Dually Nondeterministic Functions

JOSEPH M. MORRIS and MALCOLM TYRRELL

Dublin City University, Ireland and Lero - the Irish Software Engineering Research Centre

Nondeterminacy is a fundamental notion in computing. We show that it can be described by a general theory that accounts for it in the form in which it occurs in many programming contexts, among them specifications, competing agents, data refinement, abstract interpretation, imperative programming, process algebras, and recursion theory. Underpinning these applications is a theory of nondeterministic functions; we construct such a theory. The theory consists of an algebra with which practitioners can reason about nondeterministic functions, and a denotational model to establish the soundness of the theory. The model is based on the idea of free completely distributive lattices over partially ordered sets. We deduce the important properties of nondeterministic functions.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and verifying and reasoning about programs—*Logics of programs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Angelic nondeterminacy, demonic nondeterminacy, free completely distributive lattice, modeling nondeterminacy, nondeterminism, nondeterministic functions

ACM Reference Format:

Morris, J. M. and Tyrrell, M. 2008. Dually nondeterministic functions. ACM Trans. Program. Lang. Syst. 30, 6, Article 34 (October 2008), 34 pages. DOI = 10.1145/1391956.1391961 http://doi.acm.org/10.1145/1391956.1391961

1. INTRODUCTION

Here is a simple nondeterministic mechanism (written in the notation of *guarded commands* [Dijkstra 1976]):

if $x \geq 0 \rightarrow$ output $\circ$
$\Box\ x \leq 0 \rightarrow$ output $\star$
fi

This work was supported by Science Foundation Ireland, under Grant No. 03/IN_3I408C.
Authors' address: School of Computing, Dublin City University, Dublin 9, Ireland; email: joseph.morris@computing.duc.ie.

It exhibits nondeterministic behavior when x has the value 0, because in that case either $\circ$ or $\star$ can be output. Nondeterminacy can also be used to capture underspecification, as in

"Given a name n and a phone book b output a phone number for n"

This is underspecified in that it leaves open the phone number to be output when the person named has several phones. In concurrent systems, nondeterminacy may arise from the freedom given to the scheduler. For example, the system

output $\circ$ $\parallel$ output $\star$

outputs one of $\circ\star$ or $\star\circ$ depending on the relative speeds of the constituent processes.

These examples, and others to follow, illustrate that nondeterminacy occurs commonly in computing. Indeed, we see it as a fundamental notion, deserving a place alongside other fundamental notions such as recursive functions, data types, concurrency, objects, etc. As we shall show later, essentially the same notion of nondeterminacy manifests itself in a range of different contexts, among them imperative, functional, and concurrent programming, competing agents, data refinement, and fixpoint theory. We present a "plug-and-play" theory of nondeterminacy that fits all these contexts, and outline how it does so. Most significantly, we plug the theory into the notion of function to arrive at a theory of nondeterministic functions.

To motivate the work, we offer an analogy with recursion as it occurs in programming. Consider *recursion*: At the language level, programmers reason about recursion using two logical rules. The rules are typically called the *unfolding* rule, and the *induction* rule, but the details need not concern us here. *Nondeterminacy*: Analogously, our theory provides programmers with a set of reasoning rules. Programmers may use the rules informally to understand nondeterminacy and gain confidence in their use of it, or they may use them to verify their code formally.

Recursion: The rules of recursion are justified "under the hood" by a mathematical theory that guarantees that the rules are sound. The theory is that of *least fixpoints on complete partial orderings*, but again the details of this do not matter here. *Nondeterminacy*: Similarly, we will justify the rules of nondeterminacy by a theory, in this case that of *free completely distributed lattices over a poset*. This theory is under the hood in the sense that the practicing programmer need know nothing about it.

Recursion: The basic theory of recursion is applicable across most programming paradigms, such as imperative, functional, object-oriented, concurrent, etc. In this sense, the theory is fundamental. In each case, however, it has to be "wired in" to the host language, where wiring in entails some additional work such as fixing the ordering by which *least* in *least fixpoint* is given meaning. *Nondeterminacy*: Similarly, our theory accounts for nondeterminacy in many paradigms, with some local wiring in to the host language.

Recursion: Formally speaking, recursion is defined in terms of two operators, μ and ν corresponding to least and greatest fixpoints, respectively. *Nondeterminacy*: Analogously, we employ $\sqcap$ and $\bigsqcup$, each being a nondeterministic

choice operator. For example, the phone number look-up problem described above might be expressed as

$$\lambda n\colon Name, b\colon PhoneBook \cdot \sqcap\{x\colon PhoneNumber \mid (n, x) \in b \cdot x\}$$

$\{x\colon PhoneNumber \mid (n, x) \in b \cdot x\}$ denotes the set of x's (the final $\cdot x$) satisfying $(n, x) \in b$ where x is a new variable of type *PhoneNumber* (introduced by the initial x:*PhoneNumber*). $\sqcap S$, where S stands for any set whose elements are drawn from some type T, say, is itself a term of type T. It denotes *some* element of S nondeterministically chosen. The element of S resulting from an evaluation of $\sqcap S$ is not necessarily constant. For example, if an evaluation of $\sqcap\{2, 3, 4\}$ yields 3, say, it may nevertheless be the case that it will yield 4, say, the next time it is encountered.

$\sqcup S$ also represents a nondeterministic choice over set S. Our theory supports two choice operators because, in some contexts, we may need to capture the fact that choices are exercised by different agents. For example, when a client interacts with a server, the choices made by the client may need to be distinguished from those made by the server.

Note that our choice operators operate on sets of terms, and that $\sqcap S$ and $\sqcup S$ have themselves the status of terms. We have elected to root our work in the theory of terms because that is where the greater challenge and greater applicability lies. By *term* here, we mean simply an expression without side-effects as typically found in most imperative and functional programming and specification languages. We expect that nondeterminacy in terms will translate relatively straightforwardly to nondeterminacy in other domains. For example, for nondeterminacy in commands, a choice such as $x := 0 \sqcap x := 1$ can be written using term-level choice as $x := 0 \sqcap 1$ (we write $\sqcap$ and $\sqcup$ for the binary infix versions of $\sqcap$ and $\sqcup$, respectively). We will elaborate on this later.

The first part of our program is to explain the choice operators and show that they account for a wide range of uses of nondeterminacy in different contexts. We range over, among others, functions, program specifications, imperative programming, competing agents, data refinement, and process algebras. Next, we present the fundamental laws of nondeterminacy. This has been previously presented in Morris [2004], but we reprise it here for the sake of completeness. Our primary contribution is to plug the fundamental theory into the theory of functions to arrive at a theory of nondeterministic functions, and to describe the important properties of functions in the theory. The final part of the enterprise is to establish that our theory is sound.

The proof that our theory is sound is technically challenging. Our approach is to build a denotational model using the theory of free completely distributive lattices over a poset, and to show that all our laws are true in the model.

$\sqcap$ represents choices made by an agent that is classically referred to as the *demon*, while $\sqcup$ represents the choices made by an agent that is classically referred to as the *angel*. Correspondingly, we refer to the nondeterminacy represented by $\sqcap$ as *demonic*, and the nondeterminacy represented by $\sqcup$ as *angelic*.

We add a word of explanation for readers who have some experience of nondeterminacy. In some languages, angelic nondeterminacy is benign in some way, such as being failure-avoiding, whereas demonic nondeterminacy is not. We caution that, in the basic theory we present, the two choices are neutral in this regard, neither being either good or bad. Demonic and angelic nondeterminacy are simply mathematical duals of one another representing choices made by two different agents. It may be the case that when the theory is wired into some particular host language, the designer chooses to wire in some badness or goodness, but the names *angelic* and *demonic* should not mislead the reader into ascribing goodness or badness to them per se. In particular, angelic nondeterminacy in some programming languages is associated with a backtracking implementation, but backtracking is not inherent in angelic nondeterminacy. We will return to this point later (Section 2.2.2).

1.1 Outline of the Article

MOTIVATION AND CONTEXT

Section 1. Introduction and reader's guide.

Section 2. We explain our notion of dual nondeterminacy and show that it has applicability in many areas of programming.

THE ALGEBRA OF NONDETERMINACY

Section 3. We formally explain our notation and terminology, and present the basic axioms of nondeterminacy.

Section 4. We show how to plug the basic theory into functions, arriving at a theory of nondeterministic functions.

MODELING NONDETERMINACY

Section 5. We describe a denotational model for nondeterminacy based on the idea of a free completely distributive lattice over a poset.

Section 6. We show that our theory of nondeterminacy and nondeterministic functions is sound.

REVIEW

Section 7. The final section summarises the work and surveys the literature. The primary contributions of the work are as follows

—A case for treating nondeterminacy as a fundamental notion, capable of being studied in isolation.

—A theory of nondeterministic functions, and a set of properties enjoyed by nondeterministic functions.

—A denotational model based on free completely distributed lattices by which we can show that our theory of nondeterministic functions is sound.

Reader's Guide. Readers with little prior exposure to nondeterminacy may get additional motivation by an early reading of Section 7.2, which summarizes previous approaches to nondeterminacy in programming. Additionally, some readers with a background in formal semantics may opt to make an early pass through Sections 5 and 6 on the underlying model, perhaps in parallel with the

introductory material of Section 2. On the other hand, readers whose primary interest is in applications of nondeterminacy may choose to pass lightly over Sections 5 and 6 when they come to them (at least the proofs can be omitted). Not all readers will have the same degree of interest in every application area or example; in that case, it is possible to omit some of the examples in Section 2 without losing the thread of the argument.

2. UNIVERSALITY OF NONDETERMINACY

Our aim is to make a theory of nondeterminacy that has wide applicability. In this section, we range over a broad selection of programming contexts, showing that the same notion of nondeterminacy plays a role in many disparate contexts. We concentrate primarily on the role of nondeterministic functions in these various contexts.

2.1 Functions

2.1.1 *Higher-Order Functions in Specifications.* There are many generic higher-order functions that can be used to construct specifications. They may employ nondeterminacy to capture "don't care" behavior. Consider, for example, the function

$$leastWRT_T : (T \to \mathbb{Z}) \to I\!P\, T \to T,$$

which takes as arguments a function f and a set s, and selects some element x of s such that $f\ x$ is minimal in $f\ s$ ($I\!P$ is the powerset operator). We omit the type subscript in what follows. Formally, for any type T,

$$leastWRT \triangleq \lambda f{:}T \to \mathbb{Z} \cdot \lambda s{:}I\!P\, T \cdot \bigsqcap \{x \in s \mid (\forall y \in s \cdot f\ x \leq f\ y) \cdot x\}.$$

To illustrate its use, we make a function that yields a place to which two people should travel if they wish to meet as soon as possible. We assume a type *Place* whose elements represent all places of interest, and a function $time : Place \times Place \to \mathbb{N}$ which yields the traveling time (in minutes, say) between any two places. The function we want is

$$\lambda me, you{:}\ Place \cdot leastWRT\ (\lambda c{:}\ Place \cdot time(me, c)\ \mathsf{max}\ time(you, c))\ Place.$$

Here, *leastWRT* has type $(Place \to \mathbb{Z}) \to I\!P\, Place \to Place$, $(\lambda c{:}Place \cdots)$ has type $Place \to \mathbb{Z}$, and *Place* has type $I\!P\, Place$, so the result is an element of *Place*. Observe the presence of nondeterminacy in that the function may yield any acceptable place when more than one meet the criteria. It is important for specifications to retain nondeterminacy inherent in the client's requirements, leaving its resolution to a later stage in the programming process. This gives maximum freedom to the implementor.

2.1.2 *Pattern Matching in Formal Parameters.* Functional programs may make use of pattern matching in formal parameters. For example, a function f may be defined on nonempty sequences of integers by $f(v : vs) \triangleq v + 1$ (here, v stands for an integer, vs for an integer sequence, and : for sequence prefixing). Invoking $f\ \langle 2, 5, 6\rangle$, for example, yields 3 because $\langle 2, 5, 6\rangle$ is pattern matched as

$2 : \langle 5, 6\rangle$ and so v is bound to 2 in the body of f. If nondeterminacy is available the patterns need not be injective, that is, they need not be constrained to give rise to unique decompositions. For example, the following function describes the (coincidentally named) pattern matching problem, that is, the problem of finding an occurrence of one string in another, or determining that there are no such occurrences.

$$\begin{array}{l} patmatch : \mathsf{seq}\,\mathbb{C} \to (\mathsf{seq}\,\mathbb{C} \to \mathbb{Z}) \\ patmatch\ ps\ (ys +\!\!+ ps +\!\!+ zs) \ \triangleq\ \#ys \\ patmatch\ \_\ \_ \ \triangleq\ -1 \end{array}$$

$\mathsf{seq}\,\mathbb{C}$ describes the type of sequences of characters, $\#ys$ denotes the length of sequence ys, and $+\!\!+$ is sequence concatenation. If in an invocation of $patmatch\,ps\,xs$, ps occurs in xs then it is covered by the first defining clause and some index of ps in xs is returned. Otherwise (and only otherwise), the second defining clause is appealed to (always successfully) and -1 is returned. It is easy to express *patmatch* without patterns by employing demonic choice. We shall do so in a particularly neat way by making use of the operator $\overleftarrow{\sqcap}$.

First, we introduce $\top_T$ (pronounced *top*) as an abbreviation for $\bigsqcap \emptyset_T$ where $\emptyset_T$ denotes the empty set of type T. We commonly omit the type subscript in $\top_T$. We will show later that $\top$ is the unit of $\sqcap$. For t and u terms of the same type, $t \overleftarrow{\sqcap} u$ is defined to be t except in the case that t is $\top$ in which case $t \overleftarrow{\sqcap} u$ is defined to be u. *patmatch* is equivalent to

$$\lambda ps : \mathsf{seq}\,\mathbb{C} \cdot \lambda xs : \mathsf{seq}\,\mathbb{C} \cdot \bigsqcap \{ys, zs : \mathsf{seq}\,\mathbb{C} \mid xs = ys +\!\!+ ps +\!\!+ zs \cdot \#ys\} \overleftarrow{\sqcap} -1.$$

A Note on Set Notation. We write $\{x{:}T,\ y{:}U \mid P \cdot t\}$ to denote the set of t's for each x of type T and y of type U satisfying predicate P (x and y typically occur free in P and t, of course, and there may be other than two dummy variables). For example, $\bigsqcap\{x{:}\mathbb{Z} \mid 0 \le x < 3 \cdot x \sqcup 5\}$ is equivalent to $(0 \sqcup 5) \sqcap (1 \sqcup 5) \sqcap (2 \sqcup 5)$. $\{x{:}T \mid P \cdot t\}$ is more traditionally written as $\{t \mid x \in T \wedge P\}$, but this is not adequate for our purposes because it is inherently ambiguous. The ambiguity arises from the absence of information regarding dummy variables; see Gries and Schneider [1993] and Boute [2005]. In informal presentations, there is usually sufficient accompanying text to resolve any ambiguity, but we have to be more formally precise in the context of specification languages. Our notation is widely used to avoid ambiguity, for example, in the *Z* specification language [Spivey 1988], and in the textbooks [Gries and Schneider 1993; Woodcock and Loomes 1988; Morgan 1990]. We use the following abbreviations. First, we write $\{x{:}T \cdot t\}$ in place of $\{x{:}T \mid true \cdot t\}$ (this is the same as the traditional $\{t \mid x \in T\}$ if we discard the information on dummies). Second, instead of $\{x{:}T \mid x \in s \wedge P \cdot t\}$, we may write $\{x \in s \mid P \cdot t\}$, for s some set, and similarly with $\subseteq$ in place of $\in$. These abbreviations may be combined in obvious ways, such as $\{x \in s \cdot t\}$ (which ignoring information on dummies is equivalent to the traditional $\{t \mid x \in s\}$). We suggest that readers more used to the traditional notation might attune themselves by initially reading set comprehensions right-to-left.

