \text{ and} \\ \text{For all } y \in T \text{ there exists } x \in S \text{ such that } x \sqsubseteq y. \end{cases}$$

An equivalent definition of this order is:

$$S \sqsubseteq T \text{ iff } \begin{cases} S = T, \text{ or} \\ \perp \in S \text{ and } S \subseteq T \cup \{\perp\} \end{cases}$$

Exercise 3: Prove the claim made about nondeterministic XY-programs in (2) above. Show that the two definitions of $\sqsubseteq$ on P are identical, and that this partial order does make P a domain. Show that the isolated elements of P are the finite subsets.

The domain P was arrived at through the considerations listed above, but it can also be mathematically justified as being the least-constrained way of making certain subsets of $(Z \times Z)_\perp$ into a domain. This mathematical justification will also lead to a method for generating a "domain of nondeterministic values" from any domain. For now, we will postpone this mathematical

SYNTAX	SEMANTICS
1. *Deterministic Programs:* Each deterministic *XY*-program is also a nondeterministic program.	1. Let $[\![\mathrm{R}]\!]: D \to D$ be the domain morphism associated with the deterministic program R. The morphism $[R]: P \to P$ for the nondeterministic program is defined by $$[R](S) = \{[\![R]\!](x) \mid x \in S\}$$
2. *Nondeterministic Statement:* Let R_1 and R_2 be nondeterministic *XY*-programs. Then this is a nondeterministic *XY* program: $$(\mathrm{R}_1 \text{ or } \mathrm{R}_2)$$	2. For all $S \in P$: $[(\mathrm{R}_1 \text{ or } \mathrm{R}_2)](S)$ is $[\mathrm{R}_1](S) \cup [\mathrm{R}_2](S)$.
3. *Composition*: Let $\mathrm{R}_1, \mathrm{R}_2, \cdots \mathrm{R}_k$ be nondeterministic *XY*-programs. Then this is a nondeterministic *XY*-program: $$\textit{BEGIN } \mathrm{R}_1; \mathrm{R}_2; \cdots \mathrm{R}_k \textit{ END}$$	3. Let R be the composition program. Then $[\mathrm{R}]: D \to D$ is the composition function $[\mathrm{R}_k] \circ \cdots \circ [\mathrm{R}_2] \circ [\mathrm{R}_1]$.
4. *Conditional Statement:* Let R_1 and R_2 be nondeterministic *XY*-programs, and let *B* be any totally defined Boolean expression with at most two integer variables *X* and *Y* in your favorite programming language. Then this is a nondeterministic *XY* program: $$\textit{IF } B \textit{ THEN } \mathrm{R}_1 \textit{ ELSE } \mathrm{R}_2$$	4. Let $g: D \to P$ be the unique strict function such that if *B* is true at (i, j) then $$g(i, j) = [R_1](\{(i, j)\}),$$ and otherwise $$g(i, j) = [R_2](\{(i, j)\}).$$ Then $[\textit{IF } B \textit{ THEN } \mathrm{R}_1 \textit{ ELSE } \mathrm{R}_2](S)$ is $$\{g(x) \mid x \in S\}.$$
5. *Iterative Statement:* Let R be a nondeterministic *XY*-program, and let *B* be any totally defined Boolean expression with at most two integer variables *X* and *Y* in your favorite programming language. Then this is a nondeterministic *XY* program: $$\textit{WHILE } B \textit{ DO } \mathrm{R}$$	5. Let R_0 be the program *FAIL*; for any integer $n > 0$, let R_n be this program: *IF B THEN BEGIN* R; R_{n-1} *END ELSE SKIP* Then the sequence of functions $$[\mathrm{R}_0] \sqsubseteq [\mathrm{R}_1] \sqsubseteq [\mathrm{R}_2] \cdots$$ is a chain. The function [*WHILE B DO* R] is the least upper-bound of this chain.

Figure 3.1. Syntax and Semantics of the Nondeterministic *XY*-language

framework, and concentrate instead on how P (our *powerdomain!*) can be used to give the semantics of nondeterministic XY-programs. This is done in Figure 3.1, which defines a domain morphism $[\![\mathbf{R}]\!]: P \to P$ for each nondeterministic XY-program **R**.

3.2 A Toy Nondeterministic Program

For any nondeterministic XY-program and element $S \in P$, the following equation holds:

$$[\![\mathbf{R}]\!](S) = \bigcup_{x \in S} [\![\mathbf{R}]\!](\{x\})$$

In other words, the functions denoted by XY-programs are *additive*. The proof of this is an induction on the structure of the program **R**. This property is useful because it means that the behavior of a program **R** is completely determined by its behavior on deterministic (or singleton) inputs. Section 4 will further explore this property and its consequences. For now, we will simply use this property in examining the semantics of this nondeterministic XY-program:

$$\mathit{WHILE}\ (X \neq 0)\ \mathit{DO}\ (X := X - 1\ \ \mathbf{or}\ \ Y := Y + 1)$$

Let **R** be the nondeterministic statement in the body of the *WHILE*-loop. For a deterministic input (i, j), the function $[\![\mathbf{R}]\!]$ is defined by $[\![\mathbf{R}]\!]\{(i,j)\} = \{(i-1, j), (i, j+1)\}$. The function denoted by the *WHILE* loop itself is the least upper-bound of a chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \cdots$, where f_0 is the constant function which maps everything to $\{\perp\}$, and for $n > 0$, $f_n : P \to P$ is the additive domain morphism whose behavior on singleton sets is:

$$f_n\{(i,j)\} = \begin{cases} \text{if } (i=0) \text{ then } \{(i,j)\} \\ \text{else } f_{n-1}([\![\mathbf{R}]\!](i,j)) \end{cases}$$

What is the function denoted by the entire loop? It is the strict additive function $f : P \to P$ defined by

$$f\{(i,j)\} = \begin{cases} \text{if } (i<0) \text{ then } \{\perp\} \\ \text{else if } (i=0) \text{ then } \{(i,j)\} \\ \text{else } \{\perp\} \cup \{(0,k) \mid k \geq j\} \end{cases}$$

The proof that this function is an upper-bound of the f_n is by induction on n — much as in Section 2.4 for the deterministic program. The proof that this is in fact the least upper-bound is also straight-forward. The moral of the story is that all of the techniques that you are familiar with from deterministic order-theoretic semantics carry over to the nondeterministic programs with little change.

