Axiomatics for Data Refinement in Call by Value Programming Languages

John Power[1,2]

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

Miki Tanaka[3]

National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan

Abstract

We give a systematic category theoretic axiomatics for modelling data refinement in call by value programming languages. Our leading examples of call by value languages are extensions of the computational λ-calculus, such as FPC and languages for modelling nondeterminism, and extensions of the first order fragment of the computational λ-calculus, such as a CPS language. We give a category theoretic account of the basic setting, then show how to model contexts, then arbitrary type and term constructors, then signatures, and finally data refinement. This extends and clarifies Kinoshita and Power's work on lax logical relations for call by value languages.

Keywords: computational lambda calculus, premonoidal category, data refinement, lax logical relation.

1 Introduction

There have been two main category theoretic approaches to modelling data refinement. One arose from Tony Hoare's 1972 paper on data representation [5]. Hoare [6], then Hoare and He Jifeng [7] (see [16] for an account in standard category theoretic terms and see [17] for application of these ideas in practice), took as fundamental the idea that data refinements compose, i.e., if M refines N, and N refines P, then M refines P. However, that approach does not generalise easily to higher order types as for instance in the λ-calculus, as explained in [31] (but see [23] for a solution using predicate transformers). The other approach, which has many sources but which has been advocated strongly by Tennent [31], has been

[1] This author has been supported by EPSRC grants GR/M56333: The structure of programming languages : syntax and semantics and GR/586372/01:A Theory of Effects for Programming Languages.

[2] Email: ajp@inf.ed.ac.uk

[3] Email: miki.tanaka@nict.go.jp

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to use binary logical relations [21,19,4] to model data refinement. Binary logical relations model data abstraction and are well suited to higher order types, but they do not compose. So one seeks a common generalisation that both accounts easily for higher order types and is closed under composition. That has led to the notion of lax logical relation [24,15] and variants [14]. Here, we explain and develop the notion of lax logical relation in the setting of call by value languages, based on but clarifying and extending the work of [15].

For the simply typed λ-calculus generated by a signature Σ, Hermida [4] showed that to give a logical relation is equivalent to giving a strict cartesian closed functor from the cartesian closed category L determined by the term model for Σ, to Rel_2, the cartesian closed category for which an object is a pair of sets X and Y together with a binary relation R from X to Y. A lax logical relation is exactly the same except that the functor from L to Rel_2, although still required to preserve finite products strictly, equivalently, to respect contexts, need not preserve exponentials. There is a syntactic counterpart to this [24], but the above is the most compact definition.

For call by value languages, the situation is more complex. One must distinguish between values and arbitrary expressions. So rather than considering a single category L, one considers a pair of categories L_v and L_e, the former for modelling values in context and the latter for modelling arbitrary expressions in context, together with an identity on objects functor $L : L_v \longrightarrow L_e$ that allows one to see the values as possible expressions. The notion of cartesian closedness must be generalised correspondingly, yielding the notion of closed $Freyd$-category [30], and the notion of lax logical relation can and must also be generalised accordingly [15].

A leading example of a call by value language is Moggi's computational λ-calculus, or λ_c-calculus [22], for which data refinement was studied in [15]. But there are many other call by value languages, for instance FPC [3], some CPS languages [32], and languages with nondeterminism [1]. So we should like a systematic account of data refinement for call by value languages that includes a wide range of such languages, and that is the topic of this paper. So this paper clarifies and extends the work of [15], where attention was restricted to the λ_c-calculus.

We first describe models of call by value languages. This requires care. In Section 2, we recall the computational λ-calculus and show how its central feature, the distinction between values and arbitrary expressions, can be modelled in category theoretic terms, specifically in terms of categories enriched in the cartesian closed category $[\rightarrow, Set]$, the functor category of functors from the arrow category to Set. An $[\rightarrow, Set]$-category consists exactly of a pair of categories A_0 and A_1 and an identity on objects functor $A : A_0 \longrightarrow A_1$.

As outlined above, the notion of context is fundamental to data refinement, so we devote Section 3 to the category theoretic modelling of contexts in simply typed call by value languages. We recall the notion of $Freyd$-category, explain why it is of interest to us, and show how it is to be used here.

Next comes the central generalisation from the modelling of the computational λ-calculus in [15]: we must show how to model arbitrary type and term constructors, not just those of the λ_c-calculus. That requires a notion of algebraic structure, equivalently finitary monad, on the category $[\rightarrow, Set]$-Cat of small $[\rightarrow, Set]$-categories.

Once one understands algebraic structure for ordinary categories, as used to describe Hoare's approach to data refinement in [16], it is not difficult but requires a little care to generalise to $[\rightarrow, Set]$-categories with algebraic structure: we give that generalisation in Section 4.

We begged one question above in speaking of languages generated by a signature, and that was how to give a category theoretic formulation of the notion of signature. That is provided by the notion of a T-sketch for a finitary monad T. Again, once one understands that for categories, the extension to $[\rightarrow, Set]$-categories is not difficult but requires a little care. In doing so in Section 5, we also give a slightly better focused definition of the notion of T-sketch than that in the literature.

Finally, we reach the modelling of data refinement. With the above extensions or improvements of previous work, we routinely generalise the notion of lax logical relation in [15]. In doing so, we give a version of the Basic Lemma that is a much more direct generalisation of its usual formulation than appears in [15]. We also give a condition, satisfied by all our leading examples, under which lax logical relations compose; one can see immediately that lax logical relations account for higher order structure too.

We do not address representation independence, the topic of [14], in this paper, but the techniques of [14], based on the T-sketches in [18], extend to the setting of this paper. We plan to make that extension, but it is not entirely clear how to do so yet. We also do not make explicit a relationship with logic. In the case of the simply typed λ-calculus and similar languages, that can be done using fibrations with structure [4]; but it is not yet clear how to do that here, as not only do we generalise from logical to lax logical relations, but also we generalise to call by value languages, and an appropriate notion of fibration has not been developed in that setting yet: it may well be straightforward, but it remains to be investigated.

2 Modelling Call by Value Languages

Our goal in this paper is to model data refinement for call by value programming languages. So for concreteness, we shall present a leading example of a call by value language and outline the key features of its models.

We consider a version of the computational λ-calculus, or λ_c-calculus [22]. There are several equivalent formulations of the λ_c-calculus. The original formulation included a type constructor TX and associated term constructors $[e]$ and $\mu(e)$. But they are redundant, so we omit them.

The λ_c-calculus has type constructors given by

$$X ::= B \mid X_1 \times X_2 \mid 1 \mid X \Rightarrow Y \tag{1}$$

where B is a base type.

The terms of the λ_c-calculus are given by

$$e ::= x \mid b \mid e'e \mid \lambda x.e \mid * \mid (e, e') \mid \pi_i(e) \tag{2}$$

where x is a variable, b is a base term of arbitrary type, $*$ is of type 1, with π_i existing for $i = 1$ or 2, all subject to the evident typing.

It is common to see a *let* constructor in descriptions of the λ_c-calculus, with *let* $x = e$ *in* e' being syntactic sugar for $(\lambda x.e')e$. It only plays a substantial role when

one wants to consider a first-order fragment of the calculus [27], so, for simplicity, we omit it here.

The λ_c-calculus has two predicates: existence, denoted by $\downarrow$, and equivalence, denoted by $\equiv$. The $\downarrow$ rules may be expressed as saying $* \downarrow$, $x \downarrow$, $\lambda x.e \downarrow$ for all e, if $e \downarrow$ then $\pi_i(e) \downarrow$, and similarly for (e, e'). A value is a term e such that $e \downarrow$. The rules for $\equiv$ say $\equiv$ is a congruence, with variables allowed to range over values; there are also rules for the basic constructions and for unit, product and functional types. It follows from the rules that types together with equivalence classes of terms in context form a category, with a subcategory determined by values.

