Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 225 (2009) 21–38
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Category Theoretic Models of Data Refinement

Michael Johnson [1,4]

School of Mathematics and Computing
Macquarie University
Sydney 2109, Australia

David Naumann[2,5]

Stevens Institute of Technology
Hoboken NJ 07030, USA

John Power[3,6]

Department of Computer Science
University of Bath
Claverton Down, Bath BA2 7AY, UK

Abstract

We give an account of the use of category theory in modelling data refinement over the past twenty years. We start with Tony Hoare's formulation of data refinement in category theoretic terms, explain how the category theory may be made precise in generality and with elegance, using the notion of structure respecting lax transformation, for a first order imperative language, then study two main alternatives for extending that category theoretic analysis in order to account for higher order languages. The first is given by adjoint simulations; the second is given by the notion of lax logical relation. These provide techniques that can be used for a combined language, such as an imperative language with procedure passing.

Keywords: data refinement, lax natural transformation, adjoint simulation, lax logical relation.

[1] This author acknowledges the support of ARC grants
[2] This author acknowledges the support of NSF grant INT-9813854.
[3] This author acknowledges the support of EPSRC grant GR/586372/01.
[4] Email: mike@ics.mq.edu.au
[5] Email: naumann@cs.stevens-tech.edu
[6] Email: ajp@inf.ed.ac.uk

doi:10.1016/j.entcs.2008.12.064

1 Introduction

In 1987, Tony Hoare wrote a draft paper [9] in which he used category theory to provide an abstract formalism for his development of data refinement over the previous twenty years [8]. As Hoare said in [9], there was evidently a unified body of category theory underlying his constructions, but he was unaware of the details. Prompted by Hoare's question, the third author here wrote an article [32] in which he gave a partial answer by use of universal algebra on enriched categories. That answer was further developed in [17]. But that work only gave a reasonable account of data refinement for imperative languages without any procedure passing. And it was not obvious how to extend that work to account for procedure passing or to account for even a small applicative language.

From that point, there has been a divergence of approach. Naumann has applied Hoare's approach, in its original elementary form, in semantic categories where it is adequate [23,29]. Power, in collaboration with other colleagues [14,31,16,18,34], has developed the notion of lax logical relation. Both of these approaches are category-theoretic in nature, but they are quite different.

In this paper, we outline our various attempts to use category theory to model data refinement, our starting point being Hoare's idea.

In Section 2, we describe a simple imperative language in which we shall frame Hoare's account of data refinement for an imperative language. Hoare identified a language L with the category with structure $\mathbf{C}(L)$ freely generated by L, and he identified a model of L with a structure preserving functor with domain $\mathbf{C}(L)$. We explain by means of a substantial example why that is a reasonable basis on which to add an account of data refinement. Hoare framed everything in terms of an arbitrary language, but for concreteness, we study a simple language.

In Section 3, we introduce Hoare's notion of downward simulation, equivalently structure preserving lax natural transformation, for our imperative language, outline the two leading results, and give an indication how that definition extends to a more general class of imperative languages without procedure passing. Hoare also defined upward simulations, but they are a variant of downward simulations, and essentially the same techniques apply to them; so for simplicity, we shall suppress them here. Downward simulation appears to be inadequate for higher order structure. So we must modify the notion in order to incorporate procedure-passing or applicative languages into our analysis.

In Section 4, we show how lax natural transformations model data refinement for a particular class of non-higher order systems including modern database systems. This application of our categorical model for data refinement has already reached the stage of industrial application, and it shows how simply the lax natural transformation framework applies to an area normally thought of as quite different from programming language semantics.

In Section 5, we give the first of two possible ways to give a category theoretic account of data refinement that applies to higher order functional and imperative languages. This account uses a semantic category in which an additional condition can be imposed on downward simulations without losing expressiveness. We explain how Hoare's results extend to higher order imperative programs in variations on a standard model for program calculi.

Finally, in Section 6, we explain the notion of lax logical relation and show how it allows one to model data refinement for a simple applicative language, specifically one given by the simply typed λ-calculus with some base types and base operators. This is based on the work of [31]. We show how Hoare's two leading theorems extend, and we outline further results to indicate the definitiveness of the notion. The notion of lax logical relation can be extended [16,34] to account for more sophisticated languages involving both functional and imperative features.

2 An imperative language as a category with structure; a model as a structure preserving functor

In this section, for concreteness, we describe a simple imperative language that we will consider in a later section to give a category theoretic account of data refinement. The axiomatic structures we describe are not restricted to those of this language. In particular, we can account for nondeterminism although this is a deterministic language. In fact, we can account for all of Hoare's examples, and our axiomatisation is in the spirit of that he outlined.

The language L_{simple}, is essentially the simple imperative language of Tennent's book [35]. We assume we have primitive data types **unit** and **bool**, together with further primitive data types such as **nat** and abstract data types such as **stack**, and a special type **stat** that represents program states. The judgement "$F\colon \mathbf{op}[\tau,\tau']$" means "$F$ is an operator with arity (τ,τ')," and "$C\colon \mathbf{comm}$" means "C is a command." We assume we have some operators on the data types. Moreover, we assume a denumerable set $I = \{\, \iota_0, \iota_1, \ldots, \iota_n, \ldots \}$ of variable identifiers and a type assignment function π_0 from I to the set of primitive data types. We do not allow $\pi_0(\iota) = \mathbf{stat}$. With this notation, the derived and imperative syntax of L_{simple} appears in Table 1.

We have not included equations between types, between operators, or between commands in our description of L_{simple}. It would be normal to introduce equations, either directly or via an operational semantics, which for instance would imply that the composition of operators of L_{simple} is associative, with unit given by **id**. Evidently, a full language would include such equations. We tacitly assume that any reasonable semantics requires such equations to be satisfied.