2.1.3 *Function Refinement.* Consider the specification *fac0* defined as follows:

$$fac0 \triangleq \sqcap\{f{:}\mathbb{Z}\to\mathbb{Z} \mid f\ 0 = 1 \wedge (\forall n{:}\mathbb{N} \cdot f(n+1) = (n+1) * f\ n) \cdot f\}.$$

It should be intuitively plausible that the following function is an acceptable implementation of *fac0*:

$$fac \triangleq \lambda n{:}\mathbb{Z} \cdot \ \textbf{if}\ n < 0\ \textbf{then} - 47\ \textbf{else}\ \textit{code that computes}\ n!\ \textbf{fi}.$$

The assertion that *fac0* is implemented by *fac* is expressed formally as $fac0 \sqsubseteq fac$, where $\sqsubseteq$ is a partial ordering of terms called the *refinement ordering*. To assert that $fac0 \sqsubseteq fac$ is to assert that the input-output behavior of *fac* is consistent with that of *fac0*, where "consistent" means that every behavior of *fac* is a possible behavior of *fac0*. *fac* need not exhibit all the possible behaviors of *fac0* (and it doesn't). Another way to express this is that *fac0* is like *fac*, except that *fac* may offer less demonic nondeterminacy than *fac0*. We expect that the theory of nondeterminacy should suffice to prove formally that $fac0 \sqsubseteq fac$ holds.

2.2 Competing Agents

2.2.1 *Game Playing: Nim.* The two kinds of nondeterminacy may be used to capture the opposing views of an interaction between competing agents. We illustrate with the game of *Nim*, which is a game played by two players who alternately remove from 1 to 4 matches from a pile until none remain. The player who removes the last match loses. Let us refer to the players as the *home* and *away* player, respectively, where the home player goes first. A single move made by each player is represented by the functions *moveH* and *moveA*, respectively:

$$moveH, moveA : \mathbb{N}\to\mathbb{N}.$$

moveH n yields the number of matches remaining after the home player has made one move when offered *n* matches, and analogously for *moveA n*. We introduce the two-value type *Player* with elements *Home* and *Away*. The complete game played by each player is represented by functions *playH* and *playA*, respectively:

$$playH, playA : \mathbb{N}\to Player.$$

playH n yields the winner of a game in which the home player is initially offered *n* matches, and analogously for *playA n*. Formally:

$$\begin{aligned} playH \ \triangleq\ \lambda n{:}\mathbb{N} \cdot\ & \textbf{if}\ n = 0\ \textbf{then}\ Home \\ & \textbf{else}\ (playA \circ moveH)\, n \\ & \textbf{fi} \end{aligned}$$

$$\begin{aligned} playA \ \triangleq\ \lambda n{:}\mathbb{N} \cdot\ & \textbf{if}\ n = \textbf{0}\ \textbf{then}\ Away \\ & \textbf{else}\ (playH \circ moveA)\, n \\ & \textbf{fi} \end{aligned}$$

We will describe *moveH* and *moveA* with the home player's options represented by angelic choice, and the away player's represented by demonic choice:

$$moveH \triangleq \lambda n{:}\mathbb{N} \cdot \bigsqcup\{m{:}\mathbb{N} \mid 0 < n - m \leq 4\}$$

$$moveA \triangleq \lambda n{:}\mathbb{N} \cdot \bigsqcap\{m{:}\mathbb{N} \mid 0 < n - m \leq 4\}.$$

In Section 2.1.3, we indicated that term t is refined by term u, written $t \sqsubseteq u$, if u is like t except that u offers possibly less demonic nondeterminacy than t. Actually, it is a little more complex than this: $t \sqsubseteq u$ holds if u is like t except that u may offer less demonic nondeterminacy than t, *and* t may offer less angelic nondeterminacy than u. For example, all of the following hold: $1 \sqcap 2 \sqsubseteq 1, 1 \sqsubseteq 1 \sqcup 2$, and $(1 \sqcap 2) \sqcup 3 \sqsubseteq 1 \sqcup 3 \sqcup 4$ (as $(1 \sqcap 2) \sqcup 3 \sqsubseteq 1 \sqcup 3$ and $1 \sqcup 3 \sqsubseteq 1 \sqcup 3 \sqcup 4$). This extends readily to functions in the obvious pointwise way, e.g. $(\lambda x{:}\mathbb{N} \cdot x \sqcap x^2) \sqsubseteq (\lambda x{:}\mathbb{N} \cdot x)$. We will be formally precise about this later.

There is a winning strategy for the opening player in *Nim* with n matches initially (i.e., when $n > 0$), iff the home player as we have coded it can always win. This is captured formally by $Home \sqsubseteq playH\ n$. We write $Home \sqsubseteq playH\ n$ rather than $Home = playH\ n$ or $playH\ n \sqsubseteq Home$ because *playH* yields an angelic choice of possibilities of which only one need lead to $Home$. This example is interesting because it employs both kinds of nondeterminacy to express a property ("there is a winning strategy for the opening player in *Nim*") that initially would seem to have no connection with nondeterminacy.

It is easily seen that the arguments of *playH* and *playA* exhibit nondeterminacy (observe, e.g., that in the body of *playH*, *playA* is applied to $moveH\ n$). Our theory accommodates this by defining function application to distribute over choice.

2.2.2 *Protocols: Cake Cutting.* In this example, we will associate goodness with angelic nondeterminacy, and badness with demonic nondeterminacy. Each party in the competitive engagement to be described sees their opponent's actions as demonic, and their own as angelic.

Consider the classic protocol for dividing a cake fairly between two people: one person cuts and the other picks. Let us represent the cake by the set of reals from 0 to 1, upper bound excluded, that is, $[0..1)$ using interval notation. The cutter cuts by selecting an r in $[0..1]$, and the picker then selects either the left slice $[0..r)$ or the right slice $[r..1)$.

Each party places a value on slices of the cake, values being drawn from the type $\mathbb{V}$. We will take $\mathbb{V}$ to be $[0..1]$. For $0 \leq r \leq s \leq 1$, the cutter values the slice represented by $[r..s)$ as $v_c[r..s)$, and the picker values it at $v_p[r..s)$. We assume v_c is additive in the sense that $v_c[0..r) + v_c[r..1) = 1$, and continuous in the sense that for all x in $[0..1]$ we have $v_c[0..r) = x$ for some r in $[0..1]$. We make the same assumptions about v_p. The two parties regard the protocol as fair if each of $v_c\ slice_c \geq 0.5$ and $v_p\ slice_p \geq 0.5$ hold, where $slice_c$ and $slice_p$ denote the respective slices of the cutter and picker.

Now let us analyze the protocol, initially from the cutter's point of view. The cutter sees his/her slice of the cake as

$$slice_c \triangleq \bigsqcup\{r \in [0..1] \cdot [0..r) \sqcap [r..1)\}.$$

Observe that the cutter sees himself/herself as the angel, and the picker as the demon. We want to formally make the association between angelic choice and goodness on the one hand, and demonic choice and badness on the other. To that end, we extend v_c and v_p in a certain way so that they apply to nondeterministic slices.

It is usually the case that refinement coincides with equality for the elementary values in base types (e.g., $1 \sqsubseteq 1$ holds but not $1 \sqsubseteq 2$). This order is called the *discrete order*. For the present example, however, it is appropriate to adopt $\leq$ as the refinement order $\sqsubseteq_{\mathbb{V}}$ on $\mathbb{V}$. We offer the following intuitive guide as to why this imparts goodness and badness to angelic and demonic choice, respectively. For simplicity, we confine the intuitive guide to finite choice. We will show later that refinement and finite choice satisfy the classic lattice-theoretic relationship $t \sqsubseteq u \Leftrightarrow t \sqcap u = t \Leftrightarrow t \sqcup u = u$ (Theorems 13 and 14 in Section 3.3). From this, it follows that $\bigsqcap$ in $\mathbb{V}$ represents the minimum operator on reals, and $\bigsqcup$ represents the maximum operator. Now consider, for example, how the cutter values the slice $[0..0.2) \sqcap [0..0.7)$: $v_c([0..0.2) \sqcap [0..0.7))$, by distribution equals $v_c[0..0.2) \sqcap v_c[0..0.7)$, which by the argument above equals $v_c[0..0.2) \min v_c[0..0.7)$ — and indeed this is precisely the worst outcome, just as we should expect when the decision is left to the demon.

Returning to the example, let us show that the protocol is fair from the cutter's point of view (the proof anticipates some laws that are not presented until Section 3.3, and the reader is asked to accept them on trust till then).

$$
\begin{array}{ll}
 & v_c\ slice_c \geq 0.5 \\
\Leftrightarrow & \quad \text{“property of } \leq, \geq \text{; definition of } \sqsubseteq_{\mathbb{V}} \text{ (subscript omitted below)”} \\
 & 0.5 \sqsubseteq v_c\ slice_c \\
\Leftrightarrow & \quad \text{“function application distributes over choice”} \\
 & 0.5 \sqsubseteq \bigsqcup\{r \in [0..1] \cdot v_c[0..r) \sqcap v_c[r..1)\} \\
\Leftrightarrow & \quad \text{“nondeterminacy (Axiom A4 and Theorem 3 in Section 3.3)”} \\
 & (\exists r \in [0..1] \cdot 0.5 \sqsubseteq (v_c[0..r) \sqcap v_c[r..1)) \\
\Leftrightarrow & \quad \text{“nondeterminacy (Theorem 9 in Section 3.3)”} \\
 & (\exists r \in [0..1] \cdot (0.5 \sqsubseteq v_c[0..r)) \wedge (0.5 \sqsubseteq v_c[r..1))) \\
\Leftrightarrow & \quad \text{“definition of } \sqsubseteq_{\mathbb{V}} \text{”} \\
 & (\exists r \in [0..1] \cdot (0.5 \leq v_c[0..r)) \wedge (0.5 \leq v_c[r..1))) \\
\Leftrightarrow & \quad \text{“}v_c \text{ additive”} \\
 & (\exists r \in [0..1] \cdot (0.5 \leq v_c[0..r)) \wedge (v_c[0..r) \leq 0.5)) \\
\Leftrightarrow & \quad \text{“}\leq \text{ antisymmetric”} \\
 & (\exists r \in [0..1] \cdot v_c[0..r) = 0.5) \\
\Leftrightarrow & \quad \text{“continuity property of } v_c \text{”} \\
 & \mathit{true}
\end{array}
$$

Now we turn to the picker, who, of course, sees his/her slice of the cake as

$$slice_p \triangleq \bigsqcap\{r \in [0..1] \cdot [0..r) \sqcup [r..1)\}.$$

The proof that $v_p\ slice_p \geq 0.5$ is similar to the preceding proof, except that we do not need to appeal to the continuity property of v_p. We omit it for brevity. In summary, the cake-cutting protocol is fair.

The example illustrates an important point. The basic theory associates no goodness or badness with angelic and demonic nondeterminacy, respectively. Neither does goodness or badness necessarily emerge when the theory is plugged into a host language. However, we are free to introduce goodness or badness when plugging in the theory, and we do so by imposing certain nondiscrete refinement orders on one or more base types, as we have done above.

2.3 Data Refinement and Abstract Interpretation

Angelic and demonic choice can be used to make function adjoints: demonic choice for a left adjoint and angelic choice for a right adjoint. For example, let function *sqr* be the squaring function on the integers, that is, $\lambda x{:}\mathbb{Z}\cdot x^2$. We define

$$sqrt^{\sqcap} \triangleq \lambda z{:}\mathbb{Z}\cdot \bigsqcap\{x{:}\mathbb{Z} \mid sqr\ x = z\cdot x\}$$
$$sqrt^{\sqcup} \triangleq \lambda z{:}\mathbb{Z}\cdot \bigsqcup\{x{:}\mathbb{Z} \mid sqr\ x = z\cdot x\}.$$

$sqrt^{\sqcap}$ is the left adjoint of *sqr* with respect to $\sqsubseteq$, and $sqrt^{\sqcup}$ is the right adjoint, that is, they satisfy

$$sqrt^{\sqcap} \circ sqr \sqsubseteq \mathsf{Id}_{\mathbb{Z}} \sqsubseteq sqr \circ sqrt^{\sqcap}$$
$$sqr \circ sqrt^{\sqcup} \sqsubseteq \mathsf{Id}_{\mathbb{Z}} \sqsubseteq sqrt^{\sqcup} \circ sqr,$$

where Id stands for the identity function on the type identified in its subscript. It is exceptional for functions to have a left or right adjoint. With the introduction of nondeterminacy, however, large classes of λ-abstractions have both a left and a right adjoint. This has useful applications, not least in programming by data refinement [Hoare et al. 1987; Morgan and Gardiner 1991; von Wright 1994; DeRoever and Engelhardt 1999].

