Non-determinism in Functional Languages

H. SØNDERGAARD† AND P. SESTOFT‡

† *Dept. of Computer Science, University of Melbourne, Parkville, VIC. 3052, Australia*

‡ *Dept. of Computer Science, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark*

The introduction of a non-deterministic operator in even a very simple functional programming language gives rise to a plethora of semantic questions. These questions are not only concerned with the choice operator itself. A surprisingly large number of different parameter passing mechanisms are made possible by the introduction of bounded non-determinism. The diversity of semantic possibilities is examined systematically using denotational definitions based on mathematical structures called power domains. This results in an improved understanding of the different kinds of non-determinism and the properties of different kinds of non-deterministic languages.

Received August 1988, revised May 1992

GLOSSARY OF SYMBOLS

Sets

$\cup\{A \mid p\}$	distributed union of a family $\{A \mid p\}$ of sets
$\mathcal{P}A$	the power set of A, i.e. the set of all subsets of A
N	the set of positive integers: 1, 2, 3 ...
N_0	the set of non-negative integers: 0, 1, 2, 3, ...

Domains

$\bot$	bottom (least element of a domain)
$A_\bot$	flat domain (the ordered set $A \cup \{\bot\}$)
$\leqslant$	the partial ordering on a base domain
$\mathcal{P}[D]$	the power domain of domain D (with the Hoare, Smyth, or Plotkin ordering)
H	the Hoare ordering
S	the Smyth ordering
P	the Plotkin ordering
$\sqsubseteq$	any of the above three power domain orderings
$A \triangledown B$	collector of sets A and B (different for each of H, S, and P above)
$\triangledown\{A \mid p\}$	distributed collector of a family $\{A \mid p\}$ of sets

Choice operators

$a \sqcap b$	a or b, chosen at random
$\sqcap_A$	a randomly chosen element from the finite set A
amb	McCarthy's locally angelic choice operator[11]
choice(n)	Floyd's choice expression[5]

Miscellaneous

$e_1 + e_2$	e_1 plus e_2
$e_1 \rightarrow e_2, e_3$	if e_1 is non-zero then e_2 else e_3
$f(e_1, e_2)$	function f applied to the expression pair (e_1, e_2)
$g: A \rightarrow B$	a function g from A to B
$\lambda a.ga$	the function mapping each a to ga
$F[\![Q]\!]$	the denotation of program Q
$E[\![e]\!]$	the denotation of expression e
■	end of definition or example

1. INTRODUCTION

The present paper investigates the concept of non-determinism, in particular non-deterministic functional programming language constructs, their properties, and their mathematical semantics. The aim is to develop a better understanding of the properties of languages that include a non-deterministic choice operator, and to present this understanding in a precise but not overly technical way.

Non-determinism is often discussed in connection with concurrency, where it arises naturally by abstraction from the actual order of execution of different processes. A main problem when writing concurrent programs is to control this non-determinism, at least to some degree, and this is one of the reasons why semaphores, monitors, etc. have been introduced into concurrent programming languages. More recently, synchronous communication on named channels has been proposed as a simple but powerful control mechanism,[8] but even so, the risk of inadvertently creating non-determinism in concurrent programs is present. This has motivated a number of examinations of non-determinism in connection with concurrent programs.

However, this tangling of non-determinism with concurrency appears to have the unfortunate effect of blurring their distinction, perhaps owing to the fact that neither is normally understood very well. But fundamentally the two notions are independent. So to clarify notions related to non-determinism, the present work investigates them out of the context of concurrency.

Non-deterministic concepts have an established position in computer science. In the late fifties, Rabin and Scott[16] introduced non-deterministic finite automata, and Turing machines with oracles were introduced some years later by Kreider and Ritchie.[10] In both cases, non-determinism was implicit: an aspect of the device rather than part of the corresponding formalism.

Features for explicitly expressing non-determinism soon appeared in functional and sequential imperative languages. McCarthy[11] introduced a functional language with a binary choice operator *amb* which is 'ambiguous' or locally angelic, as we shall call it (cf. Section 2.3).

Floyd[5] discussed the concept of backtrack programming by introducing a non-deterministic expression *choice*(n) which yields an integer between 1 and n. In addition he introduced statements '*failure*' and '*success*'. One way of explaining these features is by means of an operational semantics based on 'splitting up machines': a statement '$x := $ *choice*(n)' will spawn n machines in accordance with the n different states possible after the statement. The statement '*failure*' eliminates the executing machine, and '*success*' forces it to output and stop.[9]

A very influential treatment of non-determinism is found in Dijkstra's book on the *guarded command* language.[4] This presentation provided operational intuition as well as mathematical (predicate transformer) semantics for an imperative language with non-determinism. In Dijkstra's language the non-determinism arises from the possibility that more than one guard is true in a generalized conditional, rather than from an explicit construct such as the *amb* operator. Recently, the field of logic programming has provided a paradigm for implicitly non-deterministic languages.[19]

When the mathematical description of programming languages was undertaken (by researchers such as Strachey, Landin, Floyd, and Scott) it appears that the treatment of non-determinism lagged behind. Thus Plotkin's seminal paper[15] that made non-determinism fit into the Scott–Strachey approach to semantics appeared some 6 years later than the papers by Scott and Strachey. Later the investigations into the semantics of concurrency have led to new treatments of non-determinism. Milner[13] has a particularly interesting discussion on experiments with non-deterministic devices.

It appears that non-determinism still is not as well understood as most other programming language concepts. This is reflected in the fact that there is no single accepted model for the concept, but rather a number of competing proposals, all of which have some drawbacks that are not very well understood.

In this paper we focus attention on the pragmatic aspects and the mathematical description of non-deterministic programming languages. We assume that the reader is acquainted with mathematical definition of programming languages in the form of denotational semantics. The aim is to remove the confusion which is abundant in many discussions of non-deterministic languages. We show how a very simple language can be interpreted in at least twelve different (natural) ways. The precise semantic definitions allow us to make general statements about the properties of these language variants.

The plan of the paper and the main points to notice are as follows. Our goal is to investigate parameter passing mechanisms and choice mechanisms in functional languages with a non-deterministic choice operator. Section 2 introduces a number of terms related to non-determinism in an informal manner. Section 3 defines and discusses a very simple functional language. In Section 4, this language is extended with a binary choice operator, and power domains are introduced as adequate semantic tools. Section 5 presents a spectrum of possible semantics for the extended language. We show the close relation between the various interpretations of a choice operator and the three standard power domains: the Hoare, Smyth, and Plotkin power domains. We also discuss the notions of definiteness, unfoldability, and referential transparency and investigate which language versions have such properties. Section 6 contains the conclusion.

2. BASIC CONCEPTS

In this section we introduce a number of concepts and distinctions related to non-determinism. This is done in a rather informal way, but many of the concepts are treated in greater depth in subsequent sections.

Assume we are given a discrete system subject to changes, or *state transitions*. If, at every point of time, the system's present state constitutes an amount of information sufficient for us to deduce its state after the next transition, then the system is said to be *deterministic*. Otherwise it is *non-deterministic*.