We give a semantics for L_{simple} in $\mathsf{Set}_\perp$, the category of pointed sets and $\perp$-preserving functions, as follows.

$$\begin{aligned}
[\![\mathbf{unit}]\!] &\stackrel{\text{def}}{=} \mathbf{1}_\perp = \langle \{\, 1, \perp_{\mathbf{1}} \}, \perp_{\mathbf{1}} \rangle, \\
[\![\mathbf{bool}]\!] &\stackrel{\text{def}}{=} \mathbf{B}_\perp = \langle \{\, \text{true}, \text{false}, \perp_{\mathbf{B}} \}, \perp_{\mathbf{B}} \rangle, \\
[\![\mathbf{stat}]\!] &\stackrel{\text{def}}{=} S \stackrel{\text{def}}{=} \Pi_{i \in I} [\![\pi_0(\iota_i)]\!], \qquad \text{the product in } \mathsf{Set}_\perp,
\end{aligned}$$

with the other data types free to be modelled by any objects of $\mathsf{Set}_\perp$. The denotation

$$\frac{}{!\colon \mathbf{op}[\mathbf{stat},\mathbf{unit}]} \quad \frac{}{\mathbf{id}\colon \mathbf{op}[\tau,\tau]} \qquad \frac{F'\colon \mathbf{op}[\tau',\tau''] \quad F\colon \mathbf{op}[\tau,\tau']}{F'\,F\colon \mathbf{op}[\tau,\tau'']}$$

$$\frac{\pi_0(\iota)=\tau}{\iota\colon \mathbf{op}[\mathbf{stat},\tau]} \qquad \frac{B\colon \mathbf{op}[\tau,\mathbf{bool}] \quad F_0\colon \mathbf{op}[\tau,\tau'] \quad F_1\colon \mathbf{op}[\tau,\tau']}{\mathbf{if}\ B\ \mathbf{then}\ F_0\ \mathbf{else}\ F_1\colon \mathbf{op}[\tau,\tau']}$$

$$\frac{}{\mathbf{skip}\colon \mathbf{comm}} \qquad \frac{}{\mathbf{diverge}\colon \mathbf{comm}}$$

$$\frac{E\colon \mathbf{op}[\mathbf{stat},\tau] \quad \pi_0(\iota)=\tau}{\iota\colon = E\colon \mathbf{comm}} \qquad \frac{C_0 : \mathbf{comm} \quad C_1 : \mathbf{comm}}{C_0\mathbf{;}C_1\colon \mathbf{comm}}$$

$$\frac{B\colon \mathbf{op}[\mathbf{stat},\mathbf{bool}] \quad C_0\colon \mathbf{comm} \quad C_1\colon \mathbf{comm}}{\mathbf{if}\ B\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1\colon \mathbf{comm}} \qquad \frac{B\colon \mathbf{op}[\mathbf{stat},\mathbf{bool}] \quad C\colon \mathbf{comm}}{\mathbf{while}\ B\ \mathbf{do}\ C\colon \mathbf{comm}}$$

Table 1
The derived and imperative syntax of a simple imperative language

of an operator of arity (τ,τ') is a morphism in $\mathsf{Set}_\perp$ from $[\![\tau]\!]$ to $[\![\tau']\!]$, with the semantics of derived operators generated as follows.

$$\begin{aligned}
[\![!]\!] &\overset{\text{def}}{=} [s \longmapsto 1],\\
[\![\mathbf{id}]\!] &\overset{\text{def}}{=} \mathrm{id}_{[\![\tau]\!]},\\
[\![F'\,F]\!] &\overset{\text{def}}{=} [\![F']\!]\circ[\![F]\!] \qquad \text{if } F\colon \mathbf{op}[\tau,\tau'] \text{ and } F'\colon \mathbf{op}[\tau',\tau''].\\
[\![\iota_n]\!] &\overset{\text{def}}{=} [s \longmapsto s_n] \qquad \text{where } s_n \text{ is the } n\text{-th component of } s,\\
[\![\mathbf{if}\ B\ \mathbf{then}\ F_0\ \mathbf{else}\ F_1]\!] &\overset{\text{def}}{=} \left[t \longmapsto \begin{cases} \perp & \text{if } [\![B]\!](t)=\perp,\\ [\![F_0]\!](t) & \text{if } [\![B]\!](t)=\text{true},\\ [\![F_1]\!](t) & \text{if } [\![B]\!](t)=\text{false} \end{cases} \right]\\
&\qquad \text{where } t \text{ is an element of } [\![\tau]\!],
\end{aligned}$$

The denotation of a command is defined to be an endomorphism on S in $\mathsf{Set}_\perp$ as follows. In the definition for $\iota_n\colon = E$, we take tupling to be strict.

$$\begin{aligned}
[\![\mathbf{skip}]\!] &\overset{\text{def}}{=} \mathrm{id}_S,\\
[\![\mathbf{diverge}]\!] &\overset{\text{def}}{=} [s \longmapsto \perp_S],\\
[\![\iota_n\colon = E]\!] &\overset{\text{def}}{=} [s_0 s_1 \ldots s_n \ldots \longmapsto s_0 s_1 \ldots s_{n-1}\,[\![E]\!](s)\,s_{n+1}\ldots],\\
[\![C_0\mathbf{;}C_1]\!] &\overset{\text{def}}{=} [\![C_1]\!]\circ[\![C_0]\!],\\
[\![\mathbf{if}\ B\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1]\!] &\overset{\text{def}}{=} \left[s \longmapsto \begin{cases} \perp & \text{if } [\![B]\!](s)=\perp,\\ [\![C_0]\!](s) & \text{if } [\![B]\!](s)=\text{true},\\ [\![C_1]\!](s) & \text{if } [\![B]\!](s)=\text{false} \end{cases} \right],
\end{aligned}$$

$$[\![\textbf{while } B \textbf{ do } C]\!] \overset{\text{def}}{=} \bigsqcup_{0 \le n} [\![C_n]\!],$$

where C_n in the definition of the while statement is defined inductively as follows:

$$C_0 \overset{\text{def}}{=} \textbf{diverge}, \qquad C_{n+1} \overset{\text{def}}{=} \textbf{if } B \textbf{ then } (C\textbf{;}C_n) \textbf{ else skip},$$

and $\bigsqcup$ means the least upper bound of this chain of morphisms in $\mathsf{Set}_\perp$.