Exercise 4: Complete the proof that the function denoted by the *WHILE*-loop is the function defined above.

4. Powerdomains

Section 3 gave a first example of a powerdomain, P, which was used to provide semantics for the nondeterministic XY-language. This section defines powerdomains in more generality, and gives a mathematical justification for using the particular powerdomain P for the XY-language.

4.1 Nondeterministic Domains

In order to provide semantics for the nondeterministic XY-language, we needed a binary operation on the domain P which corresponded to the **or** operation on XY-programs. This binary operation was the union of two sets in P. Intuitively, an element $S \cup T \in P$ is a nondeterministic choice between S and T. This view of P suggests that one important thing about a powerdomain is that it is equipped with a binary operation that we can use in this way. We generally impose some constraints on the binary operation, to match our intuition about nondeterministic choice. This leads to the definition of a nondeterministic domain in Figure 4.1.

Definition: A *nondeterministic domain* (or *ND-domain*) is a domain P together with a binary operation $\mathbf{or} : P \times P \to P$, such that for all elements $x, y, z \in P$ and all chains $x_0, x_1, \cdots$ in P:

Associativity: $((x \mathbf{or} y) \mathbf{or} z) = (x \mathbf{or} (y \mathbf{or} z))$.

Commutativity: $(x \mathbf{or} y) = (y \mathbf{or} x)$.

Idempotence: $(x \mathbf{or} x) = x$.

Continuity: $(\bigsqcup_{n=0}^{\infty} x_n) \mathbf{or} y = \bigsqcup_{n=0}^{\infty} (x_n \mathbf{or} y)$.

These ND-domains have also been called *semilattice-domains*.

Figure 4.1. Definition of a Nondeterministic-domain

Exercise 5: Show that the domain P from the last section is an ND-domain with the binary operation $\cup$. (That is, show that $\cup$ meets the four properties stated in the definition.)

4.2 ND-domain Morphisms

A nondeterministic program will be denoted by a morphism $f : P \to P$, where P is an ND-domain that is equipped with a binary operation **or**. Such a program should also preserve the **or**-structure of the ND-domain, so that $f(x \mathbf{or} y) = f(x) \mathbf{or} f(y)$. The **or**-preservation means that "running a program on a choice of several inputs is the same as running the program on several inputs and choosing between the outputs". Such a function is called an *ND-domain morphism* (see Figure 4.2).

Definition: Let P_1 and P_2 be ND-domains. A domain morphism $f : P_1 \to P_2$ is an *ND-domain morphism* provided that it meets this condition:

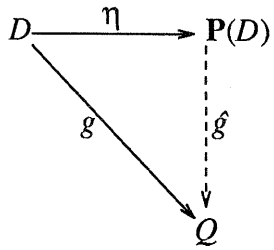
or-*Preserving:* For all elements x and y in P_1: $f(x \mathbf{or} y) = f(x) \mathbf{or} f(y)$.

Figure 4.2. Definition of an ND-domain Morphism

Preservation of the **or**-structure is the most common intuition for nondeterministic programs, although some research of Hennessy and Ashcroft has considered other ideas.

4.3 Free Generation of Powerdomains

Typically, powerdomains arise when we want to add nondeterministic features to a programming language whose deterministic semantics is already specified over some ordinary domain D (such as the deterministic XY-language over the domain $(Z \times Z)_\perp$). In this case we want to embed the deterministic domain D into an ND-domain — in other words, we are looking for an ND-domain $\mathbf{P}(D)$, together with a domain morphism $\eta: D \to \mathbf{P}(D)$. In order to make $\mathbf{P}(D)$ free from unintended constraints, the embedding morphism η should be *universal*. This means that any other possible embedding of D into an ND-domain can be obtained by factoring through η in a unique way. The universality requirement is formally stated as follows: if Q is any ND-domain and $g: D \to Q$ is any domain morphism, then there is a unique ND-domain morphism $\hat{g}: \mathbf{P}(D) \to Q$ such that for any $x \in D: \hat{g}(\eta(x)) = g(x)$, as in the following diagram:



The ND-domain $\mathbf{P}(D)$ is called the *free powerdomain generated* by D with *insertion* η. To be more precise, we should probably call $\mathbf{P}(D)$ *a* free powerdomain generated by D with insertion η, rather than *the* free powerdomain. But, it is easy to show that $\mathbf{P}(D)$ and η are unique "up to isomorphism", so that any other powerdomain freely generated by D is identical to $\mathbf{P}(D)$ except perhaps for renaming. Therefore, we usually say *the* free powerdomain.

Now we can justify the domain P that we used in Section 3 for the semantics of nondeterministic XY-programs. Recall that this domain consists of all non-empty, finite subsets of $(Z \times Z)_\perp$, plus infinite subsets which contain $\perp$. The order on P is the Egli-Milner order. Here is the result about the freeness of P:

Theorem 4.1. *Let P be the powerdomain used in Section 3, with set-union as the* **or** *-operation. Let* $\eta:(Z\times Z)_\perp \to P$ *be the domain morphism that maps each element x to the singleton set* $\{x\}$. *Then P is the free powerdomain generated by* $(Z\times Z)_\perp$ *with insertion* η.

Proof: To show the universality of η, let Q be any ND-domain and let $g:(Z\times Z)_\perp \to Q$ be a domain morphism. We must demonstrate a unique ND-domain morphism $\hat{g}:P \to Q$ with $\hat{g} \circ \eta = g$.

Now, let T be an arbitrary set in P. Since P is a domain whose isolated elements are the finite subsets of $(Z\times Z)_\perp$, it follows that T is the least-upper bound of a chain $S_0 \sqsubseteq S_1 \sqsubseteq S_2 \cdots$ of finite sets. Since each S_n is finite, these sets can be expressed as $S_n = \{x_{n,1}, x_{n,2}, \cdots, x_{n,k_n}\}$ for some integers n_k. Thus, T is $\bigsqcup_{n=0}^{\infty} (\{x_{n,1}\}$ **or** $\cdots$ **or** $\{x_{n,k_n}\})$.