One way to illustrate the difference is to say that determinism gives rise to a linear state sequence, whereas non-determinism corresponds to a branching state tree. We may consider the former a special instance of the latter. A node in the state tree corresponds to a state of the transition system. A branching point in the tree indicates that a choice among the possible subtrees will effectively take place, but we cannot foresee the result before (possibly) reaching the point.

One may imagine that to every point is attached some agent who actually performs the choice. Non-determinism is thus ultimately defined in purely operational terms.

2.1 Weak and strong non-determinism

A problem that comes up in the *use* of non-deterministic constructs is: Should they be used in such a way that the result of a computation can be predicted? The two possible attitudes are:

(1) Yes, one should use non-determinism in a way such that programs are essentially deterministic. That is, their final results should never depend on the particular choices taken by an evaluator.

(2) No, one should never care for a particular result, but be satisfied with any of a number of possible results.

Case (1) is often referred to as *weak* non-determinism: though the system includes non-deterministic components, as a whole it behaves deterministically. This corresponds to the confluence or Church–Rosser property for rewriting systems. Many examples of weak non-determinism are found in Dijkstra's book.[4] We refer to case (2) as *strong* non-determinism.

The two notions cannot be attributed to agents. They are meaningful only at a certain observational level. They concern *systems*, just as the confluence property applies to a rewriting system as a whole, not to the rewrite rules individually.

The distinction between weak and strong non-determinism will not be further discussed as the rest of this paper will deal only with strong non-determinism.

2.2 Bounded and unbounded non-determinism

Bounded non-determinism refers to the case where every computation that is known to terminate has a finite number of possible results. This implies that every agent is confined to a finite number of choices. Thus, in the case of bounded non-determinism, the choice tree is finitely branching.

Unbounded non-determinism refers to the case where even computations that are known to terminate may have a (countably) infinite number of possible results. In this case agents may or may not be limited to a finite number of choices.

Dijkstra[4] shows that in the case of unbounded non-determinism, an 'agent' does not comply with a concept of 'terminating device': Turing machines are not able to perform choices among an unbounded number of possibilities in finite time.

In accordance with this, Hoare[8] declares that his $\sqcap_A$ ('choose an element from set A') is meaningless if A is an infinite or empty set. (We later argue that the binary choice operator $\sqcap$ is associative, commutative, and absorptive, which justifies the use of $\sqcap_A$ as a shorthand notation for $a_1 \sqcap \dots \sqcap a_n$, where $A = \{a_1, \dots, a_n\}$).

2.3 Angelic, demonic, and erratic non-determinism

The question of non-terminating computations becomes more complex when non-determinism is introduced. The reason is that now we have to discuss *possibly* non-terminating computations and the way in which possible termination depends on the non-deterministic choices made during the evaluation of the program. One can distinguish three kinds of non-determinism by considering the way choices are made when non-termination is possible.

With *angelic* (or prescient) non-determinism all choices are made in favour of termination if at all possible. We can distinguish two kinds of angelicism: local and global. By *locally angelic* non-determinism, a choice $e_1 \sqcap e_2$ between two expressions is made in such a way that if e_1 is undefined, then e_2 is chosen and vice versa; if both are undefined, so is the choice expression. McCarthy's choice operator *amb*[11] is locally angelic. By *globally angelic* non-determinism, on the other hand, choices are made so as to obtain termination of the *overall* computation if at all possible. This is a somewhat stronger concept: in a choice $0 \sqcap 1$ between two perfectly well-defined values, it may still be the case that the surrounding expression is defined if 1 is chosen and not if 0 is; in that case the operator must choose 1. Non-deterministic simulation techniques as used in complexity theory, for instance, employ globally angelic non-determinism.

With *demonic* non-determinism choices are made in favour of non-termination if at all possible. This is useful if we are concerned with terminating programs only. As with angelic non-determinism, we can distinguish between *local* and *global* demonicism. With local demonicism, a choice expression $e_1 \sqcap e_2$ will fail to terminate if any of e_1 or e_2 does. With global demonicism, choices will further be made so that a value that makes the entire computation fail will be given preference.

The use of the terms 'angelic' and 'demonic' is due to C. A. R. Hoare. In the rest of the paper we shall not deal with the cases of *locally* angelic or *locally* demonic choice. The reason is that a fixed-point theoretic characterization of these kinds of semantics is very difficult and still under investigation.

Erratic non-determinism comes closest to the intuitive idea of a program running and making non-deterministic choices on its way. In this kind of non-determinism choices are made, operationally speaking, by flipping a coin, which means that nothing is done to obtain or prevent termination. The term 'erratic' is due to M. Broy.

2.4 Restrained and unrestrained non-determinism

When we want to study the properties of a non-deterministic choice operator, it is useful to distinguish between the two cases: choice among atomic values (such as integers or finite character sequences), and choice among recursively defined values (such as functions or infinite data structures). The first case shall be referred to as *restrained* non-determinism, and the second one as *unrestrained* non-determinism.

Clearly the unrestrained case is conceptually more complex: full elaboration of the values chosen between may itself involve infinitely many choices. This is reflected in the problems with modelling unrestrained non-determinism. We shall deal only with restrained non-determinism in the rest of this paper. Problems with unrestrained non-determinism are discussed elsewhere.[12]

3. A SIMPLE DETERMINISTIC LANGUAGE

In this section we define a very simple functional language L. This language is deterministic but will be extended with a non-deterministic operator in Section 4. While L has the obvious (strict or non-strict) semantics, it is far from clear what the semantics of the extended language should be, and in Section 5 we discuss 12 reasonable candidates for a definition.

Care has been taken to make the language L as simple as possible, while still interesting. Simple as it is, we feel justified in using L as a basis for our discussion, because it contains features normally found in functional programming languages, albeit only those necessary for the present treatment. Section 3.1 defines some notions needed in the following, and Section 3.2 defines L.

3.1 Preliminaries

The reader is assumed to have a basic knowledge of denotational semantics. This section merely recapitulates some important notions that will be used in the following.

A *preordering* in a set A is a binary relation, $\leqslant$, that is reflexive and transitive; that is, for all $a \in A$, $a \leqslant a$, and for all $a, b, c \in A$, $a \leqslant b \wedge b \leqslant c$ implies $a \leqslant c$. A *partial ordering* in A is a preordering that is antisymmetric, that is, for all $a, b \in A$, $a \leqslant b \wedge b \leqslant a$ implies $a = b$. A set equipped with a partial ordering is called a *poset*. Given a poset A, a subset $C \subseteq A$ is a *chain*, iff $c_1 \leqslant c_2 \vee c_2 \leqslant c_1$ holds for all $c_1, c_2 \in C$.

Given a poset A and a subset $B \subseteq A$, an element $a \in A$ is an *upper bound* for B iff $b \leqslant a$ for all $b \in B$. Dually we may define a *lower bound* for B. An upper bound $a_1 \in A$ for B is a *least upper bound* for B iff, for every upper bound a_2 for B, $a_1 \leqslant a_2$. When it exists, we let $\sqcup B$ denote the least upper bound for B.

The poset A is *chain-complete* iff a least upper bound exists for every chain $C \subseteq A$. A *domain* is a chain-complete poset. Note that a domain A has a least element $\sqcup \varnothing$, which we denote by $\perp_A$ or sometimes simply $\perp$.