It is straightforward, using the original formulation of the λ_c-calculus in [22], to spell out the inference rules required to make this formulation agree with the original one: one just bears in mind that the models are the same, and we use syntactic sugar as detailed above. We do not clutter our presentation by repeating the rules of [22].

The λ_c-calculus represents a fragment of a call by value programming language. In particular, it was designed to model fragments of ML, but is also a fragment of other languages such as FPC [3] or a nondeterministic call by value language [1]. The first-order fragment is part of the CPS calculus of [32], which in turn is a typed version of Appel's calculus for compiling ML, as explained in [32]. For category theoretic models, the key feature is that there are two entities, expressions and values, so the most direct way to model the language as we have formulated it is in terms of a pair of categories L_v and L_e, together with an identity on objects inclusion functor $L : L_v \longrightarrow L_e$. This is subject to some generalisation of the notion of finite product in order to model contexts and product types, further subject to a closedness condition to model $X \Rightarrow Y$, as we shall explain in later sections.

The key point for us is that the basic information, i.e., categories L_v and L_e and an identity on objects functor (its faithfulness is a distraction) $L : L_v \longrightarrow L_e$, amounts exactly to the data and axioms for an enriched category: let $[\rightarrow, Set]$ denote the functor category of functors from the arrow category to Set, and consider its cartesian closed structure. It is immediate from the definition of enriched category [11] that one has

Proposition 2.1 *An $[\rightarrow, Set]$-category consists of categories A_0 and A_1 and an identity on objects functor $A : A_0 \longrightarrow A_1$. An $[\rightarrow, Set]$-functor from $A : A_0 \longrightarrow A_1$ to $A' : A'_0 \longrightarrow A'_1$ consists of a pair of functors $F_0 : A_0 \longrightarrow A'_0$ and $F_1 : A_1 \longrightarrow A'_1$ making the square of functors commute. An $[\rightarrow, Set]$-natural transformation from (F_0, F_1) to (G_0, G_1) consists of a natural transformation $\alpha : F_0 \Rightarrow G_0$ with naturality extending to A_1.*

By systematic use of this observation and the theory of enriched categories [11], we can model call by value languages such as the computational λ-calculus [22], extensions [3,1], and extensions of its first order fragment such as used to model continuations in [32,30]. We shall proceed systematically to show how one can model such languages, then finally show how to extend that analysis to model data refinement.

3 Modelling Contexts

Central to our modelling of both call by value languages and data refinement is the modelling of contexts. In giving an axiomatic account of data refinement, we shall want contexts to be respected by data refinements, while not asking for any of the other structure to be respected. So we need to pay special attention to modelling contexts. That is delicate for call by value languages, requiring the notion of *Freyd*-category [30]. So in this section, we develop the machinery for *Freyd*-categories.

We must first recall the definitions of premonoidal category and strict premonoidal functor, and symmetries for them, as introduced in [28] and further studied in [1,26,32,30]. A premonoidal category is a generalisation of the concept of monoidal category: it is essentially a monoidal category except that the tensor need only be a functor of two variables and not necessarily be bifunctorial, i.e., given maps $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$, the evident two maps from $X \otimes Y$ to $X' \otimes Y'$ may differ.

Given a symmetric monoidal category C such as Set, and an object S of C, one might consider the category D with the same objects as C and with $D(X, X') = C(S \otimes X, S \otimes X')$, with composition induced by that of C. Such a construction may be used to model a functional language with side-effects [22]. The category D does not have finite products or monoidal structure, as would usually be used to model contexts, the problem being that, although one has evident functors $X \otimes -$ and $- \otimes Y$ for arbitrary objects X and Y, they do not yield a bifunctor. So we need a precise way to enunciate what structure D does have, allowing one to account for contexts in it.

Definition 3.1 A *binoidal category* is a category K together with, for each object X of K, functors $h_X : K \longrightarrow K$ and $k_X : K \longrightarrow K$ such that for each pair (X, Y) of objects of K, $h_X Y = k_Y X$. The joint value is denoted $X \otimes Y$.

Definition 3.2 An arrow $f : X \longrightarrow X'$ in a binoidal category K is *central* if for every arrow $g : Y \longrightarrow Y'$, the following diagrams commute

$$\begin{array}{ccc} X \otimes Y & \xrightarrow{X \otimes g} & X \otimes Y' \\ {\scriptstyle f \otimes Y}\downarrow & & \downarrow{\scriptstyle f \otimes Y'} \\ X' \otimes Y & \xrightarrow[X' \otimes g]{} & X' \otimes Y' \end{array} \qquad \begin{array}{ccc} Y \otimes X & \xrightarrow{g \otimes X} & Y' \otimes X \\ {\scriptstyle Y \otimes f}\downarrow & & \downarrow{\scriptstyle Y' \otimes f} \\ Y \otimes X' & \xrightarrow[g \otimes X']{} & Y' \otimes X' \end{array}$$

A natural transformation $\alpha : G \Longrightarrow H : C \longrightarrow K$ is called *central* if every component of α is central.

Definition 3.3 A *premonoidal category* is a binoidal category K together with an object I of K, and central natural isomorphisms a with components $(X \otimes Y) \otimes Z \longrightarrow X \otimes (Y \otimes Z)$, l with components $X \longrightarrow X \otimes I$, and r with components $X \longrightarrow I \otimes X$, subject to two equations: the pentagon expressing coherence of a, and the triangle expressing coherence of l and r with respect to a (see [11] for an explicit depiction of the diagrams).

Proposition 3.4 *Given a strong monad T on a symmetric monoidal category C, the Kleisli category $Kl(T)$ for T is a premonoidal category, with the functor $J : C \longrightarrow Kl(T)$ preserving premonoidal structure strictly: a monoidal category such as C is trivially a premonoidal category.*

Moggi's work on monads as notions of computation [22] provides a leading source of examples of premonoidal categories. Moggi showed that Kleisli categories for strong monads on cartesian closed categories provide a sound and complete class of models for the λ_c-calculus [22]. More specifically, one can take $C = Set$ or the category of ω-cpo's, both of which are cartesian closed; and one can take a strong monad on them, such as a lifting monad or ones for modelling side-effects, exceptions, continuations, etcetera. More specifically again, the paper [27] shows how every countable Lawvere theory gives rise to a canonical premonoidal category, including all the examples just cited.

More generally, Kleisli categories for premonoidal dyads [29] also provide a good class of examples of premonoidal categories, including a more natural class of models for side-effects.

Having defined the notion of premonoidal category, we need a subsidiary definition, that of the *centre* of a premonoidal category K, which is defined to be the subcategory of K consisting of all the objects of K and the central morphisms. This notion was fundamental to Thielecke's account of values for continuations in [32] but is of somewhat less importance here.

Given a strong monad on a symmetric monoidal category, the base category C need not be the centre of $Kl(T)$. But, modulo the condition that $J : C \longrightarrow Kl(T)$ be faithful, or equivalently, the mono requirement [22,28], i.e., the condition that the unit of the adjunction be pointwise monomorphic, it must be a subcategory of the centre.

The functors h_X and k_X preserve central maps. So we have

Proposition 3.5 *The centre of a premonoidal category is a monoidal category.*

Thus we can deduce the coherence theorem for premonoidal categories.

Theorem 3.6 *Every diagram built from the structural natural transformations in the definition of a premonoidal category commutes.*

Proof. Since the centre of a premonoidal category is a monoidal category and all the structural maps are central, the result follows immediately from coherence for a monoidal category as in Kelly's refinement [10] of Mac Lane's proof. □

All of the premonoidal categories of primary interest to us are symmetric in some reasonable sense, and we require that symmetry for a soundness proof for models of the λ_c-calculus, so we make precise the notion of a symmetry for a premonoidal category.

