



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# On Research: incremental semantics

---

**J. W. Sanders**

May 2008

## UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research **[R]**, Technical **[T]**, Compendia **[C]** or Administrative **[A]**. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations  
University

**UNU-IIST**

**International Institute for  
Software Technology**

P.O. Box 3058  
Macao

---

# On Research: incremental semantics

---

**J. W. Sanders**

## **Abstract**

The purpose of the UNU-IIST Research Day (28-ii-2008) was for us to tell each other about our research: the ideas behind it, why they are important and where they are leading. The ‘metascience’ was as important as the science and provided UNU-IIST fellows with the opportunity to be trained in that aspect of the management of research. Judging by the amount of discussion it provoked, the day was a success. This paper is an elaboration of the talk given by the author, providing more detail than it was possible to give in the talk, but which is necessary if the reader is to evaluate the case being made.

The paper makes the case for the incremental approach to program semantics using Galois connections. It considers a sustained case study, that of sequential programs and their related (specification) commands, with the final addition of probabilistic choice. ‘Structural’ decisions are discussed throughout and the conclusion reflects on several issues that arose from the day.

**Jeff Sanders** is Principal Research Fellow at UNU-IIST. His interests lie largely in Formal Methods.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Our new coffee machine</b>	<b>3</b>
<b>3</b>	<b>Modelling system development</b>	<b>6</b>
<b>4</b>	<b>Case study: sequential systems</b>	<b>8</b>
4.1	Modelling nontermination . . . . .	8
4.1.1	Behaviour . . . . .	8
4.1.2	Model . . . . .	9
4.1.3	Semantics . . . . .	10
4.2	Modelling nondeterminism . . . . .	13
4.2.1	Behaviour . . . . .	13
4.2.2	Model . . . . .	14
4.2.3	Semantics . . . . .	17
4.3	Modelling angelic choice and enabledness . . . . .	19
4.3.1	Behaviour . . . . .	20
4.3.2	Relational model . . . . .	21
4.3.3	Relational semantics . . . . .	21
4.3.4	Transformer model . . . . .	23
4.3.5	Transformer semantics . . . . .	25
4.4	Modelling probability . . . . .	26
4.4.1	Behaviour . . . . .	26
4.4.2	Distributional model . . . . .	28
4.4.3	Distributional semantics . . . . .	31
4.4.4	Transformer model . . . . .	31
4.4.5	Transformer semantics . . . . .	33
<b>5</b>	<b>Research agenda</b>	<b>33</b>
<b>6</b>	<b>Conclusion</b>	<b>35</b>



## 1 Introduction

The primary purpose of the UNU-IIST Research Day was to place emphasis on our research, for a change. To talk to each other about the things we spend much of our time thinking about; to explain the ideas that excite us and why; and to speculate where they are taking us and why that is desirable. So, far from being a day of standard seminars, it provided an opportunity for fellows to be involved in a discussion of research that concerned them and to learn about its management: a kind of *metascience*.

Scientists don't often have cause to reflect in public on the factors underpinning their work. Prize-acceptance speeches provide one important source. But, grouped with Mathematicians, we Computer Scientists have been excluded since before the birth of our discipline from giving Nobel acceptance speeches. Turing award speeches provide a valuable alternative; presumably for that reason the first twenty years' speeches have been collected [1].

Typically, a graduate student is expected to learn metascience whilst learning research—not an easy task. Perhaps that course is typically followed because in the past little attempt has been made to separate metascience from science; many scientists may even feel unsure about discussing it. The research day was an attempt to change that. This paper is an elaboration of a talk presented by the author. Its method and content are as unusual as its subject of 'metascience'. An approach to the study of program semantics is discussed. Thus it is the relationship between results that forms the plot in this brief drama: of why a certain approach is worth following. The backdrop consists of theories of programs; and the rôle of protagonist is played by the Galois connection which performs in a succession of acts involving incrementally more complex languages for sequential programming. Because the spotlight is firmly on matters of approach, proofs are frequently concealed in the darkness off stage.

The guiding technical principle of the approach is to model new behaviour by reusing existing wisdom (as much as possible). We present a sequence of behaviours in order make a convincing case for that principle. They are presented in the context of sequential programming and are:

1. firstly (and classically) a property that results from considering the distinction between programs (*i.e.* computable or recursive functions) and functions: *nontermination*;
2. the property that results from the practice of Software Engineering and its consideration of designs more abstract than code: *bounded nondeterminism*;
3. the properties that arise in Formal Methods to strengthen the utility and scope of Software Engineering: *unbounded nondeterminism*, *angelic choice* and *enabledness*;
4. finally an example of an extension to more specialised but realistic behaviour: *probabilism*.

With each step in the hierarchy our strategy is the same: to decide informally what new observations we wish to make of a computation; to capture the essence of the new behaviour in laws; to show that the laws are consistent by formulating a semantic model; and to lift as much reasoning as possible from lower in the hierarchy to the current step.

But what kind of object is it that arises at each step? It is a space of programs or their extensions, commands, partially-ordered under the binary ‘conforms to’ relation  $\sqsubseteq$ . That structure provides the foundation for:

- the notion of an implementation to meet a specification and hence the verification of an implementation or design against its specification; see for example the texts on VDM [28] and Z [45];
- the stepwise development of an implementation from its specification; see for example the texts by Dijkstra, [13], Kaldewij [29] and Morgan [32];
- a model to abstract an implementation, and then for *model checking* to confirm that the implementation satisfies some property; for example see Clarke, Grumberg and Peled’s book [9].

The result of that approach is a ‘machine’: a technique with a mathematical foundation and wide applicability that produces results not easily otherwise obtained. ‘Machines’ are important in academia and need to be identified, evaluated and understood. That is attempted in this paper for the ‘machine’ which might be called ‘the incremental approach to discrete systems’; or ‘the use of Galois connections in the theory of programming’; or, more specifically, UTP: ‘unifying theories of programming’ (see the original text by Hoare and He [22] and the more recent workshop proceedings [15]).

It is hoped that the reader will focus on the topics emphasised in the research-day talk:

- how novel behaviour has at each step been captured (here by laws);
- how details have been accumulated incrementally from step to step (using Galois connections that preserve further properties); and
- the underlying ‘machine’ (with the prospect of reusing it).

To encourage readers to apply this approach, and make the transition from reader to actor, the last increment in the case study is expressed as an exercise. Furthermore, suggestions are included for new topics. How, for instance, might the ‘machine’ be applied to more novel situations, like process algebra, real-time systems, rCOS, quantum computation, asynchronous systems, and so on?

This seminar has been made into a report so that the reader wishing to make a critical evaluation of its generalities is able to do so on the basis of the all-important details (for which there was insufficient time in the talk). Only when the details permit it can an incremental approach be taken: *i.e.* only when the features can be ‘teasing out’ and subsequently presented in an accumulative manner. It has not been easy to decide how much detail to give. Too little is to sell short the technique and render the case unconvincing; too much risks overwhelming the reader. If there is too much detail, it is hoped that the earnest reader will take the trouble to be selective and not lose sight of the plot, which is a (formal) incremental approach to the discovery and exposition of complex (discrete) phenomena.

Good luck in trying this approach on your own work. But more importantly, good luck in learning something of ‘metascience’ and the management of your own research in your own area!

## 2 Our new coffee machine

The level of water in the new *real coffee* machine is monitored by the machine itself. Such a machine may need to know how much water is in the tank: firstly to know if it can service the current task (different tasks require different amounts of water); and secondly to ensure it is not overfilled (when it may be dangerous to operate). Imagine that the water tank contains a vertical column of equi-distant water sensors. We represent the sensors as members of an abstract space  $A$  which we take to be the integers (well, it saves having to worry about how big the machine is ...<sup>1</sup>) with typical member  $n$

$$A :: n : \mathbb{Z}.$$

But the water level is actually analogue and so is modelled by a real number which we take to have typical member  $r$ . That space provides more information about water level and so is regarded as ‘concrete’

$$C :: r : \mathbb{R}.$$

A decision about water level is made on the basis of a reading that compares the actual water level  $r : C$  with that determined by the sensors,  $n : A$ . Since the types of  $A$  and  $C$  are distinct, they must first be related. Fortunately there is a natural embedding

$$\varepsilon : \mathbb{Z} \rightarrow \mathbb{R}$$

that identifies each integer  $n$  with its whole real number equivalent  $\varepsilon.n$ .

Now the comparison can be made either in  $A$ , the integers, or  $C$ , the reals. If it is made in  $C$  then it is made on the basis of an inequality in the reals

$$\varepsilon.n \leq r.$$

But to make it in  $A$  is to make it on the basis of an inequality in the integers

$$n \leq \pi.r$$

where  $\pi$  is *some* projection converting a real number to an integer.

It should not matter which of the two levels of abstraction  $A$  and  $C$  is used for making the decision. Firstly, measurement might be performed in the reals, in  $C$ , using the relationship between the ‘sensed level’—the embedding of an integer in  $\mathbb{R}$ —and the actual water level—a real number. Or it might be performed in the integers, in  $A$ , using the relationship there between an integer—the sensed level—and the projection of the real number—the actual level. The two measurements give consistent decisions about the water level iff this equivalence holds

$$(1) \quad \varepsilon.n \leq r \equiv n \leq \pi.r.$$

<sup>1</sup>But our interest here is not, of course, in an accurate model but in the principles on which models should be based.

But what is the projection  $\pi.r$ ? The most obvious choice, in the context of water level, seems to be the best (or largest) approximation in  $\mathbb{Z}$  to  $r$  (on the grounds that a sensor covered by water is activated, and the highest activated sensor is the ‘best’ approximation to the level):

$$(2) \quad \pi.r = \vee\{n : C \mid \varepsilon.n \leq r\},$$

where  $\vee$  denotes maximum (with respect to ordering  $\leq$ ) in  $\mathbb{Z}$ . For each real  $r$  the supremum on the right-hand side is nonempty and well defined<sup>2</sup> in  $\mathbb{Z}$ .

In other words, the projection is the ‘floor’ (or ‘integer part’) function:

$$\pi.r = \lfloor r \rfloor.$$

But why? That might seem obvious when only rising water is considered. But what if the water level ebbs? Perhaps, by symmetry, the projection could be the ceiling,  $\pi.r = \lceil r \rceil$ , or the average of the two. Let us see why  $\pi$  in (1) *must be* the floor function.

Firstly—and perhaps surprisingly— $\pi$  is uniquely determined by (1). For if  $\pi$  and  $\pi'$  are two solutions and  $n : \mathbb{Z}$  and  $r : \mathbb{R}$  are arbitrary then

$$\begin{aligned} n &\leq \pi.r && \\ \equiv &&& \text{Equivalence (1) for } \pi \\ \varepsilon.n &\leq r && \\ \equiv &&& \text{Equivalence (1) for } \pi' \\ n &\leq \pi'.r. \end{aligned}$$

By instantiating  $n$  to be  $\pi'.r$  and generalising over  $r$  we infer  $\pi' \leq \pi$ ; the converse follows by symmetry or the opposite instantiation.

Now, showing that the projection (2) satisfies (1) will, by that uniqueness, complete our claim that (1) forces the projection to be the floor function. We establish Equivalence (1) for floor by showing each of its two implications, that is, by reasoning cyclically.

$$\begin{aligned} n &\leq \pi.r && \\ \equiv &&& \text{Definition (2) of } \pi \\ n &\leq \vee\{m : \mathbb{Z} \mid \varepsilon.m \leq r\} \end{aligned}$$

---

<sup>2</sup>Nonemptiness is necessary because there is no least integer hence no empty supremum. In the context of the coffee machine a more accurate model would consist of only finitely many discrete sensors, and if one were placed at the lowest level then it would act as minimum.

$$\begin{array}{ll}
\Rightarrow & \text{if } \varepsilon \text{ is monotone} \\
\varepsilon.n \leq \varepsilon. \vee \{m : \mathbb{Z} \mid \varepsilon.m \leq r\} & \\
\equiv & \text{if } \varepsilon \text{ preserves such suprema} \\
\varepsilon.n \leq \vee \{\varepsilon.m : \mathbb{R} \mid \varepsilon.m \leq r\} & \\
\Rightarrow & \text{definition of supremum} \\
\varepsilon.n \leq r & \\
\Rightarrow & \text{definition of supremum again} \\
n \leq \vee \{m : \mathbb{Z} \mid \varepsilon.m \leq r\} & \\
\equiv & \text{Definition (2) of } \pi \\
n \leq \pi.r. &
\end{array}$$

That reasoning uses (a) the existence of (only) the suprema required by Definition (2) of  $\pi$  and (b) for  $\varepsilon$  to preserve them.<sup>3</sup> In other words this innocent calculation actually proves an old result:

**Theorem 1** Suppose  $(A, \leq)$  and  $(C, \leq)$  are partially ordered spaces and  $\varepsilon : A \rightarrow C$ . Suppose that suprema  $\vee E$  exist in  $A$  and that  $\varepsilon$  preserves them. Then (2) defines the unique function  $\pi : C \rightarrow A$  satisfying (1).  $\square$

In summary, the following (ancient) concept formulated by Ore [38] has been motivated. As we shall see, it forms the basis for comparing different levels of abstraction when each is endowed with a ‘refinement’ ordering.