Assume we are given a set A not containing $\perp$. Setting $A_\perp = A \cup \{\perp\}$, we can form a structure $(A_\perp, \leqslant)$ which is clearly a domain, by defining $a \leqslant b$ iff $a = b \vee a = \perp$ for all $a, b \in A_\perp$. Such a structure is called a *flat* domain. In the following, we think of A as a set of *proper* values and of $\perp$ as denoting undefinedness. We denote the power set of a set A by $\mathscr{P}A$.

Let A and B be domains and let $g: A \to B$ be a function. We say that g is *strict* iff $g\perp_A = \perp_B$. The function g is *monotonic* iff $ga \leqslant gb$ whenever $a \leqslant b$; and g is *continuous* iff $g(\sqcup C) = \sqcup \{gc \mid c \in C\}$ for all non-empty chains C. Clearly a continuous function is monotonic.

A *fixed-point* for $g: A \to A$ is an element $a \in A$ for which $a = ga$. A fixed-point a for g is a *least fixed-point* for g iff, for every fixed-point b for g, $a \leqslant b$. If g is continuous (or just monotonic) then g has a least fixed-point.

3.2 The language L

The language L is deliberately very simple. It nevertheless includes all the features needed for the following discussion, except for a choice operator which is to be introduced in Section 4. A program consists of one (and only one) function definition. The function defined is called f and has two variables x and y, both of type integer. The function definition serves as a program by the convention that an initial call $f(0, 0)$ is understood. The syntax of L is given in Definition 3.1, and an informal semantic description is given in the annotations following the definition.

Definition 3.1 The syntax of L is:

1.	$pgm \rightarrow$	$f(x, y) == exp$	
2.	$exp \rightarrow$	int	– constant
3a.		$\mid x$	– first variable
3b.		$\mid y$	– second variable
4.		$\mid exp + exp$	– addition
5.		$\mid exp \rightarrow exp,\ exp$	– conditional
6.		$\mid f(exp, exp)$	– function application ■

Annotations to Definition 3.1

1. A program consists of a function definition. An initial call $f(0, 0)$ is understood.
2. Constants are drawn from the set int of integers $\ldots, -2, -1, 0, 1, 2, \ldots$
3. The variables x and y refer to the argument values of f.
4. There is an addition operator (as an example of an operation that is strict in both arguments).
5. The conditional evaluates to the value of the third expression if the first expression evaluates to zero, otherwise it evaluates to the value of the second expression. Thus it is an 'if non-zero'. The conditional is strict in the first expression.
6. The (only) function f may be called recursively. ■

We shall use parentheses freely to disambiguate or clarify expressions in L. The semantic functions that assign denotations to programs and expressions, respectively, have types

$$F\colon pgm \rightarrow int_{\perp}$$
$$E\colon exp \rightarrow den \rightarrow den \quad \text{where} \quad den = int_{\perp}^{2} \rightarrow int_{\perp}.$$

Here den is a domain of function denotations. Each is a total mapping that maps a pair of values for f's variables into a result. We order den pointwise, so for $d_1, d_2 \in den$,

$$d_1 \leqslant d_2 \quad \text{iff} \quad \forall w \in int_{\perp}^{2} . d_1 w \leqslant d_2 w.$$

The semantics of L is given by Definition 3.2.

Definition 3.2 (Deterministic semantics)

Below, $d \in den$; $n \in int$; $u, v \in int_{\perp}$; $w \in int_{\perp}^{2}$; z_i abbreviates $E[\![e_i]\!]\, dw \in int_{\perp}$.

1.	$F[\![f(x, y) == e]\!]$	$= d(0, 0)$ where rec $d = E[e]\, d$
2.	$E[\![n]\!]\, dw$	$= n$
3a.	$E[\![x]\!]\, dw$	$= u$ where $(u, v) = w$
3b.	$E[\![y]\!]\, dw$	$= v$ where $(u, v) = w$
4.	$E[\![e_1 + e_2]\!]\, dw$	$=$ if $z_1 = \perp$ or $z_2 = \perp$ then $\perp$ else $z_1 + z_2$
5.	$E[\![e_1 \rightarrow e_2, e_3]\!]\, dw$	$=$ if $z_1 = \perp$ then $\perp$ else if $z_1 \neq 0$ then z_2 else z_3
6.	$E[\![f(e_1, e_2)]\!]\, dw$	$= d(z_1, z_2)$ ■

Annotations to Definition 3.2

1. The denotation of f is the least solution to the equation $d = E[\![e]\!]d$. Since $E[\![e]\!]$ is continuous, d is well-defined. The result of the program is the result of applying the denotation d of f to the initial argument values $(0, 0)$.
3. The w in the semantics plays the role of a state. Since there are only two variables x and y, the state is determined by the values u and v of these variables, where $(u, v) = w$.
4. The special treatment of $\perp$ makes addition strict in both arguments. Operationally, this means that evaluation of $e_1 + e_2$ terminates only when evaluation of e_1 and e_2 does.
5. The special treatment of $\perp$ makes the conditional strict in the first argument.
6. The result of a recursive function call is that of applying the denotation d of f to the values of the argument expressions. ■

The semantics of L is well-defined since d is: for every expression e, the function $E[\![e]\!]$ is continuous and so has a least fixed-point.

Example 3.3. The program

$f(x, y) == x \rightarrow x, f(1, f(x, y))$ denotes 1. ■

By Definition 3.2, f is non-strict. This, however, is easily changed so that f becomes strict, by replacing the recursive definition of d in equation 1 by

$$d(u, v) = \text{if } u = \perp \text{ or } v = \perp \text{ then } \perp$$
$$\text{else } E[\![e]\!]\, d(u, v).$$

By the resulting semantics, the program in Example 3.3 denotes $\perp$, that is, the result of evaluating the program is undefined. We shall only consider the two cases: f being strict in both arguments or in none of them. These two cases correspond to applicative order and normal order evaluation, respectively, or 'call-by-value' and 'call-by-name'. As the example shows, normal order evaluation may terminate when applicative order evaluation will not. The converse does not hold.

In the non-deterministic case discussed in Section 4, the issue of call mechanisms becomes more complicated. Sections 5.1 and 5.2 cover these problems, and in Section 5.3 the diversity of semantics is exemplified.

4. NON-DETERMINISM AND POWER DOMAINS

In this section we discuss the implications of adding a binary non-deterministic choice operator to the language L. This leads to an exposition of the three classical power domains. Since our language is first order (functions are not data objects) the choice operator works on simple values (integers) only. Technically, therefore, we have non-deterministic choice among elements from flat domains (the power domains over which are said to be *discrete*). This is referred to as *restrained* non-determinism, and turns out to have a particularly simple semantics.

We now extend L by adding the following rule to the grammar of Definition 3.1:

$$exp \to exp \sqcap exp,$$

stating that the choice between two expressions is itself an expression.