This gives the traditional semantics for L_{simple}. Observe that the locally ordered structure of the category $\mathsf{Set}_\perp$ is essential to model **while**. The semantics can also be described as a structure preserving functor with domain $\mathbf{C}(L_{\text{simple}})$, which is defined as follows.

Definition 2.1 The syntactic category $\mathbf{C}(L_{\text{simple}})$ is the locally ordered category with structure freely generated by the graph whose nodes are data types of L_{simple}, and whose edges from τ to τ' are operators of arity (τ, τ'); subject to making **skip** the identity, **diverge** the least element, assignments modelled by countable products, sequence (**;**) by composition, **if** statements by finite coproducts, **while** statements by least upper bounds, and similarly for the operators, subject to the equations determined by the full language.

Our semantic functions determine a structure preserving locally ordered functor from $\mathbf{C}(L_{\text{simple}})$ to the locally ordered category $\mathsf{Set}_\perp$. Conversely, any structure preserving locally ordered functor from $\mathbf{C}(L_{\text{simple}})$ to $\mathsf{Set}_\perp$ determines the semantic functions. So Hoare *identified* the language L_{simple} with the locally ordered category $\mathbf{C}(L_{\text{simple}})$ and he *identified* the semantics for L_{simple} with the corresponding structure preserving locally ordered functor from $\mathbf{C}(L_{\text{simple}})$ to $\mathsf{Set}_\perp$. We have only used specified structure on the locally ordered category $\mathsf{Set}_\perp$, so this correspondence generalises from semantics in $\mathsf{Set}_\perp$ to semantics in any locally ordered category A with the requisite structure: see [17] for a precise statement of a wide generality of this phenomenon and for a succession of examples.

3 Data refinement for an imperative language

We abbreviate the terminology L_{simple} for our simple imperative language to L, and we let $M, N\colon \mathbf{C}(L) \to \mathsf{A}$ be two models of L, that is, structure preserving locally ordered functors to a locally ordered category A. For M to be a refinement of N, Hoare asked for a family of maps $\{\, \rho_a\colon M(a) \to N(a) \mid a \in |\mathbf{C}(L)| \,\}$. The idea was that ρ_a says which value of $N(a)$ is represented by a given value in $M(a)$. One might ask that this be a natural transformation, but Hoare wanted to relax the naturality condition in order to allow M to be more defined than N.

To illustrate this, consider a language for which there is a type **stack** and there are operators **empty**: **op**[**unit**, **stack**] and **pop**: **op**[**stack**, **stack**]. Regarding the order as "degree of definedness," let N take **pop empty** to the least element since we want N to leave it undefined. On the other hand, we do not care what value the representation M takes. The more refined M may also leave **pop empty** undefined, or it may take it to an arbitrary element. Thus, we only require

$$N(\mathbf{pop}) \circ \rho_{\mathbf{stack}} \leq \rho_{\mathbf{stack}} \circ M(\mathbf{pop}), \tag{1}$$

not equality, as would be required for a natural transformation. This argument generalises from definedness to other notions of correctness that can be modelled by an ordering $\leq$.

More concretely, let $\mathsf{A} = \mathsf{Set}_\perp$ as in Section 2, define $M, N \colon \mathbf{C}(L) \to \mathsf{Set}_\perp$ by observing that $M(\mathbf{unit}) = N(\mathbf{unit}) = \mathbf{1}_\perp$ since this is determined by the semantics of L, and putting

$$\begin{aligned} N(\mathbf{stack}) &= \mathbf{N}_\perp, \\ N(\mathbf{empty})(1) &= 0, \\ N(\mathbf{pop})(n+1) &= n, \\ N(\mathbf{pop})(0) &= \perp_{\mathbf{N}}, \\ M(\mathbf{stack}) &= \langle\{\text{ finite binary trees }\}, \text{ the empty tree}\rangle, \\ M(\mathbf{empty})(1) &= *, \quad \text{the one point tree}, \\ M(\mathbf{pop})(\langle t_0, t_1\rangle) &= t_1, \\ M(\mathbf{pop})(*) &= *, \end{aligned}$$

and defining $\rho_{\mathbf{stack}} \colon M(\mathbf{stack}) \to N(\mathbf{stack})$ by sending a tree to the number of edges in its rightmost branch. This definition extends naturally to $\mathbf{C}(L)$ and condition (1) holds. So M is more defined than N at **pop**; $M(\mathbf{pop})$ takes $*$ to $*$ while $N(\mathbf{pop})$ takes 0 to $\perp_{\mathbf{N}}$, so $N(\mathbf{pop}) \circ \rho_{\mathbf{stack}} \leq \rho_{\mathbf{stack}} \circ M(\mathbf{pop})$. Since we do not care what value interpretations assign to illegal terms such as **pop empty**, we say M gives a representation of N, and ρ gives a refinement. Yet ρ is not natural; in fact, there is no natural transformation from M to N. However, ρ is a lax transformation.

Extending our example, when A is a locally ordered category of sets with structure, lax transformations may be understood as follows. If there is a lax transformation $\alpha \colon M \Rightarrow N \colon \mathbf{C}(L) \to \mathsf{A}$ from M to N, $M(a)$ is regarded as a representation of $N(a)$ and $\alpha_a \colon M(a) \to N(a)$ maps each value in $M(a)$ to what it represents, for each object a of $\mathbf{C}(L)$. Hoare used the term downward simulation for lax transformation. The term forward simulation has come into common use for this notion, along with backward or upward simulation for the dual [5]. The terms history and prophecy are also in common use [19] for the two. Some sources use upward and downward in the reverse senses.

A more natural category theoretic definition, and one that allows stronger results (see [18]) is to define a *downward simulation* to be a *structure respecting* lax natural transformation from the structure preserving functor M to the structure preserving functor N. So we take that to be our definition here. The main results Hoare sought were

Theorem 3.1 • *If $\alpha : M \Rightarrow N$ and $\beta : N \Rightarrow P$ are downward simulations, then the composite $\beta\alpha$ is a downward simulation from M to P*

- *Every downward simulation defined on base types and base operators extends to a downward simulation defined on all types and all operators.*

The first of these is trivial; the second requires proof and depends upon the structure of L. It is of pragmatic importance because L typically has only a small finite number of base types and base operators, so checking whether something is a structure respecting lax transformation when restricted to base types and operators is feasible, whereas $\mathbf{C}(L)$ is infinite, as one can apply the type constructors arbitrarily many times. So one cannot verify that one has a structure respecting lax transformation based on $\mathbf{C}(L)$ without some result relating the infinite to the finite, and any reasonable account of data refinement must have an account of which type constructors and operators allow such a result.