**Definition 1** If  $(A, \leq)$  and  $(C, \leq)$  are partially ordered spaces then  $\varepsilon : A \rightarrow C$  and  $\pi : C \rightarrow A$  form a *Galois connection* iff Equivalence (1) holds, in which case we write (when there is no need to make the orders explicit)

$$gc(\varepsilon, \pi; A, C).$$

A graphical representation is given in Figure 1. The functions  $\varepsilon$  and  $\pi$  are called *adjoints* (or *weak inverses*). If  $\varepsilon$  is an injection then instead we write  $ge(\varepsilon, \pi; A, C)$  and call the connection a *Galois embedding*.  $\square$

We shall call a function  $\varepsilon$  under the conditions of Theorem 1 *Galois*, in view of the fact that there is a unique function  $\pi$  forming a Galois connection with  $\varepsilon$  between the underlying partial orders.

The symmetry between  $\varepsilon$  and  $\pi$  reflected in that definition of Galois connection is violated in the definition of Galois embedding and, for example, in the case of the coffee machine where  $\varepsilon$  is injective and

<sup>3</sup>It is a standard fact—and simple to show—that preserving suprema implies monotonicity (because  $m \leq n$  iff  $m \vee n = n$ ).

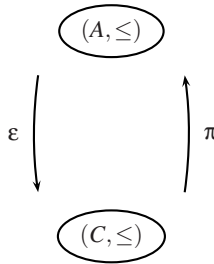


Figure 1: A Galois connection  $gc(\varepsilon, \pi; A, C)$  between abstract and concrete spaces.

$\pi$  surjective. The equivalence between  $\varepsilon$  being injective and  $\pi$  being surjective, with a justification for the title ‘weak inverses’, and the fact that the adjoints map extreme values to extreme values, follow by elementary reasoning:

**Theorem 2** If  $gc(\varepsilon, \pi; A, C)$  then

1.  $\varepsilon \circ \pi \leq id_C$  and  $\pi \circ \varepsilon \geq id_A$  (where  $id_X$  denotes the identity function on  $X$ );
2.  $\varepsilon$  is injective iff  $\pi$  is surjective, in which case  $\pi \circ \varepsilon = id_A$ ;
3.  $\varepsilon$  preserves suprema and  $\pi$  preserves infima; if  $A$  has a minimum then so does  $C$  and  $\varepsilon$  maps the former to the latter; similarly, if  $C$  has a maximum then so does  $A$  and  $\pi$  maps the former to the latter. □

### 3 Modelling system development

Apart from their intrinsic interest, models of system development are important for several reasons.

- So a design can be verified against a specification: without an understanding of conformance, what does a specification mean?
- For abstract interpretation and model checking, firstly of the abstract model and secondly of the property being checked against it.
- For incremental system comprehension by layers of successively finer detail. This approach has been traditionally used qualitatively to describe complex software (for example operating systems [31]); our task is now to exploit a quantitative version.
- For stepwise system derivation, of the kind championed over 30 years ago by Dijkstra [13] but now feasible for systems.

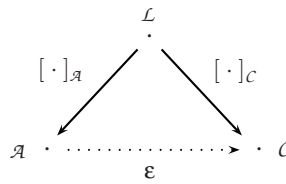


Figure 2: Lifting the semantics of a language  $\mathcal{L}$  from  $\mathcal{A}$  to  $\mathcal{C}$  using a Galois connection.

- For incorporating new behaviours when intuition alone is too risky as a basis for programming. Examples are provided by concurrency, realtime, probability and machine learning.
- To facilitate program analysis of the usual kinds: data-flow, constraint-based, abstract interpretation, type systems and effect systems [37].

Common to those situations is a formalism based on a partially-ordered space  $(\mathcal{X}, \sqsubseteq)$  of programs, or their extensions, based on a notion  $\sqsubseteq$  of conformance between elements of  $\mathcal{X}$ . It is within that space that stepwise derivations are possible

$$\text{Specification} = \text{Design}_0 \sqsubseteq \text{Design}_1 \sqsubseteq \dots \sqsubseteq \text{Design}_n = \text{Implementation}$$

including algorithmic refinements (over the same state space) and data refinements (by data representation). The important feature of  $\mathcal{X}$  is that it be powerful enough to express specifications as well as code, thus providing a uniform notation for stepwise system development.

That leads us to extend code (in which the implementation is expressed) in several increments. The resulting family of languages is summarised in Figure 3. Several of the features are not well known, and interact in subtle ways with each other (like nondeterminism and sequential composition in the presence of angelic choice; or like nondeterminism and probabilism). As the languages become more complex, so does their semantics.

By viewing the various languages hierarchically we are able to start with the simplest and import the semantics of each level in the hierarchy to the next. The technique for doing so is of course the Galois connection. It is because refinement consists of the removal of nondeterminism that each partially ordered space  $(\mathcal{X}, \sqsubseteq)$  is amenable to a Galois connection in which the embedding  $\varepsilon$  automatically preserves nondeterminism (being universally  $(\sqsubseteq, \sqsubseteq)$ -junctive). But in lifting the semantics from one level to the next, it is required that  $\varepsilon$  preserve further combinators, like sequential composition

$$\varepsilon.(r \circ s) = \varepsilon.r \circ \varepsilon.s.$$

For then  $\varepsilon$  can be used to lift the behaviour of a sequential composition from one level to the next in the hierarchy; see Figure 2.

It will then be important to characterise the lifted space  $\text{ran } \varepsilon$  as a subset of the new level and furthermore to determine—if possible—the manner in which it generates the new level. For that determines the semantics at the new level.

Language	Semantic models	Novel feature
<i>Predet</i>	$\mathcal{P}, \mathcal{Q}$	nontermination
<i>Prog</i>	$\mathcal{D}, \mathcal{T}$	nondeterminism
<i>Comm</i>	$\mathcal{R}, \mathcal{T}$	angelic choice
<i>Prob</i>	$\mathcal{H}, \mathcal{J}$	probabilism

Figure 3: The family of languages (over state space  $X$ ) together with their semantic models and novel features: predetermined functions; finitely nondeterministic programs; arbitrary (*i.e.* unboundedly non-deterministic and angelic) commands; probabilistic programs.

## 4 Case study: sequential systems

The purpose of this long section is to develop a single case study using the incremental method. Successively more complex behaviours are incorporated using the approach outlined in the previous section. The hope is that by considering a familiar example, that of sequential programs, the student can learn how to apply the technique to more novel situations (an example of which, probability, is included as the last increment here).

The first kinds of behaviour, nontermination and nondeterminism, will probably be familiar. But extensions, to angelic choice, unenabled commands and then to probabilistic choice, are likely to be less familiar. (The range of behaviours, to be introduced as we go—incrementally—is summarised in Figure 3.) So after all, this single case study may indeed indicate how to use the incremental technique to capture novel behaviour.

### 4.1 Modelling nontermination

The origins of formal methods lie in the work of Cantor with his use of set theory to model the hierarchy of mathematical concepts. In particular a function—that takes an argument of type  $X$  and returns a result of type  $Y$ —is modelled as a set of pairs satisfying ‘well-definedness’: every argument is associated with exactly one result. The consequence is the type  $X \rightarrow Y$ . But with total functions the partial order of conformance of one function to another degenerates to equality:

$$f \sqsubseteq f' \hat{=} f = f'$$

where the equality is pointwise, as functions.

#### 4.1.1 Behaviour

The theory of Computer Science began in the 1930s with the modelling, by Hilbert, Turing, Kleene, Church, Markov *et al.*, of the concept of a computation (or of recursiveness). As a consequence of modelling a ‘mechanism’ (or machine in Turing’s case) it is necessary to allow the possibility that

<b>abort</b>	nontermination
<b>skip</b>	no-op
$x := e$	assignment, with expression $e$
$P \text{ if } b \text{ else } Q$	conditional
$P \circledast Q$	sequential composition
$\mu F$	recursion

Figure 4: Syntax for the space  $Predet$  of predetermined programs. Assignment is assumed to be predetermined and recursion to be with respect to a continuous function.

the mechanism fails to terminate. Contemporary syntax for such computations, called *predeterministic* programs—each of which is, from any initial state, either nonterminating or deterministic—appears in Figure 4. The set of all predetermined programs over state space  $X$  is written  $Predet$ .

The interaction between nontermination and sequential composition is important. If a nonterminating program precedes or follows another (enabled) program the result remains nontermination. In particular, for each  $P : Predet$ ,

$$(3) \quad \mathbf{abort} \circledast P = \mathbf{abort}$$

$$(4) \quad P \circledast \mathbf{abort} = \mathbf{abort}.$$

We have concentrated on recursion, which includes tail recursion and hence iteration

$$\mathbf{do } g \rightarrow P \mathbf{ od} = \mu F$$

where  $F.X = (P \circledast X) \text{ if } g \text{ else skip}.$

#### 4.1.2 Model

In the standard theory, the semantic model of predetermined programs consists of a countable subset of *partial* functions of  $\mathbb{N} \rightarrow \mathbb{N}$ ; a typical treatment is Cutland's text [12]. A function's argument represents a program's initial state and its result represents the program's final state; an element lies in the domain of the function iff the program terminates when started at that initial state. The partial order of 'conformance' corresponds to: 'yields the same result when terminating (but may terminate from more initial states)'. In terms of partial functions, their domains and restriction:

$$(5) \quad f \sqsubseteq f' \equiv (f = f' \upharpoonright \text{dom } f)$$

which implies that  $(\text{dom } f \subseteq \text{dom } f')$ .

Typically these days we ignore, for the purposes of abstract description languages in Computer Science, the ‘constructively generated’ aspect because it will be ensured anyway by the form of the implementation (a program!). Also often we ‘compute with types’ invoking a procedure (or ‘oracle’) to perform actions that in the standard theory would have to be computable by construction, but now need not be even computable (e.g. the test  $(x = 0)$  for real  $x$ ). Throughout most of the paper the underlying state space for computations is assumed to be  $X$ .

We let  $\mathcal{P}$  denote the set of partial functions on  $X$ . With the relationship (5) of conformance, the resulting partial order is denoted  $(\mathcal{P}, \sqsubseteq)$ . It is a domain<sup>4</sup> with least element  $\{\}$ , with maximal elements the total functions and with compact elements the partial functions having finite domains.

In Recursive Function courses there is a tendency not to distinguish between computations (as in Figure 4) and the semantic space of partial functions. For us that distinction is crucial. Programs are expressed in the syntax of Figure 4 but their meaning is given in the space of partial functions. Without that distinction high-level programming would make little sense and the study of further languages would be impeded.

### 4.1.3 Semantics

The  $\mathcal{P}$  semantics of predetermined programs, *Predet*, is given in Figure 5. Program **abort**, ‘pure’ nontermination, is represented by the empty function. Program **skip** leaves every state unchanged and so is represented by the identity function. Assignment, to an expression which pointwise is either non-terminating or single-valued, is represented directly by that expression being used as a partial function to update state. A conditional program is, pointwise, a conditional with the same guard. Sequential composition is composition of functions (in the reverse order). Recursion is given, as usual, by Kleene’s recursion theorem.

Now Law (3) follows trivially

$$\begin{aligned}
 & [\mathbf{abort} \circ P]_{\mathcal{P}} \\
 & = \text{semantics of } \circ, \text{ Figure 5} \\
 & [P]_{\mathcal{P}} \circ [\mathbf{abort}]_{\mathcal{P}} \\
 & = \text{semantics of } \mathbf{abort}, \text{ Figure 5} \\
 & [P]_{\mathcal{P}} \circ \{\} \\
 & = \text{set theory} \\
 & \{\} \\
 & = \text{semantics of } \mathbf{abort} \text{ again}
 \end{aligned}$$

<sup>4</sup> By ‘domain’ here is meant a complete partial order in which each element is the supremum of its compact approximations. Recall that an element  $k$  is *compact* means that any directed set  $\mathcal{E}$  that exceeds it contains an element which does so: if  $k \sqsubseteq \sqcup \mathcal{E}$  then  $\exists e : \mathcal{E} \cdot k \sqsubseteq e$ . In the case of partial functions, the domain conditions means:  $\forall f : \mathcal{P} \cdot f = \bigcap \{k : \mathcal{P} \mid \#(\text{dom } k) < \infty \wedge k \sqsubseteq f\}$ . Indeed without loss of generality there  $k$  ranges over singleton partial functions:  $\#(\text{dom } k) = 1$ .

$$\begin{aligned}
[\mathbf{abort}]_{\mathcal{P}} &= \{\} \\
[\mathbf{skip}]_{\mathcal{P}} &= \lambda x : X \cdot x \\
[x := e]_{\mathcal{P}} &= \lambda x : X \cdot e \\
[P \mathbf{if} \ b \ \mathbf{else} \ Q]_{\mathcal{P}} &= \lambda x : X \cdot [P]_{\mathcal{P}}.x \ \mathbf{if} \ b.x \ \mathbf{else} \ [Q]_{\mathcal{P}}.x \\
[P \circ Q]_{\mathcal{P}} &= [Q]_{\mathcal{P}} \circ [P]_{\mathcal{P}} \\
[\mu F]_{\mathcal{P}} &= \cup \{f : \mathcal{P} \mid F.f \subseteq f\}
\end{aligned}$$

Figure 5: The  $\mathcal{P}$  semantics of predetermined programs, in which program  $P$  is denoted by a partial function  $[P]_{\mathcal{P}}$  and variable  $x$  is used for both its argument and the state of the program. Recursion is the least fixed point of  $F$ , as given by the first recursion theorem of Kleene (for instance [12], Theorem 10.3.1).