For example, the following shows an extract from an abstract program (on the left) and its concrete equivalent (on the right):

$$\ldots\ s : \mathsf{set}\ \mathbb{Z};\ \ s := \emptyset\ \ldots \qquad \ldots\ b : \mathbb{Z}[0..99];\ \ n : \mathbb{Z} := 0;\ \ldots$$
$$\ldots\ s := s \cup \{x\};\ \ldots \qquad \ldots\ b[n\,] := x;\ \ n := n+1;\ \ldots$$

The integer set s is implemented as an array b of 100 elements (assume we know this is sufficiently big) using an integer variable n to record the number of significant elements in b. The abstraction function F relates concrete values to abstract values, that is, it is of type $(\mathbb{Z}[0..99] \times \mathbb{Z}) \to \mathsf{set}\ \mathbb{Z}$. F is defined by $F(b,n) = \{k \in \{0..n-1\}\cdot b[k\,]\}$. Now an important question is how a function f in the abstract program translates to the concrete domain (think of $\lambda t{:}\mathsf{set}\ \mathbb{Z}\cdot t \setminus \mathbb{N}$, for example). It turns out that the concrete version of f is $F^{\mathsf{L}} \circ f \circ F$ where F^{L} denotes the left adjoint of F. Nondeterminacy is crucial here, for without it, F^{L} is not, in general, well defined. Occasionally, the abstract and concrete spaces are related by a function G from the abstract to the concrete; in that case, the concrete version of function f on the abstract space is $G \circ f \circ G^{\mathsf{R}}$ where G^{R} denotes the right adjoint of G.

Abstract interpretation [Cousot 1996] is a theory that underlies several techniques used in analyzing and implementing programs, among them code optimization in compilers, program transformation as in partial evaluation, and proofs of termination. It is the dual of data refinement: it abstracts from the

concrete program to yield a more abstract version more amenable to automatic analysis (at a price of some information loss). Our theory has applications in abstract interpretation not only because it provides support for abstraction functions, but because the abstract programs that arise may in general be nondeterministic.

2.4 Imperative Specifications and Programs

We may introduce nondeterminacy into an imperative language via its term language. To introduce it at the command level, we introduce a mapping from nondeterministic commands to standard commands that have nondeterministic terms. For example, we define $x := 0 \sqcap x := 1$ to be equivalent to $x := 0 \sqcap 1$. For this to be fully formal, it must be the case that the formal semantics of commands accommodates nondeterministic terms.

Commands are formally defined using *weakest precondition* semantics [Dijkstra 1976]. For example, the weakest precondition semantics of the assignment statement $x := t$ is $wp(x{:=}t, R) \triangleq (t \neq \bot) \wedge R[x\backslash t]$ where t stands for a term, R stands for an assertion (a Boolean term possibly using a richer language than that of the programming language), $u[x\backslash t]$ denotes term u with each free occurrence of x replaced with t (the usual caveat about avoiding variable capture applies), and $\bot$ stands for the undefined term of appropriate type. If nondeterminacy may occur in t, then $(t \neq \bot) \wedge R[x\backslash t]$ doesn't make sense and we have to formulate weakest precondition semantics more generally. In the case of assignment, it turns out that a suitable definition is $wp(x{:=}t, R) \triangleq true \sqsubseteq (\lambda x{:}T \cdot R)t$ where T stands for the type of x. (It might seem at first sight that $(\lambda x{:}T \cdot R)t$ might suffice as the righthand side of the foregoing, but that is not so because it may be nondeterministic. The role of "$true \sqsubseteq$" is to convert any demonic and angelic nondeterminacy into conjunction and disjunction, respectively.) Weakest precondition semantics is thus reduced to a theory of nondeterministic functions, described more fully in Morris et al. [2008].

Constructing a theory of imperative programs in this way offers something more than the traditional approach. Traditionally, it has been troublesome to accommodate functions in a nondeterministic imperative language because it is extremely difficult to prevent nondeterminacy leaking from the level of commands to the level of terms. That worry now disappears because nondeterminacy in terms is welcome.

2.5 Process Algebras

Process algebras are formally defined languages for the study of fundamental concepts in concurrency, including communication, synchronization, abstraction, divergence, and deadlock. Nondeterminacy is central in describing processes, and we would therefore hope that a general theory of nondeterminacy would be useful in designing and formally describing process algebras.

Let $a, b, c, \ldots$ stand for events (such as "open valve" or "release steam" or "close valve" etc.). Some events are executed locally by a single process, while others require the participation of several processes. skip is the event of terminating normally and fail is the event of terminating abnormally (we are here

conflating divergence and deadlock). A process is a sequence of events ending in skip or fail, for example, $\ulcorner a \cdot b \cdot a \cdot \mathsf{skip} \urcorner$ (we add corner brackets to help the reader pick out process terms in the running text).

Process terms may participate in various operations, typified by sequential composition (; which binds looser than $\cdot$). For example, $\ulcorner a \cdot b \cdot \mathsf{skip} \ ; \ b \cdot \mathsf{fail} \urcorner$ is equal to $\ulcorner a \cdot b \cdot b \cdot \mathsf{fail} \urcorner$, and $\ulcorner b \cdot \mathsf{fail} \ ; \ a \cdot b \cdot \mathsf{skip} \urcorner$ is equal to $\ulcorner b \cdot \mathsf{fail} \urcorner$. Processes may behave nondeterministically, as in $\ulcorner a \cdot b \cdot \mathsf{skip} \sqcap b \cdot \mathsf{fail} \urcorner$ – choice exercised internally in a process is postulated to be demonic.

Processes may be composed in parallel ($|||$ which binds looser than $\cdot$), as in, for example, $\ulcorner a \cdot b \cdot \mathsf{skip} \ ||| \ c \cdot \mathsf{skip} \urcorner$, which is equivalent to $\ulcorner a \cdot b \cdot c \cdot \mathsf{skip} \sqcup a \cdot c \cdot b \cdot \mathsf{skip} \sqcup c \cdot a \cdot b \cdot \mathsf{skip} \urcorner$. The parallel composition here gives rise to three possible behaviors in accordance with which event of the participating processes happens next. The choice arising from parallel composition is defined to be angelic to distinguish it from internal choice. The distinction is necessary because, in this case, the choice is exercised by an external agent (which we typically call the *environment*). Processes may also employ angelic choice explicitly when they are offering a menu of options that is intended to be resolved by the environment.

A more general form of parallel operator is $||_A$ where A stands for a set of events on which the participating processes must synchronize. For example, $\ulcorner a \cdot b \cdot d \cdot \mathsf{skip} \ ||_{\{b\}} \ b \cdot e \cdot \mathsf{skip} \urcorner$ is equivalent to $\ulcorner a \cdot b \cdot d \cdot e \cdot \mathsf{skip} \sqcup a \cdot b \cdot e \cdot d \cdot \mathsf{skip} \urcorner$. Here, a in the first process must happen before b in the second process as b is an event in which both processes must participate (note that b occurs just once in each of the two possible resulting processes). On the other hand, $\ulcorner a \cdot b \cdot d \cdot \mathsf{skip} \ ||_{\{b\}} \ c \cdot \mathsf{skip} \urcorner$ is equivalent to $\ulcorner a \cdot c \cdot \mathsf{fail} \sqcup c \cdot a \cdot \mathsf{fail} \urcorner$ — each fail results from the failure of the second process to synchronize on b. $|||$ as introduced earlier is equivalent to $||_\emptyset$.

Although demonic and angelic choice are duals of one another in the basic theory, it does not necessarily follow that they have symmetric properties when they are plugged in to some language. That may or may not be the case. Here we break the symmetry by postulating that demonic choice distributes over parallel composition, while angelic choice distributes only in the absence of demonic choice in the participating processes. We may be led to different process algebras by postulating alternative distribution laws.

There are several interesting ways in which to define the refinement relation on basic processes. An attractive order is one that is very close to the discrete order, differing only in that any process P which ends in fail is refined by any process Q which shares the same prefix as P up to the occurrence of fail in P (e.g., $\ulcorner a \cdot b \cdot \mathsf{fail} \urcorner \sqsubseteq \ulcorner a \cdot b \cdot c \cdot \mathsf{skip} \urcorner$). As an example, the reader might care to show that $\ulcorner (a \cdot \mathsf{skip} \sqcap b \cdot \mathsf{skip}) ||_{\{a,b\}} (a \cdot \mathsf{skip} \sqcup b \cdot \mathsf{skip}) \urcorner$ is equivalent to $\ulcorner a \cdot \mathsf{skip} \sqcap b \cdot \mathsf{skip} \urcorner$.

The above brief foray suggests that we might construct a theory of processes by embedding a theory of dual nondeterminacy in a theory of event sequences. Proceeding along the lines outlined above leads to a theory of communicating sequential processes that is similar to CSP [Hoare 1984; Roscoe 1998]. It has an attractive algebra and a mathematically elegant lattice-theoretic model, derived in large part from the theory of nondeterminacy. See Tyrrell et al. [2006] for more details.

2.6 Fixpoints and Recursion

The act of taking an adjoint is facilitated by nondeterminacy, in essence because nondeterminacy enriches each type with new artificial points (those representing *possible* outcomes). It is natural, therefore, to seek other operations whose domain of applicability is enlarged by the presence of nondeterminacy. An obvious candidate is taking fixpoints. Surprisingly, it turns out that just about every function in a "reasonable" programming or specification language has least and greatest fixpoints, and indeed these may be expressible in the language using the notation of nondeterminacy. To take an extreme example, the successor function on the naturals $\lambda x{:}\mathbb{N} \cdot x + 1$ has fixpoints $\bigsqcup\{m{:}\mathbb{N} \cdot \bigsqcap\{n{:}\mathbb{N} \mid n \geq m\}\}$ and $\bigsqcap\{m{:}\mathbb{N} \cdot \bigsqcup\{n{:}\mathbb{N} \mid n \geq m\}\}$ (recall that function application distributes over choice).

The pervasiveness of fixpoints can be exploited to give a semantics for recursive functions in terms of nondeterminacy, instead of the more common theory of complete partial orders. In essence, every function in a typical programming language can be viewed as a function in a richer nondeterministic language (just add angelic and nondeterministic choice as we have been describing), and in this language the function is guaranteed to have fixpoints. We can identify some of these fixpoints as being computationally interesting, in that they correspond to a certain operational interpretation. Indeed, we can show that wherever a recursively defined function in a typical programming language has a meaning given by the classical theory of least fixpoints in complete partial orders [Winskel 1993], it coincides with the fixpoint arrived at by viewing the function as a nondeterministic function. For more on this approach to recursion; see Morris and Tyrrell [2007].

3. THE BASIC THEORY OF NONDETERMINACY

3.1 Notation

We place ourselves in a typed specification/programming language. We use T, U, ... to stand for types, and t, u, v $\cdots$ to stand for terms. We write $t, u : T$ to assert that terms t and u are of type T, and similarly for other than two terms.

For S any set of terms of type T, say, $\bigsqcap S$ and $\bigsqcup S$ are also terms of type T and denote the demonic and angelic choice, respectively, over the constituent terms of S. $t \sqcap u$ abbreviates $\bigsqcap\{t, u\}$ and $t \sqcup u$ abbreviates $\bigsqcup\{t, u\}$.

We will write set comprehensions as $\{x{:}T \mid P \cdot t\}$, possibly using abbreviated forms as explained in Section 2.1.2. We write $(\forall X \subseteq T \ \cdots)$ as an abbreviation for $(\forall X{:}I\!P T \ \cdots)$. We will write $S :\subseteq T$ to assert that S is a set of terms each of type T.

$\bigsqcup\emptyset$ is given the special name $\bot_T$ (pronounced *bottom*), and $\bigsqcap\emptyset$ is given the name $\top_T$ (pronounced *top*). $\bigsqcup T$ is given the special name $some_T$, and $\bigsqcap T$ is given the name all_T. Again, we commonly omit the type subscripts.

Operators have the following relative precedence, higher precedence first (the list anticipates some operators we have yet to introduce):(i) function application and composition; (ii) $\bigsqcap$, $\bigsqcup$, $\bigwedge$, $\bigvee$; (iii) $\sqcap$, $\sqcup$; (iv) $\leq$, $\sqsubseteq$; (v) $\neg$; (vi) $\wedge$, $\vee$; (vii)

$=, \neq$; (viii) $\Rightarrow$; (ix) $\Leftrightarrow$. For example, the brackets in the following are superfluous:

$$\big((t \sqcap u) \sqsubseteq \bigsqcup X\big) \Leftrightarrow \big((t \sqsubseteq \bigsqcup X) \vee (u \sqsubseteq \bigsqcup X)\big)$$

We assume each type T comes equipped with a partial ordering $\sqsubseteq_T$ as explained in Section 2.1.3 (we omit the type subscript when it can be inferred from context or is not significant). $t \sqsubseteq u$ is expressed in words as "t is *refined* by u" or "*u refines t*". In the case of base types such as the integers or booleans, the refinement ordering will usually be the discrete ordering. If the reader has in mind a type with no obvious partial ordering, then it can be trivially ordered by the discrete ordering. The theory will respect the refinement ordering on base types when nondeterminacy is introduced. The theory will also impose an ordering on constructed types, e.g. the values in function types will be ordered in the usual pointwise way as we shall see.

3.2 Proper Terms

We distinguish between *proper* and *improper* terms. For base types, the proper terms are those that can be expressed without using nondeterminacy. To use the integers for illustration, the proper integers are precisely those that are equivalent to one of 0, 1, -1, 2, -2, ... For example, 1, $1 \sqcap 1$, $1 \sqcup 1$, and $\bigsqcup\{1\}$ are all proper, while $1 \sqcap 2$, $\bigsqcup\{x{:}\mathbf{Z} \mid 0 \leq x\}$, $\bot$, $\top$, *some*, and all are improper. The properness or improperness of constructed types is defined with respect to the constructors of the type. In the present case our language is limited to function types, for which the only constructor is via λ-abstractions. The propers are precisely those that are equivalent to a λ-abstraction (even if the body of the λ-abstraction employs nondeterminacy in an essential way). Note that properness is not a syntactic property: if a term is proper (or improper) then so are all equivalent terms.

In introducing nondeterminacy, we have to make clear the extent to which terms employing nondeterminacy may participate in instantiation. It is analogous to the situation in partial function theory where we accommodate "undefined" terms such as $3/0$: we expect that from $(\forall x{:}\mathbf{Z} \cdot x - x = 0)$ we can infer $3 - 3 = 0$, but probably not $3/0 - 3/0 = 0$. We adopt a similar convention here: from $(\forall x{:}\mathbf{Z} \cdot x - x = 0)$ we may infer $3 - 3 = 0$, but not $\bot_{\mathbf{Z}} - \bot_{\mathbf{Z}} = 0$ or $(2 \sqcap 3) - (2 \sqcap 3) = 0$. More generally, we adopt the convention that the range of the bound variable does not extend to improper terms. This convention applies to all bound variables, e.g. in existential quantifications, set comprehensions, etc. For example, $(\exists x{:}T \cdot x = t)$ holds iff term $t : T$ is proper.

We may occasionally use a type to represent a set, in which case we always mean it to stand for the set of its propers. For example, we may write $X \subseteq \mathbb{Z}$ to denote that X is a set of (proper) integers, or for t a term of type $\mathbb{Z}$ we may write $t \in \mathbb{Z}$ to assert that t denotes a proper value.

