CAHIERS DE
TOPOLOGIE ET GÉOMÉTRIE DIFFÉRENTIELLE
CATÉGORIQUES

B. HILKEN
D. E. RYDEHEARD

Indexed categories for program development

Cahiers de topologie et géométrie différentielle catégoriques, tome 32, n° 2 (1991), p. 165-185

<http://www.numdam.org/item?id=CTGDC_1991__32_2_165_0>

CAHIERS DE TOPOLOGIE
ET GÉOMÉTRIE DIFFÉRENTIELLE
CATÉGORIQUES

VOL. XXXII-2 (1991)

INDEXED CATEGORIES FOR PROGRAM DEVELOPMENT

by B. HILKEN and D.E. RYDEHEARD

RÉSUMÉ: Dans cet article les catégories indexées servent de cadre algèbrique et de métalangage pour la logique des programmes. Cette logique incorpore les programmes dans un langage de programmation, des conversions entre programmes, des spécifications du comportement des programmes dans un langage spécifique, et des preuves explicites qu'un programme est correct. Des méthodes de développement de programmes sont interprétées comme des constructions de catégories indexées. Deux exemples sont donnés: les invariants de type et les specifications d'opérations et de leur implémentation.

1. INTRODUCTION.

From the earliest days of electronic computers, there have been attempts to establish the behaviour of computer programs through mathematical proof. Early examples of published proofs are those of Goldstine and von Neumann [1947] and of Turing [1949], who checks a factorial definition (see [Morris, Jones 84]). Recent developments have concentrated on formal systems and 'methodologies' for constructing programs whose correctness is assured through proof. Proofs of correctness involve the following components:

- Programs written in a programming language - a language which admits a notion of computation or evaluation.
- Descriptions of program behaviour, called *specifications*. These may be written in the same language as the programs, a language related to the programming language or a different language altogether. In the latter case we need a 'program logic' linking programs and specifications.
- Proofs, either informal in the usual mathematical style or formal proofs expressed in a suitable language.

These components are widely accepted as underpinning a precise description of the process of programming, although the

practicality of using the full degree of formality is open to question.

In this paper, we propose a single general structure containing programs, specifications and formal proofs as well as other aspects of programming such as evaluation. The structure is based on categories and functors which together express the relationship between these components. Concepts such as functoriality, naturality and universality drawn from category theory describe the way that the components interact. The structure itself is not new, indeed it is an adaptation of Lawvere's [1969,70] notion of a *hyperdoctrine* - an indexed category appropriate for the description of logics.

The novelty of the paper lies in the application of indexed categories to program development. We introduce some of the basic ideas and constructions linking indexed categories and programming. The reader should be warned that the account in this paper is incomplete in that much remains to be done to give a full description both of the category theory involved and of its application to program development. This is a large project of which this paper is an early account. The links we establish between programming and categories are, in the main, fairly obvious once the context has been made clear. Despite this, interesting questions arise about the constructions of logics and the design of programs.

Why this categorical approach? One reason is that category theory allows standard algebraic techniques to be used in the analysis of logics. As an example, a standard construction in algebra is that of free extensions generated by 'indeterminates'. By this means, we introduce a degree of abstraction, allowing a single expression to cover a class of ground expressions. This describes an important aspect of modular program development whereby indeterminates for functions are introduced, delaying the definition of functions until further development has taken place. Associated with this development step is a new program logic which arises as a free construction.

The genericity achieved by category theory is important as it allows a description of programming which is independent of particular programming languages and program logics. There has been considerable interest recently in generic proof construction systems using the fact that the structure of proofs is independent of the particular set of inference rules used to generate them. For example, Edinburgh LF [Harper *et al* 87] and Isabelle [Paulson 86] are two λ-calculi with sufficient type structure to allow the encoding of infe-

rence rules. Proofs are expressed as λ-terms over these rules. This is to be contrasted with the description of logics in category theory where the genericity arises from a common format for the inference rules for the logical constants (connectives and quantifiers), namely as the bijective correspondence between arrows defining adjoints. In systems based on λ-calculi, substitution is part of the underlying algebra of expressions and all binding operations, including quantification, are described as λ-abstraction. In indexed categories, substitution is explicit as functors, and binding operations are introduced as adjoints. The usual 'side conditions' on inference rules restricting the occurrence of free variables become type constraints.

One intended result of this work is a software system to support the process of programming, enabling us to write specifications and programs and perform proofs. Software for this purpose is sometimes called a 'programming support environment'. In Computational Category Theory [Rydeheard, Burstall 88], we show how to express categorical concepts as data types in a programming language, and constructions, such as those of limits and colimits, as programs. The structures in this paper may be implemented using these ideas. The essence of this implementation lies in the construction of classifying (i.e. free, in a suitable sense) hyperdoctrines. Preliminary coding has taken place in functional languages such as ML [Harper *et al* 86] and, more appropriately with its type classes, Haskell [Hudak, Wadler *et al* 88].

Various formal systems and methodologies have been proposed for program development. An example is VDM [Jones 86]. As a formal system VDM consists of a logic, a selection of data types and a primitive functional language. There is an associated 'methodology' of program development in which development steps are accompanied by 'proof obligations' which express the preservation of correctness. VDM itself focuses on data refinement, implementing abstract types by suitable representations. In this paper, we follow some of the basic ideas of VDM-like systems, though we have yet to give a proper account of data refinement. Other systems for programming are based on type theory. For example, in Nuprl [Constable *et al* 85], programming consists of building proofs (in a constructive logic) followed by an extraction of the constructive part of a proof as a program which may be evaluated. Yet another approach to program development is derived from universal algebra and implemented in systems such as OBJ [Futatsugi, Goguen *et al* 85]. There is an abstract approach to this based upon a categorical model theory

('institutions') [Goguen, Burstall 84]. Abramsky [1987] has proposed a general framework for program logics and denotational semantics based upon topological spaces and Stone dualities. In essence, this paper proposes another development system whose meta-language is category theory and which focuses on explicit proofs and free constructions.