Definition 3.7 A *symmetry* for a premonoidal category is a central natural isomorphism with components $c : X \otimes Y \longrightarrow Y \otimes X$, satisfying the two conditions $c^2 = 1$ and equality of the evident two maps from $(X \otimes Y) \otimes Z$ to $Z \otimes (X \otimes Y)$. A *symmetric* premonoidal category is a premonoidal category together with a symmetry.

Finally, we need another supplementary definition. The key notion for us here is that of $Freyd$-category, but we need both the notions of premonoidal category and strict symmmetric premonoidal functor in order to define it.

Definition 3.8 A *strict premonoidal functor* is a functor that preserves all the structure and sends central maps to central maps.

One may similarly generalise the definition of strict symmetric monoidal functor to strict symmetric premonoidal functor.

We are finally in a position to define the notion of $Freyd$-category, which is the central definition of this section.

Definition 3.9 A $Freyd$-category consists of a category A_0 with finite products, a symmetric premonoidal category A_1, and an identity on objects strict symmetric premonoidal functor $A : A_0 \longrightarrow A_1$. A strict $Freyd$-functor consists of a pair of functors that preserve all the $Freyd$-structure strictly.

Given a category C with finite products and a strong monad T on it, $Kl(T)$ is a $Freyd$-category. A functor strictly preserving the strong monad and the finite products yields a strict $Freyd$-functor, but the converse is not true.

It is immediate from the definition that a $Freyd$-category is a $[\rightarrow, Set]$-category with extra structure. In the next section, we shall make precise the notion of $[\rightarrow, Set]$-category with algebraic structure and shall see that a $Fryed$-category can be seen as such. But first we develop the notion of $Freyd$-category a little more in its own terms.

Note that a strict $Freyd$-functor from $A : A_0 \longrightarrow A_1$ to $A' : A'_0 \longrightarrow A'_1$ need not send every central map of A_1 to a central map of A'_1: centrality is a property of a map in a premonoidal category, not a piece of structure; so we have not explicitly asked it to be preserved. The key reason for defining $Freyd$-categories as they have been defined was precisely to avoid preservation of arbitrary central maps by $Freyd$-functors. Maps in A_0, which are necessarily central in A_1, are sent to maps in A'_0, therefore to central maps in A'_1, but we specifically do not require that an arbitrary central map be sent to a central map.

Definition 3.10 A $Freyd$-category $A : A_0 \longrightarrow A_1$ is *closed* if for every object X, the functor $A(X \otimes -) : A_0 \longrightarrow A_1$ has a right adjoint. A strict *closed* $Freyd$-functor is a $Freyd$-functor that preserves all the closed structure strictly.

Observe that if A is closed, then by taking X to be the unit I, it follows that the functor $A : A_0 \longrightarrow A_1$ has a right adjoint, and so A_1 is the Kleisli category for a monad on A_0. We sometimes write A_1 for the $Freyd$-category as the rest of the structure may be implicit: often, it is given by the centre of A_1 and the inclusion.

Given a category C with finite products and a strong monad T on C, one says Kleisli exponentials exist if, for each object X of C, the functor $J(X \times -) : C \longrightarrow Kl(T)$ has a right adjoint. A variant of one of the main theorems of [26] is

Theorem 3.11 *To give a closed Freyd-category is to give a category C with finite products together with a strong monad T on C together with assigned Kleisli exponentials. To give a strict closed Freyd-functor is to give a strict map of strong monads that strictly preserves Kleisli exponentials.*

It follows from Moggi's result, but may also be proved directly, that closed $Freyd$-categories provide a sound and complete class of models for the λ_c-calculus. It is routine to define the notion of a model of the λ_c-calculus in a closed $Freyd$-category: types are modelled by objects of A_0, equivalently A_1; product and exponential types are modelled by the premonoidal and closed structures respectively; for pairing, one makes a systematic choice in modelling (e, e'), whether one operates from left to right or conversely. Left to right seems generally favoured [22,30,32].

4 Modelling Type and Term Constructors

In a call by value programming language, one has contexts as we have studied in the previous section, but one also has an arbitrary collection of type and term constructors, and these are subject to equations. For instance, both the λ_c-calculus and FPC have exponential types, FPC has coproduct types, and a language for nondeterminism has a term constructor $\vee$ to model a nondeterministic operator [1]. So we seek a general category theoretic account of modelling type and term constructors. The notion of algebraic structure, or equivalently finitary monad, on the category $[\rightarrow, Set]$-Cat provides such a unified structure for us.

Algebraic structure for $[\rightarrow, Set]$-categories generalises universal algebra, i.e., the study of sets with algebraic structure [13,2]. It has long been known that every category of algebras for a one-sorted signature, subject to equations, is equivalent to the category of algebras for a finitary monad on the category of sets [20]; the term "finitary" is a size condition: a definition is not essential to this paper, so we shall not define it. So our definition of algebraic structure for $[\rightarrow, Set]$-categories is characterised by extending that theorem from sets to $[\rightarrow, Set]$-categories. An article explaining that result in far greater generality is [13], and a version for categories with structure appears in [25]; but for work exactly at this level of generality, see [26].

In ordinary universal algebra, an algebra is a set X together with a family of basic operations $\sigma : X^n \rightarrow X$, subject to equations between derived operations. In order to define algebraic structure on $[\rightarrow, Set]$-categories, one must of course replace the set X by a $[\rightarrow, Set]$-category A. One also replaces the finite number n by a finitely presentable $[\rightarrow, Set]$-category c.

The definition of finitely presentable $[\rightarrow, Set]$-category provides the definitive generalisation of the notion of finite set for the standard category theoretic treatment of universal algebra in this setting [13]. But the definition is complex, all finite $[\rightarrow, Set]$-categories are finitely presentable, and finite $[\rightarrow, Set]$-categories are the only finitely presentable $[\rightarrow, Set]$-categories we need. So we omit the definition here, referring the interested reader to [12].

One must also generalise functions from $[\rightarrow, Set]$-$Cat(c, A)$ into the set of objects of A to allow functions from $[\rightarrow, Set]$-$Cat(c, A)$ into the sets of arrows of A_0 and A_1. These are subject to equations between derived operations. It follows that the models of all of the languages we have mentioned, i.e., extensions of the λ_c-calculus or its first-order fragment by various type and term constructors, the category of small such $[\rightarrow, Set]$-categories with structure and functors that strictly preserve the structure is equivalent to the category of algebras, T–Alg, for a finitary monad T on $[\rightarrow, Set]$-Cat.

In order to include all of our examples, specifically those involving higher order structure either explicitly as in the λ_c-calculus or implicitly as in the CPS-calculus, one needs an unenriched version of algebraic structure on the category $[\rightarrow, Set]$-Cat: so at the base level, we need to consider categories enriched in $[\rightarrow, Set]$, but at what one might call the meta-level, we then need to consider unenriched structure on $[\rightarrow, Set]$-Cat.

In calculating the details, it is easier to study the basic examples such as $Freyd$-categories using 2-categories. Any Cat-enriched algebraic structure on $[\rightarrow, Set]$-Cat qua 2-category trivially yields unenriched algebraic structure on $[\rightarrow, Set]$-Cat qua ordinary category. So, perhaps counter-intuitively, there are more ordinary algebraic structures on $[\rightarrow, Set]$-Cat than there are 2-categorical algebraic structures on $[\rightarrow, Set]$-Cat. As the basic examples are 2-categorical and are expressible more simply in those terms, we shall give the details of this section in 2-categorical terms, with the remark that all the following extends without fuss to unenriched algebraic structure.