Define $\hat{g}(T)$ to be $\bigsqcup_{n=0}^{\infty} (g(x_{n,1})$ **or** $\cdots$ **or** $g(x_{n,k_n}))$. It is easy to show that this is the unique ND-domain morphism with $\hat{g} \circ \eta = g$. □

Exercise 6: Let $C_\perp$ be any flat domain. Show that $\mathbf{P}(C_\perp)$ consists of of all non-empty finite subsets of $C_\perp$, plus any countably infinite subsets that contain $\perp$. The order is the Egli-Milner order and the **or**-operation is set-union.

5. Demonic and Angelic Powerdomains

This section provides two alternatives to the freely generated powerdomain. Each of these alternative powerdomains has a universal property of its own. These universal properties make it easy to provide semantics for nondeterministic languages using any of the powerdomains.

5.1 Demons and Angels

When a nondeterministic *XY*-program is run on some input, the result is a set of possible outputs $S \in \mathbf{P}(Z\times Z)_\perp$. Suppose I am not interested in these sets themselves — but instead I only want to ask certain kinds of questions about these sets. For example, let $b:(Z\times Z)_\perp \to \{TRUE, FALSE\}_\perp$ be a domain morphism. If we are interested in the correctness of a program, then we would ask questions like these:

Is $b(y)$ *TRUE* for every $y \in S$?
Is $b(y)$ *TRUE* for any $y \in S$?

The first question would be asked to guarantee that b will be *TRUE* for the output of a program, regardless of any nondeterministic choices that may occur during the execution. Just one state $y \in S$ where $b(y)$ is not *TRUE* will cause this question to be answered negatively. The policy of asking this kind of question has been called the *demonic* approach to nondeterminism, because it assumes that there is some malicious demon controlling the nondeterminism. If there is but one nondeterministic choice that I don't want, then the demon will find this choice and the program will fail. Therefore, if I want b to be *TRUE* of my program's output, then I must ask whether b is *TRUE* for *every* possible output.

The second question would be asked to determine whether there's any possibility that b will be *TRUE* for the output of my program. The policy of asking this kind of question has been called the *angelic* approach to nondeterminism, because it assumes that there is some benevolent angel controlling the nondeterminism. If there is but one nondeterministic choice that I want, then the angel will find this choice and the program will succeed. Therefore, if I want b to be *TRUE* of my program's output, then I need only ask whether b is *TRUE* for *any* possible output.

If a semantics will only be used to answer demonic questions, then the powerdomain $\mathbf{P}((Z \times Z)_\perp)$ can be simplified. Similarly, the powerdomain is simplified for angelic questions. These two simplifications provide two alternative ND-domains for nondeterministic XY-programs: $\mathbf{P}_{DEMON}$ and $\mathbf{P}_{ANGEL}$. These two ND-domains are defined in the two columns below.

P_{DEMON}

Consider a set S of possible outcomes, with $\perp \in S$. The answer to a demonic question about S is always "no" (because $b(\perp) = \perp$). Therefore it does no harm to add other elements to S — adding these elements won't change the "no" answer to demonic questions. In fact, I may as well add every possible state to S, making it the complete set $(Z \times Z)_\perp$. This set is also called *chaos*, since it is the most nondeterministic set possible. Hence, if $\perp$ is a possible outcome of a program with input x, then (in the demonic semantics) we say that the program maps x to *chaos* — and this does not change the answer to demonic questions.

With this in mind $\mathbf{P}_{DEMON}$ can be formed from the sets of $\mathbf{P}((Z \times Z)_\perp)$ by changing any set with $\perp$ to *chaos*. Hence, the elements of $\mathbf{P}_{DEMON}$ are all the nonempty, finite subsets of $(Z \times Z)_\perp$, plus one more set: *chaos*. The partial order is defined so that higher elements have more "yes" answers to demonic questions. Formally, $S \sqsubseteq T$ if and only if $S \supseteq T$, and **or** is set-union. The least element is *chaos*, and the least upper-bound operation is intersection of countably many sets.

P_{ANGEL}

Consider a set S of possible outcomes, with $\perp \notin S$. The answer to an angelic question about S is always the same as the answer to the same question about $S \cup \{\perp\}$ (because $b(\perp) = \perp$). Therefore it does no harm to add $\perp$ to S — adding $\perp$ won't change the answer to any angelic question.

With this in mind $\mathbf{P}_{ANGEL}$ can be formed from the sets of $\mathbf{P}((Z \times Z)_\perp)$ by adding $\perp$ to every set. Hence, the elements of $\mathbf{P}_{ANGEL}$ are all the subsets of $(Z \times Z)_\perp$, which contain $\perp$. The partial order is defined so that higher elements have more "yes" answers to angelic questions. Formally, $S \sqsubseteq T$ if and only if $S \subseteq T$, and **or** is set-union. The least element is $\{\perp\}$, and the least upper-bound operation is union of countably many sets.

These ND-domains can be used to provide a semantics for nondeterministic *XY*-programs, which will be adequate if we are only interested in program properties that can be answered with demonic or angelic questions. But before this is shown I want to discuss some universal properties that these domains possess.

5.2 Free Generation of Demonic and Angelic Powerdomains

The partial order on $\mathbf{P}_{DEMON}$ is such that a higher position in the order corresponds to more "yes" answers to demonic questions, while $\mathbf{P}_{ANGEL}$ has the same property for angelic questions. These correspondences translate to properties about the **or**-operation, namely: $(x \textbf{ or } y \sqsubseteq x)$

always holds in $\mathbf{P}_{DEMON}$, and ($x \sqsubseteq x$ **or** y) always holds in $\mathbf{P}_{ANGEL}$. We use the term *demonic ND-domain* for any ND-domain with this first property (x **or** $y \sqsubseteq x$), and *angelic ND-domain* for any ND-domain with the second property ($x \sqsubseteq x$ **or** y). When we are wanting answers to demonic questions we need a demonic ND-domain, and similarly for angelic questions.