There has been considerable interest in indexed categories in computer science as descriptions of typed programming languages in which type expression are themselves classified (by what are often called 'kinds'). Examples of such languages are those supporting parametric polymorphism (2nd-order λ-calculi) [Seely 87a], [Pitts 87] [Hyland, Pitts 89] and languages with program modules [Moggi 89a]. In this paper, we revert to Lawvere's original application of indexed categories to describe predicate logics.

Some notation and terminology: Composition is written applicatively: for $f : a \longrightarrow b$ and $g : b \longrightarrow c$, the composite is $gf : a \longrightarrow c$. We use juxtaposition or "$\circ$" for composition of (1-)arrows and the horizontal composition of 2-cells, reserving "$\cdot$" for vertical composition of 2-cells. Notation for categories: **Cat** denotes the 2-category of categories, **Ccat** the 2-category of cartesian categories (those with finite products) with functors and natural transformations preserving the cartesian structure and **BCCcat** denotes the 2-category of bicartesian closed categories with structure-preserving functors and natural transformations. The primary objects of study in this paper are indexed categories and indexed functors on a 2-category base; we shall call them simply indexed categories and indexed functors. Other structures, however, involving 2-cells receive their full 2-prefixes (2-limits, 2-colimits etc).

2. CATEGORICAL PROGRAM LOGIC

This section consists of a brief introduction to indexed categories and their extension to 2-categories. We explain how indexed categories describe logics arising in program development. In Section 3 we describe some program development strategies as constructions of indexed categories.

We wish to emphasize that indexed categories are used to define a language, rather than to classify known structures or to define a class of models. The category theory that we present should be thought of as the abstract syntax of a program logic. Proofs will take place in the term model ('classifying hyperdoctrine'). We consider objects to be

types (or, in the logic, predicates) and arrows to be terms (or proofs). We do not, therefore, define the syntax of a language and use category theory to provide a semantics. A consequence of this is that we do not need lax structures. Laxity arises when equations in a definition are replaced by isomorphisms, or more generally by 2-cells. The structures we consider are built from the syntax of programming languages and program logics and are strict (equational). However, models of the logic may require these equations to be relaxed to isomorphisms.

The base language (which in applications may be a programming language) is a 2-category T with finite 2-products. We interpret this as follows:

objects of T are tuples of types,
products in T are the concatenation of tuples of types,
arrows $t : Y \longrightarrow X$ of T are tuples of terms (programs) of types X with free variables of types Y,
2-cells $\alpha : t \Rightarrow s$ are (sequences of) conversions between terms, taking t to s. Conversions obey the laws of a 2-category [Rydeheard, Stell 87].

For the purposes of this paper, a primary example of a 2-category is provided by a typed λ-calculus. Objects in the 2-category are tuples of types of the λ-calculus. Arrows $t : Y \longrightarrow X$ in the 2-category are α-conversion classes of λ-terms of types X with free variables of types Y. The 2-cells are generated from β and η conversions, defined as follows. For each term $t : Z \times Y \longrightarrow X$ there is a term $\lambda t : Z \longrightarrow X^Y$, the λ-abstraction over Y. Moreover, for each term $s : Z \longrightarrow X^Y$, there is a term $app(s) : Z \times Y \longrightarrow X$. Both of these correspondences are natural in Z. The β and η conversions provide 2-cells

$$\beta_t : app(\lambda(t)) \Rightarrow t \quad \text{and} \quad \eta_s : s \Rightarrow \lambda(app(s))$$

which are natural in t and s and form the co-unit and unit respectively of an adjunction. This treatment of λ-calculi with explicit conversion rules follows that of Seely [1987], rather than the standard treatment via cartesian closure (see [Lambek, Scott 86]).

A point about products in categories and multiple variables in a language: The products in T are strict rather than lax because their role is to admit terms having more than one free variable rather than to represent pairing in

the language. Notice that we may allow expressions with multiple variables in a language which does not have explicit pairs (e.g. in some presentations of λ-calculus). If the language were also to have surjective pairing, the product type $X * Y$ would be laxly isomorphic (adjoint at the 2-level) to the product $X \times Y$ of the types X and Y as objects in T.

We describe a predicate logic over the base language as an indexed category over T, i.e. a functor

$$p : T^{op} \longrightarrow \mathbf{Cat}$$

where **Cat** is the (2-)category of categories. To incorporate explicit conversions, we make T (the *base* category) a 2-category and p a 2-functor acting covariantly on 2-cells. We interpret this indexed category as follows:

objects ϕ of the category $p(X)$ (the *fibre* over X) are predicates with free variables of types X

arrows $f : \phi \longrightarrow \varphi$ in $p(X)$ are proofs of φ assuming ϕ in the context of free variables X

functors $p_t : p(X) \longrightarrow p(Y)$, where $t : Y \longrightarrow X$ in T (using subscripts for this functor application), are the substitution of the (tuple of) term(s) t for the free variables in predicates and proofs in $p(X)$

natural transformations

$p(\alpha) : p_t \longrightarrow p_s$, where $\alpha : t \Rightarrow s : Y \longrightarrow X$ are, for each ϕ in $p(X)$, a proof of $p_s(\phi)$ assuming $p_t(\phi)$, i.e. properties of terms are preserved under conversion.