Let C denote the 2-category $[\rightarrow, Set]$-Cat, let C_f be the full sub-2-category of finitely presentable $[\rightarrow, Set]$-categories (we need only note that these include the finite such), and let $ob\, C_f$ denote the set of objects of that category, i.e., the set of finitely presentable $[\rightarrow, Set]$-categories. The following is a completely routine variant of the work of [25], which in turn is a special case of [13] (see also [26]).

Definition 4.1 A *signature* on C is a 2-functor S: $ob\, C_f \longrightarrow C$, regarding $ob\, C_f$ as a discrete 2-category.

For each $c \in ob\, C_f$, $S(c)$ is called the $[\rightarrow, Set]$-category of *basic operations of arity* c. Using S, we construct $S_\omega : C_f \longrightarrow C$ as follows: set

$$S_0 = J, \quad \text{the inclusion of } C_f \text{ in } C$$
$$S_{n+1} = J + \sum_{d \in ob\, C_f} C(d, S_n(-)) \times S(d),$$

and define

$$\sigma_0 : S_0 \longrightarrow S_1 \quad \text{to be} \quad inj : J \longrightarrow J + \textstyle\sum_{d \in ob\, C_f} C(d, S_0(-)) \times S(d)$$
$$\sigma_n = J + \textstyle\sum_{d \in ob\, C_f} C(d, \sigma_{n-1}(-)) \times S(d) : S_n \longrightarrow S_{n+1}$$
$$S_\omega = colim_{n<\omega} S_n.$$

The colimit exists because C is cocomplete, and it is a colimit in a functor category with base C. In many cases of interest, each σ_n is a monomorphism, so S_ω is the union of $\{ S_n \}_{n<\omega}$. For each c, we call $S_\omega(c)$ the $[\rightarrow, Set]$-category of *derived c-ary operations*.

A signature is typically accompanied by equations between derived operations. So we say

Definition 4.2 The *equations* of an algebraic theory with signature S are given by a 2-functor $E : ob\, C_f \longrightarrow C$ together with 2-natural transformations $\tau_1, \tau_2 : E \longrightarrow S_\omega(K(-))$, where $K : ob\, C_f \longrightarrow C_f$ is the inclusion.

Definition 4.3 *Algebraic structure* on C consists of a signature S, together with equations (E, τ_1, τ_2). We generally denote algebraic structure by (S, E), suppressing

τ_1 and τ_2.

We now define the algebras for a given algebraic structure.

Definition 4.4 Given a signature S, an *S-algebra* consists of a small $[\to, Set]$-category A together with a $[\to, Set]$- functor $\nu_c : C(c, A) \longrightarrow C(S(c), A)$ for each c.

So, an S-algebra consists of a carrier A and an interpretation of the basic operations of the signature. This interpretation extends canonically to the derived operations, giving an $S_\omega(K(-))$-algebra, as follows:

$$\nu_0 : C(c, A) \longrightarrow C(S_0(c), A) \quad \text{is the identity;}$$

to give $\nu_{n+1} : C(c, A) \longrightarrow C(S_{n+1}(c), A)$, using the fact that $C(-, A)$ sends colimits to limits, is to give a $[\to, Set]$-functor $C(c, A) \longrightarrow C(c, A)$, which we will make the identity, and for each d in $ob\, C_f$, a $[\to, Set]$-functor

$$C(c, A) \longrightarrow C(C(d, S_n(c)), C(S(d), A)),$$

or equivalently, $C(c, A) \times C(d, S_n(c)) \longrightarrow C(S(d), A)$, which is given inductively by

$$C(c, A) \times C(d, S_n(c)) \xrightarrow{\nu_n \times id} C(S_n(c), A) \times C(d, S_n(c)) \xrightarrow{\text{comp}} C(d, A) \xrightarrow{\nu} C(S(d), A)$$

Definition 4.5 Given algebraic structure (S, E), an *(S, E)-algebra* is an S-algebra that satisfies the equations, i.e., an S-algebra (A, ν) such that both legs of

$$C(c, A) \xrightarrow{\nu_\omega} C(S_\omega(Kc), A) \underset{C(\tau_{2c}, A)}{\overset{C(\tau_{1c}, A)}{\rightrightarrows}} C(E(c), A)$$

agree.

Given (S, E)-algebras (A, ν) and (B, δ), we define the homcategory (S, E)–$Alg((A, \nu), (B, \delta))$ to be the equaliser in of

$$\begin{array}{ccc} C(A, B) & \xrightarrow{\{\, C(c, -)\,\}_{c \in\, ob C_f}} & \prod_c C(C(c, A), C(c, B)) \\ \downarrow{\scriptstyle \{\, C(S(c), -)\,\}_{c \in\, ob C_f}} & & \downarrow{\scriptstyle \prod_c C(C(c, A), \delta_c)} \\ \prod_c C(C(S(c), A), C(S(c), B)) & \xrightarrow{\prod_c C(\nu_c, C(S(c), B))} & \prod_c C(C(c, A), C(S(c), B)). \end{array} \tag{3}$$

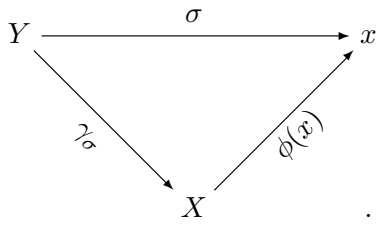
This agrees with our usual universal algebraic understanding of the notion of homomorphism of algebras, internalising it to C. (S, E)–Alg can then be made into a category in which composition is induced by that in C. An arrow in (S, E)–Alg is a $[\to, Set]$-functor $F : A \longrightarrow B$ such that for all finitely presentable c, $F\nu_c(-) = \delta_c(F-) : C(c, A) \longrightarrow C(S(c), B)$, i.e., a $[\to, Set]$-functor that commutes with all basic c-ary operations for all c.

A special case of the main result of [13] says

Theorem 4.6 *An $[\to, Set]$-category is equivalent to (S, E)–Alg for algebraic structure (S, E) on $[\to, Set]$-Cat if and only if there is a finitary 2-monad T on $[\to, Set]$-Cat such that the 2-category is equivalent to T–Alg.*

Example 4.7 Let 2 denote the discrete $[\to, Set]$-category on two objects; let $\to_0$ denote the $[\to, Set]$-category for which both A_0 and A_1 are given by the arrow category and A is the identity functor; let $Cone$ denote the $[\to, Set]$-category for which both A_0 and A_1 are given by two objects together with a cone over them, with A being the identity functor; and let $Comp$ denote the $[\to, Set]$-category with A_0 and A_1 both given by a pair of objects, a pair of cones over them, and an intermediary map from one vertex to the other, commuting with the projections, again with A being the identity functor. Define $S : ob\, C_f \longrightarrow C$ by $S(2) = Cone$, $S(Cone) = Comp$, and for all other c, $S(c)$ is the empty $[\to, Set]$-category.

An S-algebra is a small $[\to, Set]$-category A together with a functor $\phi : C(2, A) \longrightarrow C(Cone, A)$ and a functor $\psi : C(Cone, A) \longrightarrow C(Comp, A)$. The functor ϕ is to take a pair of objects to its limiting cone, and the functor ψ is to take a cone to itself, the limiting cone, and the unique comparison map. So we add equations as follows: we may add equations factoring through $S_1(2)$ and $S_1(Cone)$ respectively so that $\phi(x) : Cone \longrightarrow A$ restricts along the inclusion $2 \longrightarrow Cone$ to x, and so that ψ sends a cone $\sigma : Y \longrightarrow x$ to a commutative diagram of the form

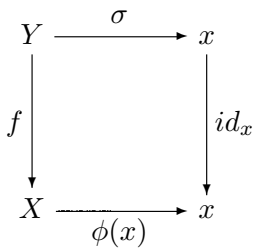


.