Example 4.1. Consider the following programs (for now, assume non-determinism is erratic):

$$Q_1: f(x, y) == 1$$
$$Q_2: f(x, y) == x \to x, f(0 \sqcap 1, 0)$$
$$Q_3: f(x, y) == f(x, y).$$

We would expect that

$F[Q_1] = \{1\}$ $-Q_1$ will terminate and return 1,

$F[Q_2] = \{\perp, 1\} - Q_2$ may fail to terminate or may terminate and return 1,

$F[Q_3] = \{\perp\}$ $-Q_3$ will fail to terminate. ■

Clearly, the range of the semantic function F can no longer be $int_\perp$, since a result may in general be any of a number of possibilities, and the semantic definition must reflect this fact. So F and E need to record all the possible outcomes:

$$F: pgm \to \mathscr{P}\, int_\perp$$
$$E: exp \to den \to den.$$

But what should den be, that is, how do we want to model the behaviour of f? There are many possibilities, but the two most worthy of consideration are

$$den = int_\perp^2 \to \mathscr{P}\, int_\perp,$$

and $$den = (\mathscr{P}\, int_\perp)^2 \to \mathscr{P}\, int_\perp,$$

according as a variable is always bound to exactly one value or is bound to a set of possible values. Both possibilities are perfectly feasible, and we give the semantics for the two cases in Sections 5.1 and 5.2, respectively. The former case yields a so-called *singular* semantics, whereas the latter yields a *plural* semantics.[3] The singular semantics will record all possible states (in the traditional sense: each state binding a name to one of its possible values), whereas the plural semantics will work with 'states' that bind a name to the set of all its possible values. The resulting languages are very different, as we shall see. Most language designers seem to consider a singular semantics more natural than a plural semantics.

We now discuss the structure of $\mathscr{P}\, int_\perp$. As is usual, we want a fixed-point characterization of the semantics, so somehow $\mathscr{P}\, int_\perp$ must be equipped with a partial ordering. The resulting structure should be a domain (thus: a *power domain*), and we denote it by $\mathscr{P}[int_\perp]$. (In fact, $\mathscr{P}[int_\perp]$ will not in general contain all the elements of $\mathscr{P}\, int_\perp$: all three power domain constructions leave certain elements out, as we shall see). Given a power domain $\mathscr{P}[D]$, we refer to D as the *base domain.*

We denote an ordering on power domains by $\sqsubseteq$. Such an ordering should reflect the usual notion of 'information content'. For instance $\{\perp\} \sqsubseteq \{1\}$ should hold. In this way, the special case of deterministic programs will be treated according to the semantics of L. The problem, however, with the ordering is that it should capture both this *and* a concept of approximation of the set of possible results.

Traditionally, three different candidate orderings combining 'information content' and 'set approximation' orderings are proposed. We shall call them H for Hoare, S for Smyth, and P for Plotkin (the last one is often called the Egli–Milner ordering or the Milner ordering[17]). The three orderings and their intended interpretations are now discussed.

Definition 4.2. Let D be a domain, partially ordered by $\leqslant$. The relations H, S, and $P \subseteq (\mathscr{P}D)^2$, all denoted by $\sqsubseteq$, are defined by

H: $A \sqsubseteq B$ iff $\forall a \in A . \exists b \in B . a \leqslant b$

S: $A \sqsubseteq B$ iff $\forall b \in B . \exists a \in A . a \leqslant b$

P: $A \sqsubseteq B$ iff $\forall a \in A . \exists b \in B . a \leqslant b \wedge$
$\forall b \in B . \exists a \in A . a \leqslant b$ ■

Of these, P is the smallest relation since $H \cap S = P$. For an arbitrary domain D, the relations are actually preorderings, so $A \sqsubseteq B \wedge B \sqsubseteq A$ does not in general imply $A = B$. This means that some additional manipulations are needed in the general case of power domain construction. However, when the base domain is flat (as is the case in this paper since we consider restrained non-determinism only), P is a partial ordering. The relations H and S will also turn out to be partial orderings in our case, but this is due to the fact that certain elements of $\mathscr{P}\, int_\perp$ will be excluded in the power domain constructions below. Flat base domains lead to simple constructions, and the resulting power domains are called *discrete.*

We now turn to the actual constructions of the three versions of $\mathscr{P}[int_\perp]$ corresponding to H, S, and P. As will be seen, the three orderings introduced above closely correspond to (globally) angelic, (globally) demonic, and erratic non-determinism, respectively. Table 1 compares the orderings (and gives all possible yes/no combinations).

Definition 4.3. The *discrete Hoare power domain* construction is this: the elements of $\mathscr{P}[int_\perp]$ are those of

Table 1. Sample relations

Case	H	S	P
$\{1\} \sqsubseteq \{2\}$	No	No	No
$\{\perp, 1\} \sqsubseteq \{2\}$	No	Yes	No
$\{1\} \sqsubseteq \{1, 2\}$	Yes	No	No
$\{\perp, 1\} \sqsubseteq \{1, 2\}$	Yes	Yes	Yes

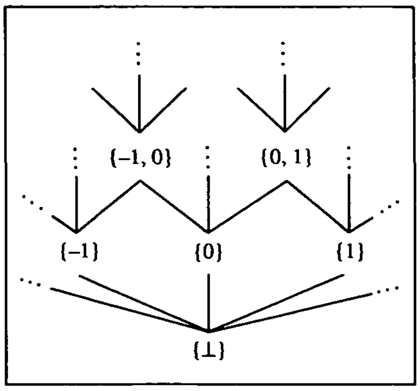

Figure 1. Hoare ordering of $\mathscr{P}[int_\perp]$.

$\mathcal{P}$ $int\backslash\{\emptyset\}$ and $\{\perp\}$. The ordering is H restrictd to $\mathcal{P}[int_\perp]$. See Figure 1.

We will use the Hoare power domain for modelling (globally) angelic non-determinism. The interpretation of $\{\perp\}$ is non-termination, and the interpretation of a non-empty set $A \neq \{\perp\}$ is that any element of A is a possible result, and non-termination is impossible. With angelic non-determinism, a computation may well be able to return infinitely many results, and therefore the Hoare power domain must include such infinite sets.

We can think of the discrete Hoare power domain as the power set $\mathcal{P}\,int_\perp$ where for any non-empty set $A \subseteq int$ we identify $A \cup \{\perp\}$ with A. If any of two sets contains a proper result (different from $\perp$) then their union will be identified with a set not containing $\perp$. This reflects the convention of angelic non-determinism that if it is possible to choose a proper result, then non-termination will be avoided.

Since $\{\perp\}$ is the bottom element, and A is identified with $A \cup \{\perp\}$, we could as well have used the usual power set $\mathcal{P}\,int$ and let $\emptyset$ play the role of $\{\perp\}$. The power domain $\mathcal{P}[int_\perp]$ and the power set $\mathcal{P}\,int$ (with subset ordering) are thus order isomorphic. However, $\mathcal{P}[int_\perp]$ is preferable in this context, since it allows for a uniform presentation of the power domains.