$[\mathbf{abort}]_{\mathcal{P}}$ .

Of course (the proof of) Law (4) is similar.

The embedding of total functions  $(X \rightarrow X, =)$  in  $(\mathcal{P}, \subseteq)$  is not Galois (since refinement of total functions is equality, Equivalence (1) would imply that the projection be defined only for  $f : \mathcal{P}$  with  $\text{pre } f = X$ ; alternatively,  $\varepsilon$  is not universally  $(\subseteq, \subseteq)$ -junctive since  $\cup\{\} \notin X \rightarrow X$ ).

In view of models soon to be presented, it is convenient to replace the model of partial functions with an equivalent. Each partial function is made total by mapping an element outside its domain to the ‘virtual’ element  $\perp$ . Let

$$X_{\perp} \hat{=} X \cup \{\perp\}.$$

Then the translation function is

$$\begin{aligned}
\varepsilon : \mathcal{P} &\rightarrow (X_{\perp} \rightarrow X_{\perp}) \\
\varepsilon.f &\hat{=} f \cup \{(x, \perp) \mid x \in X_{\perp} \setminus \text{dom } f\}.
\end{aligned}$$

For the new model to be closed under sequential composition, it must be ‘homogeneous’: the virtual state  $\perp$  must also belong to the domain.

Then for Laws (3) and (4) to hold, each function  $f : X_{\perp} \rightarrow X_{\perp}$  must be *strict*:

$$(6) \quad f.\perp = \perp.$$

Writing  $\text{pre } f$  for the set of elements of  $X$  not mapped by the extension  $f$  to  $\perp$ ,

$$(7) \quad \text{pre } f \hat{=} \{x : X \mid f.x \neq \perp\},$$

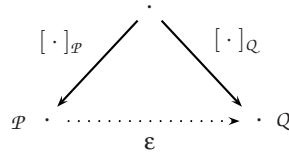


Figure 6: Using  $\epsilon$  to translate the semantics of predetermined programs from  $\mathcal{P}$  to  $\mathcal{Q}$ .

in order for  $\epsilon$  to be isotone, the partial order of conformance must translate in the new model to:

$$f \sqsubseteq f' \equiv (f \upharpoonright \text{pre } f = f' \upharpoonright \text{pre } f).$$

Let us call that model  $(\mathcal{Q}, \sqsubseteq)$ :

$$\mathcal{Q} \hat{=} \{f : X_{\perp} \rightarrow X_{\perp} \mid f.\perp = \perp\}.$$

**Theorem 3** The translation function  $\epsilon : \mathcal{P} \rightarrow \mathcal{Q}$

1. is an isotone bijection from  $(\mathcal{P}, \sqsubseteq)$  to  $(\mathcal{Q}, \sqsubseteq)$ , so that in particular  $\text{ran } \epsilon = \mathcal{Q}$ ;
2. ensures that the domain  $(\mathcal{Q}, \sqsubseteq)$  has least element the constant function  $\perp$ , maximal elements the functions  $f$  with  $\text{pre } f = X$  and compact elements the functions  $f$  with  $\text{pre } f$  finite;
3. preserves total functions (*i.e.* assignment): if  $f$  is total then  $(\epsilon.f) \upharpoonright X = f$ ;
4. preserves composition:  $\epsilon.(f \circ g) = (\epsilon.f) \circ (\epsilon.g)$ . □

Now the  $\mathcal{Q}$  semantics of *Predet* is obtained by translating the  $\mathcal{P}$  semantics with the embedding  $\epsilon$  as indicated in Figure 6. Theorem 3 ensures accuracy of the result and preservation of the laws; the result is given in Figure 7. For example, considering again Law (3), but this time in the  $\mathcal{Q}$  semantics:

$$\begin{aligned}
 & [\mathbf{abort} \circ P]_{\mathcal{Q}} \\
 & = && \text{definition of } \mathcal{Q} \text{ semantics, Figure 6} \\
 & \epsilon.[\mathbf{abort} \circ P]_{\mathcal{P}} \\
 & = && \text{previous reasoning} \\
 & \epsilon.[\mathbf{abort}]_{\mathcal{P}} \\
 & = && \text{definition of } \mathcal{Q} \text{ semantics again} \\
 & [\mathbf{abort}]_{\mathcal{Q}}.
 \end{aligned}$$

$$\begin{aligned}
[\mathbf{abort}]_Q &= \lambda x : X . \perp \\
[\mathbf{skip}]_Q &= \varepsilon. (\lambda x : X . x) \\
[x := e]_Q &= \varepsilon. (\lambda x : X . e) \\
[P \mathbf{if} b \mathbf{else} Q]_Q &= \lambda x : X . [P]_Q . x \mathbf{if} b . x \mathbf{else} [Q]_Q . x \\
[P \circledast Q]_Q &= [Q]_Q \circ [P]_Q \\
[\mu F]_Q &= \sqcup \{f : Q \mid F.f \sqsubseteq f\}
\end{aligned}$$

Figure 7: The  $Q$  semantics of predetermined programs, inferred from the  $\mathcal{P}$  semantics (Figure 5) using the technique of Figure 6.

## 4.2 Modelling nondeterminism

The advent of Software Engineering forced theoreticians (Dijkstra, Hoare, Milner *et al.*) to model nondeterminism. Nondeterminism arises for several reasons. Firstly, it might simply be inherent in the functionality being specified. For example: locate  $x$  in an array (where  $x$  may occur more than once); find a minimum spanning tree (where there may be several), a shortest path, a Hamiltonian circuit, . . . . Secondly, it might be the result of abstracting the mechanism determining a choice made at a lower level of abstraction. An example is provided by a random-number generator whose seed and mechanism of generation are concealed. Thirdly, it might be assumed in order to ensure that reasoning is local. For example, a choice determined by testing a global variable might be assumed to be a nondeterministic choice in order to avoid global reasoning.

### 4.2.1 Behaviour

Predetermined programs are extended to be finitely<sup>5</sup> nondeterministic by augmenting the language of Figure 4 with a binary combinator for nondeterministic choice; see Figure 8. The set of such programs over state space  $X$  is written  $Prog$ . The relationship of conformance is still written  $\sqsubseteq$ . Its connection with nondeterminism is:

$$(8) \quad P \sqsubseteq P' \equiv P \sqcap P' = P.$$

Important laws involving programs and nondeterminism include:

$$(9) \quad \mathbf{abort} \sqsubseteq P \quad (\text{i.e. } P \sqcap \mathbf{abort} = \mathbf{abort})$$

$$(10) \quad (P \sqcap Q) \circledast R = (P \circledast R) \sqcap (Q \circledast R)$$

$$(11) \quad P \circledast (Q \sqcap R) = (P \circledast Q) \sqcap (P \circledast R).$$

<sup>5</sup>More precisely, the nondeterministic choice is now considered of any nonempty finite set of programs. That is equivalent to the nondeterministic choice of two programs, by induction and the laws of associativity, idempotence and commutativity of binary nondeterministic choice.

$$P \sqcap Q \quad \text{nondeterministic choice}$$

Figure 8: Syntax introducing binary nondeterministic choice between programs to complete the definition of the space  $Prog$  of programs.

The first, (9), is characteristic of the Dijkstra-Hoare approach: in order to guarantee entirely correct implementations, our calculus must ensure that the (nondeterministic) possibility of an error is identified with certain error. In (10) the demonic choice responsible for the nondeterminism is made first on both sides, which are therefore indistinguishable. Law (11) is more subtle because the choice is made first on the right, but on the left only after  $P$ ; nonetheless, we expect the two programs to have identical behaviour (because  $P$  on the left-hand side is a program and so offers no behaviour which the later demonic choice can exploit).

What is the relationship between  $Prog$  and  $Predet$ , *i.e.* between programs and predetermined programs? The following law ‘quantifies’ the relationship by expressing each program as the nondeterministic combination of its predetermined refinements.

$$(12) \quad \forall P : Prog \cdot P = \sqcap \{ Q : Predet \mid P \sqsubseteq Q \}.$$

A ‘dual’ law, extended from predetermined programs to programs and hence analogous to the ‘domain law’ in Footnote 4, expresses each program (and so in particular, each predetermined program) as the supremum of the compact programs (defined in that footnote, and to be characterised semantically in Theorem 4) it refines:

$$(13) \quad \forall Q : Prog \cdot Q = \sqcup \{ K : Prog \mid K \sqsubseteq Q, K \text{ compact} \}.$$

## 4.2.2 Model

One compelling model of nondeterministic programs (Hoare *et al.* [21]) consists of the obvious extension of the model  $Q$  to relations, capturing nontermination with value  $\perp$  and nondeterminism as multi-valueness of the relation. A relation  $r$  on  $X_{\perp}$  represents a (possibly nonterminating, possibly nondeterministic but always enabled) program iff  $r$  is

1. a *total* relation on  $X$   
(just as for model  $\mathcal{P}$ , from each initial state the computation either terminates or fails to terminate):

$$(14) \quad \forall x : X \cdot \exists x' : X_{\perp} \cdot x r x'$$

2. *strict* and pointwise *upclosed* with the flat ordering on  $X_{\perp}$  (strictness is the relational version of (6) and so, as before, ensures Law (3); Condition (15) ensures Law (9) since refinement is containment):

$$(15) \quad \perp r \perp \\ xr \perp \Rightarrow \forall x' : X_{\perp} \cdot xrx'$$

3. pointwise *finitary*: the image at each initial state is either all of  $X_{\perp}$  or nonempty and finite (which captures just the nonzero finite nondeterminism we seek to express; being nonempty supersedes totality (Condition (14))):

$$(16) \quad \forall x : X \cdot \{x' : X_{\perp} \mid xrx'\} \neq X_{\perp} \Rightarrow 0 < \#\{x' : X_{\perp} \mid xrx'\} < \infty.$$

Those conditions can be abbreviated using the notation  $X \leftrightarrow X$  for the type of all relations on  $X$  and  $r.\langle x \rangle$  for the forwards relational image of  $r$  at  $x$

$$r.\langle x \rangle \hat{=} \{x' : X_{\perp} \mid xrx'\}.$$

The partial order of conformance is ‘at least as deterministic as’ between such relations

$$r \sqsubseteq s \equiv r \supseteq s,$$

where inclusion between relations is their inclusion as sets. Let us call that model  $(\mathcal{D}, \supseteq)$ :

$$\mathcal{D} \hat{=} \left\{ r : X_{\perp} \leftrightarrow X_{\perp} \mid \left( \begin{array}{l} \perp r \perp \\ \forall x : X_{\perp} \cdot \left( \begin{array}{l} xr \perp \Rightarrow r.\langle x \rangle = X_{\perp} \\ r.\langle x \rangle \neq X_{\perp} \Rightarrow 0 < \#r.\langle x \rangle < \infty \end{array} \right) \end{array} \right) \right\}.$$

There is a Galois connection from the model  $\mathcal{Q}$  for predetermined programs to the model  $\mathcal{D}$ , whose embedding is

$$(17) \quad \varepsilon : \mathcal{Q} \rightarrow \mathcal{D} \\ \varepsilon.f = f \cup (X_{\perp} \setminus \text{pre } f) \times X_{\perp}.$$

In other words,

$$x(\varepsilon.f)x' \equiv (x \in \text{pre } f \Rightarrow f.x = x').$$

Its adjoint  $\pi.r$  denotes the largest partial function in  $r$  which ‘accounts for all of  $r$ ’s results at its arguments’. It may be thought of as the largest partial function which approximates, in  $\mathcal{Q}$ , total relation  $r$ . Indeed that is the form for  $\pi$  expected by Theorem 1:

$$\pi.r = \cup \{f : \mathcal{Q} \mid \varepsilon.f \supseteq r\}.$$

We have:

**Theorem 4** The function  $\varepsilon : Q \rightarrow \mathcal{D}$

1. is an injection that preserves arbitrary suprema from  $(Q, \sqsubseteq)$  to  $(\mathcal{D}, \supseteq)$ : more generally, Definition (17) of  $\varepsilon$  makes sense if its argument is merely a relation, and then for *any*  $F \subseteq Q$  (not just those having a well-defined supremum  $\sqcup F \in Q$ ),

$$\varepsilon.\sqcup F = \cap\{\varepsilon.f \mid f \in F\};$$

2. has adjoint  $\pi : \mathcal{D} \rightarrow Q$ , thus  $gc(\varepsilon, \pi; (Q, \sqsubseteq), (\mathcal{D}, \supseteq))$ , where

$$(18) \quad \pi.r \hat{=} \{(x, y) : r \mid y \neq \perp \wedge \forall x' \neq \perp \cdot xrx' \Rightarrow x' = y\}$$

which is  $(\supseteq, \sqsubseteq)$ -continuous: if  $R$  is a  $\supseteq$ -directed subset of  $\mathcal{D}$  then

$$(19) \quad \pi.\cap R = \sqcup\{\pi.r \mid r \in R\};$$

3. has range which generates  $\mathcal{D}$  under nonempty finite unions:

$$\mathcal{D} = \{\sqcup F \mid F \subseteq \text{ran } \varepsilon \text{ is nonempty and finite}\};$$

4. ensures that the domain  $(\mathcal{D}, \supseteq)$  has least element the universal relation on  $X_\perp$ , maximal elements the (total) functions and compact elements the relations  $r$  with pre  $r$  finite (extending Definition (7) from functions to relations); thus each  $r : \mathcal{D}$  is the supremum of compact elements which it refines (a fact which is weaker than 3 since each compact element of  $\mathcal{D}$  is a nonempty finite union of elements of  $\text{ran } \varepsilon$ );
5. preserves sequential composition:  $\varepsilon.(id_X) = (id_X)_\perp$  and  $\varepsilon.(f \circ g) = (\varepsilon.g) \circ (\varepsilon.f)$ . □

It is convenient to define an embedding from relations on  $X$  to those on  $X_\perp$  to capture that part of the healthiness conditions relating to initial virtual state:

$$\begin{aligned} (\ )_\perp &: (X \leftrightarrow X) \rightarrow (X_\perp \leftrightarrow X_\perp) \\ (r)_\perp &\hat{=} r \cup \{\perp\} \times X_\perp. \end{aligned}$$

Since  $(\ )_\perp$  preserves arbitrary intersections (though only nonempty unions) it is Galois from  $(X \leftrightarrow X, \supseteq)$  to  $(X_\perp \leftrightarrow X_\perp, \supseteq)$ . Its adjoint is restriction to  $X$ :

$$\begin{aligned} \pi &: (X_\perp \leftrightarrow X_\perp) \rightarrow (X \leftrightarrow X) \\ \pi.s &\hat{=} s \cap (X \times X), \end{aligned}$$

a projection that preserves arbitrary intersections (as well as arbitrary unions as expected from Theorem 2, Part 3) and is surjective. The embedding  $(\ )_\perp$  is injective (as expected from Theorem 2, Part 2) and preserves sequential composition:

$$(20) \quad (r \circ s)_\perp = (r)_\perp \circ (s)_\perp.$$

$$[P]_{\mathcal{D}} = \cup\{\varepsilon.[Q]_{\mathcal{P}} \mid Q \in \mathit{Predet} \wedge P \sqsubseteq Q\}$$

Figure 9: Relational semantics for *Prog*, lifted from that of *Predet* (Figure 7) using the Galois connection from Theorem 4 together with union for nondeterminism.

### 4.2.3 Semantics

The semantic space  $\mathcal{D}$  is comprehensively more complex than  $\mathcal{P}$ . Our task, then, is to define the semantics of *Prog* in  $\mathcal{D}$  in such a way that the simplicity of the  $\mathcal{P}$  semantics is not obscured. That is achieved—of course—by lifting using  $\varepsilon$  via  $Q$ .

For each  $P : \mathit{Prog}$  its relational semantics  $[P]_{\mathcal{D}}$  is defined by Law (12) using union for nondeterministic choice and the lifting (Figure 6), under the Galois connection of Theorem 4, of the  $Q$  semantics of  $P$ 's predeterministic refinements. See see Figure 9.

In particular, if  $P$  is itself predeterministic then

$$[P]_{\mathcal{D}} = \varepsilon.[P]_{\mathcal{Q}}$$

For example **skip**, because it is deterministic, has semantics

$$\begin{aligned} & [\mathbf{skip}]_{\mathcal{D}} \\ & = && \text{definition of } \mathcal{D} \text{ semantics} \\ & \varepsilon.[\mathbf{skip}]_{\mathcal{Q}} \\ & = && \mathcal{Q} \text{ semantics} \\ & \varepsilon.(\lambda x : X \cdot x) \\ & = && \text{definition of } \varepsilon \\ & (\lambda x : X \cdot x)_{\perp}. \end{aligned}$$

A similar argument works for **abort**; as does the fact that  $[\mathbf{abort}]_{\mathcal{Q}}$  is the least element of  $Q$  and  $\varepsilon$  preserves minima (Theorem 2, Part 3).

Thus the  $\mathcal{D}$  semantics of *Prog* is defined by lifting on *Predet* and otherwise by union. Now the properties, that before were a matter of definition in the  $\mathcal{P}$  semantics of Figure 5, are simply inferred, though with a little more work than the  $Q$  semantics was inferred in Figure 7. See Figure 10.

Consider, for example, sequential composition. The proof relies on predeterministic computations whose  $\mathcal{P}$  semantics consists of a singleton partial function (recall Footnote 4); thus the computation terminates from just a single state. We write  $\mathit{Predet}_1(X)$  for the set of such computations. Now, for  $P, P' : \mathit{Prog}$ ,

$$\begin{aligned}
[\mathbf{abort}]_{\mathcal{D}} &= X_{\perp} \times X_{\perp} \\
[\mathbf{skip}]_{\mathcal{D}} &= (\lambda x : X \cdot x)_{\perp} \\
[x := e]_{\mathcal{D}} &= (\lambda x : X \cdot e)_{\perp} \\
[P \text{ if } b \text{ else } Q]_{\mathcal{D}} &= \{(x, x') \mid x[P]_{\mathcal{D}}x' \text{ if } b.x \text{ else } x[Q]_{\mathcal{D}}x'\} \\
[P \circledast Q]_{\mathcal{D}} &= [P]_{\mathcal{D}} \circledast [Q]_{\mathcal{D}} \\
[\mu F]_{\mathcal{D}} &= \cap \{d : \mathcal{D} \mid F.d \supseteq d\} \\
[P \sqcap Q]_{\mathcal{D}} &= [P]_{\mathcal{D}} \cup [Q]_{\mathcal{D}}
\end{aligned}$$

Figure 10: Important properties of the relational semantics for *Prog*. Function  $F$  is monotone on  $\mathcal{D}$ .

$$\begin{aligned}
& [P \circledast P']_{\mathcal{D}} \\
& = \text{Figure 9} \\
& \cup \{\varepsilon.[R]_{\mathcal{Q}} \mid R \in \text{Predet} \wedge P \circledast P' \sqsubseteq R\} \\
& = \text{Footnote 4 and set theory} \\
& \cup \{\varepsilon.[R]_{\mathcal{Q}} \mid R \in \text{Predet}_1(X) \wedge P \circledast P' \sqsubseteq R\} \\
& = \text{property of } \text{Predet}_1(X) \\
& \cup \{\varepsilon.[R]_{\mathcal{Q}} \mid \exists Q, Q' \in \text{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q' \wedge R = Q \circledast Q'\} \\
& = \text{1-point law} \\
& \cup \{\varepsilon.[Q \circledast Q']_{\mathcal{Q}} \mid Q, Q' \in \text{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q'\} \\
& = \varepsilon \text{ preserves sequential composition (Theorem 4, Part 5)} \\
& \cup \{\varepsilon.[Q]_{\mathcal{Q}} \circledast \varepsilon.[Q']_{\mathcal{Q}} \mid Q, Q' \in \text{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q'\} \\
& = \text{set theory} \\
& \cup \{\varepsilon.[Q]_{\mathcal{Q}} \mid Q \in \text{Predet}_1(X) \wedge P \sqsubseteq Q\} \circledast \cup \{\varepsilon.[Q']_{\mathcal{Q}} \mid Q' \in \text{Predet}_1(X) \wedge P' \sqsubseteq Q'\} \\
& = \text{Footnote 4 again} \\
& \cup \{\varepsilon.[Q]_{\mathcal{Q}} \mid Q \in \text{Predet} \wedge P \sqsubseteq Q\} \circledast \cup \{\varepsilon.[Q']_{\mathcal{Q}} \mid Q' \in \text{Predet} \wedge P' \sqsubseteq Q'\} \\
& = \text{Figure 9} \\
& [P]_{\mathcal{D}} \circledast [P']_{\mathcal{D}} .
\end{aligned}$$

The case of nondeterminism,  $[P \sqcap Q]_{\mathcal{D}} = [P]_{\mathcal{D}} \cup [Q]_{\mathcal{D}}$ , is similar using instead (in the third step) the property that, for  $Q : \text{Predet}_1(X)$ ,

$$P \sqcap P' \sqsubseteq Q \equiv P \sqsubseteq Q \vee P' \sqsubseteq Q.$$

Now the proofs of Laws (9) to (11) are immediate from basic set theory. For example, for Law (11),

$$[P \circledast (Q \sqcap R)]_{\mathcal{D}}$$

$$\begin{aligned}
&= && \text{Figure 10} \\
&[P]_{\mathcal{D}} \circ ([Q]_{\mathcal{D}} \cup [R]_{\mathcal{D}}) \\
&= && \text{set theory} \\
&([P]_{\mathcal{D}} \circ [Q]_{\mathcal{D}}) \cup ([P]_{\mathcal{D}} \circ [R]_{\mathcal{D}}) \\
&= && \text{Figure 10 again} \\
&[(P \circ Q) \sqcap (P \circ R)]_{\mathcal{D}}.
\end{aligned}$$

There is an alternative to our approach to the semantics of *Prog* based on Law (12) with  $\cup$  for nondeterminism. It assigns semantics by structural induction on  $P : \text{Prog}$ , ‘building in’ the equations of Figure 9 at each step. But then Law (12) must be checked and so the amount of work is equivalent. We have chosen the former approach because it is closer to that required in the probabilistic domains to follow.

### 4.3 Modelling angelic choice and enabledness

Just as Software Engineering brought to light (demonic) nondeterminism, so the formal development process discussed in Section 3 revealed the utility of ‘partially enabled’ computations and ‘angelic’ choice. We call such computations, which extend programs, *commands*.

An example of a partially-enabled command is choice of an element from a set which happens to be empty; computation cannot be started—is not enabled—in a manner that is dual to a computation that fails to terminate. This situation arises because a procedure for choosing an element from a set may be used in a context which ensures the set is nonempty. But when developed ‘in isolation’, the empty case must be considered.

Angelic choice is simply supremum  $\sqcup$ , the dual of nondeterminism  $\sqcap$ . A simple example is provided by the angelic choice of two consistent commands. The first,  $R$ , chooses  $x$  nondeterministically between 0 and 1 whilst the second,  $S$ , chooses nondeterministically between 1 and 2. Their angelic choice  $R \sqcup S$  is the weakest program stronger than both:  $x := 1$ .

If  $R$  and  $S$  had not been consistent in that example then their angelic choice, their supremum, would not have been a program. The supremum of an inconsistent set of commands is a command (though not a program) that is never enabled. Notation for the command that is never enabled and for angelic choice are introduced in Figure 11, as is our last ingredient of command space: arbitrary (rather than just binary) nondeterminism. The set of commands is written *Comm*. As usual, the relation of conformance is  $\sqsubseteq$ , satisfying (8). Of course equivalently we now have

$$(21) \quad P \sqsubseteq P' \equiv P \sqcup P' = P'.$$

<b>magic</b>	the command that is never enabled
$\sqcap \mathcal{F}$	nondeterministic choice over $\mathcal{F}$
$\sqcup \mathcal{F}$	angelic choice over $\mathcal{F}$

Figure 11: Syntax completing the space  $Comm$  of commands over state space  $X$ : the unenabled command, and arbitrary nondeterministic and angelic choices.  $\mathcal{F}$  is an arbitrary set of commands.

### 4.3.1 Behaviour

With the extension from programs to commands, the previous laws must be revisited for correctness. Law (10) remains valid: the nondeterministic choice is made initially on both sides and so the demon resolving the nondeterminism, confronted with the same choices, produces the same behaviours. But for just that reason its partner (11) does not remain valid, and must be weakened: for  $R, S, T : Comm$ ,

$$(22) \quad R \circ (S \sqcap T) \sqsubseteq (R \circ S) \sqcap (R \circ T).$$

Refinement there must of course hold by monotonicity. But equality may fail since the demon (having memory but not prescience), has more choices the later it acts. There are thus fewer choices on the right and so fewer behaviours than on the left. The choices coincide if execution of  $R$  results in no angelic choice by which the demon might profit: if  $R$  is free of angelic choice. An example of strict refinement is given in Section 4.4.1.