We alert the reader to distinguish between $t : T$ (here t may or may not be proper) and $t \in T$ (here t is proper), for T some type. A similar distinction holds between $S :\subseteq T$ (the elements of S may or may not be proper), and $S \subseteq T$ (the elements of S are proper).

3.3 Axioms of Nondeterminacy

Five axioms govern basic nondeterminacy. The first two relate refinement to demonic and angelic choice, respectively, where $t, u : T$:

$$\text{A1:} \qquad t \sqsubseteq u \Leftrightarrow \left(\forall X \subseteq T \cdot \sqcap X \sqsubseteq t \Rightarrow \sqcap X \sqsubseteq u\right)$$

$$\text{A2:} \qquad t \sqsubseteq u \Leftrightarrow \left(\forall X \subseteq T \cdot u \sqsubseteq \bigsqcup X \Rightarrow t \sqsubseteq \bigsqcup X\right)$$

The preceding axioms give formal standing to our earlier assertion that $t \sqsubseteq u$ holds when t offers at least as much demonic nondeterminacy as u, and u offers at least as much angelic nondeterminacy as t.

The second two give the axioms for demonic and angelic choice, where $X \subseteq T$, $S :\subseteq T$:

$$\text{A3:} \qquad \sqcap S \sqsubseteq \bigsqcup X \Leftrightarrow \left(\exists t \in S \cdot t \sqsubseteq \bigsqcup X\right)$$

$$\text{A4:} \qquad \sqcap X \sqsubseteq \bigsqcup S \Leftrightarrow \left(\exists t \in S \cdot \sqcap X \sqsubseteq t\right)$$

(Note that X is a set of *propers* in all the preceding axioms, while S is a set of (possibly improper) terms). Finally, we postulate that refinement is antisymmetric:

$$\text{A5:} \qquad t \sqsubseteq u \wedge u \sqsubseteq t \Rightarrow t = u$$

The axioms are not so much used in practice, once they have been used to establish a body of simpler and more practically useful laws. The more important of these are given in Figure 1. When reading Figure 1 remember that X is a set of *propers*. It might appear at first sight that Theorems 7 and 8 conflict with standard results in lattice theory, but this is not so because here X is not an arbitrary set but a set of propers. We have not formally stated the distribution laws in Theorem 15. Those for the binary choice operators are easily written, and the reader will no doubt be able to do so. Readers familiar with infinite distributions (as in, for example, distribution of arbitrary set union over arbitrary set intersection, or distribution of arbitrary joins over arbitrary meets in lattices) will know that their formal expression requires the use of a choice function. As the technique is standard but complicated, there is no additional information to be imparted by writing it formally here; the interested reader can see the details in Davey and Priestley [2002]. Theorems 23 and 24 are technically important; we will return to them later.

It is clerical routine to prove the theorems in the order presented; the proof of arbitrary distribution uses the axiom of choice, but otherwise there are no surprises. We prove Theorem 17 as an example; see Figure 2. For an example of applying the laws, see the proof in Section 2.2.2 which appeals to axiom A4 (using Theorem 3 to instantiate $\sqcap X$ as $\sqcap \{x\}$ for x a certain real), and Theorem 9.

4. THE THEORY OF NONDETERMINISTIC FUNCTIONS

4.1 Introduction and Notation

We now construct a theory of nondeterministic functions. We write function types in the usual way: $T \rightarrow U$ stands for the type of functions with domain T

Fundamental theorems

1. $t = u \Leftrightarrow (\forall X{\subseteq}T \cdot \bigsqcap X \sqsubseteq t \Leftrightarrow \bigsqcap X \sqsubseteq u)$
2. $t = u \Leftrightarrow (\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Leftrightarrow u \sqsubseteq \bigsqcup X)$
3. $\bigsqcap\{t\} = t$
4. $\bigsqcup\{t\} = t$
5. $\sqcap$, $\sqcup$ are symmetric, associative, and idempotent
6. $\sqsubseteq$ is a partial order
7. $t \sqcap u \sqsubseteq \bigsqcup X \Leftrightarrow t \sqsubseteq \bigsqcup X \vee u \sqsubseteq \bigsqcup X$
8. $\bigsqcap X \sqsubseteq t \sqcup u \Leftrightarrow \bigsqcap X \sqsubseteq t \vee \bigsqcap X \sqsubseteq u$
9. $t \sqsubseteq u \sqcap v \Leftrightarrow t \sqsubseteq u \wedge t \sqsubseteq v$
10. $u \sqcup v \sqsubseteq t \Leftrightarrow u \sqsubseteq t \wedge v \sqsubseteq t$
11. $t \sqsubseteq \bigsqcap S \Leftrightarrow (\forall u{\in}S \cdot t \sqsubseteq u)$
12. $\bigsqcup S \sqsubseteq t \Leftrightarrow (\forall u{\in}S \cdot u \sqsubseteq t)$
13. $t \sqsubseteq u \Leftrightarrow t \sqcap u = t$
14. $t \sqsubseteq u \Leftrightarrow t \sqcup u = u$
15. $\sqcap$, $\sqcup$ distribute over one another, as do $\bigsqcap$, $\bigsqcup$
16. $\bot \sqsubseteq t \sqsubseteq \top$
17. $all \sqsubseteq t \Leftrightarrow t \neq \bot$
18. $t \sqsubseteq some \Leftrightarrow t \neq \top$
19. $t \sqcap \bot = \bot$, $t \sqcap \top = t$, $t \sqcup \bot = t$, $t \sqcup \top = \top$
20. $\bot \neq \top$
21. $all \neq \bot$
22. $some \neq \top$
23. $t = \bigsqcap\{X{\subseteq}T \mid t \sqsubseteq \bigsqcup X \cdot \bigsqcup X\}$
24. $t = \bigsqcup\{X{\subseteq}T \mid \bigsqcap X \sqsubseteq t \cdot \bigsqcap X\}$

Fig. 1. Fundamental theorems (t,u,$v : T$, $X \subseteq T$, $S :\subseteq T$).

$all \sqsubseteq t$
$\Leftrightarrow$ “all-defn, A2”
$(\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Rightarrow \bigsqcap T \sqsubseteq \bigsqcup X)$
$\Leftrightarrow$ “A3”
$(\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Rightarrow (\exists y{:}T \cdot y \sqsubseteq \bigsqcup X))$
$\Leftrightarrow$ “$\bigsqcap\{y\} = y$ (Theorem 3)”
$(\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Rightarrow (\exists y{:}T \cdot \bigsqcap\{y\} \sqsubseteq \bigsqcup X))$
$\Leftrightarrow$ “A4, $\bigsqcap\{y\} = y$”
$(\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Rightarrow (\exists y{:}T \cdot (\exists x{\in}X \cdot y \sqsubseteq x)))$
$\Leftrightarrow$ “$\sqsubseteq$ reflexive (Theorem 6), logic”
$(\forall X{\subseteq}T \cdot t \sqsubseteq \bigsqcup X \Rightarrow X \neq \emptyset)$
$\Leftrightarrow$ “logic”
$(\forall X{\subseteq}T \cdot X = \emptyset \Rightarrow \neg(t \sqsubseteq \bigsqcup X))$
$\Leftrightarrow$ “logic”
$\neg(t \sqsubseteq \bigsqcup \emptyset)$
$\Leftrightarrow$ “$\bot$-defn, $\bot \sqsubseteq t$ (Theorem 16)”
$t \neq \bot$

Fig. 2. Proof of $all \sqsubseteq t \Leftrightarrow t \neq \bot$.

and codomain U. Terms of the type are written using λ-notation as in $\lambda x{:}T \cdot t$. Bodies of lambda terms may employ nondeterminacy. Function application is denoted by juxtaposition as in $t\ u$ which denotes the application of function t to argument u.

To stave off a potential inconsistency, we impose a technical limitation on the formation of λ-abstractions: we stipulate that for $\lambda x{:}T \cdot t$ to be admissible it must

satisfy the monotonicity property $(\forall y, z{:}T \cdot y \sqsubseteq z \Rightarrow t[x\backslash y] \sqsubseteq t[x\backslash z])$. This is always the case in programming languages, but requires caution in specification languages. A well-known source of non-monotonicity is strong equality on non-discrete types. For example, allowing for the moment strong equality on integer functions, the λ-abstraction $h \triangleq (\lambda f{:}\mathbb{Z}\to\mathbb{Z} \cdot f = (\lambda x{:}\mathbb{Z} \cdot 3))$ is ruled out because it is not monotonic — with $f0 \triangleq \lambda x{:}\mathbb{Z}\cdot 3 \sqcap 4$ and $f1 \triangleq \lambda x{:}\mathbb{Z}\cdot 3$ we have $f0 \sqsubseteq f1$, but not $h\,f0$ $(=false) \sqsubseteq h\,f1$ $(=true)$. We can either syntactically prevent parameters in a λ-abstraction from being used as the argument of any nonmonotonic operator, or we impose on the programmer the obligation to prove monotonicity.

Axioms A1 to A5, inclusive, are included unchanged. Using them we may infer, for example, that $f \sqcap g \sqsubseteq f$ for functions f and g of the same type, without any special information about function types.

4.2 Axioms of Nondeterministic Functions

The first axiom asserts that λ-abstractions are proper:

A6: $(\exists f{:}T\to U \cdot f = (\lambda x{:}T \cdot t))$,

where t stands for a term of type U. Observe that the theory treats every λ-term as proper, even if the body of the function employs nondeterminacy. For example, $(\lambda x{:}\mathbb{Z} \cdot x \sqcap 3)$ is proper, but $(\lambda x{:}\mathbb{Z} \cdot x) \sqcap (\lambda x{:}\mathbb{Z} \cdot 3)$ is not.

Next we define the refinement order on proper functions:

A7: $(\forall f, g{:}T\to U \cdot f \sqsubseteq g \Leftrightarrow (\forall x{:}T \cdot f\,x \sqsubseteq g\,x))$.

The third axiom states that β-reduction holds for proper arguments:

A8: $(\forall y{:}T \cdot (\lambda x{:}T \cdot t)\,y = t[x\backslash y])$

For example, $(\lambda h{:}\mathbb{Z}\to\mathbb{Z} \cdot h\,3 \sqcap h\,4)(\lambda x : \mathbb{Z} \cdot x \sqcup x^2)$ reduces to $(3 \sqcup 3^2) \sqcap (4 \sqcup 4^2)$.

The fourth and fifth axioms state that proper functions distribute over choice in their argument, that is, for $S :\subseteq T$:

A9: $(\forall f{:}T\to U \cdot f\,(\bigsqcap S) = \bigsqcap\{t \in S \cdot f\,t\})$
A10: $(\forall f{:}T\to U \cdot f\,(\bigsqcup S) = \bigsqcup\{t \in S \cdot f\,t\})$

For example, $(\lambda x{:}\mathbb{Z} \cdot x^2 \sqcup x)(3 \sqcap 4)$ reduces to $(3^2 \sqcup 3) \sqcap (4^2 \sqcup 4)$.

The sixth axiom caters for nondeterminacy on the function side of a function application: for $t : T\to U$ and $u : T$,

A11: $t\,u = (\lambda f{:}T\to U \cdot f\,u)\,t$,

where f is a fresh name. The only new information here is that function application distributes over choice on the left. For example, $((\lambda x{:}\mathbb{Z}\cdot x^2)\sqcup(\lambda x{:}\mathbb{Z}\cdot x))(2\sqcap 3)$ is by the axiom equal to $(\lambda f{:}\mathbb{Z}\to\mathbb{Z}\cdot f(2\sqcap 3))((\lambda x{:}\mathbb{Z}\cdot x^2)\sqcup(\lambda x{:}\mathbb{Z}\cdot x))$; this reduces by A10 to $(\lambda x{:}\mathbb{Z}\cdot x^2)(2\sqcap 3)\sqcup(\lambda x{:}\mathbb{Z}\cdot x)(2\sqcap 3)$, and then by A9 (twice) to $(2^2\sqcap 3^2)\sqcup(2\sqcap 3)$. Note that only applications of λ-abstractions, not arbitrary functions, distribute on the right. The import of this is that we must distribute fully on the left first and only then distribute on the right; the axioms enforce this.

Theorems about Functions

25. $\bot\, u = \bot$
26. $\top\, u = \top$
27. $t \bot = \bot \Leftrightarrow t \neq \top$
28. $t \top = \top \Leftrightarrow t \neq \bot$
29. $f(u_1 \sqcap u_2) = f\, u_1 \sqcap f\, u_2$
30. $f(u_1 \sqcup u_2) = f\, u_1 \sqcup f\, u_2$
31. $(\bigsqcap S)\, u = \bigsqcap\{f \in S \cdot f\, u\}$
32. $(\bigsqcup S)\, u = \bigsqcup\{f \in S \cdot f\, u\}$
33. $(t_1 \sqcap t_2)\, u = t_1\, u \sqcap t_2\, u$
34. $(t_1 \sqcup t_2)\, u = t_1\, u \sqcup t_2\, u$
35. $t_1 \sqsubseteq t_2 \Rightarrow t_1\, u \sqsubseteq t_2\, u$
36. $u_1 \sqsubseteq u_2 \Rightarrow t\, u_1 \sqsubseteq t\, u_2$
37. $f = g \Leftrightarrow (\forall x{:}T \cdot f\, x = g\, x)$
38. $f = (\lambda x{:}T \cdot f\, x)$
39. $(\forall x{:}U \cdot u_1 \sqsubseteq u_2) \Leftrightarrow (\lambda x{:}U \cdot u_1) \sqsubseteq (\lambda x{:}U \cdot u_2)$
40. $(\forall x{:}U \cdot u_1 = u_2) \Leftrightarrow (\lambda x{:}U \cdot u_1) = (\lambda x{:}U \cdot u_2)$
41. $(\lambda x{:}U \cdot u) = (\lambda y{:}U \cdot u[x \backslash y])$ (x not in the free variables of u)
42. $(\lambda x{:}U \cdot u_1 \sqcap u_2) \sqsubseteq (\lambda x{:}U \cdot u_1) \sqcap (\lambda x{:}U \cdot u_2)$
43. $(\lambda x{:}U \cdot u_1) \sqcup (\lambda x{:}U \cdot u_2) \sqsubseteq (\lambda x{:}U \cdot u_1 \sqcup u_2)$
44. $all_{T \to U} = (\lambda x{:}T \cdot \bot_U)$
45. $some_{T \to U} = (\lambda x{:}T \cdot \top_U)$
46. $(t \circ v)\, w = t(v\, w)$
47. $f \circ (v_1 \sqcap v_2) = f \circ v_1 \sqcap f \circ v_2$
48. $f \circ (v_1 \sqcup v_2) = f \circ v_1 \sqcup f \circ v_2$
49. $(t_1 \sqcap t_2) \circ v = t_1 \circ v \sqcap t_2 \circ v$
50. $(t_1 \sqcup t_2) \circ v = t_1 \circ v \sqcup t_2 \circ v$
51. $t_1 \sqsubseteq t_2 \Rightarrow t_1 \circ v \sqsubseteq t_2 \circ v$
52. $v_1 \sqsubseteq v_2 \Rightarrow t \circ v_1 \sqsubseteq t \circ v_2$

Fig. 3. Functions ($t, t_1, t_2 : T \to U, u, u_1, u_2 : T$, $f, g \in T \to U, v, v_1, v_2 : V \to T, w : V, S :\subseteq T \to U$).