Now, suppose I want to give a demonic nondeterministic semantics for a language whose deterministic semantics is already specified over some domain D. In this case, I must search for a demonic ND-domain $\mathbf{P}_{DEMON}(D)$, together with a domain morphism $\eta: D \to \mathbf{P}_{DEMON}(D)$. As in Section 4.3, I want $\mathbf{P}_{DEMON}(D)$ to be free from any unintended constraints. This is achieved by requiring the embedding η to be universal with respect to other possible embeddings of D into demonic ND-domains. This is formally described in the left-hand column below, and the same ideas are defined for angelic ND-domains on the right.

Free Demonic Powerdomain

Let D be any domain. The *free demonic powerdomain generated by D* is a demonic ND-domain $\mathbf{P}_{DEMON}(D)$ together with a domain morphism $\eta: D \to \mathbf{P}_{DEMON}(D)$ (called the insertion). The insertion η is universal, so that if Q is any demonic ND-domain and $g: D \to Q$ is any domain morphism, then there is a unique ND-domain morphism $\hat{g}: \mathbf{P}_{DEMON}(D) \to Q$ such that

$$\hat{g}(\eta(x)) = g(x) \text{ for any } x \in D.$$

Free Angelic Powerdomain

Let D be any domain. The *free angelic powerdomain generated by D* is an angelic ND-domain $\mathbf{P}_{DEMON}(D)$ together with a domain morphism $\eta: D \to \mathbf{P}_{DEMON}(D)$ (called the insertion). The insertion η is universal, so that if Q is any angelic ND-domain and $g: D \to Q$ is any domain morphism, then there is a unique ND-domain morphism $\hat{g}: \mathbf{P}_{ANGEL}(D) \to Q$ such that

$$\hat{g}(\eta(x)) = g(x) \text{ for any } x \in D.$$

Exercise 7: Let $C_\perp$ be any flat domain. Show that $\mathbf{P}_{DEMON}(C_\perp)$ consists of the non-empty finite subsets of C plus chaos (all of $C_\perp$), with the insertion mapping $\perp$ to chaos, and each other element is mapped to the corresponding singleton. The order is the superset order ($S \sqsubseteq T$ if and only if $S \supseteq T$) and the **or**-operation is set-union. Hence, $\mathbf{P}_{DEMON}$ (from Section 5.1) is the free demonic powerdomain generated by $(Z \times Z)_\perp$.

Exercise 8: Let $C_\perp$ be any flat domain. Show that $\mathbf{P}_{ANGEL}(C_\perp)$ consists of the countable subsets of $C_\perp$ which contain $\perp$. The insertion maps each element x to $\{x, \perp\}$. The order is the subset order and the **or**-operation is set-union. Hence, $\mathbf{P}_{ANGEL}$ (from Section 5.1) is the free angelic

powerdomain generated by $(Z \times Z)_\perp$.

6. Semantics via Free Powerdomains

A flat domain $D_\perp$ generates three powerdomains:

$\mathbf{P}(D)$ — the freely generated powerdomain,

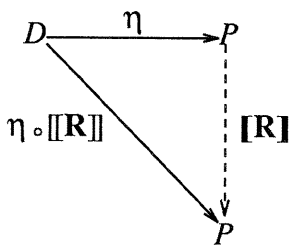
$\mathbf{P}_{DEMON}(D)$ — the freely generated demonic powerdomain,

$\mathbf{P}_{ANGEL}(D)$ — the freely generated angelic powerdomain.

There is an insertion from D to each of these domains, with a certain universal property. When this kind of property exists, we can usually give the semantics of a nondeterministic language simply by using the universal property. We don't even need to know what the powerdomain looks like. That's what we'll do in this section for the nondeterministic XY-language.

To start things off, let P be *any* of the three powerdomains $\mathbf{P}((Z \times Z)_\perp)$ or $\mathbf{P}_{DEMON}((Z \times Z)_\perp)$ or $\mathbf{P}_{ANGEL}((Z \times Z)_\perp)$. Let $\eta: D \to P$ be the corresponding universal insertion. For each nondeterministic XY-program $\mathbf{R}$, we will give the ND-domain morphism $\mathbf{[R]}: P \to P$ denoted by $\mathbf{R}$.

Deterministic Programs: Each deterministic XY-program $\mathbf{R}$ is also a nondeterministic program. Suppose $[\![\mathbf{R}]\!]: D \to D$ is the deterministic domain morphism associated with a program $\mathbf{R}$. Notice that the composition function $\eta \circ [\![\mathbf{R}]\!]$ maps D to P. The ND-domain morphism $\mathbf{[R]}: P \to P$ is the unique ND-domain morphism such that $\mathbf{[R]} \circ \eta = \eta \circ [\![\mathbf{R}]\!]$:



This unique morphism exists because η is universal. Intuitively, $\mathbf{[R]}: P \to P$ is the unique powerdomain morphism whose value at a "singleton" $\{x\}$ is $[\![\mathbf{R}]\!](x)$.

Exercise 9: Prove that $\mathbf{[}FAIL\mathbf{]}: P \to P$ maps every element of P to $\perp$, and that $\mathbf{[}SKIP\mathbf{]}$ is the identity on P.

Nondeterministic Choice: Let $\mathbf{R}_1$ and $\mathbf{R}_2$ be nondeterministic XY-programs. Then $(\mathbf{R}_1 \textbf{ or } \mathbf{R}_2)$ is a new nondeterministic XY-program. The ND-domain morphism $[\![(\mathbf{R}_1 \textbf{ or } \mathbf{R}_2)]\!]: P \to P$ maps an element $S \in P$ to $[\![\mathbf{R}_1]\!](S)$ **or** $[\![\mathbf{R}_2]\!](S)$.