If L has only finite product types and constant commands, a unique extension of a structure respecting lax transformation from base types and operations to all of $\mathbf{C}(L)$ always exists. However, if, for instance, L had a contravariant construction or a construction of mixed variance such as that given by higher order types, then an extension generally does not exist. So Hoare asked whether one can give a precise, general account of which structures allow such unique extensions. The paper [17] was devoted to answering Hoare's question in terms of enriched algebraic structure on the category of small locally ordered categories, and by an analysis of a precise concept of sketch of such structure.

The key points are

- the structures one may consider in defining a language L and its semantics are those given by *enriched algebraic structure* $\langle S, E\rangle$ on the category LocOrd_l
- a category theoretic formulation of the notion of base types and base operators for specified algebraic structure $\langle S, E\rangle$ is that of $\langle S, E, \mathcal{D}\rangle$-sketch.

These definitions support the two theorems sought by Hoare; the paper [17] gives the details and the examples: the examples include all of Hoare's examples in which the structure is covariant, so that includes all the structure of the simple imperative language L_{simple}, together with that of a powerdomain to model nondeterminism. But it specifically does not include function types, and there have been two approaches to including them, which we investigate in Sections 5 and 6.

4 Database data refinement and lax natural transformations

Before we proceed to deal with function types, this section presents an example application of our categorical model of data refinement. The example illustrates the use of a simple covariant database language defined in the framework described above — the structure of the database is presented as a category with structure, models (also called "snapshots") of the database are structure preserving functors (usually valued in the category of finite sets, or of finite sets and partial functions), and data refinements are given by (structure preserving) lax natural transformations. In the following paragraphs we describe each of these in more detail, and we end the section by indicating how this approach has been of benefit in industry. A more rigorous presentation of the basics of this approach can be found in [12].

Databases are frequently specified by giving entities, relationships between entities, and attributes of entities. This information can be summarised in a directed graph which has entities, relationships, and attribute value sets as nodes, together with edges from each entity to each of its attribute value sets, and from each relationship to each of the entities that it relates. The widely used *entity-relationship modelling technique* is an attempt to discover a graph which in this way best represents the structure of the data which are to be stored in the database. But of course in a real world system the data will be required to satisfy many constraints which are not recorded in the ER graph, including requirements that attributes can only take a certain number of known values, that some attributes depend on several entities (not just one), and that certain composites of relations must be equal. Write L for the graph along with the extra constraints. L is a representation of the language available for a particular database.

What structure should the category with structure $\mathbf{C}(L)$ bear? We will need a terminal object 1 so that we can talk about elements, finite coproducts so that we can construct attribute value sets as fixed coproducts of 1, finite products to support attributes that depend upon several entities, and pullbacks to support relational composition. So we require $\mathbf{C}(L)$ to be the category with finite coproducts and finite limits freely generated by L. By the way, it was an important observation early in the development of categorical specifications of information systems that the category $\mathbf{C}(L)$ has an object corresponding to each of the structural queries that can be applied to a database with structure L [4]. Thus $\mathbf{C}(L)$ is a language in another sense — it embodies the query language for the database with structure L.

As usual, models of $\mathbf{C}(L)$ are structure preserving functors $\mathbf{C}(L) \to \mathsf{A}$ for a locally ordered category A, typically the category of pointed sets, or equivalently the category of sets and partial functions. Because the functors preserve finite coproducts the attribute value sets remain constant up to canonical isomorphisms in all models of a given database, while an entity or a relationship may be modelled by sets of various cardinalities in different models as instances of that entity or relationship are inserted or deleted from the database.

If there is a lax transformation $\alpha\colon M \Rightarrow N\colon \mathbf{C}(L) \to \mathsf{A}$ between two models M and N we say that M is a data refinement of N. Since the use of a natural transformation here is well understood, we concentrate for a moment of the effect of the laxness of that transformation. The laxness, a generalisation of inequality (1) above, says, if we view the order on morphisms of A as definedness, that M may be more defined than N. This is particularly important in database applications where attributes which have unknown values in one model are often updated to take known (more defined) values in another model.

How has this approach been useful in industry? As noted above entity-relationship modelling, useful as it is, neglects a range of important constraints. Including those constraints in a model, in a formal and consistent way, not only completes the model but frequently reveals inadequacies in the model under development. This development approach has been used in large scale contracts with industry including a major telecommunications carrier, an oil company, and a government department of health, among other organisations. It has consistently added value in the analysis and specification stage.

Of course this example applications has only involved first order operations. We now proceed to extend data refinement to higher order constructs.

5 Downward simulations for higher order imperative programs

Hoare noted that in general a downward simulation on base types and operators may fail to have an extension to the full language, if the constructs include contravariant structure such as exponentials. Exploiting the notion of embedding-projection pair familiar from domain theory, Hoare defined "total simulations" as downward simulations α such that each component α_a is a projection, and showed that extensions exist for total simulations even when the language includes contravariant constructs. Naumann noted that is sufficient to require each α_a to be an internal right adjoint, i.e., there exists $\alpha_a^o\colon N(a) \to M(a)$ with $id \leq \alpha_a \circ \alpha_a^o$ and $\alpha_a^o \circ \alpha_a \leq id$. This observation was also made by Kinoshita and Power in [15,17], who showed that Theorem 3.1 also holds for contravariant constructs if simulations are restricted to *adjoint simulations*.

In his dissertation [22], Naumann also generalizes Hoare's development to 2-categories, in the following way. The semantic category is taken to be a 2-category, so inequalities are replaced by 2-cells. Structures are required to satisfy standard coherence conditions, as are downward and adjoint simulations. The leading application is the idea that a 2-cell represents a proof of refinement, but this application is not developed formally. In the elementary style of Hoare's manuscript, it is shown that the coherence conditions are preserved by all the constructs considered by Hoare, as well as various additional ones that arise in 2-categories. Until now a precise general formulation has only been developed for locally ordered categories, and there is no pressing need for more.