Definition 4.4. The *discrete Smyth power domain* construction is this: the elements of $\mathcal{P}[int_\perp]$ are $\{\perp\}$ and the finite elements of $\mathcal{P}\,int\backslash\{\emptyset\}$. The ordering is S restricted to $\mathcal{P}[int_\perp]$. See Figure 2. ■

The discrete Smyth power domain will be used for modelling globally demonic non-determinism. Again, $\{\perp\}$ is interpreted as non-termination, any other element as a set of possible results. With demonic non-determinism, no computation can return infinitely many different results, and therefore the Smyth power domain includes only finite sets.

The discrete Smyth power domain can be thought of as containing the finite subsets of $int \cup \{\perp\}$, when we identify any set containing $\perp$ with $\{\perp\}$. If any of two sets contains $\perp$ (representing non-termination), their union will be identified with $\{\perp\}$. This reflects the convention of demonic non-determinism that if it is possible to choose non-termination, then it will be chosen.

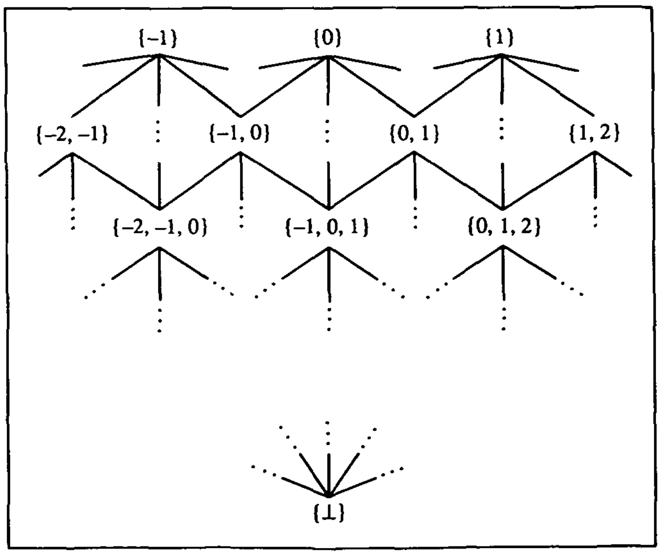

Figure 2. Smyth ordering of $\mathcal{P}$ $[int_\perp]$.

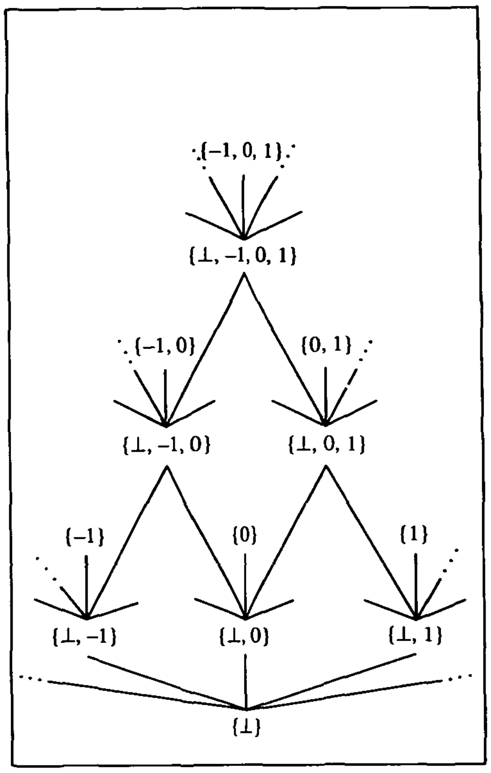

Figure 3. Plotkin ordering of $\mathcal{P}$ $[int_\perp]$.

Definition 4.5. The *discrete Plotkin power domain* construction is this: we take as the elements of $\mathcal{P}[int_\perp]$ all elements of $\mathcal{P}\,int_\perp\backslash\{\emptyset\}$ which are finite or contain $\perp$. The ordering is P restricted to $\mathcal{P}[int_\perp]$. See Figure 3. ■

The discrete Plotkin power domain also has $\{\perp\}$ as bottom element. A set A is interpreted as the set of results which must be possible, and may include $\perp$ indicating that non-termination is possible. All infinite sets also contain $\perp$ to reflect that with erratic non-determinism, if a computation can return infinitely many results, then it can also go on forever.

The Plotkin power domain is obviously useful for modelling erratic non-determinism, since the ordering does not use $\perp$ to identify sets of results, but rather treat non-termination ($\perp$) as a 'result' like any other.

The reason why the empty set $\emptyset$ is not in any of the power domains is clear: we must have a result (thinking of non-termination as a 'result'). Note that the restriction to *finite* sets in the Smyth and Plotkin cases means that only the Hoare case can model unbounded non-determinism.

A clarification of the difference between the Smyth and Plotkin power domains is offered by Apt and Plotkin.[1] They show (for a non-deterministic sequential language) that a denotational semantics based on a Plotkin power domain of states is equivalent to an operational (state transition) semantics. Correspondingly, a denotational semantics based on the Smyth power domain is shown equivalent to a predicate transformer ('weakest pre-condition') semantics.

So the Plotkin power domain (and operational semantics) can represent computations that *may fail to terminate*, whereas the Smyth power domain (and 'weakest precondition' predicate transformer semantics) is concerned only with computations that *will terminate*. Operationally, therefore, the Plotkin power domain allows for the best description of non-deterministic constructs. In particular, it is the only power domain (of those discussed here) that allows the three programs in Example 4.1 to be distinguished.

Finally, Winskel[20] discusses the relation between powerdomains and modality. He identifies three simple modal logics as the logical counterparts of the three powerdomains.

The powerdomains are usually presented in a way which is more uniform than ours, namely as quotients over the equivalence relations generated by $\sqsubseteq$, rather than as "powersets with certain elements left out". However, generality may sometimes impair intelligibility, and our aim here has been to make the introduction of the powerdomains as painless as possible.

5. TWELVE VARIANTS OF A NON-DETERMINISTIC LANGUAGE

In this section we give denotational definitions for a variety of possible semantics of the extended language. Section 5.1 covers the singular semantics, Section 5.2 covers the plural semantics, and Section 5.3 discusses the various languages' properties and makes some comparisons.

5.1 Singular semantics

The characterization of the orderings given in Section 4 may be made more concrete now that we have the three proposals for $\mathscr{P}[int_\perp]$. Below we give these characterizations. Also, we define for each case the 'collector' operator ∇ which proves useful in the semantic definitions to come.

Definition 5.1. Let $A \subseteq \mathscr{P}(int_\perp)\backslash\{\varnothing\}$ and define $\nabla A \in \mathscr{P}[int_\perp]$ for each of the three powerdomains as follows:

Discrete Hoare power domain:

$A \sqsubseteq B$ iff $A = \{\perp\} \vee A \subseteq B$

$$\begin{aligned}\nabla A &= \{\perp\} && \text{if } A = \{\{\perp\}\}\\ &= (\textstyle\bigcup A)\backslash\{\perp\} && \text{otherwise}\end{aligned}$$

Discrete Smyth power domain:

$A \sqsubseteq B =$ iff $A' = \{\perp\} \vee B \subseteq A$

$$\begin{aligned}\nabla A &= \{\perp\} && \text{if } \perp \in \textstyle\bigcup A \text{ or } \textstyle\bigcup A \text{ is infinite}\\ &= \textstyle\bigcup A && \text{otherwise}\end{aligned}$$

Discrete Plotkin power domain:

$A \sqsubseteq B$ iff $A = B \vee (\perp \in A \wedge A \subseteq B \cup \{\perp\})$

$$\begin{aligned}\nabla A &= \{\perp\} \cup \textstyle\bigcup A && \text{if } \textstyle\bigcup A \text{ is infinite}\\ &= \textstyle\bigcup A && \text{otherwise}\end{aligned}$$

We write $\nabla\{A, B\}$ as $A \nabla B$. Note that ∇ is commutative, associative, and absorptive in all three cases. Also, even though ∇ is not monotonic with respect to $\sqsubseteq$ in the Smyth and Plotkin cases, ∇ *is* monotonic in all three cases. Furthermore, in the Plotkin case, the result of the ∇ operator is the union of its results in the Hoare and the Smyth cases.