This is a description of typed logic where types are used to control the occurrence of free variables within expressions. Notice that proofs are components of the structure, so that the concept of a proof rather than the derived concept of entailment is made primary. This is in keeping with the application to program development where we want to construct and manipulate formal proofs. However, we have not considered conversions between proofs. To study the proof theory of the system, proof conversions would be included as 2-cells in the fibres.

The functor p acts covariantly on 2-cells, so we admit only predicates which are preserved under conversions. There is thus an interplay between behaviour describable by predicates and preservations of behaviour under conversions. In a logic with a suitable equality, we can also reason contravar-

iantly, as we show in the next section, so that conversions are intensional equalities. However, we do not always want an equality of this form in the logic. Consider a programming language with a non-deterministic choice operator $\sqcap$. There are conversions $0 \sqcap 1 \Rightarrow 0$ and $0 \sqcap 1 \Rightarrow 1$. The term $0 \sqcap 1$ should not have all the properties of 0 (in particular, that of being equal to 0), but it may have properties common to both 0 and 1.

Propositional logic.

We have described the predicate structure without a propositional logic. To incorporate propositional logics we restrict the fibres to a sub-2-category $\mathcal{L}$ of **Cat**. The categories in $\mathcal{L}$ contain sufficient structure to interpret the logic, and the functors and natural transformations in $\mathcal{L}$ preserve this (distinguished) structure. For example, for an intuitionistic logic we choose $\mathcal{L}$ to be the category of bicartesian closed categories, interpreting conjunction $\wedge$ as binary product, disjunction $\vee$ as binary coproduct, true T as terminal object, false $\mathbf{F}$ as initial object and implication as exponential (see [Lambek, Scott 86] for details). Other logics arise by a suitable choice of $\mathcal{L}$.

The indexing functor p is to factor through the inclusion of $\mathcal{L}$ in **Cat**

$$p : T^{op} \longrightarrow \mathcal{L} \hookrightarrow \mathbf{Cat} .$$

This says that substitution preserves all propositional structure, for example, $p_t(\phi \wedge \psi) = p_t(\phi) \wedge p_t(\psi)$.

We now show that in a logic with suitable propositional structure and equality predicate, a conversion

$$\alpha : t \Rightarrow s : Y \longrightarrow X$$

yields, for any $\phi \in p(X)$, a proof of $p_t(\phi)$ from $p_s(\phi)$.

Proposition 1. *Let* $p : T^{op} \longrightarrow \mathcal{L} \subseteq \mathbf{Ccat}$ *be an indexed category. If* $=_X$ *is a predicate in* $p(X \times X)$ *with reflexivity and substitutivity axioms:*

$$\mathbf{refl} : \mathsf{T} \longrightarrow p_{\delta_X}(=_X) \quad \text{in} \quad p(X) ,$$

$$\mathbf{eq}_\phi : =_X \wedge p_{\pi_1}(\phi) \longrightarrow p_{\pi_2}(\phi) \quad \text{in} \quad p(X \times X)$$

$$\text{for} \quad \phi \in p(X)$$

where $\delta_X : X \to X \times X$ *is the diagonal arrow, then for each* $\alpha : t \Rightarrow s : Y \to X$ *and each* $\phi \in p(X)$ *there is an arrow from* $p_s(\phi)$ *to* $p_t(\phi)$.

Proof.

$$p_t(\mathbf{refl}) : \top \to p_{\langle t,t\rangle}(=_X) \quad \text{in} \quad p(Y) \tag{*}$$

$$p((\mathrm{id}_X \times t) \circ \langle \alpha, \mathrm{id}_{\mathrm{id}_Y} \rangle)_{=_X} : p_{\langle t,t\rangle}(=_X) \to p_{\langle s,t\rangle}(=_X) \text{ in } p(Y) \tag{†}$$

$$p_{\langle s,t\rangle}(\mathbf{eq}_\phi) : p_{\langle s,t\rangle}(=_X) \wedge p_s(\phi) \to p_t(\phi) \quad \text{in} \quad p(Y) \tag{‡}$$

The composition of (*) , (†) and (‡) gives an arrow of the required type.

Quantifiers and equality

We treat quantifiers and equality in a standard way. $\exists$ is left adjoint to projection and natural in the unquantified variable, giving the bijective inference rule:

$$\frac{\phi \longrightarrow p_{\pi_2}(\psi) \quad \text{in} \quad p(X \times Y)}{\exists_{XY}(\phi) \longrightarrow \psi \quad \text{in} \quad p(Y)}$$

and, for $t : Y' \to Y$, the condition:

$$p_t \exists_{XY}(\phi) = \exists_{XY'} p_{\mathrm{id}_X \times t}(\phi) \tag{§}$$

Similarly, $\forall$ is right adjoint to projection and natural in the unquantified variable. If the logic has conjunction $\wedge$, we define equality so that $(=_X) \wedge p_{\pi_1}(\text{-})$ is left adjoint to the diagonal. This equality satisfies the above axioms of reflexivity and substitutivity. For details of the adjoint formulation of quantifiers and predicates, see [Seely 83].

Some treatments of hyperdoctrines demand left and right adjoints to all functors p_t , satisfying the Beck conditions for pullbacks in the base. Here we introduce adjoints to particular classes of functors in order to define particular logical connectives, and change the Beck condition: programming languages do not have pullbacks other than those arising from the product structure, so we are left with the naturality condition (§).

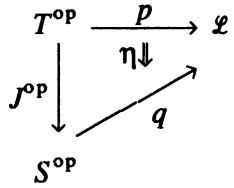
Notice that quantifiers do not, in general, preserve the propositional structure so are not functors in $\mathcal{L}$. The constructions below of new logics take place within the category

$\mathcal{L}$ so that a propositional structure is imposed on the constructed logic. However, this does not work for quantifiers, and in general the constructed logics do not have quantifiers (or equality) even if the original does. We discuss a possible solution to this in the conclusions.