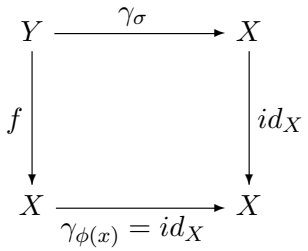
Finally, we add an equation factoring through $S_2(2)$ so that, for each $x : 2 \longrightarrow A$, we have $\gamma_{\phi(x)} = id_X$.

Putting this together, let $E(2)$ be the $[\to, Set]$-category for which both A_0 and A_1 are given by $Cone+ \to_0$ with A being the identity, let $E(Cone)$ have both A_0 and A_1 be given by $Cone + Cone$ with A being the identity, and let $E(c)$ be the empty $[\to, Set]$-category for all other c, and define τ_1 and τ_2 to force the equations as described above: on most components, the τ's factor through $S_1(c)$, but for one of them, we need to factor through $S_2(c)$.

It then follows that for any $x : 2 \longrightarrow A$, $\phi(x)$ is a limiting cone: given any cone $\sigma : Y \longrightarrow x$, the diagram $\psi(\sigma)$ provides a comparison map; and given any comparison map $f : Y \longrightarrow X$, functoriality of ψ applied to the arrow



in $C(Cone, A)$ shows that



commutes, so $f = \gamma_\sigma$.

An (S, E)-algebra is precisely a small category A_0 with assigned binary products, together with an identity on objects functor $A : A_0 \longrightarrow A_1$. An (S, E)-algebra map is a $[\rightarrow, Set]$-functor that sends assigned binary products to assigned binary products.

Observe in the above that all of the finitely presentable $[\rightarrow, Set]$-categories we considered have $A_0 = A_1$ with A being the identity functor. That is no coincidence. In fact, one can use the above observation to prove

Theorem 4.8 *Let T be any finitary monad on Cat. Then there exists a finitary monad T' on $[\rightarrow, Set]$-Cat for which a T'-algebra consists of a T-algebra (A_0, a) together with an identity on objects functor $A : A_0 \longrightarrow A_1$.*

This result allows us to extend known examples of categories with algebraic structure to give $[\rightarrow, Set]$-categories with algebraic structure, providing our only concern is with structure on A_0.

Some examples of categories with algebraic structure that routinely extend the above example are small categories with finite products, small categories with finite coproducts, small monoidal categories and small symmetric monoidal categories. As mentioned above, we can account for exponentials if we drop the enrichment in Cat. Another example of algebraic structure (S, E) is that for which an (S, E)-algebra is a small category together with a monad on it. The construction is not difficult. For instance, for an endofunctor, one puts $S(c) = 1$ if $c = 1$ and makes it empty otherwise, with no equations.

This gives us part of the structure of a $Freyd$-category and extensions, such as finite coproducts as in the models of FPC. However, all the structure exemplified so far has been structure on A_0, so we need to consider non-trivial structure on A_1.

Theorem 4.9 *There is algebraic structure on $[\rightarrow, Set]$-Cat for which an algebra is a small premonoidal category A_1 together with a monoidal A_0 and an identity on objects strict premonoidal functor $A : A_0 \longrightarrow A_1$.*

Proof. Extending the notation of Example 4.7, let $\rightarrow$ denote the $[\rightarrow, Set]$-category with two objects, with one arrow from the first to the second in A_1, and with A_0 discrete. Recall that we let $\rightarrow_0$ denote the $[\rightarrow, Set]$-category with two objects and an arrow from one to the other in both A_0 and A_1. Let 1 denote the discrete $[\rightarrow, Set]$-category with one object.

Let $A : A_0 \longrightarrow A_1$ be an arbitrary small $[\rightarrow, Set]$-category. Then the category $[\rightarrow, Set]$-$Cat(1, A)$ is isomorphic to A_0. Also, an object of the category

$[\rightarrow, Set]$-$Cat(\rightarrow, A)$ is an arrow of A_1, and an arrow is a pair of arrows in A_0 that together with the domain and codomain, form a commutative square in A_1. The category $[\rightarrow, Set]$-$Cat(\rightarrow_0, A)$ maps faithfully into $[\rightarrow, Set]$-$Cat(\rightarrow, A)$ and is given by the arrows of A_0.

So if we put

- $S(1+ \rightarrow) = \rightarrow$,
- $S(\rightarrow +1) = \rightarrow$, and
- $S(c) = 0$ for all other c,

then an S-algebra would consist of a $[\rightarrow, Set]$-category $A : A_0 \longrightarrow A_1$, together with the data for functors $h_X : A_1 \longrightarrow A_1$ and $k_X : A_1 \longrightarrow A_1$ for each object X, with corresponding data for each map in A_0, subject to naturality conditions that will force each map in A_0 to be central. One can extend S by operations and equations to force the above data to give A_1 the structure of a binoidal category: one needs to ensure that the object functions of the two functors are well defined and agree as required by the binoidal definition, and that composition and identities are preserved. So, for instance one puts $E(2) = 1$ and defines τ_1 and τ_2 to force $h_X(Y) = k_Y(X)$; similarly for composition and identities for h_X and k_X; one must extend the signature S and add further equations to give the structural isomorphisms of a premonoidal category, but these are given along the lines of Example 4.7, extending the algebraic structure for monoidal categories. In doing so, the image of A_0 is forced to lie in the centre of A_1. Then one can routinely add operations and equations to give the coherent structural isomorphisms a, l, and r, making A_1 premonoidal. □

Combining the constructions of Example 4.7 and Theorem 4.9, we have

Corollary 4.10 *There is algebraic structure on $[\rightarrow, Set]$-Cat for which an algebra is a small $Freyd$-category.*

We have expressed the technical details of this section almost entirely in terms of 2-categories and algebraic structure with respect to 2-categories. Closed $Freyd$-categories are not included in that, just as cartesian closed categories are not given by algebraic structure on Cat seen as a 2-category [13]: the reason is that closed structure is contravariant, whereas Cat-enrichment requires covariance, as in all the above examples. But, just as cartesian closed categories are given by algebraic structure on Cat as an ordinary category [13,25], closed $Freyd$-categories are given by algebraic structure on $[\rightarrow, Set]$-Cat seen as an ordinary category, cf [26], a proof given by a routine, albeit careful extension of the proof for closedness of a cartesian category. Summarising, we have

Corollary 4.11 *There is algebraic structure on $[\rightarrow, Set]$-Cat, seen as an ordinary category, for which an algebra is a small closed $Freyd$-category.*

These results suggest one consider $[\rightarrow, Set]$-categories with algebraic structure as a way to provide semantics of call by value languages. One may use the results of enriched category theory to do so. For instance, $[\rightarrow, Set]$ is a $[\rightarrow, Set]$-category, and plays a similar role to that of Set in ordinary category theory, so one can speak of presheaves, free cocompletions, etcetera. Moreover, $[\rightarrow, Set]$-Cat is a locally finitely

presentable 2-category, so one has access to the theory of 2-monads, in particular to the treatment of functors that preserve structure only up to coherent isomorphism. In particular, for the purposes of this paper, for any monad T on $[\rightarrow, Set]$-Cat and for any $[\rightarrow, Set]$-category A, one has a free T-algebra on A. As the models of our various languages are to be taken in T-algebras, this fact allows us to give a category theoretic account of the language generated by a signature. We develop that in the next section.