Important laws involving the new combinators include:

$$(23) \quad R \sqsubseteq \mathbf{magic} \quad (i.e. \ R \sqcup \mathbf{magic} = \mathbf{magic})$$

$$(24) \quad \mathbf{magic} \circ R = \mathbf{magic}$$

$$(25) \quad (R \sqcup S) \circ T = (R \circ T) \sqcup (S \circ T)$$

$$(26) \quad R \circ (S \sqcup T) \sqsupseteq (R \circ S) \sqcup (R \circ T).$$

The first, (23), says that **magic** is indeed dual to **abort** and so is the greatest (or ‘most refined’) command (and thus equals the empty angelic choice  $\sqcap\{\}$ ). The second says that an unenabled command cannot be enabled by any sequential successor (even **abort**). In (25) the choice is made initially on both sides so, reasoning as above (with the angel in place of the demon), equality holds. But (26) is dual to (22): on the right the angel acts early and—having prescience but not memory—has more choices and so produces more behaviours; alternatively, the refinement follows by monotonicity. The choices coincide if execution of  $R$  results in no nondeterministic choices by which the angel might profit: if  $R$  is predetermined.

The relationship between commands and programs is given by the law analogous to (13) (evidently the

analogue of (12) fails): for any command  $R$

$$(27) \quad R = \sqcup \{ P : Prog \mid P \sqsubseteq R \}.$$

In fact the domain property holds: without loss of generality, program  $P$  can be assumed to be compact.

### 4.3.2 Relational model

In the relational model, angelic choice must be intersection and partial enabledness must therefore be captured by partial-ness of a relation. But that means the healthiness condition of totality, (14), no longer holds. Because nondeterminism is now arbitrary, the finitary condition (16) also fails (at both ends of the inequality, in view of lack of totality). Thus all that remains is strictness and upclosure (15). The extension to  $\mathcal{D}$  consisting of relations satisfying just strictness and upclosure, but with the same criterion of conformance, we call  $(\mathcal{R}, \supseteq)$ .

The space  $(\mathcal{R}, \supseteq)$  is a domain and a complete lattice with same least element as  $\mathcal{D}$  but greatest element  $(\{\})_{\perp}$  and compact elements the cofinite subsets of  $X_{\perp} \times X_{\perp}$ . Moreover it is a Boolean algebra under the complement  $r \mapsto (X_{\perp} \times X_{\perp} \setminus r)_{\perp}$ . However the natural embedding of  $\mathcal{D}$  in  $\mathcal{R}$  is not Galois. Otherwise its adjoint  $\pi$  would map the greatest element in  $\mathcal{R}$  to a greatest element of  $\mathcal{D}$  (by Theorem 2, Part 3); but no such element exists.<sup>6</sup>

Nonetheless the injection of  $\mathcal{D}$  in  $\mathcal{R}$  does generate  $\mathcal{R}$  under arbitrary intersections, reflecting Law (27) (recall that from Theorem 4 nonempty finite unions were used to generate  $\mathcal{D}$  from  $\mathcal{Q}$ , reflecting Law (12)):

$$\mathcal{R} = \{ \cap F \mid F \subseteq \mathcal{D} \}.$$

### 4.3.3 Relational semantics

The relational semantics of *Comm* may be thought of—like the semantics for *Prog*—as follows.

1. Firstly, the  $\mathcal{R}$  semantics equals the  $\mathcal{D}$  semantics for commands that are code (like **skip**). In other words the  $\mathcal{R}$  semantics *extends* the  $\mathcal{D}$  semantics.
2. Secondly, the  $\mathcal{R}$  semantics is inferred from the  $\mathcal{D}$  semantics by extending the combinators of code to commands (as in the case of sequential composition, or even arbitrary nondeterministic choice, on *Prog* from *Predet*). This is possible because the natural embedding preserves those combinators.

---

<sup>6</sup>Since the natural embedding from  $\mathcal{D}$  to  $\mathcal{R}$  preserves arbitrary unions, why is it not Galois by Theorem 1? Because suprema in  $\mathcal{R}$  (arbitrary unions) are not the same as suprema in  $\mathcal{D}$  (consider for example the empty union).

$$\begin{aligned}
[\mathbf{magic}]_{\mathcal{R}} &= (\{\})_{\perp} \\
[\sqcap \mathcal{F}]_{\mathcal{R}} &= \cup \{ [P]_{\mathcal{R}} \mid P \in \mathcal{F} \} \\
[\sqcup \mathcal{F}]_{\mathcal{R}} &= \cap \{ [P]_{\mathcal{R}} \mid P \in \mathcal{F} \}
\end{aligned}$$

Figure 12: Relational semantics for *Comm*; this augments the extension of the semantics in Figure 9 from  $\mathcal{D}$  to  $\mathcal{R}$  using the natural embedding.

3. Thirdly, it is defined for the (new) combinator of angelic choice by edict, to be intersection.

Thus the  $\mathcal{R}$  semantics of *Comm* is provided by Figures 9 (thus extended) and 12 (which also includes arbitrary nondeterminism and its empty case, **magic**).

The proofs of Laws (23), (24) and (26) are now straightforward using basic set theory. For example, for Law (26),

$$\begin{aligned}
& [P \circledast (Q \sqcup R)]_{\mathcal{R}} \\
& = \hspace{15em} \mathcal{R} \text{ semantics of } \sqcup \text{ and } \circledast \text{ from Figure 12} \\
& [P]_{\mathcal{R}} \circledast ([Q]_{\mathcal{R}} \cap [R]_{\mathcal{R}}) \\
& \subseteq \hspace{15em} \text{set theory} \\
& ([P]_{\mathcal{R}} \circledast [Q]_{\mathcal{R}}) \cap ([P]_{\mathcal{R}} \circledast [R]_{\mathcal{R}}) \\
& = \hspace{15em} \mathcal{R} \text{ semantics of } \circledast \text{ and } \sqcup \text{ again} \\
& [(P \circledast Q) \sqcup (P \circledast R)]_{\mathcal{R}}.
\end{aligned}$$

Moreover equality holds in the middle step if, pointwise, the relation  $[P]_{\mathcal{R}}$  either maps to  $\perp$  (and hence to all of  $X_{\perp}$ ) or is single valued; in other words, the command  $P$  is predeterministic, as required.

Unfortunately, for Identity (25) the analogous argument establishes only  $\sqsupseteq$ , unless relation  $[R]_{\mathcal{R}}$  is a total function; in other words, command  $R$  is deterministic. Furthermore in Law (22) equality always holds (the existential quantification of  $\circledast$  distributing the  $\cup$  of nondeterminism). We infer that the relational model  $\mathcal{R}$  does not fully capture angelic behaviour.

Thus stretching the relational model  $\mathcal{R}$  from programs to commands reveals deficiencies. The situation is analogous to the introduction of nondeterminism: the model  $\mathcal{P}$  was simply not expressive enough and so was extended to  $\mathcal{D}$ . Now with the introduction of angelic choice, the relational model is in turn not expressive enough and must be extended.

#### 4.3.4 Transformer model

The *predicate-transformer* model (Dijkstra [13]) views each command as transforming postconditions (predicates on final states) to preconditions (predicates on initial states). For command  $P$ , the operational interpretation of its transformer semantics  $[P]_{\mathcal{T}}$  is: for any postcondition  $q$  and any initial state  $x$

$[P].q.x$  holds iff computation of  $P$  terminates from  $x$  in a state satisfying  $q$ .

Of course that is sufficient to motivate a formal definition of the semantics. But our interest here lies in reusing the relational semantics to infer the transformer semantics, as far as that is possible.

Let  $(\mathfrak{p}X, \leq)$  denote the space of all predicates (*i.e.* conditions) on  $X$  partially ordered by implication. The *predicate-transformer* model,  $(\mathcal{T}, \leq)$ , of commands (Dijkstra [13], Nelson [36], Back [4] and Morgan [33]) consists of the space of predicate transformers ordered under the lifting of the ordering on predicates

$$t \leq t' \hat{=} \forall q : \mathfrak{p}X \cdot t.q \leq t'.q$$

that satisfy just one healthiness property, that of monotonicity

$$q \leq q' \Rightarrow t.q \leq t.q'.$$

Then  $(\mathcal{T}, \leq)$  is a domain and complete lattice with least and greatest elements the constant functions *false* and *true* respectively. Its compact elements are the transformers  $t$  for which there is a finite subset  $F \subseteq X$  such that

$$(28) \quad \forall q : \mathfrak{p}X \cdot t.q = \bigvee \{ q.x \mid x \in F \}.$$

The space  $\mathcal{T}$  is endowed with an involution (see Back and von Wright [3])

$$t^*.q \hat{=} \neg t.\neg q$$

that preserves sequential composition but interchanges nondeterministic and angelic choice, enabledness and termination and **magic** and **abort**.

The embedding from relations  $\mathcal{R}$  to transformers  $\mathcal{T}$  is traditionally called the *weakest precondition*

$$\begin{aligned} wp : \mathcal{R} &\rightarrow \mathcal{T} \\ wp.r.q.x &\hat{=} \forall x' : X_{\perp} \cdot xrx' \Rightarrow (x' \neq \perp \wedge q.x'). \end{aligned}$$

It is Galois, but with orders reversed.

**Theorem 5** The function  $wp : \mathcal{R} \rightarrow \mathcal{T}$

1. is an injection that preserves arbitrary suprema from  $(\mathcal{R}, \subseteq)$  to  $(\mathcal{T}, \geq)$ : for any subset  $R \subseteq \mathcal{R}$ ,

$$(29) \quad wp.\bigcup R = \bigwedge \{ wp.r \mid r \in R \};$$

2. has adjoint the *relational projection*,  $rp : \mathcal{T} \rightarrow \mathcal{R}$  so that  $gc(wp, rp; (\mathcal{R}, \subseteq), (\mathcal{T}, \geq))$ , where

$$(30) \quad x(rp.t)x' \hat{=} x = \perp \vee \forall q : pX \cdot t.q.x \Rightarrow q.x'$$

which of course preserves infima (Theorem 2, part 3): for any subset  $T \subseteq \mathcal{T}$ ,

$$(31) \quad rp.\bigvee T = \bigcap \{rp.t \mid t \in T\}$$

but moreover preserves suprema:

$$(32) \quad rp.\bigwedge T = \bigcup \{rp.t \mid t \in T\};$$

3. satisfies merely

$$(33) \quad wp.(r \cap s) \geq (wp.r) \vee (wp.s)$$

rather than equality (in contrast to the identities (29), (31) and (32));

4. has range  $\text{ran } wp$  consisting of the conjunctive transformers,

$$(34) \quad t \in \text{ran } wp \equiv \forall q, q' : pX \cdot t.(q \wedge q') = t.q \wedge t.q',$$

and that generates  $\mathcal{T}$  under angelic choice:<sup>7</sup>

$$(35) \quad \mathcal{T} = \{\bigvee F \mid F \subseteq \text{ran } wp\};$$

5. ensures that the domain  $(\mathcal{T}, \geq)$  has least element the constant function  $\lambda q : pX \cdot \text{true}$ , greatest element the constant function  $\lambda q : pX \cdot \text{false}$  and with compact elements the transformers analogous (because of the reversal of orders) to those described in (28);

6. preserves sequential composition:  $wp.(id_X)_\perp = id_{pX}$  and  $wp.(r \circ s) = (wp.r) \circ (wp.s)$ , as does its adjoint  $rp$  in the reverse direction.  $\square$

As expected from Theorem 1, the projection  $rp.t$  defined by (30) is the largest relation that approximates  $t$  under  $wp$ .

<sup>7</sup>The space  $\mathcal{T}$  is also generated by the composition of  $wp$  with its involution [3]— $\forall t : \mathcal{T} \cdot \exists u, v : \text{ran } wp \cdot t = u^* \circ v$ —but that fact appears less useful here because the transformer involution is not the lifting of an involution on relations [39].

$$\begin{aligned}
[\mathbf{abort}]_{\mathcal{T}} &= \mathit{false} \\
[\mathbf{magic}]_{\mathcal{T}} &= \mathit{true} \\
[\mathbf{skip}]_{\mathcal{T}} &= \lambda q : pX \cdot q \\
[x := e]_{\mathcal{T}} &= \lambda q : pX \cdot q[e/x] \\
[P \mathbf{if} \ b \ \mathbf{else} \ Q]_{\mathcal{T}} &= [P]_{\mathcal{T}} \ \mathbf{if} \ b \ \mathbf{else} \ [Q]_{\mathcal{T}} \\
[P \circledast Q]_{\mathcal{T}} &= [P]_{\mathcal{T}} \circ [Q]_{\mathcal{T}} \\
[\mu F]_{\mathcal{T}} &= \bigvee \{ t : \mathcal{T} \mid F.t \leq t \} \\
[\sqcap \mathcal{F}]_{\mathcal{T}} &= \bigwedge \{ [P]_{\mathcal{T}} \mid P \in \mathcal{F} \} \\
[\sqcup \mathcal{F}]_{\mathcal{T}} &= \bigvee \{ [P]_{\mathcal{T}} \mid P \in \mathcal{F} \}
\end{aligned}$$

Figure 13: Transformer semantics for commands, inferred from Figure 12 using the  $wp$  Galois connection.