Composition: Let $\mathbf{R}_1, \mathbf{R}_2, \cdots, \mathbf{R}_k$ be nondeterministic XY-programs. Then

$$\textit{BEGIN } \mathbf{R}_1; \mathbf{R}_2; \cdots \mathbf{R}_k \textit{ END}$$

is a nondeterministic XY-program whose ND-domain morphism is the composition

$$[\![\mathbf{R}_k]\!] \circ \cdots \circ [\![\mathbf{R}_2]\!] \circ [\![\mathbf{R}_1]\!]: P \to P$$

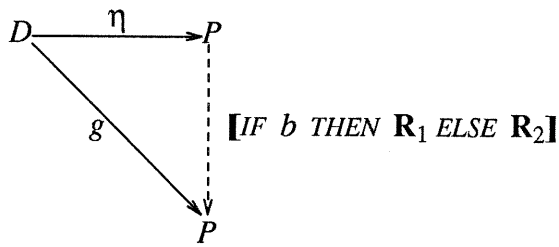
Conditional Statements: Let $\mathbf{R}_1$ and $\mathbf{R}_2$ be XY-programs and let B be any Boolean expression in the XY-language. Then this is a nondeterministic XY-program:

$$\textit{IF } b \textit{ THEN } \mathbf{R}_1 \textit{ ELSE } \mathbf{R}_2$$

To define the semantics of the conditional, let $b: (Z \times Z) \to \{\textit{TRUE}, \textit{FALSE}\}$ be the function which corresponds to the Boolean expression B. Next, define a domain morphism $g: D \to P$ by this:

$$g(x) = \begin{cases} \text{if } x = \perp \textit{ then } \perp \\ \text{else if } b(x) \textit{ is TRUE then } [\![\mathbf{R}_1]\!](\eta(x)) \\ \text{else } [\![\mathbf{R}_2]\!](\eta(x)) \end{cases}$$

The function $[\![\textit{IF } b \textit{ THEN } \mathbf{R}_1 \textit{ ELSE } \mathbf{R}_2]\!]: P \to P$ is the unique ND-domain morphism such that $[\![\textit{IF } b \textit{ THEN } \mathbf{R}_1 \textit{ ELSE } \mathbf{R}_2]\!] \circ \eta = g$, as shown here:



Exercise 10: Prove that each of the functions defined on P in the above paragraphs is an ND-domain morphism.

Iterative Statements: Let $\mathbf{R}$ be a nondeterministic XY-program, and let B be a Boolean expression in the XY-language. Then this is a nondeterministic XY-program:

WHILE B DO **R**

The ND-domain morphism ⟦*WHILE B DO* **R**⟧, is defined as a least upper-bound of a sequence, just as we did in the other examples of semantics. Namely, $\mathbf{R}_0$ is the *XY*-program *FAIL*, and for any integer $n>0$, $\mathbf{R}_n$ is the *XY*-program:

IF B THEN BEGIN **R**; $\mathbf{R}_{n-1}$ *END*
ELSE SKIP

The meaning of the iterative *XY*-program is the least upper-bound of the chain

$$[\![R_0]\!] \sqsubseteq [\![R_1]\!] \sqsubseteq [\![R_2]\!] \sqsubseteq \cdots$$

Exercise 11: Justify the following statement: In order to use a powerdomain, it is generally enough to know that the powerdomain is freely generated by a fixed domain D.

7. A Free Powerdomain Construction

So far, we have used powerdomains to define the semantics of nondeterministic programming languages. These examples used powerdomains of the form $\mathbf{P}(D)$, $\mathbf{P}_{DEMON}(D)$ or $\mathbf{P}_{ANGEL}(D)$, where D was a flat domain. What if D is a more complicated domain? Are there always free powerdomains, generated by D, with the desired universal properties?

For $\mathbf{P}(D)$, $\mathbf{P}_{DEMON}(D)$ and $\mathbf{P}_{ANGEL}(D)$, the answer is *yes*! The easiest way to prove this existence makes use of some category theory, and will not be given here — but see the bibliography if you're interested. Usually, this existence of freely generated powerdomains is sufficient to define the semantics of nondeterministic languages – since once the existence of a free powerdomain is known, we can make use of its universal properties without really knowing what the powerdomain looks like in set-theoretic terms — as was done in Section 6.

But, sometimes a more concrete construction of a free powerdomain is needed – or at least comforting. This section provides such a construction for my favorite powerdomain: $\mathbf{P}_{DEMON}(D)$.

7.1 What $\mathbf{P}_{DEMON}(D)$ Looks Like.

Let D be any domain. In order to describe $\mathbf{P}_{DEMON}(D)$, we need some definitions about subsets of D.

Definitions. The *upward-closure* of a subset $S \subseteq D$ is the set

$$S\uparrow = \{y \mid \text{ There exists some } x \in S \text{ with } x \sqsubseteq y\}.$$

A subset $S \subseteq D$ is called *upward-closed* if $S = S\uparrow$. A set C *covers* another set S provided that $C\uparrow \supseteq S$. A set S is called *Scott-compact* provided that whenever a set C of isolated elements covers S, then some finite subset of C also covers S.

Figure 7.1 Definitions of Upward-Closed and Scott-Compact Sets

Now we can describe $\mathbf{P}_{DEMON}(D)$. The elements of this domain are certain subsets of D. Namely, the elements of $\mathbf{P}_{DEMON}(D)$ are the non-empty, upward-closed, Scott-compact subsets of D. Intuitively, such a subset $S \subseteq D$ is a nondeterministic choice between all of the elements of S. The intuition behind upward-closure comes from the idea of asking demonic questions, as defined in Section 5. Part of a demonic question is the implicit idea that whenever we are willing to accept an outcome x from our program, then we should also accept any outcome y with $x \sqsubseteq y$. Hence, it does no harm to add a possible outcome y to a set of nondeterministic choices which already contains some lower element x The intuition behind Scott-compactness is essentially that Scott-compact sets are the kinds of nondeterministic choices that can occur in a program with finitely-branching nondeterminism (similar to the way that $\perp$ had to be added to each of the infinite sets in $\mathbf{P}(C_\perp)$).

The partial-order on $\mathbf{P}_{DEMON}(D)$ is $S \sqsubseteq T$ iff $S \supseteq T$. Again, this is justified by the idea that we will use this domain to answer demonic questions, and that a higher location in the semantic order corresponds to more "yes" answers to demonic questions. The least element in this order is the set D itself. If $S_1 \sqsubseteq S_2 \sqsubseteq S_3 \cdots$ is a chain in $\mathbf{P}_{DEMON}(D)$, then $\bigsqcup_{n=0}^{\infty} S_n$ is the intersection $\bigcap_{n=0}^{\infty} S_n$.