5 Modelling Signatures

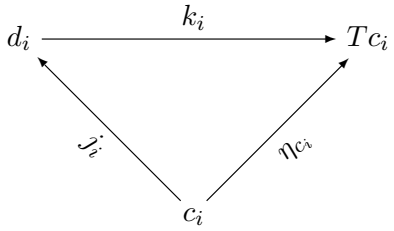
There are two different notions of signature in this paper. We introduced one such notion, consistently with the relevant literature, in Section 4. The other, consistent with a different body of literature, is given by basic types and expressions for a call by value programming language in the spirit of Section 2. In this section, we characterise the notions of signature in the latter sense, language generated by a signature, and model, in category theoretic terms. The key notions for this are those of T-sketch $\mathcal{S}$, the theory $Th(\mathcal{S})$ of a sketch, and a model of a T-sketch, for a given finitary monad T on $[\rightarrow, Set]$-Cat.

A programming language may be freely generated by a signature, i.e., basic data types and basic expressions. For a recent account and use of the idea, see [14]. For a category theoretic formulation of the notion of signature, we give, for any finitary monad T on $[\rightarrow, Set]$-Cat, a notion of a T-sketch $\mathcal{S}$, which we identify with the notion of signature Σ. We then prove that each T-sketch $\mathcal{S}$ generates a free model $\iota : \mathcal{S} \longrightarrow Th(\mathcal{S})$. The free model $Th(\mathcal{S})$ represents the programming language generated by the signature Σ.

We first need a supplementary definition. Although it is the first definition in this section, it is not to be taken as being of central importance, just as the notion of binoidal category is supplementary to the notion of premonoidal category.

Just as in the previous section, the leading examples are more easily seen in terms of 2-monads rather than ordinary monads: recall that 2-monads on $[\rightarrow, Set]$-Cat have underlying ordinary monads, so enrichment amounts to a restriction, but one that includes our leading examples. So, for convenience, we express ourselves in terms of 2-monads: the description of the examples in terms of ordinary monads is routine but tedious.

Definition 5.1 Given a finitary 2-monad T on $[\rightarrow, Set]$-Cat, a *family $\mathcal{D}$ of diagram types* is a small family of 4-tuples $(c_i, d_i, j_i : c_i \rightarrow d_i, k_i : d_i \rightarrow Tc_i)$, where c_i and d_i are finitely presentable $[\rightarrow, Set]$-categories, and j_i and k_i are both $[\rightarrow, Set]$-functors, subject to the condition that the following diagram commutes:



We generally suppress j_i and k_i, leaving them implicit in c_i and d_i, just as one often refers to a category in terms of its set of the objects, with the rest of the data implicit. So we speak of (c_i, d_i).

Example 5.2 Let T be the 2-monad for small $[\to, Set]$-categories $A : A_0 \longrightarrow A_1$ for which A_0 has finite products. With the notation of Ex. 4.7, let $\mathcal{D}$ consist of one pair $(2, Cone)$, with j the (ordered) inclusion of 2 into the base of the cone, and k the inclusion of $Cone$ into $T(2)$ yielding that part of $T(2)$ that gives the product cone over the two base objects. That it satisfies the condition on a family of diagram types amounts to the assertion that k sends $Cone$ to the product cone of the two objects given by 2.

If one began directly with algebraic structure (S, E) rather than a finitary 2-monad T, it would be natural to give a mildly stronger definition of family of diagram types: one would demand that the $[\to, Set]$-functors $k : d_i \longrightarrow Tc_i$ have codomain $S_\omega c_i$, then rewrite the condition so that the top $[\to, Set]$-functor is replaced by the composite of $k_i : d_i \longrightarrow S_\omega c_i$ with the universal map $t : S_\omega \longrightarrow T_{(S,E)}$ evaluated at c_i. Thus a family of diagram types for algebraic structure would immediately give rise to a family of diagram types for the induced 2-monad, but they would not a priori be equivalent. We only need the latter concept here, so shall not formalise the former. However, all constructions we make here are immediately expressible directly in terms of algebraic structure (S, E).

Now assume we are given a finitary 2-monad T.

Definition 5.3 A *T-sketch* $\mathcal{S}$ consists of a small $[\to, Set]$-category X, a family of diagram types $\mathcal{D}$, and a $\mathcal{D}$-indexed family of $[\to, Set]$-functors $\phi_i : d_i \longrightarrow X$. A model of (X, ϕ_i) in a T-algebra (A, a) is a $[\to, Set]$-functor $f : X \longrightarrow A$ such that the following diagrams commute:

$$\begin{array}{ccc} d_i & \xrightarrow{k_i} & Tc_i \\ {\scriptstyle \phi_i}\downarrow & & \downarrow{\scriptstyle a \cdot T(f\phi_i j_i)} \\ X & \xrightarrow[f]{} & A \end{array}$$

If one began with algebraic structure (S, E), then this definition of model would be expressible directly in terms of (S, E): the algebra (A, a) would be replaced by (A, ν), and in the condition, the expression $a \cdot T(f\phi j)$ would be replaced by $\nu(f\phi j)$, with the codomain of k replaced by $S_\omega c$ as above.

Definition 5.4 Given a T-sketch $\mathcal{S}$, the category $Mod(\mathcal{S}, (A, a))$ is defined to be the limit in Cat of the diagram with vertex $[\to, Set]$-$Cat(X, A)$ and for each ϕ_i, two maps from $[\to, Set]$-$Cat(X, A)$ to $[\to, Set]$-$Cat(d_i, A)$, the first given by composition with ϕ_i, the second given by first precomposing with $\phi_i j_i$, then applying $a \cdot T(\,)$, then precomposing with $k_i : d_i \longrightarrow Tc_i$.

The main result of [18] yields

Theorem 5.5 *Let T be a finitary 2-monad on $[\rightarrow, Set]$-Cat. Then for any T-sketch $\mathcal{S}$, there is a model $\iota : \mathcal{S} \longrightarrow Th(\mathcal{S})$ of $\mathcal{S}$ such that composition with ι induces an isomorphism of categories from T–$Alg(Th(\mathcal{S}), (A, a))$ to $Mod(\mathcal{S}, (A, a))$.*

We have expressed this result in terms of $[\rightarrow, Set]$-functors that strictly preserve structure, but it is fairly routine, by mild adaptation of the results of [2], to extend it to $[\rightarrow, Set]$-functors preserving structure in the usual sense, i.e., to T–Alg_p.

Example 5.6 Consider Example 5.2. In it, T is the 2-monad for small $[\rightarrow, Set]$-categories A for which A_0 has finite products, and $\mathcal{D}$ has one element, giving one cone. Let $\mathcal{S}$ be an arbitrary T-sketch with family of diagram types given by the singleton $\mathcal{D}$, i.e., a small $[\rightarrow, Set]$-category X together with a pair of objects of X. By Theorem 5.5, $\mathcal{S}$ freely generates a $[\rightarrow, Set]$-category A for which A_0 has finite products, in particular having a product of the specified pair of objects of X. Theorem 5.5 tells us that there an isomorphism of categories between the category of models of $\mathcal{S}$ in any T-algebra (B, b) and the category of $[\rightarrow, Set]$-functors from A to B that strictly preserve the finite products of A_0.

If we extend to arbitrary monads rather than 2-monads, the notion of T-sketch allows one to speak of the free closed $Freyd$-category generated by a signature as in Corollary 4.11 and the discussion preceding it. So, for instance, if one starts with the λ_c-calculus and some basic types and terms such as those for the natural numbers, we would let T be the monad on $[\rightarrow, Set]$-Cat for small closed $Freyd$-categories, and let $\mathcal{S}$ be the sketch determined by the given basic types and terms. Then $Th(\mathcal{S})$ would be the free closed $Freyd$-category determined by the base types and terms, hence the free model for the λ_c-calculus with those types and terms. See [14] for examples of sketches for monads on Cat and their use for modelling signatures in call by name programming languages, and see [18] for more detail of this idea.