Finally, we define function composition. For $t : U \to V$ and $u : T \to U$:

A12: $$t \circ u = (\lambda f{:}U \to V \cdot \lambda g{:}T \to U \cdot \lambda x{:}T \cdot f(g\, x))\, t\, u.$$

The axiom is expressed as it is, with the participating functions on the lefthand side emerging as arguments on the righthand side, to capture the fact that composition distributes over choice (as Axioms A9 to A11 are now applicable).

The most important theorems that follow from the axioms are listed in Figure 3; when reading it note that f and g stand for *proper* functions ($f, g \in T \to U$), such as λ-abstractions as guaranteed by Axiom A6. The proofs are clerical routine in each case; as an example a proof of Theorem 33 is given in Figure 4.

Perhaps the most significant conclusion to be drawn from the axioms and theorems is how much of the standard theory of functions is retained following the introduction of nondeterminacy. In particular, β-reduction survives almost intact, as does function composition. Extensional equality (Theorem 37) and η-equivalence (Theorem 38) continue to hold for proper functions. They do not hold in general as the following argument shows. Clearly, $(\lambda x{:}\mathbb{Z} \cdot x \sqcap 3)$ and $(\lambda x{:}\mathbb{Z} \cdot x) \sqcap (\lambda x{:}\mathbb{Z} \cdot 3)$ are extensionally equal. Nevertheless they can be distinguished by the higher order function $F \triangleq \lambda h{:}\mathbb{Z} \to \mathbb{Z} \cdot h\, 1 + h\, 2$. We leave it to the reader to verify that $F\ (\lambda x{:}\mathbb{Z} \cdot x \sqcap 3)$ yields $3 \sqcap 4 \sqcap 5 \sqcap 6$ while $F\ ((\lambda x{:}\mathbb{Z} \cdot x) \sqcap (\lambda x{:}\mathbb{Z} \cdot 3))$

$$
\begin{array}{ll}
 & (t_1 \sqcap t_2)\, u \\
\Leftrightarrow & \quad \text{“definition of } \sqcap\text{”} \\
 & (\bigsqcap\{t_1, t_2\})\, u \\
\Leftrightarrow & \quad \text{“A11”} \\
 & (\lambda f{:}T{\to}U \cdot f\, u)(\bigsqcap\{t_1, t_2\}) \\
\Leftrightarrow & \quad \text{“A9”} \\
 & \bigsqcap\{(\lambda f{:}T{\to}U \cdot f\, u)\, t_1, (\lambda f{:}T{\to}U \cdot f\, u)\, t_2\} \\
\Leftrightarrow & \quad \text{“A11 twice”} \\
 & \bigsqcap\{t_1\, u, t_2\, u\} \\
\Leftrightarrow & \quad \text{“definition of } \sqcap\text{”} \\
 & t_1\, u \sqcap t_2\, u
\end{array}
$$

Fig. 4. Proof of $(t_1 \sqcap t_2)\, u = t_1\, u \sqcap t_2\, u$.

yields $3 \sqcap 6$ (remember that the application of F in the latter case distributes over the choice in the argument). It follows that function equivalence via extensionality does not hold in general. Without appealing to extensionality, we could infer the equivalence of $(\lambda x{:}\mathbb{Z} \cdot x \sqcap 3)$ and $(\lambda x{:}\mathbb{Z} \cdot x) \sqcap (\lambda x{:}\mathbb{Z} \cdot 3)$ if we were to appeal to η-equivalence (by applying it to the latter term). It follows that η-equivalence does not hold in general.

As a final comment, we point out that there is no one unique way in which to plug nondeterminacy into function theory. The above treatment seems to us to be the simplest, generally useful approach.

4.3 Operators

It will usually be the case that the operators of a type will be defined to distribute over choice. In the case of binary operators, we may choose to distribute left argument first, or right argument first. For example, if for some reason we wanted integer subtraction to distribute right argument first we would postulate $t - u = (\lambda x{:}\mathbb{Z} \cdot \lambda y{:}\mathbb{Z} \cdot y - x)\, u\, t$.

The theory does not exclude the possibility that operators might deal with choice in other ways — there is no obligation to employ the distribution laws of function application. For example, if it is important for a symmetric binary argument to continue to be symmetric in the presence of nondeterminacy then we might define it to distribute demonically first and then angelically (or vice versa). By “demonically first” we mean that all demonic choices in either argument are distributed, after which all angelic choices are distributed. This is possible because of the Theorems 23 and 24 in Figure 1. These technically important theorems assert that every term can be expressed in demonic or angelic normal form, respectively. A term is in *demonic normal form* if it is a demonic choice over angelically proper terms, where an angelically proper term is an angelic choice of propers. *Angelic normal form* is defined dually. Roughly speaking, the theorems state that all demonic choices can be moved “to the outside”, and similarly for angelic choices.

5. DOMAINS FOR NONDETERMINACY

5.1 Overview

Our final task is to build a model by which we can show that the theory is sound. The key to this is a technique to embed a poset C in a certain complete

and completely distributive lattice denoted by $\mathbf{FCD}(C)$. $\mathbf{FCD}(C)$ is what is called the *free completely distributive lattice* over C. The mathematics of these lattices is the subject of this section.

5.2 Lattice Fundamentals

Everything in this subsection is standard and is available in more detail in any standard text (such as Davey and Priestley [2002] and Birkhoff [1967]).

A *complete lattice* L is a partially ordered set (we'll use $\leq$ to represent the partial ordering) such that every subset of L has a greatest lower bound in L with respect to $\leq$, and similarly has a least upper bound in L with respect to $\leq$. For $S \subseteq L$, the greatest lower bound of S is denoted by $\bigwedge S$, and the least upper bound is denoted by $\bigvee S$. Greatest lower bounds are also called *meets*, and least upper bounds are also called *joins*. A complete lattice is *completely distributive* iff meets distribute over joins, and vice-versa. When we say that a lattice is completely distributive, we mean to imply that it is also complete.

A function f from poset C to poset D is *monotonic* iff $x \leq_C y \Rightarrow f\,x \leq_D f\,y$ for all $x, y \in C$, and an *order embedding* iff $x \leq_C y \Leftrightarrow f\,x \leq_D f\,y$ for all $x, y \in C$. An order embedding from poset C to a complete lattice is said to be a *completion* of C. We denote the space of monotonic functions from C to D by $C \xrightarrow{\text{m}} D$, which is ordered pointwise.

Let f be a function from complete lattice L to complete lattice M. f is $\bigwedge$*-distributive* iff $f(\bigwedge S) = \bigwedge(f\,S)$ for all $S \subseteq L$; $\bigvee$*-distributive* is defined dually. By $f\,S$ above and in what follows, we mean the image of S through f, that is, $\{x \in S \cdot f\,x\}$. f is a *complete homomorphism* if f is $\bigwedge$- and $\bigvee$-distributive.

5.3 Free Completely Distributive Lattices

There are various ways in which a partially ordered set can be embedded in a complete lattice, the one of interest to us being the free completely distributive completion. A completely distributive lattice L is called the *free completely distributive lattice over a poset* C iff there is a completion $\phi : C \xrightarrow{\text{m}} L$ such that for every completely distributive lattice M and monotonic function $f : C \xrightarrow{\text{m}} M$ there is a unique complete homomorphism $\phi^*_M f : L \xrightarrow{\text{m}} M$ satisfying $f = (\phi^*_M f) \circ \phi$. We call this a *free completely distributive completion* of C.

We offer some pictorial insight into free completely distributive lattices. Figure 5 depicts a small poset and its free completely distributive lattice. Each of the small circles can be labeled in several ways because of distributivity, for example, the upper circle represents both $(c \wedge d) \vee (a \vee b)$ and $(c \vee b) \wedge d$. We make some general observations. $\bot$ is always connected to *all* and no other, and $\top$ is always connected to *some* and no other. Observe $a \leq c$ in the poset and hence c is the least upper bound of a and c; c continues to be the least upper bound in the lattice. Contrast that with a and b for which a third element d is the least upper bound in the poset. In this case, a new least upper bound is created in the lattice below d. Unbounded chains behave similarly (although not illustrated here): if $\{i \in \mathbb{N} \cdot x_i\}$ is a chain in the poset (i.e., $x_i \leq x_{i+1}$ for all natural i) with a least upper bound z not in the chain, then the least upper

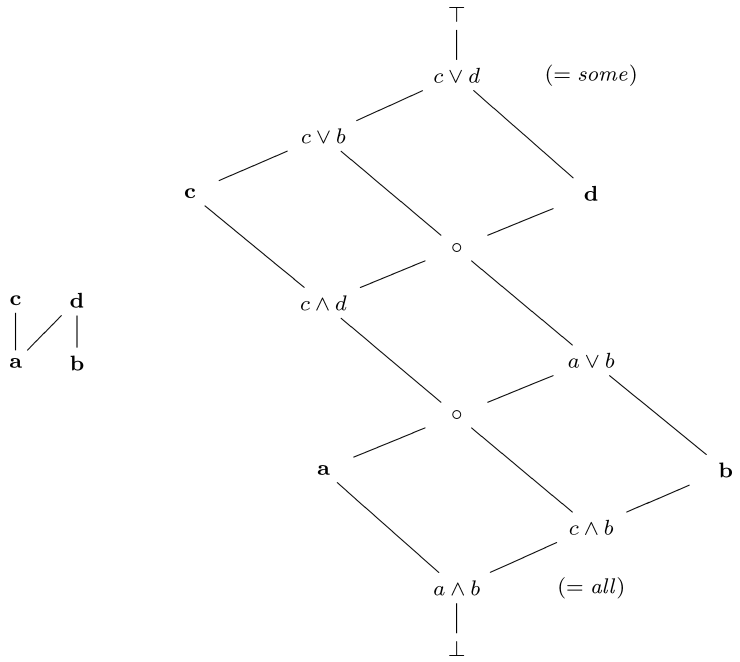

Fig. 5. A poset and its free completely distributive lattice.

bound of the chain in the lattice is not z but a new element below z. Similar remarks hold for greatest lower bounds.

THEOREM 5.1. *For every poset C, the free completely distributive lattice over C exists and is unique up to isomorphism.*

PROOF. Tunnicliffe [1985] gives the following construction. Let D be the set of downclosed subsets of C (a set is downclosed when it contains all the elements below x whenever it contains x) ordered by set inclusion, and let E be the set of downclosed subsets of D. Then, E is a construction of the free completely distributive lattice over C under the completion which takes each x in C to the family of sets in D which do not contain x. There is a similar construction with detailed proof in Morris [2004], and an alternative construction in Bartenschlager [1995]. If $\phi : C \xrightarrow{\text{m}} L_0$ and $\theta : C \xrightarrow{\text{m}} L_1$ are two completions, then L_0 and L_1 are isomorphic as $\phi^*_{L_1}\theta : L_0 \xrightarrow{\text{m}} L_1$ and $\theta^*_{L_0}\phi : L_1 \xrightarrow{\text{m}} L_0$ are complete homomorphisms and inverses of one another. □

Theorem 5.1 means that we can talk about *the* free completely distributive lattice over C, which we denote $\mathbf{FCD}(C)$. We typically drop the lattice subscript in ϕ^*_M when it can be determined by context. We need the properties of free completely distributive lattices as encapsulated in the following four theorems. The first theorem states that every element in $\mathbf{FCD}(C)$ for C a poset can be constructed from C-elements, that is, from ϕC where ϕ denotes the completion, either by taking meets followed by joins or vice-versa.

THEOREM 5.2. *Let* $\phi : C \xrightarrow{\text{m}} \mathbf{FCD}(C)$ *be an FCD completion of* C*. For all* $t \in \mathbf{FCD}(C)$*, (i)* $t = \bigvee\{X \subseteq \phi C \mid \bigwedge X \leq t \cdot \bigwedge X\}$*, and (ii)* $t = \bigwedge\{X \subseteq \phi C \mid t \leq \bigvee X \cdot \bigvee X)\}$.

PROOF. Let L be the complete sublattice of $\mathbf{FCD}(C)$ obtained by taking any elements of $\mathbf{FCD}(C)$ that can be expressed as arbitrarily nested meets and joins over ϕC. As a complete sublattice of a completely distributive lattice, L is completely distributive. The image of ϕ is in L, and for any completely distributive lattice M and function $f : C \xrightarrow{\text{m}} M$, $(\phi^* f) \restriction L$ defines a complete homomorphism from L to M with the property that $((\phi^* f) \restriction L) \circ \phi = f$. The behavior of $(\phi^* f) \restriction L$ is completely determined by its behavior on ϕC, so $(\phi^* f) \restriction L$ is unique on L. Thus, ϕ is a free completely distributive completion of C in L, and hence $\mathbf{FCD}(C) \cong L$.