Exercise 12: Prove that this intersection is indeed a non-empty, upward-closed, Scott-compact set. Therefore, $\mathbf{P}_{DEMON}(D)$ has least upper-bounds of all chains, and hence is a CPO.

Exercise 13: Let S be a non-empty finite set of isolated elements from D. Show that $S\uparrow$ is an element of $\mathbf{P}_{DEMON}(D)$. As a further exercise, show that these elements are isolated in the CPO

$\mathbf{P}_{DEMON}(D)$, and that this CPO is actually a domain with these elements forming the base.

The binary operation **or** in $\mathbf{P}_{DEMON}(D)$ is union: $S \,\mathbf{or}\, T = S \cup T$. Since $\bigcap_{n=0}^{\infty} (S_n \cup T) = (\bigcap_{n=0}^{\infty} S_n) \cup T$, this is a continuous operation.

Exercise 14: Prove that $S \cup T$ is non-empty, upward-closed and Scott-compact whenever S and T are. Therefore, this addition operation is well-defined on $\mathbf{P}_{DEMON}(D)$.

7.2 The Universal Insertion Function, and Some Preliminaries

We want to show that $\mathbf{P}_{DEMON}(D)$ is the free demonic powerdomain generated by D, with some universal insertion function $\eta: D \to \mathbf{P}_{DEMON}(D)$. For this construction, the insertion function η maps each element $x \in D$ to the set $\{x\}\!\uparrow \in \mathbf{P}_{DEMON}(D)$.

Exercise 15: Clearly the set $\eta(x)$ is non-empty and upward-closed for any $x \in D$. Show that it is Scott-compact. Also show that the function η is a domain morphism.

The idea behind a proof that $\mathbf{P}_{DEMON}(D)$ is freely generated by D is this: Intuitively, the sets of $\mathbf{P}_{DEMON}(D)$ are those sets which can be formed from the upward-closure of singletons – using only the **or**-operation (union) and putting in least upper bounds (intersections) whenever chains are formed. A more formal proof will be aided by the following preliminary definitions and results.

Definition. A subset S of a domain D is called *directed* provided that any two elements in S have an upper bound in S. That is: whenever $x, y \in S$, then there exists some $z \in S$ with $x \sqsubseteq z$ and $y \sqsubseteq z$. Note that every chain is an example of a directed set.

Exercise 16: Let $S \subseteq D$ be a directed subset of a domain D. Prove that S has a least upper-bound (which we will denote by $\bigsqcup S$). Also prove that every domain morphism preserves the least upper-bound of every directed set.

Definition. Let D be a domain, Q a demonic ND-domain, and $g:D \to Q$ a domain morphism. Then for any finite subset $S = \{x_1, \cdots, x_n\} \subseteq D$, define $g(S)$ to be the element $g(x_1)$ **or** $\cdots$ **or** $g(x_n)$ of Q.

Lemma 7.1. *Let D be a domain, Q a demonic ND-domain, and $g:D \to Q$ a domain morphism. Then for any set $T \in \mathbf{P}_{DEMON}(D)$, the set*

$$\{g(S) \mid S \text{ is a finite set of isolated elements which covers } T\}$$

is a directed subset of Q.

Proof: Let U be the indicated subset of Q, which we must show to be directed. Suppose S_1 and S_2 are finite sets of isolated elements which cover T, so that $g(S_1)$ and $g(S_2)$ are elements of U. We must find another finite set S of isolated elements which covers T, and with $g(S)$ an upper bound for $g(S_1)$ and $g(S_2)$. Before we define this set S, we define another set of isolated elements:

$$V = \{x \in BASE_D \mid \text{For some } x_1 \in S_1,\ x_2 \in S_2, \text{ and } t \in T : (x_1 \sqsubseteq x) \text{ and } (x_2 \sqsubseteq x) \text{ and } (x \sqsubseteq t)\}.$$

It is easy to show that V covers T, and since T is Scott-compact there is some finite subset $S \subseteq V$ which also covers T. For this S, the element $g(S)$ is in U, and it is an upper-bound for $g(S_1)$ and $g(S_2)$. Therefore, U is directed, as required. □

Exercise 17: In the proof of the last lemma, show that V covers T.

Lemma 7.2. *Every domain morphism between demonic ND-domains is also a ND-domain morphism.*

Proof: The proof relies on the fact that in a demonic ND-domain, x **or** y is always the greatest lower-bound of x and y. (It is a lower bound since x **or** $y \sqsubseteq x$ is an axiom in a demonic ND-domain. It is the greatest lower bound since whenever $z \sqsubseteq x$ and $z \sqsubseteq y$ then also $z = (z \text{ or } z) \sqsubseteq (x \text{ or } y)$.) But this g.l.b. can also be written as the least upper-bound of the directed set $\{z \mid z \sqsubseteq x \text{ and } z \sqsubseteq y\}$. (The set is directed since it contains its own upper-bound, namely x **or** y.) Since a domain morphism preserves the least upper-bound a of directed set, it also preserves the greatest lower-bound of a pair of elements, hence it also preserves the sum of a pair of elements. □

7.3 The Proof that $\mathbf{P}_{DEMON}(D)$ is Free

Let D be a domain. We will show that $\mathbf{P}_{DEMON}(D)$ is generated by D with the insertion $\eta: D \to \mathbf{P}_{DEMON}(D)$, defined above. To do this, let Q be a demonic ND-domain, and let $g: D \to Q$ be a domain morphism. We must show that there is a unique ND-domain morphism $\hat{g}: \mathbf{P}_{DEMON}(D) \to Q$ such that $\hat{g}(\eta(x)) = g(x)$ for all $x \in D$, as in this commuting diagram:

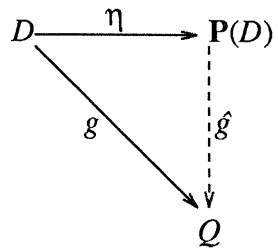


Figure 7.2

With the aid of Lemma 7.1, the definition of $\hat{g}$ is simple:

For any $T \in \mathbf{P}_{DEMON}(D)$: $\hat{g}(T) = \bigsqcup\{g(S) \mid S$ is a finite set of isolated elements which covers $T\}$.