Hoare wanted to classify which proof techniques can be used with which constructs. The connection between contravariance and adjoint simulation could be part of such a classification, and it is not surprising that higher order structure imposes stronger requirements than those needed for first order. On the other hand, adjoint simulations are unacceptably restrictive in some of the leading models. This led Power to pursue what appears to be a very different approach, lax logical relations (see Section 6). Naumann focused on semantic categories where adjoint simulation is useful. To introduce those categories in a way that highlights the connection with logical relations, we begin by reconsidering the results of Section 3.

Recall from Section 3 that Hoare's formulation involves a single semantic category A in which the downward simulation condition can be expressed as an inequation $N(\mathbf{S}) \circ \alpha \leq \alpha \circ M(\mathbf{S})$ where program denotations, such as $N(\mathbf{S})$, are composed with morphisms α that connect two interpretations of the language. This is not adequate if there are not enough morphisms to make the desired connections. (In this paper we treat adequacy informally, but our remarks are justified by completeness results in the literature.) If we take A to be $\mathsf{Set}_\perp$ and restrict ourselves to adjoint simulations, we are left with little more than isomorphisms, which are certainly not adequate to account for different data representations that are observably indistinguishable.

Let us consider the alternative of logical relations, well known for functional programs. For simplicity let us ignore divergence and consider a language such as simply typed lambda calculus, with non-divergent base operators, so we can use interpretations M, N into Set rather than $\mathsf{Set}_\perp$. A logical relation consists of, for each type a, a binary relation $R_a \subseteq M(a) \times N(a)$, such that for each program $\mathbf{S} : a \to b$ we have

$$x\ R_a\ y \Longrightarrow M(\mathbf{S})x\ R_b\ N(\mathbf{S})y \qquad \text{for all } x, y. \tag{2}$$

Some authors prefer to emphasize that R and $M(\mathbf{S})$ play different roles; this has led to the formulation found in Section 6. But the logical relation condition (2) can also be expressed as a downward simulation. Let Rel be the category of binary relations between sets, locally ordered by $\subseteq$. Owing to the inclusion of Set in Rel, we can consider the semantic functors M, N to be into Rel. So we can compose their images with relations, as in the following.

(3)
$$\begin{tikzcd} M(a) \arrow[r, "M(\mathbf{S})"] \arrow[d, "R"'] \arrow[dr, phantom, "\subseteq"] & M(b) \arrow[d, "R"] \\ N(a) \arrow[r, "N(\mathbf{S})"'] & N(b) \end{tikzcd} \qquad \text{i.e.,} \quad N(\mathbf{S}) \circ R \subseteq R \circ M(\mathbf{S})$$

This inequality is equivalent to (2). By embedding the category for program semantics (in this case, Set) in a larger category (Rel), we have expressed the desired connection as a downward simulation square.

In Rel, the adjoint morphisms are the total functions, and total functions are not even adequate for deterministic first order programs [5,19]. But the embedding idea can be carried further. We can embed Rel in a still larger category, where relations are not only morphisms but adjoint ones. The particular examples studied by Naumann are categories of predicate transformers. Before giving their general construction, we follow the historical path and describe predicate transformers in elementary terms.

The weakest-precondition function $wp(\mathbf{S})$ associated with a command $\mathbf{S}$ maps postconditions to preconditions. As a semantic model, predicate transformers have the attraction that they adequately model divergence and nondeterminacy without the need for lifting or powersets [30,10]. For the language of Section 2, let us take "predicate" to mean a subset p of $[\![\mathbf{stat}]\!]$, not containing $\perp$. Then $wp(\mathbf{S})(p)$ is the inverse image of $M(\mathbf{S})$ on p. Because predicate transformers are functions from post-conditions to pre-conditions, it is natural to describe them in terms of an opposite category. Let PSpec have sets as objects, and let $\mathsf{PSpec}(X, Y)$ be the set of monotonic functions from the powerset $\mathbb{P}(Y)$ to $\mathbb{P}(X)$. Each homset is locally ordered, with $\leq$ the pointwise order with respect to inclusion $\subseteq$ of sets. The name

PSpec alludes to the fact that this category contains not only the denotations of programs—it also models total correctness specifications [1,5] in such a way that the order $\leq$ represents both satisfaction of specifications and algorithmic refinement of programs.

The right adjoint morphisms in PSpec are the completely disjunctive functions. The completely disjunctive functions in $\mathsf{PSpec}(X, Y)$ are in order isomorphism with $\mathsf{Rel}(Y, X)$; the isomorphism sends a relation to its direct image function, the left adjoint of which is its inverse image function. This is the key to Naumann's approach to higher order structure:

- Embed the semantic category in one where the desired downward simulations are adjoint simulations, and check that
- the larger category has the structure and properties needed for extension of adjoint simulations.

The example at hand is an instance of a general construction. Recall that Rel can be constructed as spans over Set, and Set embeds into Rel as the right adjoint morphisms (right or left, depending on how one chooses to formulate the definitions). The span construction can be generalized to a notion of "skew span" which can be applied to Rel to yield PSpec [6].

Besides the availability of more adjoint simulations, the benefit of embedding in a category with more morphisms on the same objects is that a richer language can be interpreted, e.g., PSpec models both angelic and demonic nondeterminacy. The cost is that some structures have weaker properties. Cartesian product of sets is an important example of this weakening. In PSpec, product forms a very lax kind of adjunction. For example, one expects the inequality $\pi \circ (f, g) \leq f$ (because g could diverge) and the inequality $((\pi \circ h), (\pi' \circ h)) \leq h$ (because h could be nondeterministic); these were among Hoare's leading examples. But in PSpec even these inequalities are conditional. Because PSpec includes arbitrary monotonic functions, it includes morphisms that exhibit "miracles" and "angelic nondeterminacy" [10]. This is needed to model specifications [1], but it means, for example, that the first projection law $\pi \circ (f, g) \leq f$ holds only if f, g are $\emptyset$-strict. Strictness of f, sometimes called the "law of the excluded miracle", can be expressed as $f \circ \mathbf{diverge} = \mathbf{diverge}$.