Interpretations

Indexed functors (morphisms between indexed categories) provide a notion of an interpretation of one logic in another.

Definition 1. An **indexed functor** from $p : T^{op} \to \mathcal{L}$ to $q : S^{op} \to \mathcal{L}$ consists of a 2-functor $J : T \to S$ which preserves finite 2-products, and a 2-natural transformation η :



The 2-functor $J : T \to S$ is a translation from language T to language S, translating types, terms and conversions. The components $\eta_X : p(X) \to qJ(X)$ of the 2-natural transformations η interpret predicates in p over X as predicates in q over $J(X)$. They are functors in $\mathcal{L}$, so preserve the propositional structure. The action of this natural transformation on proofs ensures that the interpretation of predicates is sound.

Indexed functors may be composed via functor composition and a composition of natural transformations. The resulting category is a 2-category with 2-cells

$$(\alpha,\nu) : (J,\eta) \to (J',\eta') : p \to q$$

consisting of a 2-natural transformation $\alpha : J \to J'$ and a modification $\nu : \eta \to (q \circ \alpha^{op}).\eta'$ (see [Gray 74]).

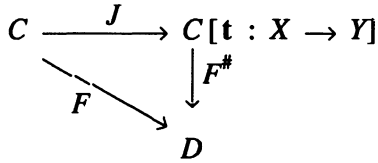
Kan extensions provide an important construction of indexed functors as follows: Let $p : T^{op} \to \mathcal{L}$ be an indexed category and $J : T \to S$ a 2-functor. We define an indexed category $q : S^{op} \to L$ and indexed functor $(J,\eta) : p \to q$ as the left Kan extension of p along J^{op}. The Kan extension exists if $\mathcal{L}$ has indexed colimits (see [Kelly 82]). For logics which are finitely presented, this is the case.

3. PROGRAM DEVELOPMENT.

In this section we introduce some program development techniques and corresponding constructions of program logics as indexed categories. For those not familiar with program development, we illustrate the constructions with simple examples.

The constructions of program logics depend upon 'adding indeterminate arrows' to categories as follows:

Definition 2. Let C be an object in a 2-category $\mathcal{E} \subseteq \mathbf{Cat}$, and let X and Y be objects of C. A category $C[\mathbf{t} : X \longrightarrow Y]$ with functor $J : C \longrightarrow C[\mathbf{t}]$ is the free extension of C with an ('indeterminate') arrow $\mathbf{t} : X \longrightarrow Y$ if and only if J is the 2-universal functor in $\mathcal{E}$ with an arrow $\mathbf{t} : J(X) \longrightarrow J(Y)$ in $C[\mathbf{t}]$. In other words, for any functor $F : C \longrightarrow D$ in $\mathcal{E}$ and arrow $t : F(X) \longrightarrow F(Y)$ in D, there is a unique functor $F^{\#}$ in $\mathcal{E}$ such that



commutes, and $F^{\#}(\mathbf{t}) = t$. Moreover, for any 2-natural transformation $\alpha : F_1 \longrightarrow F_2$ between 2-functors $F_1, F_2 : C \longrightarrow D$ in $\mathcal{E}$ and arrows $t_1 : F_1(X) \longrightarrow F_1(Y)$ and $t_2 : F_2(X) \longrightarrow F_2(Y)$ such that $\alpha_Y \circ t_1 = t_2 \circ \alpha_X$, there is a unique 2-natural transformation $\alpha^{\#} : F_1^{\#} \longrightarrow F_2^{\#}$ in $\mathcal{E}$ such that $\alpha_Y^{\#} \circ t_1 = t_2 \circ \alpha_X^{\#}$ and $\alpha^{\#} J = \alpha$.

This is an example of an *indexed colimit.* Street [1974, 76] establishes results on the existence and construction of indexed colimits. Letting $\mathcal{E}$ be a 2-category of 2-categories, this defines the addition of indeterminate 1-arrows to 2-categories.

Type Invariants

Types in a programming language (objects in a base category) may not be sufficiently discriminating for use in specifications, so additional properties of programs are imposed as predicates over the types. These predicates are called 'type invariants'. The concept of 'invariance' arises from

the fact that operations over the type must preserve the properties of the type as described in the invariant.

As a simple example, consider the concept of a *date* presented as a day (the position in a year) and a year (simplifying the day-month-year or month-day-year format). Both the day and the year are natural numbers, so dates are pairs of natural numbers. The restriction on the number of days in each year, which distinguishes dates from arbitrary pairs of natural numbers, is not captured through type constructors but is expressed as a predicate over $\mathbb{N} \times \mathbb{N}$, where $\mathbb{N}$ is the type of natural numbers:

$$\textbf{valid-date}(d,y) = \textbf{if } \textbf{leapyear}(y) \textbf{ then } d \leq 366 \textbf{ else } d \leq 365$$

where **leapyear** is the predicate determining whether a year is a leap year[1]. The type **Date** of dates is then the type $\mathbb{N} \times \mathbb{N}$ together with this predicate.

Now consider a function on dates, such as that yielding the next day:

$$\begin{aligned}&\textbf{next-day}(d,y) = \\ &\quad \textbf{if } (\textbf{leapyear}(y) \wedge d = 366) \vee (\neg\textbf{leapyear}(y) \wedge d = 365) \\ &\quad \textbf{then } (1, y+1) \textbf{ else } (d+1, y)\end{aligned}$$

This term not only takes pairs of natural numbers to pairs of natural numbers, but preserves the invariant. This means that there is a proof f of the sequent

$$\textbf{valid-date}(d,y) \vdash \textbf{valid-date}(\textbf{next-day}(d,y)) \ .$$

Working with explicit proofs, it is the pair (**next-day**, f) that constitutes an arrow **Date** $\rightarrow$ **Date**.