Lemma 7.1 is used to guarantee that the set on the right is indeed directed, and hence has a least upper-bound in Q.

Now we will demonstrate that this definition of $\hat{g}$ meets the properties stated above. The first property required of $\hat{g}$ is that $\hat{g}(\eta(x)) = g(x)$ for any $x \in D$. This is shown here for any $x \in D$:

$$\hat{g}(\eta(x)) = \hat{g}(\{x\}\uparrow)$$

$$= \bigsqcup\{g(S) \mid S \text{ is a finite set of isolated elements which covers } \{x\}\uparrow\}$$

$$= \bigsqcup\{g(S) \mid S \text{ is a finite set of isolated elements with some } y \in S \text{ and } y \sqsubseteq x\}$$

$$= g(x)$$

Exercise 18: Show that $g(x)$ is indeed the least upper-bound indicated above.

Next we must show that $\hat{g}$ is an ND-domain morphism. By Lemma 7.2, every domain morphism on demonic ND-domains is also an ND-domain morphism, so really we only need to show that $\hat{g}$ is a domain morphism — *i.e.*, that $\hat{g}$ is strict, monotonic and continuous. Strictness and monotonicity are easy (try them and see!). For continuity, let $T_0 \sqsubseteq T_1 \sqsubseteq T_2 \cdots$ be a chain in

$\mathbf{P}_{DEMON}(D)$. The equality $\hat{g}(\bigsqcup_{n=0}^{\infty} T_n) = \bigsqcup_{n=0}^{\infty} (\hat{g}(T_n))$ is shown here:

$$\hat{g}(\bigsqcup_{n=0}^{\infty} T_n) = \hat{g}(\bigcap_{n=0}^{\infty} T_n)$$

$$= \bigsqcup\{g(S) \mid S \text{ is a finite set of isolated elements which covers } \bigcap_{n=0}^{\infty} T_n\}$$

$$= \bigsqcup\{g(S) \mid S \text{ is a finite set of isolated elements which covers some } T_n\}$$

$$= \bigsqcup_{n=0}^{\infty} \bigsqcup\{g(S) \mid S \text{ is a finite set of isolated elements which covers } T_n\}$$

$$= \bigsqcup_{n=0}^{\infty} (\hat{g}(T_n))$$

Exercise 19: Demonstrate in detail the above equalities. Hint: the second equality uses the fact that whenever S is a finite set of isolated elements from D, then $S\uparrow$ is an isolated element in $\mathbf{P}_{DEMON}(D)$.

So, we have shown that $\hat{g}$ is an ND-domain morphism which makes the triangle in Figure 7.2 commute. To finish the free construction proof, we only need to show that it is the only ND-domain morphism with this property. For this purpose, let $h: \mathbf{P}_{DEMON}(D) \to Q$ be an ND-domain morphism with $h(\eta(x)) = g(x)$ for all $x \in D$. We need to show that $\hat{g}$ and h are identical. It is sufficient to show that $\hat{g}$ and h are identical when applied to isolated elements of $\mathbf{P}_{DEMON}(D)$. As stated above, an isolated element of $\mathbf{P}_{DEMON}(D)$ has the form $S\uparrow$ where S is a non-empty finite subset of isolated elements from D. If $S = \{s_1, \cdots, s_n\}$, then $S\uparrow$ can also be written as $\eta(s_1)$ **or** $\cdots$ **or** $\eta(s_n)$. Hence we have these equalities:

$$\hat{g}(S\uparrow) = \hat{g}(\eta(s_1) \textbf{ or } \cdots \textbf{ or } \eta(s_n))$$

$$= \hat{g}(\eta(s_1)) \textbf{ or } \cdots \textbf{ or } \hat{g}(\eta(s_n))$$

$$= g(s_1) \textbf{ or } \cdots \textbf{ or } g(s_n)$$

$$= h(\eta(s_1)) \textbf{ or } \cdots \textbf{ or } h(\eta(s_n))$$

$$= h(\eta(s_1) \textbf{ or } \cdots \textbf{ or } \eta(s_n))$$

$$= h(S\uparrow)$$

7.4 Constructions of $\mathbf{P}(D)$ and $\mathbf{P}_{ANGEL}(D)$

The powerdomains $\mathbf{P}(D)$ and $\mathbf{P}_{ANGEL}(D)$ also have constructive characterizations similar to the one given here for the demonic powerdomain. Here's what these characterization are:

For any subset $S \subseteq D$, the *-closure of S, written $CLOSURE(S)$ is the set:
$\{x \in D |$ For any isolated y with $y \sqsubseteq x$ there exists $z \in X$ *with* $y \sqsubseteq z\}$
The powerdomain $\mathbf{P}_{ANGEL}(D)$ consists of those non-empty subsets of D with $S = CLOSURE(S)$. The order is defined by $S \sqsubseteq T$ if and only if $S \subseteq T$, and the insertion $\eta{:}D \to \mathbf{P}_{ANGEL}(D)$ maps $x \in D$ to $CLOSURE(\{x\})$. The binary **or** -operation is union.

Figure 7.3 Characterization of the Free Angelic Powerdomain

$\mathbf{P}(D)$ consists of those non-empty subsets of D with $S = S\uparrow \cap CLOSURE(S)$. The order is the Egli-Milner order, and the insertion $\eta{:}D \to \mathbf{P}(D)$ maps $x \in D$ to $\{x\}\uparrow \cap CLOSURE(\{x\})$. The binary **or** -operation is union.

Figure 7.4 Characterization of the Free Powerdomain

These characterizations are taken from Gordon Plotkin's postgraduate notes. See the historical bibliography for details.

8. Other Powerdomains

A powerdomain is a domain together with extra structure for handling nondeterminism. In $\mathbf{P}(D)$, this structure is a binary operation **or** which is associative, commutative, idempotent, and continuous. For demonic and angelic powerdomains, the same kind of operation is used, but with additional constraints.