Using elementary proofs in the manner of Hoare's manuscript, Naumann showed that downward simulations extend even with these very lax products [23]. It is believed that the general theory applies to them [13], but the details have not been spelled out explicitly.

For a treatment of higher types in PSpec, the skew span construction can be used to lift structure from Set to PSpec [6]. This lifting does not create new objects, so the lifted exponent is just the function space. That is enough to treat a simply typed lambda calculus using adjoint simulations, by composing a semantics $M : \mathbf{C}(L) \to \mathsf{Set}$ with the embedding of Set in PSpec. But for imperative programs, which denote morphisms of PSpec, the arrow type should be interpreted by the internal hom of PSpec. This is a very weak adjoint to the lax product, and it is difficult to find a useful axiomatization even using conditional inequations [24]. In particular, it is not functorial, so we cannot apply the extension result for which we sought adjoint simulations.

A reason for the weak properties can be found in the definition of PSpec: its objects are powersets, but taking the powerset of an ordered homset utterly neglects its order structure. This brings to mind a slightly more refined category Spec: Objects are posets, and $\mathsf{Spec}(X, Y)$ is the set of monotonic functions $\mathcal{U}Y \to \mathcal{U}X$ where $\mathcal{U}X$ is the lattice of updeals on X, ordered by $\subseteq$. Naumann originally approached Spec in these elementary terms, but it was later found to be an instance of the skew span construction, over Poset instead of Set [25].

The lax product in Spec behaves like the one in PSpec, but exponents have slightly better properties; in particular, the action on morphisms is functorial. Finally, we get Naumann's treatment of higher types [23] for simply typed higher order imperative languages.

Theorem 5.1 *Every adjoint simulation on base types extends to one for the language including exponents (Currying, application, and the arrow functor).*

Apropos the first item in Theorem 3.1, the situation is clear cut: adjoint simulations do compose.

Although Theorem 5.1 is expressed in terms of exponents, the original interest in predicate transformers came from imperative languages. The first author used the model Spec to give semantics for a conventional imperative language like Modula-3 in which procedures can be stored in state variables and passed as arguments [27], and he showed that extensions for adjoint simulations exist for this language [29]. But, because procedures are allowed to have global variables as well as parameters, the connection with exponents is somewhat indirect, and categorical aspects are suppressed in the cited work. Only later was the connection with lambda calculus presented explicitly [28].

As a bridge to the next section, let us review the preceding discussion. The starting point is a pair of structure preserving functors M, N from $\mathbf{C}(L)$ to a semantic category A, along with a downward simulation α defined only on the base types and operations of L. Inequation squares like (3) are the primary objects of interest, so let us make them into a category Sims in a standard way: objects are morphisms of A, morphisms are pairs of morphisms making inequation squares. With δ_0, δ_1 the evident projections from Sims to A, a downward simulation $\alpha : M \Rightarrow N$ amounts to a functor $\hat{\alpha}$ from L to Sims such that $M = \delta_0 \circ \hat{\alpha}$ and $N = \delta_1 \circ \hat{\alpha}$. The picture looks like this.

$$\begin{tikzcd} & \mathsf{Sims} \arrow[d, "(\delta_0,\delta_1)"] \\ \mathbf{C}(L) \arrow[ur, "\hat{\alpha}"] \arrow[r, "(M,N)"'] & \mathsf{A} \times \mathsf{A} \end{tikzcd}$$

(Although this description of downward simulations has been used primarily for the case where A is Cartesian closed [7], the example of Spec shows that it works more generally.) The main objective is to extend $\hat{\alpha}$, defined only on base types and base operations, to all of $\mathbf{C}(L)$. As soon as A has the requisite structure for $M(\mathbf{S}), N(\mathbf{S})$ to be definable for some program $\mathbf{S}$, that structure can also be used to define the relevant components of α. Whether the resulting square is an inequality, so that $\hat{\alpha}(\mathbf{S})$ is defined, depends on properties of the structure on A —that is the content of the extension theorems. As indicated in the statement of Theorem 3.1, the goals are for an extension to exist, and for simulations to compose. This does not imply that simulations must respect all structure used for obtaining extensions, and this is exploited in Section 6.

To conclude this section, let us consider a sort of higher order data refinement, the refinement of a construct, specifically the exponent. Readers familiar with predicate transformer semantics will have noted that the denotations of programs are not arbitrary morphisms in Spec; they enjoy additional healthiness conditions—preserving $\emptyset$ and distributing through nonempty intersections and unions of ascending chains [30,10]. The main motivation to use all monotonic functions is to model specifications in calculi of program refinement. The internal hom of Spec is then useful due to its strong properties; besides Theorem 5.1, we mention that it yields Lawvere's recursion theorem for inductive data types [26]. On the other hand, for fully abstract program semantics we should pay attention of the embedding of program denotations as some full on objects subcategory Prog of Spec. Here Prog could be, e.g., Poset, for simply typed functional programs, or the subcategory of Spec given by the above healthiness conditions for nondeterministic imperative programs. If we take the semantics of arrow types to be the internal hom of Prog, then we need an account of the connection with the internal hom of Spec; this essentially internalizes the embedding of Prog in Spec. An account has been given in [26, Section 7] in elementary terms at a level of generality similar to Hoare's draft, and an account is given in [28] for the specific example of Poset and simply typed lambda calculus. A more general account would be very welcome. In particular, to treat recursive types one would like an account that applies to skew span categories over CPOs.

6 Data refinement for an applicative language using lax logical relations

In this section, we wish to maintain Hoare's basic approach as best we can, but specifically accounting for higher order types. So, for simplicity of exposition (but see [16] and [34] for more sophisticated treatments), we restrict attention to the simply typed λ-calculus with products on some base types and some base operators, and we call such a language L_λ. As in Section 2, Hoare's identification of a language with the category with structure freely generated by it applies equally here, as does his identification of a model with a structure preserving functor out of L_λ: so we may identify L_λ with the free cartesian closed category $\mathbf{C}(L_\lambda)$ on L_λ, and we may identify a model of L_λ with a cartesian closed functor with domain $\mathbf{C}(L_\lambda)$. We have already explained how to treat recursion in Section 2, so we shall not clutter this section with a repetition of it. So, for ease of exposition here, we shall ignore local order structure, and take our models in Set. And for notational simplicity, we shall abbreviate L_λ by L.