### 4.3.5 Transformer semantics

Now we find that the semantic space  $\mathcal{T}$  appears deceptively simple but the manner of expressing a computation is radically different from that in relations. Naturally we use the Galois connection to bridge the gap.

The Galois connection can be used to lift much of the relational semantics to transformers following our standard approach of Figure 2. As usual (Theorem 2 part 3), it maps the least element  $(\{\})_{\perp}$  in  $(\mathcal{R}, \subseteq)$  to the least element, the constant transformer  $\mathit{true}$  in  $(\mathcal{T}, \geq)$ , thus providing the semantics of **magic**. For sequential composition, we find

$$\begin{aligned}
& [P \circledast Q]_{\mathcal{T}} \\
& = \hspace{20em} \text{definition of } \mathcal{T} \text{ semantics} \\
& wp \cdot [P \circledast Q]_{\mathcal{R}} \\
& = \hspace{10em} \text{definition of } \mathcal{R} \text{ semantics, Figures 9 and 12} \\
& wp \cdot ([P]_{\mathcal{R}} \circledast [Q]_{\mathcal{R}}) \\
& = \hspace{15em} \text{property of } wp, \text{ Theorem 5.6} \\
& (wp \cdot [P]_{\mathcal{R}}) \circ (wp \cdot [Q]_{\mathcal{R}}) \\
& = \hspace{15em} \text{definition of } \mathcal{T} \text{ semantics again} \\
& [P]_{\mathcal{T}} \circ [Q]_{\mathcal{T}} .
\end{aligned}$$

It maps arbitrary unions in  $\mathcal{R}$  to arbitrary conjunctions in  $\mathcal{T}$ , by (29), thus providing the semantics of arbitrary nondeterminism. But the lack of equality in (33) means that  $wp$  can not be used to lift angelic choice from  $\mathcal{R}$  to  $\mathcal{T}$ . That must simply be defined to be disjunction. The resulting transformer semantics is given in Figure 13.

The proofs of Laws (23), (24) and (25) are now straightforward using elementary logic. For Law (26),

$$P_{r \oplus} Q \quad \text{equals} \quad \begin{cases} P \text{ with probability } r \\ Q \text{ with probability } 1-r \end{cases}$$

Figure 14: The probabilistic combinator chooses between its two arguments with probabilities that sum to 1. The expression  $r$  is a function of state.

we find

$$\begin{aligned} & [P \circledast (Q \sqcup R)]_{\mathcal{T}} \\ & = && \mathcal{T} \text{ semantics of } \sqcup \text{ and } \circledast \text{ from Figure 13} \\ & [P]_{\mathcal{T}} \circ ([Q]_{\mathcal{T}} \vee [R]_{\mathcal{T}}) \\ & \geq && \text{monotonicity} \\ & ([P]_{\mathcal{T}} \circ [Q]_{\mathcal{T}}) \vee ([P]_{\mathcal{T}} \circ [R]_{\mathcal{T}}) \\ & = && \mathcal{T} \text{ semantics of } \circledast \text{ and } \sqcup \text{ again} \\ & [(P \circledast Q) \sqcup (P \circledast R)]_{\mathcal{T}}. \end{aligned}$$

Moreover equality holds in the middle step if the transformer  $[P]_{\mathcal{T}}$  is disjunctive; in other words, the command  $P$  is predetermined as required.

Thus the transformer model  $\mathcal{T}$  derives from the relational model  $\mathcal{R}$  but does not exhibit its deficiencies discussed in the previous section.

## 4.4 Modelling probability

Our treatment of programs and their extension to commands has indicated how the incremental method works. But what about more complex, novel, functionality? Here, as an example, we consider probabilistic choice of the sort that is achieved by use of a random-number generator to select between alternatives. Since any (nonvoid finite) multi-way probabilistic choice is obtained by nested binary choices, binary choice is by itself sufficiently expressive. Syntax is introduced in Figure 14. The space *Prog* of programs extended by such probabilism we denote *Prob*. For a thorough study of *Prob* (and also *Comm* similarly extended) we refer to Morgan and McIver's careful and entertaining text [34].

### 4.4.1 Behaviour

Laws characteristic of programs with probabilism are contained in Figure 15.

We have already considered Law (48) in Section 4.3 as Law (22). There we reasoned that equality (which held for nondeterministic code) was violated by commands more general than code (angelic choice in

- (36)  $P_{1\oplus}Q = P$   
(37)  $P_r\oplus Q = (Q_{1-r}\oplus P)$   
(38)  $P_r\oplus P = P$   
 $(P_a\oplus Q)_b\oplus R = P_c\oplus(Q_d\oplus R)$  where  $c = a\times b$ ,  
 $d = ((1-a)\times b)/(1-a\times b)$   
(39)  $(P_r\oplus Q)\circlearrowleft R = (P\circlearrowleft R)_r\oplus(Q\circlearrowleft R)$   
(40)  $(P\circlearrowleft Q)_r\oplus(P\circlearrowleft R) \sqsubseteq P\circlearrowleft(Q_r\oplus R)$   
(41)  $P \text{ if } b \text{ else } Q = P_b\oplus Q$   
(42)  $P_r\oplus(Q \text{ if } b \text{ else } R) = (P_r\oplus Q) \text{ if } b \text{ else } (P_r\oplus R)$   
(43)  $P\sqcap Q = \sqcap_{0\leq r\leq 1} P_r\oplus Q$   
(44)  $P\sqcap Q \sqsubseteq P_r\oplus Q$   
(45)  $(P\sqcap Q)_r\oplus R = (P_r\oplus R)\sqcap(Q_r\oplus R)$   
(46)  $(P\sqcap R)_r\oplus(Q\sqcap R) \sqsubseteq (P_r\oplus Q)\sqcap R$   
(47)  $(P\sqcap Q)\circlearrowleft R = (P\circlearrowleft R)\sqcap(Q\circlearrowleft R)$   
(48)  $P\circlearrowleft(Q\sqcap R) \sqsubseteq (P\circlearrowleft Q)\sqcap(P\circlearrowleft R)$

Figure 15: Some laws for probabilism. In Law (41)  $b$  is treated on the left as a Boolean and on the right numerically. In (44)  $r$  is, as usual, an expression on state with values in the unit interval.

particular). Now we see that equality is violated by (probabilistic) *code*. Again the (weak) refinement follows by monotonicity, so we focus on showing inequality. Consider

$$\begin{aligned} P &\hat{=} x := 0 \frac{1}{2} \oplus x := 1 \\ Q &\hat{=} y := 0 \\ R &\hat{=} y := 1, \end{aligned}$$

and consider the probability with which each side in (22) achieves the postcondition  $x = y$ . The left-hand side

$$(x := 0 \frac{1}{2} \oplus x := 1) \circlearrowleft (y := 0 \sqcap y := 1)$$

does so with probability 0: the demon chooses after the probabilistic choice has been made and so can choose  $y$  to ensure  $y \neq x$ . But the right-hand side

$$(x := 0 \frac{1}{2} \oplus x := 1) \circlearrowleft y := 0 \sqcap (x := 0 \frac{1}{2} \oplus x := 1) \circlearrowleft y := 1$$

does so with probability  $\frac{1}{2}$ : the demon's initial choice is one of the two sequential compositions, each of which has probability  $\frac{1}{2}$  of establishing  $x = y$ . Remember: the demon is not prescient and so is unable to take advantage of the probabilistic choice still to be made.

From that informal argument it is clear that the definition of refinement must be revised, to make it numerical. The revision must of course be consistent with previous definitions. The effect of the new

version of refinement  $P \sqsubseteq Q$  is that if  $P$  satisfies a postcondition with some probability then  $Q$  will satisfy it with a probability at least as great; and the probability that  $Q$  terminates is at least as great as the probability that  $P$  does.

The program  $x := 0 \frac{1}{2} \oplus x := 1$  is, in spite of popular notation to the contrary (particularly in quantum computing where ‘nondeterministic’ is synonymous with ‘probabilistic’) deterministic: it has no proper refinements. Thus *Prob* has more maximal elements than *Prog* in the refinement ordering.

The equivalent for probabilistic programs of Law (12) is

$$(49) \quad P = \sqcap \{ Q : \text{Prob} \mid P \sqsubseteq Q \wedge Q \text{ predeterministic} \},$$

where  $Q$  is predeterministic means that it is expressed in the sublanguage *Predet* extended by the combinator for probabilism. Thus from each initial state it either aborts or its behaviour is maximal in the refinement ordering.

#### 4.4.2 Distributional model

The result of executing a *deterministic* probabilistic program from a given initial state is a probability distribution over its state space. (A deterministic program with no probabilism results in a distribution whose mass is concentrated at its single final state.) That view is extended to allow the program to be *predeterministic* by allowing the distribution to sum to less than 1. In this model no virtual state  $\perp$  is required: the probability of nontermination is given by the difference from 1 of the total mass of the distribution. In a refinement that probability is diminished by increasing the probabilities of some final states.

We can then view the result of executing a *nondeterministic* probabilistic program from a given initial state as a set of (such) distributions, one member for each resolution of the nondeterminism. This intuition is of course captured by Law (49). A probabilistic program can therefore be thought of as a function from its initial state to a set of distributions over final state. The result is a model formalised as follows.

We write  $(\Delta, \sqsubseteq)$  for the space of distributions<sup>8</sup> over  $X$ , with pointwise order

$$\begin{aligned} \Delta &\hat{=} \{ f : X \rightarrow [0, 1] \mid \sum_{x:X} f.x \leq 1 \} \\ f \sqsubseteq f' &\hat{=} \forall x : X \cdot f.x \leq f'.x. \end{aligned}$$

In order for refinement to be containment, each set  $\mathcal{F}$  of distributions must be closed under refinement of its elements: it must be *upclosed* in the sense that it contains any distribution that dominates a member:

$$\text{upc}(\mathcal{F}) \hat{=} \forall f, f' : \Delta \cdot (f \in \mathcal{F} \wedge f \sqsubseteq f') \Rightarrow f' \in \mathcal{F}.$$

<sup>8</sup>Throughout we consider only functions on  $X$  with mass at most 1 and so do not introduce notation to distinguish that case from the more traditional one in which mass equals 1.

A set  $\mathcal{F}$  of distributions is called *convex* if it contains all convex combinations of its member distributions:

$$cvx(\mathcal{F}) \hat{=} \forall f, f' : \mathcal{F} \cdot 0 \leq r \leq 1 \Rightarrow r \times f + (1 - r) \times f' \in \mathcal{F} .$$

Finally a set  $\mathcal{F}$  of distributions is *closed*,  $cld(\mathcal{F})$ , if it is topologically closed as a subset of the product topological space  $[0, 1]^X$  with the usual topology on the real interval  $[0, 1]$  and the discrete topology on  $X$ .

The distributional model  $(\mathcal{H}, \sqsubseteq)$  of *Prob* consists of the functions from  $X$  to sets of distributions that, pointwise, are nonempty and are upclosed, convex and closed:

$$\begin{aligned} \mathcal{H} &\hat{=} \{F : X \rightarrow \mathbb{P}\Delta \mid \forall x : X \cdot F.x \neq \{\} \wedge upc(F.x) \wedge cvx(F.x) \wedge cld(F.x)\} \\ F \sqsubseteq F' &\hat{=} \forall x \in X \cdot F.x \supseteq F'.x . \end{aligned}$$

Although not complete,  $(\mathcal{H}, \sqsubseteq)$  has least element the constant function  $\lambda x \cdot \Delta$  and maximal elements the functions whose values are singletons of distributions having mass 1.

The relational model  $\mathcal{D}$  is embedded in the distributional model  $\mathcal{H}$  as follows. Firstly, the state space  $X$  is embedded in the space  $\Delta$  of distributions on  $X$  via point-masses: for  $x : X$  its point mass  $\delta.x : \Delta$  assigns 1 to  $x$  and 0 to any other state

$$\begin{aligned} \delta : X &\rightarrow \Delta \\ \delta.x.y &= (x = y) . \end{aligned}$$

Then an embedding  $\varepsilon$  is defined as follows. Suppose  $r : \mathcal{D}$  and  $x : X$ . If  $r$  represents a nonterminating command,  $xr \perp$ , then the embedding  $\varepsilon.r$  at  $x$  equals the set of all distributions:  $\varepsilon.r.x = \Delta$ . Otherwise  $\varepsilon.r.x$  consists of the closure, with respect to the three properties above<sup>9</sup> of the set of point masses of final states:

$$(50) \quad \begin{aligned} \varepsilon : \mathcal{D} &\rightarrow \mathcal{H} \\ \varepsilon.r.x &\hat{=} \Delta \text{ if } xr \perp \text{ else } closure\{\delta.y \mid xry\} . \end{aligned}$$

The function  $\varepsilon$  is Galois from  $(\mathcal{D}, \sqsubseteq)$  to  $(\mathcal{H}, \sqsubseteq)$ , but also preserves unions.