By complete distributivity, any element $t \in L$ can be expressed in the form $\bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}$ where $x_{i,j} \in \phi C$. Thus, (i) will follow if we prove $\bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j} = \bigvee\{X \subseteq \phi C \mid \bigwedge X \leq (\bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}) \cdot \bigwedge X\}$. The $\geq$ case follows directly from the property of joins, so we give only the $\leq$ case:

$$
\begin{array}{ll}
 & \bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j} \;\leq\; \bigvee \left\{X \subseteq \phi C \mid \bigwedge X \leq \left(\bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}\right) \cdot \bigwedge X\right\} \\
\Leftarrow & \qquad \text{“properties of joins”} \\
 & \forall i \in I \cdot \exists X \subseteq \phi C \cdot \left(\bigwedge X \leq \bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}\right) \wedge \left(\bigwedge_{j \in J_i} x_{i,j} \leq \bigwedge X\right) \\
\Leftarrow & \qquad \text{“for each } i \in I \text{, let } X = \{j \in J_i \cdot x_{i,j}\}\text{”} \\
 & \forall i \in I \cdot \left(\bigwedge_{j \in J_i} x_{i,j} \leq \bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}\right) \wedge \left(\bigwedge_{j \in J_i} x_{i,j} \leq \bigwedge_{j \in J_i} x_{i,j}\right) \\
\Leftrightarrow & \qquad \text{“reflexivity of } \leq\text{, property of joins”} \\
 & \bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j} \leq \bigvee_{i \in I} \bigwedge_{j \in J_i} x_{i,j}
\end{array}
$$

The result follows from the reflexivity of $\leq$. (ii) is similarly proved by expressing t in the form $\bigwedge_{i \in I} \bigvee_{j \in J_i} y_{i,j}$ where $y_{i,j} \in \phi C$. □

THEOREM 5.3. *Let* $\phi : C \xrightarrow{\text{m}} \mathbf{FCD}(C)$ *be an FCD completion of* C*. For all* $t, u \in \mathbf{FCD}(C)$*, (i)* $t \leq u \Leftrightarrow (\forall X \subseteq \phi C \cdot \bigwedge X \leq t \Rightarrow \bigwedge X \leq u)$*, and (ii)* $t \leq u \Leftrightarrow (\forall X \subseteq \phi C \cdot u \leq \bigvee X \Rightarrow t \leq \bigvee X)$

PROOF. For (i), the implication from left to right is trivial, depending only on the transitivity of $\leq$. For the right-to-left implication, use the preceding theorem to express each of t and u as a join of meets, and appeal to an elementary property of joins, viz. $S_0 \subseteq S_1 \Rightarrow \bigvee S_0 \leq \bigvee S_1$ for arbitrary sets S_0 and S_1. The proof of (ii) is similar. □

THEOREM 5.4. *Let* $\phi : C \xrightarrow{\text{m}} \mathbf{FCD}(C)$ *be an FCD completion of* C*. For all* $X \subseteq \phi C$ *and* $S \subseteq \mathbf{FCD}(C)$*, (i)* $\bigwedge X \leq \bigvee S \Leftrightarrow (\exists s \in S \cdot \bigwedge X \leq s)$*, and (ii)* $\bigwedge S \leq \bigvee X \Leftrightarrow (\exists s \in S \cdot s \leq \bigvee X)$.

PROOF. (i) and (ii) are dual, so we consider only (i). The $\Leftarrow$ implication follows directly from transitivity and the properties of joins. For the $\Rightarrow$ implication, let $X \subseteq \phi C$ and $S \subseteq \mathbf{FCD}(C)$ be given. The truth-value lattice $(\mathbb{B}, \Rightarrow, \exists, \forall)$ is a completely distributive lattice. Define a function (predicate) $P : C \to \mathbb{B}$ on

elements of C by $P\,x \triangleq (\bigwedge X \leq \phi x)$. It is easy to prove that P is a monotonic function and hence can be lifted to $\phi^* P : \mathbf{FCD}(C) \to \mathbb{B}$.

Step 1. By the monotonicity and distributivity of $\phi^* P$ we have $\bigwedge X \leq \bigvee S \Rightarrow ((\phi^* P)(\bigwedge X) \Rightarrow (\phi^* P)(\bigvee S)) \Leftrightarrow ((\forall x \in X \cdot (\phi^* P)x) \Rightarrow (\exists s \in S \cdot (\phi^* P)s))$. By the definition of P, it is straightforward to show $\forall x \in X \cdot (\phi^* P)x$ is true. Thus, $\bigwedge X \leq \bigvee S \Rightarrow \exists s \in S \cdot (\phi^* P)s$.

Step 2. Using Theorem 5.2, $(\phi^* P)s \Leftrightarrow (\phi^* P)(\bigvee\{Y \subseteq \phi C \mid \bigwedge Y \leq s \cdot \bigwedge Y\})$. Distributing $(\phi^* P)$ over the $\bigvee$ and $\bigwedge$ gives $(\exists Y \subseteq \phi C \cdot (\bigwedge Y \leq s) \wedge (\forall y \in Y \cdot (\phi^* P)y))$. By the definition of P and properties of meets, it follows straightforwardly that $(\forall y \in Y \cdot (\phi^* P)y) \Leftrightarrow \bigwedge X \leq \bigwedge Y$. Using transitivity of $\leq$, we have $(\phi^* P)s \Rightarrow \bigwedge X \leq s$.

Step 3. By transitivity of $\Rightarrow$, the conclusions of Steps 1 and 2 give $\bigwedge X \leq \bigvee S \Rightarrow (\exists s \in S \cdot \bigwedge X \leq s)$ as required. □

THEOREM 5.5. *Let $\phi : C \to \mathbf{FCD}(C)$ be an FCD completion of C and let M be a completely distributive lattice. Then, ϕ^*_M is a monotonic function from $C \xrightarrow{\text{m}} M$ to $\mathbf{FCD}(C) \xrightarrow{\text{m}} M$.*

PROOF. Let $f, g : C \xrightarrow{\text{m}} M$ be functions such that $f \leq g$. By definition of the ordering on $\mathbf{FCD}(C) \xrightarrow{\text{m}} M$, $\phi^*_M f \leq \phi^*_M g$ iff $(\forall t \in \mathbf{FCD}(C) \cdot (\phi^*_M f)t \leq (\phi^*_M g)t)$. Pick an arbitrary t. It is straightforward to prove that $(\phi^*_M f)t \leq (\phi^*_M g)t$ follows from $f \leq g$ by expressing t as a join of meets of elements of ϕC using Theorem 5.2. □

One of the referees has pointed out to us that the free completely distributive lattice over a poset is folklore in category theory, and that its properties follow easily from a category-theoretic presentation. Briefly, the construction is as follows. Let POS and CDL be the categories of posets and completely distributive lattices, respectively, and let $U : \text{CDL} \to \text{POS}$ be the forgetful functor from CDL to POS. The problem of proving the existence of the free completely distributive lattice over a poset becomes that of proving the existence of a functor $F : \text{POS} \to \text{CDL}$ which is left-adjoint to the functor U. This can be done by defining F as the composition of two Yoneda embeddings (corresponding, in order-theoretic terms, to taking down-closed and up-closed sets). The operator which lifts monotonic functions $C \xrightarrow{m} M$ to $\text{FCD}(C) \xrightarrow{m} M$ corresponds to a Kan extension along F. We retain the order-theoretic presentation given above to make our results more accessible to a wider audience.

6. SOUNDNESS

6.1 The Model

The hierarchy of types in the language will be modeled by a corresponding hierarchy of complete lattices. The interpretation of each type T is denoted by $[\![\text{T}]\!]$, where $[\![\text{T}]\!]$ is constructed as the free completely distributive lattice over a certain poset $[T]$, that is, $[\![\text{T}]\!]=\text{FCD}([\text{T}])$. Each closed term of type T will have a denotation in $[\![\text{T}]\!]$.

For every base type T, we assume that the poset $[T]$ is given, and that its ordering $\leq_{[T]}$ agrees with $\sqsubseteq_T$ when restricted to propers of T (this ordering will almost always be the discrete ordering). Each constant c in base type T has a representative $[c]$ in $[T]$, and hence a representative $[\![c]\!]$ in $[\![T]\!]$, where $[\![c]\!]$ is ϕ [c] and $\phi : [T] \xrightarrow{\text{m}} [\![T]\!]$ is the FCD completion.

For each function type $T \to U$, poset $[T \to U]$ is defined to be [T] $\xrightarrow{\text{m}}$ $[\![U]\!]$. Therefore $[\![T \to U]\!]$ is **FCD**($[T] \xrightarrow{\text{m}} [\![U]\!]$). We show later how to construct a representative in $[\![T \to U]\!]$ for each λ-term of type $T \to U$.

We reiterate for clarity that $[T]$ contains a representative for every *proper* of type T, while $[\![T]\!]$ contains denotations for all (closed) terms of type T. The denotations of proper terms will lie in the image of the poset $[T]$ under the FCD completion $\phi : [T] \xrightarrow{\text{m}} [\![T]\!]$, that is, for every (closed) proper term t of type T, $[\![t]\!] = \phi[t]$ where $[t]$ denotes the representation of t in $[T]$. $\sqcap$ and $\sqcup$ in T will be modeled by meet and join in $[\![T]\!]$, respectively.

That the ordering relation in $[\![T]\!]$, for each type T, respects the given ordering relation on the original terms of T (i.e., those employing no choice) follows from a straightforward inductive argument on the grammar of types (i.e., basic types and function types).

6.2 General Axioms

Let us fix on a type T. The five general axioms A1 to A5 translate to the model as follows, where $t, u \in [\![T]\!]$, $S \subseteq [\![T]\!]$, and $X \subseteq \phi[T]$ with $\phi : [T] \xrightarrow{\text{m}} [\![T]\!]$ the FCD completion of $[T]$:

$$
\begin{aligned}
&t \leq u \Leftrightarrow \left(\forall X \subseteq \phi[T] \cdot \textstyle\bigwedge X \leq t \Rightarrow \bigwedge X \leq u\right)\\
&t \leq u \Leftrightarrow \left(\forall X \subseteq \phi[T] \cdot u \leq \textstyle\bigvee X \Rightarrow t \leq \bigvee X\right)\\
&\textstyle\bigwedge S \leq \bigvee X \Leftrightarrow \left(\exists s \in S \cdot s \leq \bigvee X\right)\\
&\textstyle\bigwedge X \leq \bigvee S \Leftrightarrow \left(\exists s \in S \cdot \bigwedge X \leq s\right)\\
&t \leq u \wedge u \leq t \Rightarrow t = u.
\end{aligned}
$$

The translation of each axiom is straightforward. The one point to note is that in the first two translated formulae X is presented as a subset of ϕ [T] rather than $[\![T]\!]$; this is because all bound variables range over propers only. All of the formulae above follow immediately from the properties of $[\![T]\!]$ as the free completely distributive lattice over [T]. See Theorem 5.3 for the first two, and Theorem 5.4 for the following two; the final one is just antisymmetry of the lattice order.

6.3 Function Axioms

Under an environment ρ, a term t of type T has a denotation in $[\![T]\!]$ written $[\![t]\!]\rho$. To minimize cluttering the presentation, we will usually leave environments implicit. When we need to make an environment explicit, we will only expose the assignments relevant in the context. For example, we write $[\![t]\!]^x_v$ for the denotation of u when x is bound to v in $[T]$ (note variables are bound to propers only and so have a representative in $[T]$).

Let us now fix the types T and U and turn to finding a representative in $[\![T \to U]\!]$ for $(\lambda x{:}T \cdot t)$ of type $T \to U$. For the rest of this section, let

$$\begin{aligned} \phi &: [T] \xrightarrow{\mathrm{m}} [\![T]\!] \\ \theta &: [T \to U] \xrightarrow{\mathrm{m}} [\![T \to U]\!] \end{aligned}$$

be the FCD completions of $[T]$ and $[T \to U]$, respectively. We define

$$[\![\lambda x{:}T \cdot t]\!] \;=\; \theta(\lambda v{:}[T] \cdot [\![t]\!]^x_v)$$

(We trust the reader is not put out by our re-use of λ-notation in the semantic domain.) It must be the case that $\lambda v{:}[T] \cdot [\![t]\!]^x_v$ is monotonic; that follows readily from our requirement that the body of λ-abstractions in the language be monotonic in the parameter variable.

To model function application, we introduce

$$\mathsf{app}_{T,U} : [\![T \to U]\!] \xrightarrow{\mathrm{m}} [\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!],$$

such that for $t : T \to U$ and $u : T$

$$[\![t\ u]\!] = \mathsf{app}_{T,U}\ [\![t]\!][\![u]\!].$$

The appropriate definition of app is

$$\mathsf{app}_{T,U} \;=\; \theta^*_{[T] \xrightarrow{\mathrm{m}} [U]}\, \phi^*_{[U]}.$$

We typically omit the subscript in $\mathsf{app}_{T,U}$ when it can be inferred from context. We explain the definition further. Referring to Theorem 5.5, observe that $\phi^*_{[U]}$ has type $([T] \xrightarrow{\mathrm{m}} [\![U]\!]) \xrightarrow{\mathrm{m}} ([\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!])$. Next observe that $\theta^*_{[T] \xrightarrow{\mathrm{m}} [U]}$ has type $(([T] \xrightarrow{\mathrm{m}} [\![U]\!]) \xrightarrow{\mathrm{m}} ([\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!])) \xrightarrow{\mathrm{m}} ([\![T \to U]\!] \xrightarrow{\mathrm{m}} ([\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!]))$. It follows that $\theta^*_{[T] \xrightarrow{\mathrm{m}} [U]}\, \phi^*_{[U]}$ has type $[\![T \to U]\!] \xrightarrow{\mathrm{m}} [\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!]$ as required. The definition of $\mathsf{app}_{T,U}$ requires that $[\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!]$ be a completely distributive lattice (otherwise $\theta^*_{[T] \xrightarrow{\mathrm{m}} [U]}$ is not well defined), and it is so in the standard way, that is, for S a set of functions each of type $[\![T]\!] \xrightarrow{\mathrm{m}} [\![U]\!]$, define $\bigvee S$ to be $\lambda x{:}[\![T]\!] \cdot \bigvee\{f \in S \cdot f\ x\}$, and similarly for meets. For readers who want to gain added confidence in the definition of app, we prove in Figure 6 that $[\![f(x \sqcap y)]\!]$ simplifies to $[f]\,[x] \wedge [f]\,[y]$ for f a proper function of type $T \to U$ and x and y propers of type T; this property is not required in what follows.