As remarked by Hoare in [9], downward simulation is not respected by higher order structure. But the more general notion of *logical relation* was designed specifically so that higher order structure respects it [21], and therefore the notion of logical relation yields the second part of Hoare's Theorem 3.1. However, logical relations do not compose, so they do not admit the first part of Theorem 3.1, and therefore do not model data refinement as Hoare and as we understand it. So, after a few attempts, notably [14], the third author here with some colleagues developed the notion of *lax logical relation* [31], a mild generalisation of the notion of logical relation, but one that satisfies both of Hoare's criteria. So we outline that development in category theoretic terms here.

Claudio Hermida, in his thesis [7], showed that logical relations may be expressed in category theoretic terms. The heart of his analysis consists of the following definition and proposition when considered in the setting of our cartesian closed category $\mathbf{C}(L)$.

Definition 6.1 The category Rel_2 is defined as follows: an object consists of a pair (X, Y) of sets and a binary relation R from X to Y; a map from (X, R, Y) to (X', R', Y') is a pair of functions $(f : X \longrightarrow X', g : Y \longrightarrow Y')$ such that $x\,R\,y$ implies $f(x)\,R'\,g(y)$; composition is given by ordinary composition of functions. We denote the forgetful functor from Rel_2 to $\mathsf{Set} \times \mathsf{Set}$ sending (X, R, Y) to (X, Y) by $(\delta_0, \delta_1) : \mathsf{Rel}_2 \longrightarrow \mathsf{Set} \times \mathsf{Set}$.

It is folklore and routine to verify that the category Rel_2 is cartesian closed, and the cartesian closed structure is preserved by (δ_0, δ_1).

Proposition 6.2 *To give a logical relation from M to N is equivalent to giving a functor $R : \mathbf{C}(L) \longrightarrow \mathsf{Rel}_2$ strictly preserving cartesian closed structure, such that $(\delta_0, \delta_1)R = (M, N)$.*

$$\begin{tikzcd}
 & \mathsf{Rel}_2 \arrow[d, "{(\delta_0,\delta_1)}"] \\
\mathbf{C}(L) \arrow[ur, "R"] \arrow[r, "{(M,N)}"'] & \mathsf{Set} \times \mathsf{Set}
\end{tikzcd}$$

We take Hermida's category theoretic formulation of the notion of logical relation as a *definition* of logical relation, and we generalise it to define a notion of lax logical relation as follows.

Definition 6.3 A *lax logical relation* from M to N is a functor $R : \mathbf{C}(L) \longrightarrow \mathsf{Rel}_2$

strictly preserving finite products such that $(\delta_0, \delta_1)R = (M, N)$.

$$\begin{tikzcd}
 & \mathsf{Rel}_2 \arrow[d, "(\delta_0,\delta_1)"] \\
\mathbf{C}(L) \arrow[ur, "R"] \arrow[r, "(M,N)"'] & \mathsf{Set} \times \mathsf{Set}
\end{tikzcd}$$

The notion of Henkin model is closely related to this definition. A Henkin model of simply typed λ-calculus is a finite product preserving functor from $\mathbf{C}(L)$ to Set such that the induced lax maps are injective. This is a kind of lax model, but is not quite the same as giving a unary lax logical relation; nevertheless, it is a natural and useful generalisation of the notion of model we have used, and one to which our results routinely extend.

Hoare's Theorem 3.1 extends to lax logical relations.

Theorem 6.4 • *If $R : M \Rightarrow N$ and $S : N \Rightarrow P$ are lax logical relations, then pointwise composition of relations yields a lax logical relation $R; S$ from M to P*

- *Every lax logical relation defined on base types and base operators extends to a lax logical relation defined on all types and all operators.*

We have further theorems to indicate the definitiveness of the notion of lax logical relation too.

Theorem 6.5 • *(The Basic Lemma for Lax Logical Relations) A family of relations*

$$R(a) \subseteq M(a) \times N(a)$$

for every object a of $\mathbf{C}(L)$ determines a lax logical relation from M to N if and only if for every arrow $f : a \longrightarrow b$ in $\mathbf{C}(L)$, if $x\,R(a)\,y$, then $M(f)x\,R(b)\,N(f)y$.

- *A family of relations*

$$R(a) \subseteq M(a) \times N(a)$$

for every object a of $\mathbf{C}(L)$ determines a lax logical relation from M to N if and only if it determines a pre-logical relation from M to N

- *Every lax logical relation is a composite of at most three logical relations.*

The last of these results is the central mathematical result about pre-logical relations in [11].

7 Further Work

We have done our best in this paper to present our various approaches to data refinement in a unified way, focusing on what unites our work rather than what divides it. But for further work, it is perhaps more instructive to consider briefly what divides us as that yields loose ends, open problems, and possible further directions.

Naumann and Power's developments of Hoare's ideas have most in common as both have studied higher-order programming. But they have used different semantics: Naumann has adapted predicate transformer semantics whereas Power has adapted logical relations. So one sensible task would be to make more precise the relationship between the two extensions they have proposed. Such a relationship would provide theoretical support to complementary perspectives, allowing either to be used, depending upon the specific question at hand.

Johnson, in contrast, has focused on database refinement. That has the attribute of being closer to practice than either Naumann or Power's work, but it also has correspondingly less theoretical development. So one wonders whether Naumann and Power's higher-order analysis might impact on Johnson's database work, in particular on entity-relationship modelling. More precisely, can Naumann and Power account for databases? and can Johnson see a role for higher-order techniques?

All three authors of this paper have been surprised and impressed by the amount we have in common. But all three developments give rise to their own questions too. For instance, data refinement is an ubiquitous concept, so for any of the many extensions of higher-order programming, one can ask for an extended theory of data refinement. Equally, industry uses sophisticated combinations of database languages, and each of them requires an analysis of data refinement too.