**Theorem 6** Recall that state space  $X$  is assumed to be finite. The function  $\varepsilon : \mathcal{D} \rightarrow \mathcal{H}$

1. is a bijection that preserves *arbitrary* suprema and infima from  $(\mathcal{D}, \supseteq)$  to  $(\mathcal{H}, \supseteq)$ : for any  $R \subseteq \mathcal{D}$ ,

$$(51) \quad \varepsilon.\cap R = \cap\{\varepsilon.r \mid r \in R\}$$

$$(52) \quad \varepsilon.\cup R = \cup\{\varepsilon.r \mid r \in R\};$$

<sup>9</sup>If  $D$  is a set of distributions its *closure*,  $closure(D)$ , means the smallest set containing  $D$  which is upclosed, convex and (topologically) closed.

2. in particular has an adjoint  $\pi : \mathcal{H} \rightarrow \mathcal{D}$  so that  $gc(\varepsilon, \pi; (\mathcal{D}, \sqsupseteq), (\mathcal{H}, \sqsupseteq))$ ;
3. preserves sequential composition of total functions in this sense (contorted by the fact that the members of  $\mathcal{H}$  have different types for source and target): omitting the set parentheses in Definition (50), if  $g, g' : \mathcal{D}$  are total functions then

$$(53) \quad \varepsilon.g = \delta \circ g \quad \text{so that}$$

$$(54) \quad \varepsilon.(id_X)_\perp = \delta$$

$$(55) \quad \varepsilon.(g \circledast g') = \delta \circ g' \circ g = (\varepsilon.g') \circ g.$$

□

Sequential composition is clearly more complicated in this than in previous models. It is treated more fully as follows. The first step is to consider deterministic (probabilistic) programs: commands represented in  $\mathcal{H}$  by functions whose value at each initial state is a single distribution (and so we elide the set containing it). For such commands  $F$ , the model is readily made homogeneous by:

- (a) replacing each initial state  $x$  by its point mass  $\delta.x$  and lifting  $F$  to  $\delta.x$  in the obvious way

$$F.(\delta.x) = F.x$$

- (b) extending  $F$  from point masses to distributions  $f$  (i.e. to linear combinations of point masses that are *subconvex* in the sense that their positive coefficients sum to at most 1 (rather than to exactly 1)) by averaging (or convolution); we call the result  $F^\dagger$ :

$$(56) \quad F^\dagger.f.y \hat{=} \sum_{x:X} (F.x.y) \times f.x.$$

In other words, representing a distribution as a subconvex combination of point masses, the first step has resulted in a lifting  $F^\dagger$  from distributions to distributions,

$$F^\dagger.f = \sum_{y:X} (\sum_{x:X} (F.x.y) \times f.x) \times \delta.y,$$

and so facilitates a definition of sequential composition for deterministic programs:

$$F \circledast G \hat{=} G^\dagger \circ F.$$

Of course a further application of  $^\dagger$  produces a homogeneous result,  $(G^\dagger \circ F)^\dagger$ .

The second step completes the definition of sequential composition by using (a special case of) Law (49) to express, semantically, a (general) probabilistic program as the nondeterministic choice of its deterministic refinements, using the  $^\dagger$  operation

$$(F \circledast G).x \hat{=} \{D^\dagger.f \mid f \in F.x \wedge G \sqsubseteq \varepsilon.D \wedge D \in \mathcal{D} \text{ deterministic}\}.$$

There the Galois connection  $\varepsilon$  translates  $D$  from a deterministic program in  $\mathcal{D}$  to one in  $\mathcal{H}$  so that it can be compared with  $G$  in the refinement ordering.

$$\begin{aligned}
[\mathbf{skip}]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \{\delta.x\} \\
[\mathbf{abort}]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \Delta \\
[x := e]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \{\delta.e\} \\
[P_r \oplus Q]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \{r \times f + (1-r) \times g \mid f \in [P]_{\mathcal{H}}.x \wedge g \in [Q]_{\mathcal{H}}.x\} \\
[P \sqcap Q]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \cup\{[P_r \oplus Q]_{\mathcal{H}}.x \mid 0 \leq r \leq 1\} \\
[P \circ Q]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \{\sum_{x' \in X} f.x' \times g.x' \mid f \in [P]_{\mathcal{H}}.x \wedge g.x' \in [Q]_{\mathcal{H}}.x'\} \\
[P \text{ if } b \text{ else } Q]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot [P]_{\mathcal{H}}.x \text{ if } b.x \text{ else } [Q]_{\mathcal{H}}.x \\
[\mu F]_{\mathcal{H}} &\hat{=} \lambda x : X \cdot \cap\{G : \mathcal{H} \mid F.G.x \supseteq G.x\}
\end{aligned}$$

Figure 16: Distributional probabilistic semantics of code. Expression  $e$  is assumed to be deterministic.

#### 4.4.3 Distributional semantics

As is to be expected by now, the  $\mathcal{H}$  semantics of *Prob* is obtained, where possible, by lifting the  $\mathcal{D}$  semantics with  $\varepsilon$ ; see Figure 16. The semantics  $[P_r \oplus Q]_{\mathcal{H}}$  of the combinator for probabilism is given by the set of convex combinations (determined by the probabilistic expression  $r$ ) of distributions from its two arguments; the result is upclosed.

If  $[P \sqcap Q]_{\mathcal{H}}$  were to be defined by lifting its definition in  $\mathcal{D}$  then Law (43) would fail. So having defined  $[P_r \oplus Q]_{\mathcal{H}}$  we simply define  $[P \sqcap Q]_{\mathcal{H}}$  in order to ensure Law (43).

#### 4.4.4 Transformer model

The calculus of predicate transformers is Boolean. But now, to handle probability and the convex combinations that result, that must be extended. The Booleans *false* and *true* are replaced by the numeric values 0 and 1 respectively, and each predicate is lifted to be real-valued:

$$\begin{aligned}
[\ ] : \text{p}X &\rightarrow (X \rightarrow \mathbb{R}) \\
[q].x &\hat{=} 1 \text{ if } q.x \text{ else } 0.
\end{aligned}$$

An *expectation* is a non-negative-real-valued function on state space  $X$ . It arises in the present context by closing predicates (*i.e.* pre and postconditions) under convex combinations and so may be thought of as a ‘probabilistic predicate’. The set of all expectations is

$$Q \hat{=} X \rightarrow \{r : \mathbb{R} \mid r \geq 0\}.$$

The transformer model of probabilistic programs is like the transformer model of standard programs, but with conditions (*i.e.* predicates) replaced by expectations. Thus the space  $(\mathcal{J}, \sqsubseteq)$  of *expectation transformers*, named for Claire Jones but due to Kozen [30], Jones [27] and Morgan *et al.* [34], is

$$\mathcal{J} \hat{=} Q \rightarrow Q$$

with pointwise ordering

$$j \sqsubseteq k \hat{=} \forall q: Q. [j.q] \leq [k.q].$$

In translating from  $\mathcal{H}$  to  $\mathcal{J}$  we must relate distributions and expectations. The expectation of a function  $\phi$  against a distribution  $f$  (already used in a special situation in (56)) is defined to be the discrete Stieltjes integral

$$\int \phi df \hat{=} \sum_{x \in X} (\phi.x) \times (f.x).$$

We can recover, from expectations, the ordinary notion of evaluating a state at a predicate by specialising to standard expectation  $[q]$  and a point-mass distributions  $\delta.x$ :

$$\begin{aligned} & \int [q] d(\delta.x) \\ \equiv & && \text{definition of integral} \\ & \sum_{x' \in X} [q].x' \times \delta.x.x' \\ \equiv & && \text{definition of point mass} \\ & [q].x. \end{aligned}$$

In fact for standard expectations and arbitrary distribution the above integral calculates the probability that (with respect to the distribution) the predicate is satisfied.

The translation function between the lattices  $(\mathcal{H}, \supseteq)$  and  $(\mathcal{J}, \sqsubseteq)$  is now formalised

$$\begin{aligned} wp: \mathcal{H} & \rightarrow \mathcal{J} \\ wp.h.q.x & \hat{=} \sqcap \{ \int q df \mid f \in h.x \}, \end{aligned}$$

for probabilistic relation  $h: \mathcal{H}$ , expectation  $q: Q$ , and state  $x: X$ . It gives the least expected value of postexpectation  $q$  against all final distributions  $f$  achievable by probabilistic command  $h$  from initial state  $x$ .

The translation  $wp$  has the following operational interpretation: for a program  $h$ , and  $q$  a (standard) predicate,  $wp.h.[q].x$  represents the greatest guaranteed probability that  $q$  is satisfied finally. Not surprisingly,  $wp$  is Galois:

**Theorem 7** The function  $wp: (\mathcal{H}, \supseteq) \rightarrow (\mathcal{J}, \sqsubseteq)$

1. is an injection that preserves suprema: for any set  $Q$  of expectations,

$$wp. \sqcap Q = \sqcup \{ wp.q \mid q \in Q \};$$

2. has adjoint  $rp : \mathcal{J} \rightarrow \mathcal{H}$  defined, for expectation transformer  $j \in \mathcal{J}$  and state  $x \in X$ ,

$$rp.j.x \hat{=} \{f : \Delta.X \mid \forall q : \mathcal{Q} \cdot j.q.x \leq \int_f q\};$$

thus  $ge(wp, rp; (\mathcal{H}, \supseteq), (\mathcal{J}, \sqsubseteq))$ ;

3. ensures that the lattice  $(\mathcal{J}, \sqsubseteq)$  has least element the constant expectation transformer  $\lambda q : \mathcal{Q} \cdot 0$ ;
4. has range characterised by sublinearity:  $t \in \text{ran } wp$  iff  $t$  is *sublinear*: for all expectations  $q, q'$  and non-negative reals  $a, b, c$ ,

$$(57) \quad [t.((a \times q + b \times q') \ominus \mathbf{c})] \geq (a \times [t.q] + b \times [t.q']) \ominus \mathbf{c},$$

where the ‘truncated difference’ operator  $\ominus$  is defined<sup>10</sup> on non-negative reals (and lifted pointwise to expectations) to ensure that difference remains non-negative:  $a \ominus b \hat{=} (a - b) \sqcup 0$ ;

5. when restricted to predeterminedistic (probabilistic) programs is characterised by *linearity*:  $t$  is the image of a predeterminedistic distributional program (*i.e.* whose image under  $wp$  assigns to each initial state a single distribution) iff  $t$  is *linear*: equality holds in (57).

The restriction of  $rp$  to  $\text{ran } wp$  is well defined, but  $rp$  need not otherwise be defined (recall that relations in  $\mathcal{H}$  are non-empty).

#### 4.4.5 Transformer semantics

The expectation-transformer semantics of *Prob* is given in Figure 17. This section, the end of the technical material of the paper, consists of two training exercises.

1. First, simple practice in this last increment: *justify each definition intuitively*.
2. Second, practice in the central theme of this paper: *define the semantics of Figure 17 as a lifting using the Galois connection just defined*.

## 5 Research agenda

Here are just a few areas in which the approach of this paper seems assured of making progress.

<sup>10</sup>A typical use of  $\ominus$  in our context is to translate between addition of bits and their minimum:  $a \sqcap b = (a + b) \ominus 1$ .

$$\begin{aligned}
[\mathbf{abort}]_j &\hat{=} \lambda q : Q \cdot 0 \\
[\mathbf{skip}]_j &\hat{=} \lambda q : Q \cdot q \\
[x := e]_j &\hat{=} \lambda q : Q \cdot q.e \\
[P \mathbin{;} Q]_j &\hat{=} [P]_j \circ [Q]_j \\
[P \mathbf{if} b \mathbf{else} Q]_j &\hat{=} [b] \times [P]_j + [\neg b] \times [Q]_j \\
[P \sqcap Q]_j &\hat{=} [P]_j \sqcap [Q]_j \\
[P_r \oplus Q]_j &\hat{=} \lambda q : Q \cdot r \times [P]_j.q + (1-r) \times [Q]_j.q \\
[\mu F]_j &\hat{=} \text{least fixed point of } F : j \rightarrow j .
\end{aligned}$$

Figure 17: Expectation-transformer semantics for *Prob*.

### 1. Concurrent systems: Process algebra.

Perhaps the cleanest area of program semantics in which to apply the incremental approach is that of process algebra, and in particular Communicating Sequential Processes, CSP (see Hoare's reflection [23]). Since its inception, CSP's semantic models have been presented incrementally: first, traces to capture safety information; then failures to capture deadlock and so distinguish internal and external choice; and finally divergences to distinguish livelock from deadlock. But only relatively recently has that hierarchy been presented incrementally with its three levels connected by Galois connections; see Seidel and Morgan [43] (which in fact achieves more: the addition of probabilism at each increment).