We are now ready to give a semantics for the language of our definition extended with non-deterministic choice. In the singular semantics we have

F: $pgm \to \mathscr{P}[int_\perp]$

E: $exp \to den \to den$ where $den = int_\perp^2 \to \mathscr{P}[int_\perp]$.

The domain *den* of function denotations is equipped with the respective orderings taken pointwise. Note that $\perp_{den} = \lambda w.\ \{\perp\}$ irrespective of the power domain used. The revised semantic functions are given in Definition 5.2.

Definition 5.2. (Singular semantics)

Below, $d \in den$; $n \in int$; $u, v \in int_\perp$; $w \in int_\perp^2$; z_i abbreviates $E\llbracket e_i\rrbracket dw \in \mathscr{P}[int_\perp]$. The operator ∇ was defined previously. The operator $\bar{\nabla}$ is the distributed version of ∇.

$$\begin{array}{lll}
1. & F\llbracket f(x, y) == e\rrbracket = d(0, 0) \text{ where rec } d = E\llbracket e\rrbracket d & \\
2. & E\llbracket n\rrbracket dw & = \{n\}\\
3a. & E\llbracket x\rrbracket dw & = \{u\} \text{ where } (u, v) = w\\
3b. & E\llbracket y\rrbracket dw & = \{v\} \text{ where } (u, v) = w\\
4. & E\llbracket e_1 + e_2\rrbracket dw & = \{\text{if } r_1 = \perp \text{ or } r_2 = \perp \text{ then } \perp\\
 & & \quad \text{else } r_1 + r_2 \mid r_1 \in z_1 \wedge r_2 \in z_2\}\\
5. & E\llbracket e_1 \to e_2, e_3\rrbracket dw & = \{\perp \mid \perp \in z_1\}\\
 & & \quad \nabla\{r_2 \mid r_2 \in z_2 \wedge z_1\backslash\{\perp, 0\} \neq \varnothing\}\\
 & & \quad \nabla\{r_3 \mid r_3 \in z_3 \wedge 0 \in z_1\}\\
6. & E\llbracket f(e_1, e_2)\rrbracket dw & = \bar{\nabla}\{d(u, v) \mid u \in z_1 \wedge v \in z_2\}\\
7. & E\llbracket e \sqcap e_2\rrbracket dw & = z_1 \nabla z_2
\end{array}$$ ■

Annotations to Definition 5.2

1. Here $d \in den$ is the denotation of the (only) function f. The definition gives a non-strict f. As in the deterministic case, a strict f is obtained by replacing the recursive definition of d by

   $d(u, v) =$ if $u = \perp$ or $v = \perp$
   then $\perp$ else $E\llbracket e\rrbracket d(u, v)$.

4. Addition is strict. The semantics takes all possible combinations of sums. In this way $+$ distributes over $\sqcap$, that is,

   $$e_1 + (e_2 \sqcap e_3) = (e_1 + e_2) \sqcap (e_1 + e_3) \qquad (5.1)$$

   $$(e_1 \sqcap e_2) + e_3 = (e_1 + e_3) \sqcap (e_2 + e_3) \qquad (5.2)$$

   Here and below, '$=$' (also) denotes strong equality of programs or expressions. That is, either both sides are undefined, or they are defined and equal. Note that does not distribute over $+$, as can be seen from the example:

   $$\begin{aligned}E\llbracket(1+1) \sqcap 2\rrbracket &= \lambda d.\lambda w.\{2\}\\ &\neq \lambda d.\lambda w.\{2, 3, 4\} = E\llbracket(1 \sqcap 2) + (1 \sqcap 2)\rrbracket.\end{aligned}$$

5. By this, $\sqcap$ distributes over conditions, so

   $$(e_1 \sqcap e_2) \to e_3, e_4 = (e_1 \to e_3, e_4) \sqcap (e_2 \to e_3, e_4). \qquad (5.3)$$

   Furthermore, the following holds in the demonic case:

   $$(e_1 \to e_2, e_3) \sqcap e_4 = e_1 \to (e_2 \sqcap e_4), (e_3 \sqcap e_4). \qquad (5.4)$$

   It does not hold otherwise, witness the case where e_1 evaluates to $\{\perp\}$, while e_4 evaluates to $\{3\}$, say. For example,

   $$f(x, y) == (f(x, y) \to 1, 2) \sqcap 3$$

   does not define the same function as does

   $$f(x, y) == f(x, y) \to (1 \sqcap 3), (2 \sqcap 3)$$

   in the angelic and erratic cases, since the former may yield 3.

6. With singular semantics, a variable always has exactly

one value and this makes function application distribute over $\sqcap$, that is,

$$f(e_1 \sqcap e_2, e_3) = f(e_1, e_3) \sqcap f(e_2, e_3) \qquad (5.5)$$
$$f(e_1, e_2 \sqcap e_3) = f(e_1, e_2) \sqcap f(e_1, e_3). \qquad (5.6)$$

Note that in the Smyth and Plotkin cases, ∇ is applied to finite collections of finite sets only. Thus $E\llbracket e \rrbracket$ is continuous in all cases and F is well-defined.

7. This makes $\sqcap$ commutative, associative, and absorptive. ∎

Note that the semantic definition is incomplete until we state which version of ∇ is meant. Proofs of (5.1)–(5.6) are straightforward.

5.2 Plural semantics

The collection of semantics given by Definition 5.2 are singular: a variable is always bound to a single value, so $x+x$ would always evaluate to an even number. Now we give several plural semantics; with these a variable may be bound to a *set of possible* values and hence $x+x$ may evaluate to 3 in case x is bound to $\{1, 2\}$. For the plural semantics we have

$$F{:}pgm \rightarrow \mathscr{P}[int_\perp]$$
$$E{:}exp \rightarrow den \rightarrow den$$

as before, but now $den = (\mathscr{P}[int_\perp])^2 \rightarrow \mathscr{P}[int_\perp]$ because a variable may be bound to a set of possible values. The semantic functions are given in the following definition.