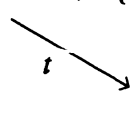
We now construct, as an indexed category, a logic with type invariants. The starting point is the idea of a comprehension schema.

Definition 3. An indexed category $p : T^{op} \rightarrow \mathcal{L} \subseteq \mathbf{Ccat}$ satisfies the **comprehension schema** for $X \in T$ if there is a functor $\{X \mid \_\} : p(X) \rightarrow T/X$ and a bijection

---

[1] The current method of calculating leap years began with the Gregorian calendar. This could be incorporated as an additional clause in the invariant, restricting years to be after 1582.

$$\frac{\mathsf{T} \longrightarrow p_t(\phi) \quad \text{in } p(Y)}{Y \longrightarrow \{X \mid \phi\} \quad \text{in } T}$$



natural in $\phi \in p(X)$ and $t \in T/X$.

This is a variant of Lawvere's [1970] comprehension schema, but does not depend on p_t having a left adjoint.

The following proposition is based on the Grothendieck construction of a fibration.

Proposition 2. *Let* $p : T^{op} \longrightarrow \mathcal{L} \subseteq \mathbf{Ccat}$ *be an indexed category. The indexed category* $p' : T'^{op} \longrightarrow \mathcal{L}$ *, defined as follows, satisfies the comprehension schema for all objects.*

From p construct a category T' *, which is the base category with invariants:*

objects (X,ϕ) *where* $X \in T$ *and* $\phi \in p(X)$ *,*
type X under invariant ϕ *,*

arrows $(t,f) \in (X,\phi) \longrightarrow (Y,\psi)$ *where* $t : X \longrightarrow Y$ *in* T *,*
$f : \phi \longrightarrow p_t(\psi)$ *in* $p(X)$ *,*
t is a term, f is a proof that it preserves the invariant,

2-cells $\alpha : (t,f) \Rightarrow (s,g)$ *where* $\alpha : t \Rightarrow s$ *,*
$g = p(\alpha)_\psi \circ f$ *,*
t converts to s and the same conversion turns f into g ,

products $(X,\phi) \times (Y,\psi) = (X \times Y,\ p_{\pi_1}(\phi) \wedge p_{\pi_2}(\psi))$ *,*
the invariant on the product holds if each invariant holds on the corresponding component.

Define $p' : T'^{op} \longrightarrow \mathcal{L}$ *as follows:*

objects $p'(X,\phi) = p(X)[\mathbf{ax} : \mathsf{T} \longrightarrow \phi]$ *in* $\mathcal{L}$ *where* $\eta_X : p(X) \longrightarrow p(X)[\mathbf{ax} : \mathsf{T} \longrightarrow \phi]$ *is the free extension by an axiom* $\mathbf{ax} : \mathsf{T} \longrightarrow \phi$ *saying that* ϕ *holds over* (X,ϕ) *,*

arrows $p'_{(t,f)}$ *by universality of the free extension:*

$$\begin{array}{lll} X & p(X) \longrightarrow p(X)[\mathbf{ax} : \mathsf{T} \longrightarrow \phi] & \\ t\downarrow & \mapsto p_t\uparrow \qquad\qquad \uparrow p'_{(t,f)} & \textit{since } f \circ \mathbf{ax}{:}\mathsf{T} \longrightarrow p_t(\psi)\ , \\ Y & p(Y) \longrightarrow p(Y)[\mathbf{ax}' : \mathsf{T} \longrightarrow \psi] & \end{array}$$

2-cells *similarly, by the universal property of the indeterminate arrow on natural transformations.*

Note: There are two 2-functors:

$$F : T \longrightarrow T' : X \mapsto (X,\mathsf{T})\ ,$$
$$G : T' \longrightarrow T : (X,\phi) \mapsto X\ .$$

The first translates expressions in the base language to expressions in the language with type invariants; the second functor allows us to implement, in the base language, terms written using invariants, by omitting the correctness information. The construction also provides the indexed functor $(F,\eta) : p \longrightarrow p'$, a universal interpretation of p in p' .

Proof. The comprehension schema for p' is defined at (X,ϕ) through the universality of the free extension $\eta_X : p(X) \longrightarrow p(X)[\mathbf{ax} : \mathsf{T} \longrightarrow \phi]$:

$$\{(X,\phi)\,|\,\psi\} = (\mathrm{id},\pi_1) : (X,\phi \wedge \psi) \longrightarrow (X,\phi)\ ,$$
$$\{(X,\phi)\,|\,\mathbf{ax}\} = (\mathrm{id},\delta) : (X,\phi) \longrightarrow (X,\phi \wedge \phi)\ .$$

Note that $p'_{(t,f)}(\phi) = p_t(\phi)$ so the bijection takes

$$(t,f) : (Y,\psi) \to (X,\theta \wedge \phi) \text{ in } T'$$
$$\text{to } \pi_2 \circ f \circ \mathbf{ax} : \mathsf{T} \to p_t(\phi) \text{ in } p'(Y,\psi)\ .$$

Quotient types, as in Nuprl [Constable *et al* 85], may be handled similarly. Quotients are defined as an equivalence relation on each type, differing in general from the inherent equality on terms of the types. A category is constructed with objects as triples $(X,\sim,\alpha)$ where $X \in T$, $\sim \in p(X \times X)$ and, because all proofs are explicit, α is a proof that $\sim$ is an equivalence relation. Arrows in this category are arrows in T which preserve the equivalence relations and transform the equivalence proof in the source to that in the target. Again we may construct an indexed category over this new base. Partial equivalence relations enable us to combine both subtypes and quotient types.