An industrially critical language feature not modelled in the work discussed here is mutable heap objects. For data refinement, one seeks a notion of relation that is locally supported [20,2] in the sense of separation logic. To extend to heaps the categorical approaches to data refinement, it could be fruitful to draw on recent work by Birkedal and Yang [3] and by Power [33].

Essentially, data refinement is fundamental to our understanding of programming, and the central assertion of this paper is that Hoare's ideas provide a fine springboard for theoretical, specifically category theoretic, support for it.

References

[1] Back, R., *A calculus of refinements for program derivations*, Acta Informatica **25** (1988), 593–624.

[2] Banerjee, A., and D. A. Naumman, *Ownership Confinement Ensures Representation Independence for Object-Oriented Programs*, J. ACM **52** (2005) 894–960.

[3] Birkedal, L., and H. Yang, *Relational Parametricity and Separation Logic*, "Proc. FOSSACS 2007'," Lecture Notes in Computer Science **4423** (2007) 93–107.

[4] Dampney, C., M. Johnson, and G. Monro, *A mathematical foundation for ERA*, "Proceedings of the Institute for Mathematics and its Applications" **35**, Oxford Univ. Press, 1992, 77–84.

[5] de Roever, W. P., and K. Engelhardt, "Data Refinement: Model-Oriented Proof Methods and their Comparison, " Cambridge Univ. Press, 1998.

[6] Gardiner, P. H., C. E. Martin, and O. de Moor, *An algebraic construction of predicate transformers*, Science of Computer Programming, **22** (1994), 21–44.

[7] Hermida, C. A., "Fibrations, logical predicates, and indeterminates," PhD thesis, The University of Edinburgh, 1993, published as CST–103–93, also as ECS–LFCS–93–277.

[8] Hoare, C., *Proof of correctness of data representations*, Acta Informatica **1** (1972) 271–281.

[9] Hoare, C., "Data refinement in a categorical setting," unpublished manuscript, 1987.

[10] Hoare, C. A. R., *Some properties of predicate transformers*, J. ACM **25** (1978) 461–480.

[11] Honsell. F., and D. Sannella, *Pre-logical relations*, "Computer Science Logic 1999," Lecture Notes in Computer Science **1683** (1999), 546–561.

[12] Johnson, M., and R. Rosebrugh, *View updatability based on the models of a formal specification.* Lecture Notes in Computer Science **2021** (2001), 534–549.

[13] Kelly, G. M., and A. J. Power, *Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads*, Journal of Pure and Applied Algebra **89** (1993), 163–179.

[14] Kinoshita, Y., P. O'Hearn, A. Power, M. Takeyama, and R. Tennent, *An axiomatic approach to binary logical relations with applications to data refinement*, "Proc. Theoretical Aspects of Computer Science," Lecture Notes in Computer Science **1281** (1997), 191–212.

[15] Kinoshita, Y., and A. Power, *Lax naturality through enrichment*, Journal of Pure and Applied Algebra **112** (1996) 53–72.

[16] Kinoshita, Y., and J. Power, *Data-refinement for call-by-value programming languages*, "Computer Science Logic 1999," Lecture Notes in Computer Science **1683** (1999), 562–576.

[17] Kinoshita, Y., and J. Power, *Data refinement and algebraic structure*, Acta Informatica **36** (2000), 693–719.

[18] Kinoshita, Y., and J. Power, *A general completeness result in refinement*, "Recent Trends in Algebraic Development Techniques," Lecture Notes in Computer Science **1827** (2000), 201–218.

[19] Lynch, N., and F. Vaandrager, *Forward and backward simulations part I: Untimed systems*, Information and Computation **121** (1995).

[20] Mijajlović, I., and H. Yang, *Data Refinement with Low-Level Pointer Operations*, "Proc. APLAS 2005," Lecture Notes in Computer Science **3780** (2005) 19–36.

[21] Mitchell, J., "Type Systems for Programming Language Volume A," MIT Press, Elsevier, 1990, 365–458.

[22] Naumann, D. A., "Two-categories and program structure: Data types, refinement calculi, and predicate transformers," PhD thesis, University of Texas at Austin, 1992.

[23] Naumann, D. A., *Data refinement, call by value, and higher order programs*, Formal Aspects of Computing **7** (1995), 652–662.

[24] Naumann, D. A., *Predicate transformers and higher order programs*, Journal of Theoretical Computer Science **150** (1995) 111–159.

[25] Naumann, D. A., *A categorical model for higher order imperative programming*, Mathematical Structures in Computer Science **8** (1998) 351–399.

[26] Naumann, D. A., *Towards squiggly refinement algebra*, "Programming Concepts and Methods," Proc. IFIP PROCOMET '98 (1998), 346–365.

[27] Naumann, D. A., *Predicate transformer semantics of a higher order imperative language with record subtyping*, Sci. Comput. Programming **41** (2001), 1–51.

[28] Naumann, D. A., *Ideal models for pointwise relational and state-free imperative programming*, "Principles and Practice of Declarative Programming" (2001).

[29] Naumann, D. A., *Soundness of data refinement for a higher order imperative language*, Journal of Theoretical Computer Science **278** (2002) 271–301.

[30] Plotkin. G. D., *Dijkstra's predicate transformers and Smyth's powerdomains*, "Abstract Software Specifications," Lecture Notes in Computer Science **86** (1979), 527–553.

[31] Plotkin, G. D., J. Power, D. Sannella, and R. Tennent, *Lax logical relations*, "Proc. ICALP 2000," Lecture Notes in Computer Science **1853** (2000) 85–102.

[32] Power, A. J., *An algebraic formulation for data refinement*, "Proc. MFPS 1989," Lecture Notes in Computer Science **442** (1990), 390–401.

[33] Power, A. J., *Semantics for Local Computational Effects*, "Proc. MFPS 2006," Electronic Notes in Theoretical Computer Science **158** (2006) 355–371.

[34] Power, J., and M. Tanaka, *Axiomatics for data-refinement in call by value programming languages*, to appear in this volume.

[35] Tennent, R., "Semantics of Programming Languages," Prentice-Hall, 1991.