6 Modelling Data Refinement

In this section, we finally model data refinement, extending the analysis of [15]. We assume we have a call by value language with models given by algebraic structure, equivalently a finitary monad T, on $[\rightarrow, Set]$-Cat, and that T extends $Freyd$-structure, which is used to model contexts. Examples are given by extensions of the λ_c-calculus such as FPC [3] and call by value languages with nondeterminism [1], and extensions of the first order fragment of the λ_c-calculus such as CPS-languages [32].

For concreteness, we shall consider Set-based models: our results here do not strictly require that, as all our results generalise by use of sconing [4,19,24]. As outlined in Section 3, a good source of examples of semantic models for call by value languages is given by taking a monad M on Set and considering the Kleisli category of the monad. Every monad on Set has a unique strength, and Kleisli exponentials always exist. So if we denote the Kleisli category by Set_M, then Set_M is a $Freyd$-category (leaving Set and the canonical functor $J : Set \longrightarrow Set_M$ implicit by the convention we mentioned in Section 3); in fact, it is a closed $Freyd$-category. We assume that Set_M has T-structure. That is true for example for FPC as Set has

finite coproducts, and it is true for languages with nondeterminism [1] by choice of M as given by a powerdomain.

We further assume we are given a signature (= T-sketch) Σ for a call by value language. Extending our convention for the λ_c-calculus [15], and following Hoare's convention in his modelling of data refinement [6,16], we identify the language generated by Σ with $Th(\Sigma)$, so for the purposes of this section, we denote $Th(\Sigma)$ by $L : L_v \longrightarrow L_e$, the idea being that L_v denotes our category of values, and L_e denotes the category of arbitrary expressions.

Definition 6.1 A *model* N of L in Set_M is a map of T-algebras from L to Set_M.

We need to model relations between two models N and P of L. So, in principle, we need to send a type σ, i.e., an object of L, to a relation R_σ from N_σ to P_σ. We then need to add conditions to the effect that the structure of both L_v and L_e is respected. To put this in category theoretic terms, we first denote by Rel_2 the category for which an object consists of a triple (X, R, Y) where X and Y are sets and R is a binary relation from X to Y, and where a map $(f, g) : (X, R, Y) \longrightarrow (X', R', Y')$ consists of functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$ that respect the relations. The category Rel_2 has finite products, and that they are preserved by the two projections to Set.

Proposition 6.2 *Given a monad M on Set, the following data forms a Freyd-category Rel_{2M} together with a pair of strict Freyd-functors from Rel_{2M} to Set_M:*

- *the category Rel_2*
- *the category Rel_{2M} with the same objects as Rel_2 but with an arrow from (X, R, Y) to (X', R', Y') given by maps $f : X \longrightarrow MX'$ and $g : Y \longrightarrow MY'$ such that there exists a map $h : R \longrightarrow MR'$ commuting with the projections, with the evident composition*
- *the canonical functor $J : Rel_2 \longrightarrow Rel_{2M}$ taking an object of Rel_2 to itself and taking an arrow (f, g) to its composite with the X'- and Y'-components of the unit of M.*
- *the projections $\delta_0, \delta_1 : Rel_{2M} \longrightarrow Set_M$.*

The functor $J : Rel_2 \longrightarrow Rel_{2M}$ has a right adjoint and so Rel_{2M} is the Kleisli category for a monad on Rel_2, but we do not use that fact. Observe that we make no mention of T-structure beyond that for $Freyd$-structure for modelling contexts.

We do not assume that M preserves jointly monic pairs as it does not hold for our nondeterminism example: a powerdomain is a construct for modelling nondeterminism, a slightly simplified version of one being the endofunctor on Set that sends a set X to its set of finite subsets, $P_f(X)$, with the operation of the endofunctor on maps given by taking the image of each finite subset. A jointly monic pair in Set amounts to a pair of sets (X, Y) together with a subset R of $X \times Y$. Our point here is that the set of finite subsets $P_f(R)$ of R need not be exhibited by the functor P_f as a subset of $P_f(X) \times P_f(Y)$, as for instance can be seen by taking X and Y both to be two element sets with R their product. For notational simplicity, we abbreviate Rel_{2M} by Rel_M where the context is clear.

One could define a logical relation for L as a functor from L to Rel_M that strictly

preserves all the T-structure and commutes with the projections, providing Rel_M has and the projections preserve T-structure. But that is not our immediate concern here as logical relations need not compose, and we want composition in order to model data refinement. So we now define lax logical relations. The central idea is preservation of that structure required to model contexts, i.e., *Freyd*-structure.

Definition 6.3 A binary *lax logical relation* from N to P is a strict *Freyd*-functor $R : L \longrightarrow Rel_M$ such that $(\delta_0, \delta_1)R = (N, P)$.

Our definition restricts to the notion of lax logical relation, or equivalently pre-logical relation, in [15], [24], and [8] if M is the identity.

It is not automatically the case that a pointwise composite of binary lax logical relations is again a binary lax logical relation. That requires an extra condition on the monad M on Set. The central point is that we must consider when the composite of two binary relations extends from Rel_2 to Rel_{2M}; the condition we need is that M weakly preserves pullbacks, i.e., that if

$$\begin{array}{ccc} W & \xrightarrow{h} & X \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle k'} \\ Y & \xrightarrow[h']{} & Z \end{array}$$

is a pullback, then the diagram

$$\begin{array}{ccc} MW & \xrightarrow{Mh} & MX \\ {\scriptstyle Mk}\downarrow & & \downarrow{\scriptstyle Mk'} \\ MY & \xrightarrow[Mh']{} & MZ \end{array}$$

satisfies the existence part of the definition of pullback. This condition is the central condition used to analyse functional bisimulation in [9] with several of the same examples. Examples of such monads are powerdomains, $S \Rightarrow (S \times -)$ for a set S, as used for modelling side-effects, and similarly for monads used for modelling partiality, or exceptions, or combinations of the above. It does not seem to hold of the monad $(- \Rightarrow R) \Rightarrow R$ as has been used to model continuations; but that does not concern us greatly, as data refinement for continuations seems likely to follow a different paradigm to that adopted here anyway.

Theorem 6.4 *Let M be a monad on Set that weakly preserves pullbacks. Then for any lax logical relations $R : L \longrightarrow Rel_M$ and $S : L \longrightarrow Rel_M$ such that $\delta_1 R = \delta_0 S$, the pointwise composite of relations yields a lax logical relation $R \circ S$.*

The proof of Theorem 6.4 is given by routine checking that the pointwise composite satisfies the conditions required to be a lax logical relation. At one critical

point in the proof, one uses the fact that strong epimorphisms in Set are retracts. Unfortunately, the fact that a strong epimorphism is a retract seems unavoidable, contrary to a remark in [15]. moreover, we cannot see any alternative that does not require the condition but retains the same spirit as we have here. So this result appears not to extend to arbitrary toposes for example. But there seems no difficulty in routinely extending the result to categories given by sconing [4,19,24]. There would be more difficulty if we demanded that a lax logical relation preserve not merely $Freyd$-structure but also the monad, as one would need a condition such as M preserving strong epimorphisms, contradicting examples such as $M = S \Rightarrow (S \times -)$.

We now give a generalised Basic Lemma for lax logical relations.