Specifications.
A major theme in program development is the passage between abstract descriptions and more specific instances. In formalism, abstraction arises through the addition of indeterminates (variables) so that a single expression with indeterminates covers a class of ground expressions (i.e. those built from constants). In modular program development, we introduce names (indeterminates) and types for operations, delaying the definition of the operation until further development has taken place. For correctness of programs involving this operation, we include a specification of its behaviour. Associated with this development step is a new program logic which we construct freely using a Kan extension.

We begin with the idea of a specification of an operation. In programming, specifications are often presented as pairs of a pre-condition and a post-condition, where the pre-condition describes the range of valid arguments and the post-condition relates the arguments and the result.

Definition 4. A **specification** of type X , Y is a pair of predicates

$$\begin{aligned} \textbf{pre} &\in p(Y) , \\ \textbf{post} &\in p(X \times Y) \end{aligned}$$

A term $t : Y \to X$ **satisfies** the specification if there is a proof

$$f : \textbf{pre} \to p_{\langle t,Y\rangle}(\textbf{post}) \quad \text{in} \quad p(Y) .$$

As an example of an operator specified in terms of pre- and post-conditions, consider integer division.

$$\begin{aligned} &\textbf{integer-divide}(n : \mathbb{Z}, d : \mathbb{Z}) r : \mathbb{Z} = \\ &\quad \textbf{pre } d \neq 0 , \\ &\quad \textbf{post } \exists x : \mathbb{Z}.(0 \leq x \leq d) \wedge (r \times d + x = n) . \end{aligned}$$

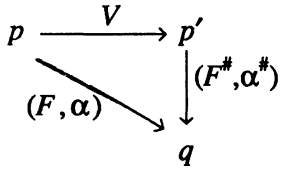
The pre-condition gives the range of valid arguments, whilst the post-condition is a predicate relating the arguments n and d to the result r . This is not an integer division algorithm, but specifies the required behaviour. An algorithm, such as 'long division', is expressed as a term in the programming language and will satisfy the specification in the sense of Definition 4. In general, not only will many algorithms satisfy a given specification, but also operations with different functional behaviour may satisfy the same

specification. Specifications allow us to give definitions at a suitable level of abstraction, rather than be constrained to algorithmic code as the only formal definition of required behaviour.

We now describe the construction of the logic associated with a specification as a construction of an indexed category. In an indexed category, an indeterminate arrow may be added either to the base or to a fibre, or to both; each case yields a new indexed category. A specification introduces a new arrow in the base language (the name of the operation being specified) and a new arrow in a fibre, extending the fibre with a proof that the operation satisfies its specification. The universality of this construction defines what is meant by an implementation of a specification. We define the extension of the base category (extension of a fibre, in parentheses) as follows.

Definition 5. Let $\mathcal{L} \subseteq \mathbf{Cat}$ be a 2-category and $p : T^{op} \to \mathcal{L}$ be an indexed category. For objects $X,Y \in T$ (object $X \in T$ and objects ϕ,φ in fibre $p(X)$), an indexed category $p' : T'^{op} \to \mathcal{L}$ (indexed category $p' : T^{op} \to \mathcal{L}$) and indexed functor $V = (J,\eta) : p \to p'$ (indexed functor $V = (I_T,\eta) : p \to p'$) is the free extension of p with arrow $\mathbf{t} : Y \to X$ (with arrow $\mathbf{f} : \phi \to \varphi$ over X) if and only if V is the 2-universal indexed functor with an arrow $\mathbf{t} : J(Y) \to J(X)$ in T' (with an arrow $\mathbf{f} : \phi \to \varphi$ in $p'(X)$).

In other words, for any indexed category $q : S^{op} \to \mathcal{L}$, indexed functor $(F,\alpha) : p \to q$ and arrow $t : F(Y) \to F(X)$ in S (arrow $f : \alpha_X(\phi) \to \alpha_X(\varphi)$ in $q(F(X))$), there is a unique indexed functor $(F^{\#},\alpha^{\#}) : p' \to q$ such that



commutes and $F^{\#}(\mathbf{t}) = t$ (and $\alpha^{\#}_X(\mathbf{f}) = f$). Moreover, this is universal on 2-cells: For any $(\varepsilon,\nu) : (F_1,\alpha_1) \to (F_2.\alpha_2)$ with $F_1, F_2 : T \to S$ and arrows $t_1 : F_1(Y) \to F_1(X)$ and $t_2 : F_2(Y) \to F_2(X)$ in S such that $\varepsilon_X \circ t_1 = t_2 \circ \varepsilon_Y$ (arrows $f_1 : (\alpha_1)_X(\phi) \to (\alpha_1)_X(\varphi)$ in $q(F_1(X))$, and $f_2 : (\alpha_2)_X(\phi) \to (\alpha_2)_X(\varphi)$ in $q(F_2(X))$ such that

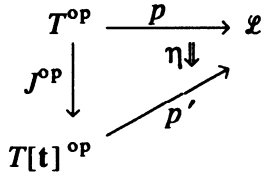
$$(\nu_X)_\varphi \circ f_1 = (q \circ \in^{op})_X(f_2) \circ (\nu_X)_\phi ,$$

there is a unique

$$(\varepsilon^{\#},\nu^{\#}) : (F_1^{\#},\alpha_1^{\#}) \longrightarrow (F_2^{\#},\alpha_2^{\#})$$

such that $(\varepsilon^{\#},\nu^{\#}) \circ V = (\varepsilon,\nu)$, and $\varepsilon_X^{\#} \circ t_1 = t_2 \circ \varepsilon_Y^{\#}$ (and $(\nu_X^{\#})_{\varphi} \circ f_1 = (q \circ \varepsilon^{\#op})_X(f_2) \circ (\nu_X^{\#})_{\phi})$) .