But much remains. The last section of [43] contains a discussion of the way in which (real) time is incorporated incrementally. But other behaviours seem ripe for consideration: probability and time together, angelic choice (alone or with probability and/or time), the violation of atomicity, or even time reversal, and the incorporation of behaviour pertinent to long-running transactions.

### 2. Ensemble engineering.

The working group WG1, under the coordination action 'Interlink' (see the interim report of Hölzl and Wirsing [24]), have been considering a call for proposals in the area of *software intensive systems and new computing paradigms*. One area of interest is the engineering of large and complex systems—which they have named *ensembles*. Ensembles, complex by definition, provide the perfect opportunity for an incremental approach; or do they? If they are complex enough to exhibit emergent behaviour, how can Formal Methods apply? For by its definition, emergent behaviour lies beyond the scope of a description of the system components. However a proposal has been made by Hu *et al.* [26] to reconcile that apparent contradiction and so for the application of incremental methods.

Ensembles are of particular interest because, although they are composed of discrete components, they typically embody statistical behaviour. Thus it is of interest to see how the incremental methods apply in that situation; the techniques of the present paper should suffice. Special cases include hybrid systems and multiagent systems that adapt.

### 3. *Hardware systems.*

There is a well-established hierarchy of models for hardware devices driven synchronously. However for asynchronous timing, the models and handshaking protocols are not usually thought of hierarchically. There seems therefore to be an opportunity to exploit the incremental approach. The details are certainly not trivial but the reward worth the effort. Timed automata appear to provide an appropriate vehicle; see UPAAL [6]. For a study involving simulation in UPAAL and first thoughts of a hierarchical approach, see Boumaza *et al.* [7].

### 4. *Quantum systems.*

Formal Methods have not played a large part in quantum computation. Indeed most of the proposed ‘quantum programming languages’ do not even have formal semantics, let alone a statement of laws. The two languages supported by semantics and laws, Selinger’s language [44] and that based on the language *Prob* of this paper, qGCL [41], fail to capture all the behaviour nowadays required in expressing and reasoning about quantum computation.

Firstly, some kind of quantum process algebra would be desirable for expressing and reasoning about quantum cryptographic protocols. Secondly, the recent use of traditional concepts from Mathematical Physics—completely positive operators—due to D’Hondt and Panangaden [25] provides an enticing formalism for quantum semantics. It would be interesting to use it for quantum process algebra as well as quantum algorithms, and to relate it via Galois connections to the semantic models of the two languages above.

### 5. *Component-based systems: rCOS.*

Perhaps the most obvious extension to the ideas of this paper lie in the direction of object orientation. The relational calculus of object systems, rCOS, has been proposed by He *et al.* [19]. Its semantics is described relationally, following the UTP style of Hoare and He [22]. However with more recent extensions to rCOS, due to Liu *et al.* [8], it would be interesting and informative to consider an incremental account of the semantics of current rCOS. Either the UTP approach, or the more general one promoted here, could be taken.

## 6 Conclusion

Systems are inherently complicated so theories must be as simple as possible. Since detail cannot ultimately be avoided, the simplicity must come from the method. In the areas of traditional engineering, where relationships between entities are assumed to be differentiable, the method is based on approximation by simpler behaviours whose outcome approximate closely that of the real system. In the case of discrete systems such approximation is of little use (how do you approximate a bit?), and the method must describe the complexity *exactly* but stepwise. That can be done in two ‘orthogonal’ ways—by modularisation at a *given* level of abstraction or incrementally *across* levels of abstraction. The former approach is reasonably well understood and much practiced. The purpose of this paper is to promote the latter, in the guise of Galois connections.

A potted history of Galois connections in Computer Science is given by Backhouse [5]. Particularly

worthy of mention are their use for ‘abstract interpretation’ by Cousot and Cousot [11]; their use in automata theory, via the related concept of ‘pair algebra’ by Hartmanis and Stearns [18], and via ‘factors’ by Conway [10]; and in program semantics: see for example Nielsen, Nielsen and Hankin [37], Chapter 4, and Scott *et al.*’s use of embedding-projection pairs [17].

We have presented a single sustained case for the incremental approach, moving from predetermined (*i.e.* computable) computations through finitely nondeterministic programs to angelic and arbitrary nondeterministic commands and finally to probabilistic programs. At most of the increments the semantic intuition and laws have been able to be lifted by Galois connections. Where that has not been the case, valuable insight has been provided by the property that fails (for example failure of *wp* to map intersections to disjunctions).

But much has been omitted. We have not dealt with data refinement, nor properly with recursion. In overlooking data refinement we have neglected the important technique of ‘spans’ an alternative elaboration of the construction of  $\mathcal{T}$  from functions via relations (see Gardiner *et al.*’s [16]). Nor have we systematically used laws to determine healthiness conditions, as is typical of the UTP approach. The current approach—by comparison with UTP, more ‘back to basics’—may as a result be more flexible in use. We have also not considered the use of Galois connections to relate the various languages algebraically. Where we have done so semantically, an alternative would be to have defined them directly between the languages. This remains future work.

Two questions arose from the UNU-IIST research day. The less interesting one is ‘*What constitutes research?*’ Evidently not all work is research. But a definition of ‘research’ must remain elusive: at the very least it is area dependent. What is research for a technician may not be research for an engineer; and research for an engineer may not be research for a scientist. One purpose of the UNU-IIST research day was to celebrate the range of research we do.

A more interesting question that arose is ‘*What constitutes use of formal methods?*’ Evidently such use forms a spectrum. At the light end, it is realised by use of a formal notation, provided some degree of reasoning is performed. Just to use some formal notation is of course not necessarily to use Formal Methods. The easiest way to achieve some reasoning is by (automated) type checking of a specification, which confers some (mild) degree of confidence in the construction of the specification. Of course it ensures nothing about the functionality of the specification (let alone an implementation)!

A more substantial way in which some form of reasoning may be achieved is by model checking; then *some* functionality of the specification can be confirmed. Again, little can be said about a (putative) implementation except in trivial cases where the specification *is* the implementation. Prototyping has its uses, but does not confer correctness of the final implementation.

The only way to infer that a putative implementation behaves correctly (*i.e.* that it is in fact an implementation) is to prove that it conforms to the specification. For realistic systems that involves an amount of work which is prohibitive unless the system is critical in some way. One compromise is to verify the design to a level of increment that balances the tradeoff between confidence and expense. For that the techniques considered here apply.

Thus we conclude that the degree of use of formal methods is in general captured by the degree of reasoning performed. To perform no reasoning is not to use formal methods.

The process of reflecting on ‘his’ work is something each graduate student must learn. The reasoning that forms part of Formal Methods provides one way for the practising Software Engineer to do so. The content of this paper provides an example: if the work consists of constructing a semantics then the process of reflecting on the work consists of thinking how that semantics can be explained/introduced as simply and elegantly as possible. The importance of reflection lies far beyond the specific topics discussed here and is perhaps the most vital skill a beginning research student can learn.

## References

- [1] R. Ashenurst, editor. *ACM Turing Award Lectures: The First Twenty Years: 1966 to 1985*. ACM Press Anthology Series, 1987.
- [2] R.-J. R. Back. Combining angels, demons and miracles in program specifications. *Theoretical Computer Science*, **100**(2):365–383, 1992.
- [3] R.-J. R. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, **27**(7):583–625, 1990.
- [4] R.-J. R. Back and J. von Wright. *Refinement Calculus: a Systematic Introduction*. Springer Verlag, 1998.
- [5] R. J. Backhouse. Pair algebras and Galois connections. *Information Processing Letters*, **67**(4):169–175, 1998.
- [6] J. Bengtsson, K. Larsen, F. Larsson and P. Pettersson. UPAAL—a tool suite for automatic verification of real-time systems. *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*. SLNCS, Springer Verlag, 1995.
- [7] K. Boumaza, Wang Xu and J. W. Sanders. Forthcoming UNU-IIST Technical Report, 2008.
- [8] Z. Chen, A. H. Hannousse, D. V. Hung, I. Knoll, X. Li, Z. Liu, Y. Liu, Q. Nan, J. C. Okika, A. P. Ravn, V. Stolz, L. Yang, and N. Zhan. Modelling with Relational Calculus of Object and Component Systems—rCOS. UNU-IIST Technical Report **382**, September 2007.
- [9] E. M. Clarke, O. Grumberg and D. A. Peled. *Model Checking*. MIT Press, 2002.
- [10] J. H. Conway. *Regular Algebras and Finite Machines*, Chapman and Hall, London, 1971.
- [11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Conference Record of the *Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 238–252, Los Angeles, California. ACM Press, New York, 1977.
- [12] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

- [13] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1974.
- [14] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.
- [15] S. E. Dunne and W. J. Stoddart, editors. *UTP'06, First International Symposium on Unifying Theories of Programming*, SLNCS, **4010**, Springer Verlag, 2006.
- [16] P. H. B. Gardiner, C. E. Martin and O. de Moor. An algebraic construction of predicate transformers. In *Mathematics of Program Construction, LNCS*, **669**:100-121, Springer-Verlag, 1993.
- [17] G. Gierz, K. H. Hoffmann, K. Keimel, J. D. Lawson, M. Mislove and D. S. Scott. *A Compendium of Continuous Lattices*, Springer Verlag, 1980.
- [18] J. Hartmanis and R. E. Stearns. Pair algebras and their application to automata theory. *Information and Control*, **7**(4):485–507, 1964.
- [19] He Jifeng, X. Li, and Z. Liu. rCOS: A Refinement Calculus for Object Systems. *Theoretical Computer Science*, **365**(1–2):109–142, 2006.
- [20] W. H. Hesselink. *Programs, Recursion and Unbounded Choice*. Cambridge Tracts in Theoretical Computer Science 27, C. U. P., 1992.
- [21] C. A. R. Hoare *et al.* The laws of programming. *Communications of the ACM*, **30**(8):672–686, 1987.
- [22] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [23] C. A. R. Hoare. Why ever CSP? In Proceedings of the Workshop “Essays on Algebraic Process Calculi” (APC 25), *Electronic Notes in Theoretical Computer Science*, **162**:209–215, 2006.
- [24] M. Höltzl and M. Wirsing. State of the art for the engineering of software-intensive systems. Deliverable number D3.1, 31 January 2007. Interlink: Software intensive systems and new computing paradigms. <http://interlink.ics.forth.gr/central.aspx>.
- [25] E. D’Hondt and P. Panangaden. Quantum weakest preconditions. In P. Selinger, editor, *Proceedings of the 2nd Workshop on Quantum Programming Languages (QPL04)*, Turku Centre for Computer Science, 2004.
- [26] Hu Jun, Zhiming Liu, G. M. Reed and J. W. Sanders. Ensemble engineering and emergence. Submitted. Draft available as UNU-IIST Technical Report, **390**, December 2007. <http://www.iist.unu.edu/>.
- [27] C. Jones. *Probabilistic nondeterminism*. Monograph ECS-LFCS-90-105, The University of Edinburgh, 1990. (*PhD*. thesis.)
- [28] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1990.
- [29] A. Kaldewij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.
- [30] D. Kozen. Semantics of probabilistic programs. *Journal of Computer System Sciences*. **22**:328–350, 1981.

- [31] A. M. Lister. *Fundamentals of Operating Systems*, second edition. Macmillan, 1979.
- [32] C. C. Morgan. *Programming from Specifications*, second edition. Prentice-Hall International, 1994.
- [33] C. C. Morgan and T. Vickers. *On the Refinement Calculus*. FACIT series in Computer Science, Springer Verlag, 1994.
- [34] A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer Verlag, 2005.
- [35] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, **9**(3):287–306, 1987.
- [36] G. Nelson. A generalisation of Dijkstra’s calculus. *ACM Transactions on Programming Language and Systems*, **11**(4):517–561, 1989.
- [37] F. Nielsen, H. R. Nielsen and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.
- [38] O. Ore. Galois connexions. *Transactions of the American Mathematical Society*, **55**:494–513, 1944.
- [39] I. M. Rewitzky and J. W. Sanders. Involutions on relational program calculi. UNU-IIST Research Report **391**, February 2008.
- [40] W.-P. de Roeper and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.
- [41] J. W. Sanders and P. Zuliani. Quantum Programming. *Mathematics of Program Construction, 2000*, edited by J. N. Oliveira and R. Backhouse, *LNCS 1837*:80–99, Springer-Verlag, 2000.
- [42] J. W. Sanders. Computations and relational bundles. In *Proceedings of Relational Methods in Computer Science 2006*, edited by R. A. Schmidt, *LNCS 4136*:30–62, Springer-Verlag, 2006.
- [43] K. Seidel and C. C. Morgan. Hierarchical reasoning in probabilistic CSP. *Programmirovaniye*, (Russian journal of Programming and Computer Software), **23**(5), Mezhdunarodnaya Kniga, 1997. Available (in English) at:  
<http://www.cse.unsw.edu.au/~carrollm/probs/Papers/Seidel-97.pdf>.
- [44] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, **14**(4):527–586, 2004.
- [45] J. M. Spivey. *The Z Notation: a reference manual*, second edition. Prentice-Hall International, 1992.