Theorem 6.5 (The Basic Lemma) *To give a lax logical relation from N to P is to give for each type σ of L, a relation*

$$R_\sigma \subseteq N_\sigma \times P_\sigma \tag{4}$$

such that

(i) *for every expression in context, $\Gamma \vdash e : \sigma$, if $x\, R_\Gamma\, y$, then $N(\Gamma \vdash e : \sigma)x$ is related to $P(\Gamma \vdash e : \sigma)y$ by the relation generated by MR_σ, and*

(ii) *if the expression e is a value, then if $x\, R_\Gamma\, y$, one has the stronger result that $N(\Gamma \vdash e : \sigma)x\, R_\sigma\, P(\Gamma \vdash e : \sigma)y$*

where $x\, R_\Gamma\, y$ is an abbreviation for $x_i\, R_{\sigma_i}\, y_i$ for all i when $\sigma_1, \cdots, \sigma_n$ is the sequence of types given by Γ.

Proof. For the forward direction, suppose Γ has sequence of types $\sigma_1, \cdots, \sigma_n$. Since R preserves $Freyd$-structure, $x\, R_\Gamma\, y$ implies $x\, R_{\sigma_1 \times \cdots \times \sigma_n}\, y$. The expression $\Gamma \vdash e : \sigma$ is a map in L from $\sigma_1, \cdots, \sigma_n$ to σ, so R sends it to the unique map from $R_{\sigma_1 \times \cdots \times \sigma_n}$ to R_σ that lifts $(N(\Gamma \vdash e : \sigma), P(\Gamma \vdash e : \sigma))$. The first part of the result is now immediate as N and P strictly preserve $Freyd$-structure. The second part is similar.

For the converse, first taking Γ to be a singleton, the two conditions say that the family R_σ extends, necessarily uniquely, to give graph morphisms from L_e to Rel_M and from L_v to Rel_2. the former restricting to the latter, such that $(\delta_0, \delta_1)R = (N, P)$. Such a pair of graph morphisms trivially forms a $[\to, Set]$-functor as compositions and identities are preserved trivially. Taking $\Gamma \vdash e : \sigma$ to be $\emptyset \vdash * : 1$, where $*$ is the unique element of type 1, the second condition yields $*\, R_1\, *$, so R preserves the unit of the $Freyd$-structure. Taking $\Gamma \vdash e : \sigma$ to be $a : \sigma_0, b : \sigma_1 \vdash (a, b) : \sigma_0 \times \sigma_1$ yields that if $x_0\, R_{\sigma_0}\, y_0$ and $x_1\, R_{\sigma_1}\, y_1$, then $(x_0, x_1)\, R_{(\sigma_0 \times \sigma_1)}\, (y_0, y_1)$. And taking $\Gamma \vdash e : \sigma$ to be $a : \sigma_0 \times \sigma_1 \vdash \pi_i a : \sigma_i$ for $i = 0, 1$ gives the converse. So R strictly preserves $Freyd$-structure. □

Finally, we shall consider an example to see how this all works in practice.

Example 6.6 Consider the computational λ-calculus L_{Stack} generated by the data for a stack. We have base types $Stack$ and Nat, and we have base terms including pop and $push$. The intended semantics of the unCurrying of pop is a partial function from $N(Stack)$ to $N(Stack)$, with $N(Stack)$ being the usual set of stacks. The partiality of the intended semantics for pop is the reason we use the λ_c-calculus here rather than the ordinary λ-calculus. Let N be the intended semantics for stacks in $Set_\perp$, where $\perp$ is the usual lifting monad on Set. The functor $\perp$ preserves pullbacks,

so our composability result holds. Let P be a model of L_{Stack} in $Set_\perp$ generated by modelling stacks in terms of trees, so $P(Stack)$ is the set of non-empty finite trees. Define a logical relation from N to P by defining it on base types as the identity on Nat and on $Stack$, by the usual relationship between stacks and trees. This respects base terms, so it automatically lifts to higher types. We might further define a model Q of L_{Stack} in $Set_\perp$ by modelling stacks by lists of natural numbers. We then have a logical relation S from P to Q generated by the identity on Nat and on $Stack$, by relating finite trees with lists. Now taking the pointwise composite $R \circ S$, we have a lax logical relation from N to Q.

References

[1] Anderson, S. O., and A. J. Power, *A representable approach to finite nondeterminism*, Theoretical Computer Science **177** (1997), 3–25.

[2] Blackwell, R., G. M. Kelly, and A. J. Power, *Two-dimensional monad theory*, J. Pure Appl. Algebra **59** (1989), 1–41.

[3] Fiore, M., and G. D. Plotkin, *An axiomatisation of computationally adequate domain-theoretic models of FPC*, "Proc LICS 1994," IEEE Press, 1994, 92–102.

[4] Hermida, C. A., "Fibrations, Logical Predicates and Indeterminates," Ph.D. thesis, The University of Edinburgh, 1993, available as CST–103–93, also as ECS–LFCS–93–277.

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

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

[7] Hoare, C. A. R., and H. Jifeng, "Data refinement in a categorical setting," Oxford University Technical Mongraph PRG-90,1990.

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

[9] Johnstone, P. T., A. J. Power, T. Tsujishita, H. Watanabe, and J. Worrell, *An Axiomatics for Categories of Transition Systems as Coalgebras*, " Proc LICS 1998," IEEE Press, 1998, 207–213.

[10] Kelly, G. M., *On Mac Lane's conditions for coherence of natural associativities, commutativities, etc.*, J. Algebra **1** (1964), 397–402.

[11] Kelly, G. M., "Basic concepts of enriched category theory," Cambridge University Press, 1982.

[12] Kelly, G. M., *Structures defined by finite limits in the enriched context I*, Cahiers de Topologie and Geometrie Differentielle **23** (1982), 3–41.

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

[14] Kinoshita, Y., P. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent, *An Axiomatic Approach to Binary Logical Relations with Applications to Data Refinement*, "Proc TACS 1997," Lecture Notes in Computer Science **1281** (1997), 191–212.

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

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

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

[18] Kinoshita, Y., A. J. Power, and M. Takeyama, *Sketches*, J. Pure Appl. Algebra **143** (1999), 275–291.

[19] Ma, Q., and J. C. Reynolds, *Types, abstraction and parametric polymorphism 2*, "Mathematical Foundations of Computer Science 1991," Lecture Notes in Computer Science **598** (1991), 1–40.

[20] Mac Lane, S., "Categories for the working mathematician," Graduate Texts in Mathematics **5**, Springer, 1971.

[21] Mitchell, J., "Foundations for programming languages," Foundations of Computing Series, MIT Press, 1996.

[22] Moggi, E., *Computational Lambda-calculus and Monads*, "Proc LICS 1989," IEEE Press (1989), 14–23.

[23] Naumann, D. A., *Data refinement, call by value, and higher order programs*, Formal Aspects of Computing **7** (1995), 752–762.

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

[25] Power, A. J., *Categories with algebraic structure*, "Computer Science Logic 1997," Lecture Notes in Computer Science **1414** (1998), 389–405.

[26] Power, A. J., *Premonoidal categories as categories with algebraic structure*, Theoretical Computer Science **278** (2002), 303–321.

[27] Power, A. J., *Generic models for computational effects*, Theoretical Computer Science **364** (2006) 254–269.

[28] Power, A. J., and E. P. Robinson, *Premonoidal categories and notions of computation*, Math. Structures in Computer Science **7** (1997), 453–468.

[29] Power, A. J., and E. P. Robinson, *Modularity and Dyads*, Electronic Notes in Theoretical Computer Science **20** (1999) 1–14.

[30] Power, A. J., and H. Thielecke, *Environments, Continuation Semantics and Indexed Categories*, "Proc TACS 1997," Lecture Notes in Computer Science **1281** (1997), 391–414

[31] Tennent, R.D., *Correctness of data representations in ALGOL-like languages*, "A Classical Mind, Essays in Honour of C.A.R. Hoare," Prentice-Hall, 1994, 405–417.

[32] Thielecke, H., *Continuations semantics and self-adjointness*, Electronic Notes in Theoretical Computer Science **6** (1997).