Definition 5.3. (Plural semantics)

Below, $d \in den$; $u, v \in \mathscr{P}[int_\perp]$; $w \in (\mathscr{P}[int_\perp])^2$; z_i abbreviates $E\llbracket e_i \rrbracket dw \in \mathscr{P}[int_\perp]$. The semantic functions are as shown in Definition 5.2, except:

1. $F\llbracket f(x, y) == e \rrbracket = d(\{0\}, \{0\})$ where $\operatorname{rec} d = E\llbracket e \rrbracket d$

3a. $E\llbracket x \rrbracket dw = u$ where $(u, v) = w$

3b. $E\llbracket y \rrbracket dw = v$ where $(u, v) = w$

6. $E\llbracket f(e_1, e_2) \rrbracket dw = d(z_1, z_2)$ ∎

Annotations to Definition 5.3

1. Both initial arguments are the singleton set $\{0\}$. The definition gives a non-strict function f. A strict f is obtained by replacing the recursive definition of d by

$$d(u, v) = \text{if } u = \{\perp\} \text{ or } v = \{\perp\} \text{ then } \{\perp\}$$
$$\text{else } \; (E\llbracket e \rrbracket d(u, v)) \cup (\{\perp\} \cap (u \cup v)).$$

In the erratic case, this definition is justified as follows: if non-termination of argument evaluation is possible, then non-termination must be a possible result of the function call. If it is the *only* possibility, then it is the only possible result of the function call as well. Note that in the angelic and demonic cases, the 'else' part is just $E\llbracket e \rrbracket d(u, v)$, since neither u nor v contain $\perp$.

3. Variables are now bound to *sets* of values.
4. The equations (5.1) and (5.2) hold in this case too. Still does not distribute over +, since the above counter-example applies.
5. Equation (5.3) still applies, and in the demonic case so does (5.4).
6. The equations (5.5) and (5.6) are no longer valid. For example,

$$f(x, y) == x \rightarrow x+x, f(1 \sqcap 2, y)$$

does not define the same function as does

$$f(x, y) == x \rightarrow x+x, (f(1, y) \sqcap f(2, y)),$$

since the former may yield 3, whereas the latter cannot. ∎

Note that equations 3a, 3b, and 6 are exactly as in the deterministic semantics of Definition 3.2

5.3 Summary of language properties

Sections 5.1 and 5.2 made the existence of several unallied semantic notions apparent. We now discuss and recapitulate these from an operational point of view. We have seen that an important part of defining a non-deterministic language with function application is to fix the semantics along three dimensions:

(1) the usual strict/non-strict dimension of functions known from deterministic languages;
(2) the singular/plural dimension of variables in a non-deterministic language, and
(3) the angelic/demonic/erratic dimension of non-deterministic choice.

First we illustrate each of the dimensions by examples; then we show that the $12 = 2 \cdot 2 \cdot 3$ semantics are all different.

Strict or non-strict functions

- When f is strict, the program

$$f(x, y) == x \rightarrow x, f(1, f(0, 0))$$

fails to terminate because of an attempt to evaluate $f(0, 0)$ infinitely often. When f is non-strict, the program will terminate with result 1 because the value of variable y is not needed when x is non-zero and the argument expression $f(0, 0)$ need not be evaluated.

Singular or plural semantics

- This second dimension is sometimes referred to as *call-time-choice/run-time-choice*[6,7] to indicate at what time the variable's value is fixed. We prefer the shorter singular/plural since in general variables may become bound by other constructs than function calls, such as let clauses, *etc.* With any singular semantics, the program

$$f(x, y) == x \rightarrow x+x, f(1 \sqcap 2, 0) \qquad (5.7)$$

has the set $\{2, 4\}$ of possible results because x must be bound to either 1 or 2. With any plural semantics, the set of possible results is $\{2, 3, 4\}$ since one occurrence of x may evaluate to 1 and the other to 2.

Angelic, demonic, or erratic choice

- With angelic semantics, the program

$$f(x, y) == y \rightarrow f(0, 1), (x \rightarrow x, f(1, 0) \sqcap f(1, 1))$$

will terminate with result 1 because the choice operator will choose the left call $f(1, 0)$ to make the program terminate. With demonic semantics, the right call $f(1, 1)$ will be chosen, resulting in non-termination. With erratic semantics, the program either terminates with result 1 or it fails to terminate.

Items (1), (2) and (3) above span 12 semantics (all compactly given by Definitions 5.2 and 5.3 and their annotations). Interestingly, the 12 semantics *are all different*. This is an important point of this paper: very often we confuse the notions involved, because we try to maintain a too simplistic view of non-deterministic programs. For example it is common to think that the strict *vs.* non-strict distinction is just the same as the singular *vs.* plural distinction. This misunderstanding comes from considering programs such as (5.7) and reasoning that since both 'strict' and 'singular' seem to have some affinity with 'call-by-value'. they must be equivalent. This is not correct, and the problem is that in reasoning like this, we let 'call-by-value' (which is familiar from the more narrow deterministic case) serve as the bed of Procrustes, into which we force the richer notions from the non-deterministic case. We hope to have demonstrated the advantage of a more rigorous mathematical analysis of the notions.

To demonstrate that the 12 semantics are all different, we give three example programs. First, the reader may want to verify that the following program yields 8 different values depending on whether the semantics is strict or non-strict, angelic or erratic, and singular or plural:

$$f(x, y) == y \to f(x, y), (x \to x+x, (f(1 \sqcap 2, 0) \sqcap f(3 \sqcap f(0, 0), f(0, 1)))) \quad (5.8)$$

The eight values are tabulated in Table 2. Yet a ninth value is $\{\perp\}$ which results from any demonic semantics. Notice that in the (non-strict) angelic cases there is an infinite set of possible results, and non-termination is not a possible result. Hence we have an example of unbounded non-determinism. In the (non-strict) erratic cases, non-termination ($\perp$) is an additional possibility in the cases where there is an infinity of possible results.

It is not hard to exemplify the variety of demonic semantics either. To show that they are different, we evaluate the following two programs in the four cases (strict/non-strict and singular/plural):

$$f(x, y) == x \to x+x, f(1 \sqcap 2, f(x, y)) \quad (5.9)$$

$$f(x, y) == x \to x+x, f(1 \sqcap 2, y). \quad (5.10)$$

The program (5.9) denotes $\{2, 4\}$ if non-strict singular, $\{2, 3, 4\}$ if non-strict plural, and $\{\perp\}$ if strict. But also strict singular and strict plural are different, as witnessed by (5.10).

We end this section by categorizing the various language variants according to some important properties. We first define these properties, then summarize the result in Table 3.

A language has the *unfolding property* if a function call $f(e_1, e_2)$ may be replaced by the function's body expression with the argument expressions substituted for the formal parameters, without disturbing the meaning of the enclosing expression. More precisely, let σ (e, e_1, e_2) be the result of substituting expression e_1 for every occurrence of x and expression e_2 for every occurrence of y in expression e. Then the requirement is that

$$E\llbracket f(e_1, e_2)\rrbracket dw = E\llbracket \sigma(e, e_1, e_2)\rrbracket dw \text{ for all } w \in int_\perp^2 \quad (5.11)$$

where $d = E\llbracket e\rrbracket d$ is the denotation of f.