The first five function axioms A6 to A10 translate to the model as follows, where $t{:}U$, $\mathrm{V} \subseteq [\![]\!]T$:

(1) $\quad (\exists f{:}\theta[T \to U] \cdot f = [\![\lambda x{:}T \cdot t]\!])$

(2) $\quad (\forall f, g{:}\theta[T \to U] \cdot f \le g \Leftrightarrow (\forall x{:}\phi[T] \cdot \mathsf{app}\ f\ x \le \mathsf{app}\ g\ x))$

(3) $\quad (\forall w{:}\phi[T] \cdot \mathsf{app}\,(\theta(\lambda v{:}[T] \cdot [\![t]\!]^x_v))\, w = [\![t[x \setminus y]]\!]^y_z)$ $\quad$ where $\phi z = w$

(4) $\quad (\forall f{:}\theta[T \to U] \cdot \mathsf{app}\ f\ (\bigwedge V) = \bigwedge\{t \in V \cdot \mathsf{app}\ f\ t\})$

(5) $\quad (\forall f{:}\theta[T \to U] \cdot \mathsf{app}\ f\ (\bigvee V) = \bigvee\{t \in V \cdot \mathsf{app}\ f\ t\})$

The proof of each of these is without difficulty. (1) is equivalent to $[\![\lambda x{:}T \cdot t]\!] \in \theta[T \to U]$, which is the case. (2) can be rewritten by a shift of the bound variable

$$
\begin{array}{ll}
 & [\![f(x \sqcap y)]\!] \\
= & \qquad \text{semantics of function application} \\
 & \mathsf{app}\,[\![f]\!]\,[\![x \sqcap y]\!] \\
= & \qquad \mathsf{app};\ f,\ x, \text{ and } y \text{ proper; denotation of } \sqcap \\
 & \theta^*\phi^*(\theta[f])(\phi[x] \wedge \phi[y]) \\
= & \qquad \text{function composition} \\
 & (((\theta^*\phi^*) \circ \theta)[f])(\phi[x] \wedge \phi[y]) \\
= & \qquad \text{defining property of } \theta^* \ (\phi^* \text{ monotonic by Theorem 5.5}) \\
 & (\phi^*[f])(\phi[x] \wedge \phi[y]) \\
= & \qquad \phi^* g \text{ a homomorphism for all } g \\
 & (\phi^*[f])(\phi[x]) \wedge (\phi^*[f])(\phi[y]) \\
= & \qquad \text{function composition} \\
 & ((\phi^*[f]) \circ \phi)[x] \wedge ((\phi^*[f]) \circ \phi)[y] \\
= & \qquad \text{defining property of } \phi^* \\
 & [f][x] \wedge [f][y]
\end{array}
$$

Fig. 6. Proof of $[\![f(x \sqcap y)]\!] = [f]\,[x] \wedge [f]\,[y]$ for f, x, y proper.

to

$$(\forall f, g{:}[T \to U] \cdot \theta f \leq \theta g \Leftrightarrow (\forall x{:}[T] \cdot \mathsf{app}\,(\theta f)\,(\phi x) \leq \mathsf{app}\,(\theta g)\,(\phi x))),$$

which, by applying the definitions of app, θ^*, and ϕ^*, simplifies to

$$(\forall f, g{:}[T \to U] \cdot f \leq g \Leftrightarrow (\forall x{:}[T] \cdot f\ x \leq g\ x)).$$

This follows immediately from the definition of the ordering on $[T \to U]$. (3) can be equivalently written as follows, by a shift of the bound variable:

$$\big(\forall z{:}[T] \cdot \mathsf{app}\,(\theta(\lambda v{:}[T] \cdot [\![t]\!]^x_v))\,(\phi z) = [\![t[x \backslash y]]\!]^y_z\big).$$

The left-hand side readily simplifies to $(\lambda v{:}[T] \cdot [\![t]\!]^x_v)z$ by applying the definitions of app, θ^*, and ϕ^*, and further simplifies to $[\![t]\!]^x_z$. So it remains to prove

$$[\![t]\!]^x_z = [\![t[x \backslash y]]\!]^y_z.$$

The proof has therefore reduced to a syntactic requirement on substitutivity in the terms of the language. In fact, this is a standard substitution rule that is widely applicable (see Reynolds [1998] for example), and so it is reasonable to require that our language obey it. (4) can be equivalently written as follows, by a simple shift of the bound variable:

$$(\forall g{:}[T \to U] \cdot \mathsf{app}\,(\theta g)\,\big(\bigwedge V\big) = \bigwedge\{t \in V \cdot \mathsf{app}\,(\theta g)\,t\}),$$

which, by applying the definitions of app and θ^*, simplifies to

$$(\forall g{:}[T \to U] \cdot (\phi^* g)\,\big(\bigwedge V\big) = \bigwedge\{t \in V \cdot (\phi^* g)\,t\}),$$

which follows from the fact that $\phi^* g$ is a complete homomorphism. The proof of (5) is dual.

We tackle Axiom A11 indirectly. First, we show that the following theorems hold in the model:

$$\big(\bigsqcap S\big)\,t = \bigsqcap\{f \in S \cdot f\ t\}, \tag{1}$$

$$\big(\bigsqcap S\big)\,t = \bigsqcap\{f \in S \cdot f\ t\}, \tag{2}$$

where $S :\subseteq T \to U$ and $t{:}T$. The two theorems translate into the semantics as follows:

$$\begin{aligned} \mathsf{app}\,(\bigwedge S)\,t &= \bigwedge\{f \in S \cdot \mathsf{app}\,f\,t\} \\ \mathsf{app}\,(\bigvee S)\,t &= \bigvee\{f \in S \cdot \mathsf{app}\,f\,t\}, \end{aligned}$$

where $S \subseteq [\![T \to U]\!]$ and $t \in [\![T]\!]$. These hold since app is by definition $\theta^*\phi^*$ and therefore distributes over meets and joins. We will be able to conclude that A11 holds in the model, that is, $t\,u = (\lambda f{:}T \to U \cdot f\,u)\,t$, if we can show that it can be inferred from (1) and (2). This is a routine exercise that we leave to the reader (hint: express t in normal form using either of the final two theorems in Figure 1).

For Axiom A12, we can define an operator comp to model function composition analogous to our introduction of app, but no new insight is gained and so for brevity we will simply treat A12 as a rewrite rule. We have now shown that the axioms hold in the model and we conclude that the theory is sound.

6.4 A Note on Strictness and Continuity

Functions are distributive with respect to both kinds of choice and hence they are $\bot$- and $\top$-strict, as well as being continuous (i.e., they are distributive with respect to $\bigsqcup\{i \in I \cdot t_i\}$ where the t_i's form an ascending chain, and dually for demonic choice). It might appear at first sight that these properties are stronger than we might wish for in some circumstances, but this is not necessarily so.

It is convenient to use $\bot$ to represent undefined terms such as $3/0$ because $\bot$ comes with nondeterminacy for free. In most commonly-used programming languages, functions are indeed strict with respect to "undefined", corresponding to a call-by-value semantics. A call-by-name semantics requires nonstrict functions which would entail a somewhat different denotational semantics.

The fact that functions are continuous with respect to nondeterministic choice does not mean that they coincide with continuous functions as the term is used in the theory of programming languages. In what follows, we assume familiarity with the standard denotational semantics of programming functions in terms of CPO's (complete partial orders) as described in Winskel [1993], for example. In the CPO semantics, functions of type $U \to V$ are interpreted in the space $\langle U\rangle \xrightarrow{c} \langle V\rangle_\varepsilon$, where the decoration c indicates continuous functions, $\langle U\rangle$ is a CPO associated with the type U, and $\langle U\rangle_\varepsilon$ is $\langle U\rangle$ "lifted" with ε (to avoid confusion, we are here using ε instead of the more commonly used $\bot$ to represent the added bottom element in the CPO semantics). This should be compared with our representation of proper functions in $[U] \xrightarrow{\mathrm{m}} [\![V]\!]$. The main difference between $\langle U\rangle \xrightarrow{c} \langle V\rangle_\varepsilon$ and $[U] \xrightarrow{\mathrm{m}} [\![V]\!]$ is that the latter accommodates monotonic (which includes continuous) functions. To construct a distinguishing example, let us assume the language includes a type $\mathbb{N}^\omega$ consisting of the natural numbers with an added element ω, and for which the refinement order coincides with the usual $\le$ order with additionally $i < \omega$ for all $i \in \mathbb{N}$. Define $f : \mathbb{N}^\omega \to \mathbb{N}^\omega \triangleq \lambda n : \mathbb{N}^\omega \cdot$ **if** $n < \omega$ **then 0 else 1 fi**. Function f is obviously monotonic, but is not continuous as $f\,i = 0$ for all $i < \omega$ but $f\,(\lim_{i<\omega} i) = f\,\omega = 1$.

Unbounded nondeterminacy is a source of discontinuity in this sense [Dijkstra 1982]; consider, for example, the function $F : (\mathbb{Z}\to\mathbb{Z})\to(\mathbb{Z}\to\mathbb{Z})$:

$$\begin{aligned} F \;\triangleq\; \lambda f{:}\ \mathbb{Z}\to\mathbb{Z}\ \cdot\ & \\ \lambda n{:}\mathbb{Z}\ \cdot\ & \textbf{if } n < 0 \textbf{ then } f\,(\sqcap\mathbb{N}) \\ & \textbf{else if } n = 0 \textbf{ then } 47 \\ & \textbf{else } f\,(n-1) \textbf{ fi} \end{aligned}$$

F is not continuous, as can be verified by considering the sequence f_i for $i \in \mathbb{N}$ defined by $f_i \triangleq \lambda n{:}\mathbb{Z} \cdot \textbf{if } 0 \le n < i \textbf{ then } 47 \textbf{ else } \bot \textbf{ fi}$. The discontinuity is attributable to the presence of unbounded nondeterminacy: if $\sqcap\mathbb{N}$ is replaced by $\sqcap\mathbb{N}_K$ where K stands for 516749637, say, and $\mathbb{N}_K$ stands for the set of naturals less than K, then the discontinuity disappears.

7. CONCLUSION AND RELATED WORK

7.1 Summary

We have presented a theory of nondeterministic functions based on what we suggest is a general theory of nondeterminacy. The theory supports unbounded angelic and demonic nondeterminacy. It is presented as a relatively small set of axioms whose soundness we have shown. The soundness proof relies on a new model based on free completely distributive lattices over posets. We have argued that nondeterminacy is a fundamental notion in computing, and that nondeterminacy as it occurs in functions is similar to nondeterminacy as manifested in many other contexts, among them imperative programming, game playing, recursive function theory, data refinement, and process algebras. There are other areas of applicability we have yet to explore, such as logic and constraint programming. We note that the introduction of nondeterminacy into functions makes the methods of the imperative refinement calculus [Back 1980; Morgan 1988; Morris 1987; Back and von Wright 1998] available to functional programming, and indeed we see no impediment into introducing functional refinement without difficulty into the refinement calculus.

7.2 Nondeterminacy in Programming

Nondeterminacy occurs in many branches of computing, and so it is not surprising that it has a vast literature. Here, we will confine our attention to approaches that combine angelic and demonic nondeterminacy, or those that seek to combine nondeterminacy with functions.

Demonic nondeterminacy has had a big presence in imperative programming since [Dijkstra 1976], while angelic nondeterminacy became prominent a little later [Jacobs and Gries 1985; Broy 1986; Morris 1987; Back and von Wright 1990]. Both play a central role in the imperative refinement calculus. Demonic nondeterminacy is used for underspecification, and angelic nondeterminacy is used, for example, in data refinement (e.g., Gardiner and Morgan [1991] and von Wright [1994]) and in modeling competing processes (e.g., Back and von Wright [1990] and Nelson [1992]).

The incorporation of nondeterminacy into functions has been more difficult. There are two obvious ways in which to proceed from classic functions to something richer that will handle choice: one is to move to set-valued functions (e.g., Hughes and O'Donnell [1991]) and the other is to generalize functions to binary relations (e.g., Bird and de Moor [1997]). Set-valued functions are not attractive in a programming calculus. First, they lead to frequent packing and unpacking of values into and out of sets even though often the set is just a singleton set. Second, while the idea is simple for nondeterminacy over discrete types, it is less simple for more complex types. In any event, it does not solve the problem of accommodating demonic and angelic nondeterminacy simultaneously.

The pointfree relational calculus of Bird and de Moor [1997] provides an elegant basis for a calculus of functional programming, and Bird and de Moor [1997] illustrate its application over a wide range of examples. However, the pointfree style is more suited to establishing general laws, and less convenient in constructing individual programs. This has led to some recent effort at employing pointwise relations [de Moor and Gibbons 2000; Naumann 2001a; Martin et al. 2004]. Classical relational approaches cannot accommodate both kinds of nondeterminacy, however. Martin et al. [2004] seek to address this with a more general type of relation called a *multirelation*. Multirelations are significantly more complex than simple relations, to the extent that Martin et al. [2004] recommend staying within classical relations when that it sufficient. The approach put forward in Martin et al. [2004] does not provide operators for angelic or demonic choice. It would seem to be a challenge to merge relations, multirelations, and functions into a seamless calculus that supports functional programming, and ideally imperative programming as well.

There have been quite a few efforts at combining nondeterminacy with functions without moving into sets or relations, (e.g., Partsch [1990], Hoogerwoord [1989], Norvell and Hehner [1993], Hehner [1993], Larsen and Hansen [1996], Lassen [1998], and Hughes and Moran [1995]), but they almost universally impose significant limitations on how nondeterminacy may be employed, or they lack a theory, or they lack a model, or they confine themselves to demonic nondeterminacy. Nondeterminacy sometimes occurs unavoidably in functional programming such as in parallelism (e.g., Bois et al. [2002]) or exception handling (e.g., Jones et al. [1999]), but in these situations the approach has been to cage it rather than embrace it. More ambitious approaches to adding nondeterminacy to functions include Morris and Bunkenburg [1999] and Ward [1994]. Morris and Bunkenburg [1999] includes a theory and a model but deals only with demonic nondeterminacy. (The model of Morris and Bunkenburg [1999] is set-theoretic in which terms are denoted by certain upclosed sets. We surmize that it could be equivalently expressed in terms of free complete semilattices over a poset.) Ward [1994] supports both kinds of nondeterminacy, but offers only a model and not a theory. Definitions are introduced and laws are proved with respect to the model. The laws as presented are not comprehensive, and harbor inconsistency [Morris and Bunkenburg 2002]. Both Morris and Bunkenburg [1999] and Ward [1994] were motivated by a desire to introduce the notion of refinement into functions. Naumann [2001b] gives a presentation of the refinement calculus with higher-order procedures added, and in

which refinement of expressions is considered in the special case of procedure expressions.

de Moor and Gibbons [2000] offers nondeterminacy (of just one kind) within a functional programming setting primarily by providing noninjective pattern matching in function parameters. Naumann [2001a] identifies some shortcomings in its relational semantics, however, and sets out to address them, giving in addition a model based on predicate transformers. Naumann shows that it is possible to define angelic and demonic choice operators in the language (even unbounded ones), and presents many properties of them. The primary focus of Naumann [2001a] is on noninjective patterns, and in offering simulations that connect functional, relational, and predicate transformer interpretations of lambda and pattern terms.

7.3 Models for Nondeterminacy

Free completely distributive lattices over posets represent a new model for nondeterminacy. They do not appear to be well documented in the literature, other than as cited in the proof of Theorem 5.1. They are not mentioned in the encyclopaedic paper of Freese et al. [1995]. The thesis of Bonsangue [1998] uses the simpler structure of the free completely distributive lattices over a *set* to model a first-order language with both kinds of nondeterminacy. In Cattani and Winskel [1996] and Nygaard and Winskel [2002, 2004], concurrent languages with one kind of nondeterminacy are given a denotational semantics using *presheaf* models. Whereas we take a poset as the base of our free stucture, presheaf models generalise the base to arbitrary categories, but only considering non-strict join-preserving maps and not complete maps.