Each of these extensions may be expressed as an indexed colimit, constructions of which are given in [Street 74,76]. The extension of the base can be described in terms of a Kan extension: To add arrow $\mathbf{t} : Y \longrightarrow X$ to the base of $p : T^{op} \longrightarrow \mathcal{L}$, let $J : T \longrightarrow T[\mathbf{t}]$ be a free extension of T by $\mathbf{t}$. Construct the indexed category p' as a left Kan extension of p along J^{op} :



Consider now a specification **<pre, post>** of type X,Y in indexed category p. We construct an indexed category p'' and interpretation (indexed functor) of p in p'' by composition: Extend the base of p with an arrow $\mathbf{t} : Y \longrightarrow X$ to form indexed category p' and indexed functor $(J,\eta) : p \longrightarrow p'$, then freely adjoin a proof $\mathbf{f} : \mathbf{pre} \longrightarrow p'_{<\mathbf{t},Y>}(\mathbf{post})$ over $J(Y)$ to p' to form indexed category p'' and indexed functor $(I_{T'},\eta') : p' \longrightarrow p''$. From the programming point of view, the universality of this construction captures what is meant by an implementation of a specification. An implementation consists of a term t in a base language S. In general, S may differ from the underlying language T of the specification. The term t satisfies the specification in that there is a proof f of satisfaction. The term t together with the proof f provide, through the universality, a sound interpretation of the freely constructed program logic p''. In a stepwise development, the implementing term may be an indeterminate satisfying a stronger specification, so that the base category S is itself constructed from a specification.

CONCLUSIONS.

This is a rather unsatisfactory account as so much is

left to be done. However, as a preliminary paper, it does set the context of the investigation and give some relevant constructions. Let us look at some of the open questions and topics left undone.

Programming languages are treated as 2-categories in this paper. Work by several authors under the heading of categorical semantics shows how various sequential languages may be treated as categories (see e.g. [Manes, Arbib 86]), and in many cases the definitions can be modified to give 2-category versions. Polymorphic languages, such as 2nd-order λ-calculus, may be treated as indexed categories [Seely 87a], [Pitts 87], [Hyland, Pitts 89], [Moggi 89a]; in each case the Grothendieck construction yields an appropriate category.

In this paper we have concentrated on VDM-like logics [Jones 86]. We have not considered how the great variety of program logics, process logics and temporal logics can be described as indexed categories. The generality of our definitions should make these descriptions possible, and also provide a framework for the definition of new logics.

As remarked earlier, the constructions in Section 3 do not define logics with quantifiers. When the base category is cartesian closed, Andrew Pitts has shown that some quantifiers exist in the constructed logics (private correspondence). A more general solution is under investigation allowing quantifiers to be imposed on logics. For this, we use a somewhat different presentation, with all constructions and logics defined as monads on a category of indexed categories. This reduces the problem to that of finding colimits of monads, constructions of which are known.

In the current version of this work, the 2-cells in the base category are rather 'grafted' on to the familiar definition of an indexed category. In particular, the interaction of 2-cells with the logic is weak. Andrew Pitts' work on program logics for categories with a computational monad [Pitts 90] shows how a notion of computation can be internalised in the logic. Evaluation is a basic mechanism of computation and 2-cells are an explicit description of evaluation steps. Program logics should provide suitable primitives for reasoning about evaluation, using predicates of 'termination' or 'definedness' and a notion of 'values' of programs. We have some results in this direction, indicating that we can define such predicates in terms of 2-cells, and that the logic of values is simpler than the logic of programs.

A major omission in the description of program development techniques is data refinement, whereby 'abstract' types

in specifications are instantiated as 'concrete' representation types. An example is the implementation of real numbers by floating point numbers, used in most programming languages. This involves not only a correspondence between types, but also between languages and ultimately between logics. In the example, real numbers are an object in the category of sets, with a classical set theory as an appropriate logic, whereas floating point numbers are a type in a programming language, whose logic is a process logic. We are currently investigating this more general form of data abstraction in the setting of indexed categories.

ACKNOWLEDGEMENTS

We are especially indebted to Andrew Pitts (Cambridge) who has provided us with guidance on categorical matters and solved some of the problems we raised. He kindly allowed us early access to his work on logics associated with Moggi's monads. John Power (Edinburgh) read an early version of this paper and provided useful comments. We are grateful to the referee for the constructive comments. The project is funded by the Science and Engineering Research Council. The SERC Logic for IT Initiative is to be commended for fostering work in this area.

REFERENCES

ABRAMSKY, S. (1987) Domain Theory in Logical Form. *Proc. Symposium on Logic in Computer Science, June 22-25,* I.E.E.E., Ithaca, NY.

CONSTABLE, R.L. *et al.* (1985) *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ.

CROLE, R. & PITTS, A.M. (1990) New foundations for fixpoint computations. *Proc. Symposium on Logic in Computer Science, June 4-7, 1990.* I.E.E.E., Philadelphia, PA.

FUTATSUGI, K., GOGUEN, J.A., JOUANNAUD, & MESEGUER. (1985) Principles of OBJ2. In H.K. Reid (ed.) *Proc. 12th ACM Symposium on Principles of Programming Languages,* pp. 52-66. A.C.M.

GOGUEN, J.A. & BURSTALL, R.M. (1984) Introducing Institutions In E. Clarke and D. Kozen (eds) *Logics of Programs,* pp. 221-256, Springer LNCS 164.