These three powerdomains have not exhausted the possible structures for handling nondeterminism. For example, Kozen, Saheb-Djahromi and Graham have all proposed probabilistic structures that can be placed on top of a domain. David Benson and I have suggested the algebraic structure of a semiring-module for describing certain kinds of nondeterminism. At a recent talk at the Workshop on Mathematical Semantics of Programming, Gordon Plotkin suggested an algebraic structure containing several operations including nondeterministic choice and parallel composition.

The basic idea of a universal or free construction should be applicable to all these suggested structures. In each of these cases, we should be looking for a way to take a domain D and embed it in another domain which has additional algebraic structure. Let's call this latter domain a *structured domain*. So, what we are looking for is a structured domain $\mathbf{S}(D)$, together with a domain morphism $\eta: D \to \mathbf{S}(D)$. As usual, we want η to be universal so that if Q is another structured domain and $g: D \to Q$ is a domain morphism, then there exists a unique structure-preserving domain morphism $\hat{g}: \mathbf{S}(D) \to Q$ such that $\hat{g} \circ \eta = g$.

Several papers in the historical bibliography show conditions under which the existence of such free structured domains is guaranteed.

9. Historical Bibliography

A. Origins. The problem of defining domains of nondeterministic values has its origin in R. Milner's research on denotational semantics of nondeterministic and parallel programs [A.1,A.2]. He proposed a solution for flat domains, which was also proposed by H. Egli in unpublished notes. G. Plotkin extended Milner's proposal to a class of domains which he called SFP — although the construction was not characterized as a universal construction in his original paper [A.3]. M. Smyth refined Plotkin's construction — characterizing it in terms of a completion by ideals of a pre-domain [A.4]. Smyth also defined the demonic powerdomain construction — which he called the "weak"

powerdomain, and others have sometimes called the Smyth powerdomain. Smyth later recognized the angelic powerdomain as the dual to the demonic powerdomain, and connected all three powerdomains to topological constructions proposed by Vietoris in the 1920's [A.5]. The view which I've presented — powerdomains as freely generated ND-domains — came from Plotkin's postgraduate notes and a paper of Plotkin and M. Hennessy [A.6,A.7]. They also used the term Hoare powerdomain for the angelic powerdomain — because of a connection of the angelic powerdomain with Hoare's partial correctness logic for nondeterministic programs. D. Schmidt's text on Denotational Semantics also has a chapter presenting these constructions [A.8].

[A.1]

R. Milner, An approach to the semantics of parallel programs. In: *Proceedings of the Convegno di Informatica Teorica*, Instituto di Elaborazione della Informazione, Pisa (1973).

[A.2]

R. Milner, Processes: A mathematical model of computing agents. In: *Logic Colloquium '73*, North-Holland, Amsterdam (1973).

[A.3]

G. Plotkin, A powerdomain construction, *Siam J. Computing* 5 (1976), 452-487.

[A.4]

M. Smyth, Powerdomains, *Journal of Computer and System Sciences* 16 (1978), 23-36.

[A.5]

M. Smyth, Powerdomains and predicate transformers: a topological view, In: *Automata, Languages and Programming, 10th Colloquium*, Lecture Notes in Computer Science 154, Springer-Verlag, Berlin (1983), 662-675.

[A.6]

G. Plotkin, *Computer Science Postgraduate Notes*, University of Edinburgh, 1980-81.

[A.7]

M. Hennessy and G. Plotkin, Full abstraction of a simple parallel programming language, In: *Mathematical Foundations of Computer Science '79*, Lecture Notes in Computer Science 74, Springer-Verlag, Berlin (1979), 108-120.

[A.8]

D. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, Boston (1986).

B. Characterizations of Powerdomains. Apart from Smyth's topological characterization of

powerdomains, and the Plotkin/Hennessy view as a universal construction, there have been several other views of powerdomains. S. Abramsky constructed the powerdomains as capturing the discriminations that can be made by different kinds of finite experiments on nondeterministic programs [B.1]. G. Winskel defined the powerdomains in terms of statements in a modal logic [B.2]. In fact, these two papers — presented at the same conference — are the origin for the demonic questions and angelic questions which I used as motivation in Section 5. For a special kind of domain (called consistently complete) a topological characterization in terms of the Lawson topology of a domain has been given by M. Mislove [B.3].

[B.1]

S. Abramsky, Experiments, powerdomains and fully abstract models for applicative multiprogramming, In: *Foundations of Computation Theory*, Lecture Notes in Computer Science 158, Springer-Verlag, Berlin (1983), 1-13.

[B.2]

G. Winskel, A note on powerdomains and modality, In: *Foundations of Computation Theory*, Lecture Notes in Computer Science 158, Springer-Verlag, Berlin (1983), 505-514.

[B.3]

M. Mislove, On the Smyth powerdomain, In: *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, Springer-Verlag, Berlin (1987), to appear.

Alternative Algebraic Structures. Here are some of the alternative algebraic structures that have been proposed for handling nondeterminism in a domain.

[C.1]

S. Graham, Closure properties of a probabilistic powerdomain construction, In: *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, Springer-Verlag, Berlin (1987), to appear.

[C.2]

K. Hrbacek, A powerdomain construction, In: *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, Springer-Verlag, Berlin (1987), to appear.

[C.3]

D. Kozen, Semantics of probabilistic programs, *Journal of Computer and System Sciences* 22 (1981), 328-350.

[C.4]

M. Main, Free constructions of powerdomains, In: *First Workshop on the Mathematical*

Foundations of Programming Language Semantics, Springer-Verlag, Berlin (1985), 162-183.

[C.5]

A. Poinge, A remark on variations of powerdomains, *Bulletin of the EATCS* 31 (1987), 38-41.

[C.6]

G. Plotkin, A powerdomain for countable nondeterminism, In: *Automata, Languages and Programming, 9th Colloquium*, Lecture Notes in Computer Science 140, Springer-Verlag, Berlin (1982), 418-428.

[C.7]

N. Saheb-Djahromi, CPO's of measures for nondeterminism, *Theoretical Computer Science* 12 (1980), 19-37.