Power domains [Plotkin 1976; Smyth 1978; Smyth 1983] have been used for many years in denotational models of nondeterminacy. See Heckmann [1991b] for a comprehensive account in an algebraic setting. The classical power domains deal with bounded nondeterminacy. Apt and Plotkin [1986] deals with unbounded demonic nondeterminacy but limited to discretely-ordered domains; see Laird [2006] for further developments of this approach using bidomains [Berry 1978]. The presentation of power domains as free structures is developed in Hennessy and Plotkin [1979], Main [1985], Hoofman [1987], and Heckmann [1991b]; there is a comprehensive account in Abramsky and Jung [1994].

Intimately related to bounded nondeterminacy is continuity of the functions being modelled [Apt and Plotkin 1986; Dijkstra 1976]. Power domains describe nondeterminacy in continuous functions as they occur in programming languages, whereas we have placed our nondeterministic functions in the more general setting of specification languages which are monotonic but not necessarily continuous. The base of our construction is a poset in contrast with the directed complete posets of power domain theory, and our injection functions are not continuous. On the other hand, our construction enjoys many of the properties of a power domain, and might liberally be viewed as one. In particular, our structure can be expressed as a Kleisli triple and so admits of a treatment in terms of monads (see Heckmann [1991c]).

It is known (see Flannery and Martin [1990] and Heckmann [1991a]) that two of the classic power domains, the lower and upper ones, commute to yield a free structure called a *frame* that can encapsulate both kinds of nondeterminacy. However, frames have bounds on the nondeterminacy, take complete posets (CPOs) rather than posets as their starting point, and do not possess all the distribution properties we require. We are not aware of work seeking to use frames in a denotational semantics of functions with both kinds of nondeterminacy, or as the basis for an algebra of nondeterminacy.

Harmer and McCusker [1999] and Levy [2005] model nondeterminacy using game semantics, but limited to demonic nondeterminacy. Other models that handle both kinds of nondeterminacy in a functional setting include multirelations [Martin et al. 2004] and predicate transformers [Ward 1994; Naumann 2001a]. Actually, it is known that multirelations are in essence the same as predicate transformers (see Hesselink [2004]). Additionally, there is an isomorphism between predicate transformers and denotational semantics including power domains [Smyth 1983; Plotkin 1979; Apt and Plotkin 1986; Bonsangue and Kok 1994], and (as a referee has pointed out to us) there is a similarity between Theorem 5.2 above and the well-known factorization of predicate transformers [Hesselink 1990; Gardiner et al. 1994]. It suggests that a detailed comparison of the various models would be fruitful, although we do not pursue it further here.

ACKNOWLEDGMENTS

Many people have helped us in this work, particularly in suggesting many improvements to the technical presentation. We especially thank the anonymous referees who went to great pains to understand the work in detail and made many important observations and suggestions, and in addition David Naumann and Eric Hehner. The research on process algebras has been joint work with Andrew Butterfield and Arthur Hughes.

REFERENCES

Abramsky, S. and Jung, A. 1994. Domain theory. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Vol. 3. Clarendon Press, 1–168.

Apt, K. R. and Plotkin, G. D. 1986. Countable nondeterminism and random assignment. *J. ACM 33,* 4, 724–767.

Back, R.-J. R. 1980. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam.

Back, R.-J. R. and von Wright, J. 1990. Duality in specification languages: a lattice-theoretical approach. *Acta Inf. 27,* 7, 583–625.

Back, R.-J. and von Wright, J. 1998. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, New York.

Bartenschlager, G. 1995. Free bounded distributive lattices over finite ordered sets and their skeletons. *Acta Math. Univ. Comen. 64*, 1–23.

Berry, G. 1978. Stable models of typed lambda-calculi. In *Proceedings of the 5th Colloquium on Automata, Languages and Programming*. Lecture Notes in Comput. Science, vol. 62. Springer-Verlag, New York, 72–89.

Bird, R. and de Moor, O. 1997. *Algebra of Programming*. Prentice Hall, London. ISBN 0-13-507245-X.

BIRKHOFF, G. 1967. *Lattice Theory*, 3rd ed. Colloquium Publications, vol. 25. American Mathematical Society.

BOIS, A. R. D., POINTON, R., LOIDL, H.-W., AND TRINDER, P. 2002. Implementing declarative parallel bottom-avoiding choice. In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing*, A. F. de Souza, Ed. IEEE Computer Society Press, Los Alamitos, CA.

BONSANGUE, M. 1998. *Topological Duality in Semantics*. Electronic Notes in Theoretical Computer Science, vol. 8. Elsevier, Amsterdam.

BONSANGUE, M. M. AND KOK, J. N. 1994. The weakest precondition calculus: Recursion and duality. *Formal Asp. Comput. 6*, 788–800.

BOUTE, R. T. 2005. Functional declarative language design and predicate calculus: a practical approach. *ACM Trans. Program. Lang. Syst. 27,* 5, 988–1047.

BROY, M. 1986. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoret. Comput. Sci. 45,* 1, 1–61.

CATTANI, G. L. AND WINSKEL, G. 1996. Presheaf models for concurrency. In *Computer Science Logic*, D. van Dalen and M. Bezem, Eds. Lecture Notes in Computer Science, vol. 1258. Springer, 58–75.

COUSOT, P. 1996. Abstract interpretation. *ACM Comput. Surv. 28,* 2, 324–328.

DAVEY, B. AND PRIESTLEY, H. 2002. *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press.

DE MOOR, O. AND GIBBONS, J. 2000. Invited talk: Pointwise relational programming. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science. Vol. 1816. Springer-Verlag, New York, 371–390.

DEROEVER, W.-P. AND ENGELHARDT, K. 1999. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA.

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

DIJKSTRA, E. W. 1982. The equivalence of bounded nondeterminacy and continuity. In *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York.

FLANNERY, K. E. AND MARTIN, J. J. 1990. The Hoare and Smith power domain constructors commute under composition. *J. Comput. Syst. Sci. 40,* 2, 125–135.

FREESE, R., JEZEK, J., AND NATION, J. 1995. *Free Lattices*. Mathematical Surveys and Monographs, vol. 42. American Mathematical Society.

GARDINER, P. H. B., MARTIN, C. E., AND DE MOOR, O. 1994. An algebraic construction of predicate transformers. *Sci. Comput. Program. 22,* 1-2, 21–44.

GARDINER, P. H. B. AND MORGAN, C. C. 1991. Data refinement of predicate transformers. *Theoret. Comput. Sci. 87*, 143–162.

GRIES, D. AND SCHNEIDER, F. B. 1993. *A Logical Approach to Discrete Math*. Springer-Verlag, New York.

HARMER, R. AND MCCUSKER, G. 1999. A fully abstract game semantics for finite nondeterminism. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 422–430.

HECKMANN, R. 1991a. Lower and upper power domain constructions commute on all cpos. *Inf. Process. Lett. 40,* 1, 7–11.

HECKMANN, R. 1991b. Power domain constructions. *Sci. Comput. Program. 17,* 1-3, 77–117.

HECKMANN, R. 1991c. An upper power domain construction in terms of strongly compact sets. In *MFPS*, S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 598. Springer-Verlag, New York, 272–293.

HEHNER, E. C. R. 1993. *A Practical Theory of Programming*. Springer Verlag, New York, London. 2nd ed. 2004 at http://www.cs.toronto.edu/ hehner/aPToP/.

HENNESSY, M. AND PLOTKIN, G. D. 1979. Full abstraction for a simple parallel programming language. In *MFCS*, J. Becvár, Ed. Lecture Notes in Computer Science, vol. 74. Springer, 108–120.

HESSELINK, W. H. 1990. Modalities of nondeterminacy. In *Beauty is our Business: A Birthday Salute to E.W. Dijkstra*, W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, Eds. Springer-Verlag, New York, 182–192.

HESSELINK, W. 2004. Multirelations are predicate transformers. Tech. rep., Dept. of Computing Science, University of Groningen, The Netherlands.

HOARE, C. A. R. 1984. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ.

HOARE, C. A. R., HE, J., AND SANDERS, J. W. 1987. Prespecification in data refinement. *Inf. Process. Lett. 25,* 2, 71–76.

HOOFMAN, R. 1987. Powerdomains. Tech. Rep. RUU-CS-87-23, Institute of Information and Computing Sciences, Utrecht University.

HOOGERWOORD, R. R. 1989. The design of functional programs: a calculational approach. Ph.D. thesis, Technische Universiteit Eindhoven.

HUGHES, J. AND MORAN, A. 1995. Making choices lazily. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 108–119.

HUGHES, J. AND O'DONNELL, J. 1991. Nondeterministic functional programming with sets. In *Proceedings of the 4th Higher Order Workshop Banff 1990* (Sept. 10–14, 1990, Alberta, Bc, Canada). Springer-Verlag, New York.

JACOBS, D. AND GRIES, D. 1985. General correctness: A unification of partial and total correctness. *Acta Inf. 22,* 1, 67–83.

JONES, S. P., REID, A., HENDERSON, F., HOARE, T., AND MARLOW, S. 1999. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM, New York, 25–36.

LAIRD, J. 2006. Bidomains and full abstraction for countable nondeterminism. In *Proceedings of the Foundations of Software Science and Computation Structures 2006*. Lecture Notes in Computer Science, vol. 3921. Springer-Verlag, New York.

LARSEN, P. G. AND HANSEN, B. S. 1996. Semantics of under-determined expressions. *Form. Asp. Comput. 8,* 1, 47–66.

LASSEN, S. B. 1998. Relational reasoning about functions and nondeterminism. Ph.D. dissertation. Dept of Computer Science, University of Aarhus.

LEVY, P. B. 2005. Infinite trace equivalence. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science, vol. 155. Springer-Verlag, New York, 195–209.

MAIN, M. G. 1985. Free constructions of powerdomains. In *Mathematical Foundations of Programming Semantics*, A. Melton, Ed. Lecture Notes in Computer Science, vol. 239. Springer-Verlag, New York, 162–183.

MARTIN, C. E., CURTIS, S. A., AND REWITZKY, I. 2004. Modelling nondeterminism. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, D. Kozen and C. Shankland, Eds. Lecture Notes in Computer Science, vol. 3125. Springer-Verlag, New York, 228–251.

MORGAN, C. 1988. The specification statement. *ACM Trans. Prog. Lang. Syst. 10*, 403–419.

MORGAN, C. 1990. *Programming from Specifications*. Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

MORGAN, C. AND GARDINER, P. H. B. 1991. Data refinement by calculation. *Acta Informatica 27*, 481–503.

MORRIS, J. M. 1987. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Prog. 9*, 287–306.

MORRIS, J. M. 2004. Augmenting types with unbounded demonic and angelic nondeterminacy. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, D. Kozen and C. Shankland, Eds. Lecture Notes in Computer Science, vol. 3125. Springer-Verlag, New York, 274–288.

MORRIS, J. M. AND BUNKENBURG, A. 1999. Specificational functions. *ACM Trans. Prog. Lang. Syst. 21*, 677–701.

MORRIS, J. M. AND BUNKENBURG, A. 2002. A source of inconsistency in theories of nondeterministic functions. *Sci. Comput. Program. 43,* 1, 77–89.

MORRIS, J. M., BUNKENBURG, A., AND TYRRELL, M. 2008. Term transformers: A new approach to state. submitted.

MORRIS, J. M. AND TYRRELL, M. 2007. Dual unbounded nondeterminacy, recursion, and fixpoints. *Acta Inf. 44,* 5, 323–344.

NAUMANN, D. A. 2001a. Ideal models for pointwise relational and state-free imperative programming. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM, New York, 4–15.

NAUMANN, D. A. 2001b. Predicate transformer semantics of a higher-order imperative language with record subtyping. *Sci. Comput. Prog. 41,* 1, 1–51.

NELSON, G. 1992. Some generalizations and applications of Dijkstra's guarded commands. In *Programming and Mathematical Method*, M. Broy, Ed. NATO ASI Series F: Computer and Systems Sciences, vol. 88. Springer-Verlag New York.

NORVELL, T. S. AND HEHNER, E. C. R. 1993. Logical specifications for functional programs. In *Proceedings of the 2nd International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 669. Springer-Verlag, New York, 269–290.

NYGAARD, M. AND WINSKEL, G. 2002. Linearity in process languages. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Press, Los Alamitos, CA, 433–441.

NYGAARD, M. AND WINSKEL, G. 2004. Domain theory for concurrency. *Theoret. Comput. Sci. 316,* 1, 153–190.

PARTSCH, H. A. 1990. *Specification and Transformation of Programs*. Springer-Verlag, New York.

PLOTKIN, G. 1976. A powerdomain construction. *SIAM J. Comput. 5,* 3, 452–487.

PLOTKIN, G. 1979. Dijkstra's predicate transformers and smyth's powerdomains. In *Proceedings of the Copenhagen Winter School on Abstract Software Specifications*, D. Bjorner, Ed. Lecture Notes in Computer Science, vol. 96. Springer-Verlag, New York, 527 – 553.

REYNOLDS, J. C. 1998. *Theories of Programming Languages*. Cambridge University Press, Cambridge, UK.

ROSCOE, A. W. 1998. *The Theory and Practice of Concurrency*. Prentice-Hall, Englewood Cliffs, NJ.

SMYTH, M. B. 1978. Power domains. *J. Comput. Syst. Sci. 16,* 1, 23–26.

SMYTH, M. B. 1983. Power domains and predicate transformers: A topological view. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, J. Diaz, Ed. Lecture Notes in Computer Science, vol. 153. Springer-Verlag, London, UK, 662–675.

SPIVEY, J. 1988. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, UK.

TUNNICLIFFE, W. R. 1985. The free completely distributive lattice over a poset. *Algebra Univ. 21*, 133–135.

TYRRELL, M., MORRIS, J. M., BUTTERFIELD, A., AND HUGHES, A. 2006. A lattice-theoretic model for an algebra of communicating sequential processes. In *Proceedings of the 3rd International Colloquium on Theoretical Aspects of Computing*, K. Barkaoui, A. Cavalcanti, and A. Cerone, Eds. Lecture Notes in Computing Science, vol. 4281. Springer-Verlag, New York.

VON WRIGHT, J. 1994. The lattice of data refinement. *Acta Inf. 31*, 105–135.

WARD, N. 1994. A refinement calculus for nondeterministic expressions. Ph.D. dissertation, University of Queensland.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

WOODCOCK, J. AND LOOMES, M. 1988. *Software engineering mathematics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Received December 2005; revised October 2006, July 2007; accepted February 2008