Variables in a language are *definite* if all occurrences of a variable (within its scope) always have the same value. Lack of definiteness implies a slightly unnatural semantics. For example, consider the conditional expression $x \to x$, 1 and assume that (by plural semantics) x is bound to $\{0, 1\}$. The expression then denotes $\{0, 1\}$ although one might feel that $x \to x$, 1 should not yield 0 (recall that the conditional is an 'if non-zero'). With non-strict plural semantics, a language has the unfolding property (as can be verified by proving (5.11) by induction on the structure of e). To see that the property is not shared by any language with *strict* semantics, consider the program

$$f(x, y) == x \to 1, f(1, f(x, y))$$

which has the denotation $\{\perp\}$. Unfolding the *outermost* function call yields

$$f(x, y) == x \to 1, (1 \to 1, f(1, f(1, f(x, y)))),$$

which has the denotation $\{1\}$. To see that all languages with *singular* semantics lack the unfolding property, consider the program

$$f(x, y) == x \to x+x, f(1 \sqcap 2, y)$$

which has the denotation $\{2, 4\}$ by any singular semantics. Unfolding yields

$$f(x, y) == x \to x+x, ((1 \sqcap 2) \to (1 \sqcap 2)+(1 \sqcap 2), f(1 \sqcap 2, y)),$$

which has the denotation $\{2, 3, 4\}$.

Table 2. Results from evaluating (5.8)

	Strict		Non-strict	
	Singular	Plural	Singular	Plural
Angelic	$\{2, 4\}$	$\{2, 3, 4\}$	$\bigcup_{n\in\mathcal{N}} \{2^n, 3\cdot 2^n\}$	$\mathcal{N}\backslash\{1\}$
Erratic	$\{\perp, 2, 4\}$	$\{\perp, 2, 3, 4\}$	$\{\perp\} \cup \bigcup_{n\in\mathcal{N}} \{2^n, 3\cdot 2^n\}$	$\mathcal{N}_\perp\backslash\{1\}$

Table 3. Properties of the various language variants

	Having definite variables	Having indefinite variables
Having the unfolding property	None	Those having non-strict plural semantics
Lacking the unfolding property	Those having singular semantics	Those having strict plural semantics

Our language has *referential transparency* by all 12 semantics, at least in Quine's sense, since no construct destroys referentiality.[14] A quoting operator such as *QUOTE* in Lisp or a string constructor such as '...' in Pascal would destroy referentiality. We note, however, that just like the notions discussed in this paper, that of referential transparency has caused much confusion. We discuss referential transparency and its relation to similar but distinct notions elsewhere.[18]

6. CONCLUSION

We have presented some concepts related to non-determinism in programming languages, exemplified by a small non-deterministic functional language. In particular, we have demonstrated the various possible mechanisms for parameter passing and some possible interpretations of a non-deterministic choice operator. These possibilities have all been formally described (using denotational semantics) to allow for a precise discussion of language properties, and we have shown that no less than 12 different semantics are possible, spanned by the 3 dimensions:

(1) strict/non-strict functions;
(2) definite/indefinite variables, and
(3) angelic/demonic/erratic choice.

This richness is not often recognized and as a result there is generally much confusion about non-deterministic languages. This is understandable; the unexpectedly large number of combinations shows that the introduction of a non-deterministic choice operator into a simple functional language complicates its semantics considerably. Seemingly minor changes in the interpretation of non-deterministic choice and call mechanisms have great impact on the properties of the resulting language.

The formal discussion of the non-deterministic language by denotational semantics required us to deal with sets of possible results of a computation instead of a single result. Therefore we presented the standard so-called power domains; these are mathematical structures intended to describe such sets of possible results. We related the three well-known kinds of power domain, the Hoare, Smyth, and Plotkin power domains to the three possible interpretations of a non-deterministic choice operator (dimension number (3) above).

Finally we summarized the various semantics, and the properties with which they endow the language were discussed. We have not formally discussed locally angelic or locally demonic choice which were informally introduced in Section 2.3. A (rather complicated) proposal for a formal semantics of a language with an erratic and a locally angelic choice operator has been given by Broy.[2]

Several other aspects of non-determinism have not been treated formally here. For example, we have discussed only restrained non-determinism: choice among basic values (numbers), so that the particularly simple so-called discrete power domain constructions would do. Modelling unrestrained choice (choice between functional values, say) poses some delicate problems. Hence we have not discussed general power domains either. The original papers on power domains are by Plotkin[15] and Smyth;[17] an introduction to (discrete and non-discrete) power domains can be found in Main's tutorial paper.[12]

ACKNOWLEDGEMENTS

A number of people have contributed to our understanding and have commented on earlier versions of the paper, notably Manfred Broy, Olivier Danvy, Hans Dybkjær, Fritz Henglein, Neil D. Jones, and Kim Marriott.

REFERENCES

1. K. Apt and G. D. Plotkin, Countable non determinism and random assignment. *Journal of the ACM* **33** (4), 724–767 (1986).
2. M. Broy, A theory for non determinism, parallelism, communication, and concurrency. *Theoretical Computer Science* **45** (1), 1–61 (1986).
3. W. Clinger, Non deterministic call by need is neither lazy nor by name. *Proc. 1982 ACM Symp. LISP and Functional Programming*, Pittsburgh, Pennsylvania, August 1982, pp. 226–234 (1982).
4. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs (1976).
5. R. W. Floyd, Non-deterministic algorithms. *Journal of the ACM* **14** (4), 636–644 (1967).
6. M. C. B. Hennessy, Power domains and non deterministic recursive definitions. In M. Dezani-Ciancaglini and U. Montanari (eds.): *International Symposium on Programming, Turin, Italy, 1982. Springer Lecture Notes in Computer Science* **137**, 178–193 (1982).
7. M. C. B. Hennessy and E. A. Ashcroft, Parameter-passing mechanisms and non determinism. *Proc. 9th ACM Symp. Theory of Computing*, Boulder, Colorado, May 1977, pp. 306–311 (1977).
8. C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (1985).
9. P. Johansen, Non-deterministic programming. *BIT* **7**, 289–304 (1967).
10. D. L. Kreider and R. W. Ritchie, Predictably computable functionals and definition by recursion. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* **10**, 65–80 (1964).
11. J. McCarthy, A basis for a mathematical theory of computation. In P. Brafford and D. Hirschberg (eds.): *Computer Programming and Formal Systems*. North-Holland, pp. 33–70 (1963).
12. M. G. Main, A powerdomain primer. *Bulletin of the EATCS* **33**, 115–147 (1987).
13. R. Milner, *A Calculus of Communicating Systems. Springer Lecture Notes in Computer Science* **92** (1980).
14. W. V. O. Quine, *Word and Object*. MIT Press, Cambridge Mass. (1960).
15. G. D. Plotkin, A powerdomain construction. *SIAM Journal on Computing* **5** (3), 452–487 (1976).
16. M. O. Rabin and D. Scott, Finite automata and their decision problems. *IBM Journal of Research* **3** (2), 115–125 (1959).
17. M. B. Smyth, Power domains. *Journal of Computer and System Sciences* **16** (1), 23–26 (1978).
18. H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica* **27**, 505–517 (1990).
19. M. H. van Emerden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM* **23** (4), 733–742 (1976).
20. G. Winskel. On powerdomains and modality. *Theoretical Computer Science* **36**, 127–137 (1985).