GOLDSTINE, H.H. & VON NEUMANN, J. (1947) "Planning and Coding of Problems for an Electronic Computing Instrument". Report of U.S. Ord. Dept. In A. Traub (ed.) *Collected Works of J. von Neumann,* New York, Pergamon, Vol. 5, pp 80-151.

GRAY, J.W., (1974) *Formal Category Theory: Adjointness for 2-Categories.* Springer LNM 391.

HARPER, R., HONSELL, F. & PLOTKIN, P. (1987) A Framework for Defining Logics. *Proc. Symposium on Logic in Computer Science, June 22-25,* I.E.E.E., Ithaca, NY.

HARPER, R., MACQUEEN, D.B. AND MILNER, R. (1986) *Standard ML.* Technical Report, ECS-LFCS-86-2, Edinburgh University, Department of Computer Science.

HUDAK, P. & WADLER, P. *et al.* (1988) *Report on the Functional Programming Language, Haskell.* Draft proposed standard. Preprint, Dept. Computer Science, University of Glasgow, U.K.

HYLAND, J.M.E. & PITTS, A.M. (1989) Theory of constructions: categorical semantics and topos-theoretic models. *Proc. A.M.S. Conference on Categories in Computer Science and Logic, Boulder, Colorado (1987).* A.M.S.

JOHNSTONE, P.T. & PARE, R. (1978) (eds.) *Indexed Categories and their Applications,* Springer LMS 661.

JONES, C.B. (1986) *Systematic Software Development Using VDM.* Prentice-Hall International Series in Computer Science (ed. C.A.R. Hoare), Hemmel Hempstead.

KELLY, G.M. (1982) *Basic Concepts of Enriched Category Theory.* London Math. Soc., Lecture Note Series, 64. C.U.P.

LAMBEK, J. & SCOTT, P.J. (1986) *Introduction to higher order categorical logic.* Cambridge studies in advanced mathematics 7. C.U.P.

LAWVERE, F.W. (1969) Adjointness in Foundations. *Dialectica* 23, 3/4. pp. 281-296.

LAWVERE, F.W. (1970) Equality in Hyperdoctrines and the Comprehension Schema as an Adjoint Functor. *Proc. Symp. in Pure Math., XVII: Applications of Categorical Algebra,* A.M.S. pp 1-14.

MAC LANE, S. (1971) *Categories for the Working Mathematician.* Springer-Verlag, New York.

MANES, E.G. & ARBIB, M.A. (1986) *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science, AKM Series, Springer-Verlag.

MARTÍ-OLIET, N. & MESEGUER, J. (1989) From Petri Nets to Linear Logic. *Proc. Conference on Category Theory and Computer Science, Manchester, 1989, Springer LNCS* 389.

MOGGI, E. (1989) Computational lambda-calculus and monads. *Proc. 4th Symp. Logic in Computer Science, 1989,* I.E.E.E.

MOGGI, E. (1989a) A categorical account of program modules. *Proc. Summer Conference on Category Theory and Computer Science,* Manchester 1989,Springer LNCS 389.

MORRIS, F.L. & JONES, C.B. (1984) An Early Program Proof by Alan Turing. *Annals of the History of Computing* Vol. 6, Number 2, pp.139-143.

de PAIVA, V.C.V. (1989) A Dialectica-like Model of Linear Logic. *Proc. Summer Conference on Category Theory and Computer Science,* Manchester 1989, Springer LNCS 389.

PAULSON, L.C. (1986) Natural deduction as higher-order resolution. *J. Logic Programming,* 3, pp. 237-258.

PITTS, A.M. (1987) Polymorphism is Set Theoretic, Constructively. *Proc. Summer Conference on Category Theory and Computer Science,* Edinburgh, 1987, Springer LNCS 283.

RYDEHEARD, D.E. & BURSTALL, R.M. (1988) *Computational Category Theory.* Prentice-Hall International Series in Computer Science (ed. C.A.R. Hoare), Hemmel Hempstead.

RYDEHEARD, D.E. & STELL, J.G. (1987) Foundations of Equational Deduction: A Categorical Treatment of Equational Proofs and Unification Algorithms. *Proc. Summer Conference on Category Theory and Computer Science,* Edinburgh, 1987, Springer LNCS 283.

SEELY, R.A.G. (1983) Hyperdoctrines, Natural Deduction and the Beck Condition. *Z. Math. Logik,* 29, pp. 505-542.

SEELY, R.A.G. (1984) Locally Cartesian Closed Categories and Type Theory. *Math. Proc. Camb. Phil. Soc.,* 95, pp. 33-48.

SEELY, R.A.G. (1987) Modelling Computations: A 2-Categorical Framework. *Proc. Symp. Logic in Computer Science, June 22-25, 1987,* I.E.E.E., Ithaca, NY.

SEELY, R.A.G. (1987a) Categorical semantics for higher order polymorphic lambda calculus. *J. Sym. Logic 52, 4.*

SEELY, R.A.G. (1987b) Linear Logic, *-Autonomous Categories and Cofree Co-algebras. In J.W. Gray and A. Scedrov (eds.), *Proc. A.M.S. Conference on Categories in Computer Science and Logic,* Boulder, Colorado.

STREET, R.H. (1974) Elementary Cosmoi. *Springer LNM* 420, pp. 104-133.

STREET, R.H. (1976) Limits Indexed by Category-Valued 2-Functors. *J. Pure and Applied Algebra, 8.* pp. 149-181.

TURING, A.M. (1949) "Checking a Large Routine". In *Report of a Conference on High Speed Automatic Calculating Machines*, Univ. Math. Lab., Cambridge, pp. 67-69.

Department of Computer Science
University of Manchester
Oxford Road
MANCHESTER M13 9PL
U.K.