Appears in *LISP and Symbolic Computation* 9(1):77–107, February 1996

Categorical Models for Local Names

Ian Stark*
University of Cambridge Computer Laboratory
Ian.Stark@cl.cam.ac.uk

July 1994

Abstract

This paper describes the construction of categorical models for the *nu-calculus*, a language that combines higher-order functions with dynamically created *names*. Names are created with local scope, they can be compared with each other and passed around through function application, but that is all.

The intent behind this language is to examine one aspect of the imperative character of Standard ML: the use of local state by dynamic creation of references. The nu-calculus is equivalent to a certain fragment of ML, omitting side effects, exceptions, datatypes and recursion. Even without all these features, the interaction of name creation with higher-order functions can be complex and subtle; it is particularly difficult to characterise the *observable* behaviour of expressions.

Categorical monads, in the style of Moggi, are used to build denotational models for the nu-calculus. An intermediate stage is the use of a computational metalanguage, which distinguishes in the type system between values and computations.

The general requirements for a categorical model are presented, and two specific examples described in detail. These provide a sound denotational semantics for the nu-calculus, and can be used to reason about observable equivalence in the language. In particular a model using logical relations is fully abstract for first-order expressions.

1 Introduction

Programming languages that combine higher-order functions and state create certain difficulties for traditional methods of semantics and reasoning about programs. For operational semantics, the presence of state means that simple methods for analysing observational equivalence in purely functional languages break down. For example Milner's 'Context Lemma' [15] becomes invalid and the technique of applicative bisimulation [1] is no longer complete. For denotational semantics, there are difficulties of abstraction. Models may give

*Supported by UK SERC studentship 91307943 and CEC SCIENCE project PL910296.

different denotations to expressions that are *observationally equivalent*; that is, they can replace each other in a program without affecting its result. This happens even in the absence of non-termination, which is well known to cause similar problems.

Our particular concern is to strengthen methods for reasoning about the programming language Standard ML [16]. This is a typed language with higher-order functions as first-class values and a call-by-value semantics. Among other features, expressions can dynamically create typed *references*, which are private mutable storage cells. The use of call-by-value semantics means that a reference may be shared by several functions, or by successive invocations of the same function (as with 'own' variables in Algol).

We want to highlight the effects that arise purely from locality of state, ignoring other features such as side effects, exceptions and non-termination due to recursion. To do this we work with a language called the *nu-calculus*. This is a typed call-by-value lambda-calculus extended with the notion of a *name*; names can be created, passed around and compared with each other. They correspond to values in ML of type unit ref, cells that can only contain the value (). Higher-order functions and booleans are also available, but functions cannot be defined recursively. The nu-calculus is described in Section 2. We give an operational semantics taken from that of Standard ML [16] and derive a suitable notion of observational equivalence, together with several examples.

It might reasonably be expected that a language as bare as the nu-calculus would easily admit fully-abstract models and simple, complete reasoning methods. As has been shown elsewhere, in [27] and [28], this is not the case. The particular interaction of dynamically generated names with a call-by-value semantics leads to some subtle behaviour, so that the scope and visibility of names is not easily tracked by traditional techniques. In this earlier work there is an operational method for proving observational equivalence, using a variant of logical relations. Here we look at denotational methods through categorical models, expanding on the brief sketch in Section 4 of [28].

We follow Moggi [19] in using categorical *monads* to encapsulate a notion of computation appropriate to the nu-calculus. Section 3 introduces a *computational metalanguage* for the nu-calculus, which distinguishes in its type system between the denotations of values and of computations. An associated equational logic permits some simple reasoning about observational equivalence. Using this logic we can state conditions for a sound and adequate interpretation of the nu-calculus (Figure 4). In Section 4 we turn these into requirements for a categorical model, and describe how the nu-calculus can be interpreted in any category that satisfies them. This two stage technique of

$$\text{nu-calculus} \longrightarrow \text{computational metalanguage} \longrightarrow \text{category with strong monad}$$

makes the construction simpler and allows us to build more than one categorical model on the same foundations. Both parts are significant: the metalanguage serves as the internal language of the category, while the existence of categorical models proves the consistency of the metalanguage. This method is also extensible; if we wish to replace the nu-calculus with a larger fragment of ML, then we need only adjust the notion of 'computation'. Of course, finding the correct adjustment, and models that satisfy it, is not trivial. But the framework remains the same.

Sections 5 and 6 present two examples of categories that model the nu-calculus. The first is a functor category $Set^{\mathcal{I}}$, which captures the basic properties of dynamic name creation. The second refines this, using categories with relations, to give a model that is fully abstract for ground and first-order types (Theorem 4).

2 The Nu-Calculus

The *nu-calculus* is a simple language providing higher-order functions and the dynamic creation of names. It was identified by Pitts as a sensible subset of ML, and is close to Stoughton's *identity calculus*. This section describes the syntax, type system and operational semantics of the language. We then define a notion of observational equivalence, and give a collection of sample equivalences and inequivalences of increasing sophistication.

2.1 Syntax

The syntax of the nu-calculus extends a simply-typed lambda-calculus. Types σ are built up from ground types o of *booleans* and ν of *names* by formation of *function types* $\sigma \to \sigma'$. Expressions have the form

M	$::=$	x	variable
	$\mid$	n	name
	$\mid$	$true \mid false$	truth values
	$\mid$	$if\ M\ then\ M\ else\ M$	conditional
	$\mid$	$M = M$	compare names
	$\mid$	$\nu n.M$	create new name n in expression M
	$\mid$	$\lambda x{:}\sigma.M$	function abstraction
	$\mid$	MM	function application

There are separate infinite supplies of typed variables and names. Function abstraction $\lambda x{:}\sigma.M$ binds the variable x of type σ, and name creation $\nu n.M$ binds the name n. We implicitly identify expressions which only differ in their choice of bound variables and names (α-conversion). A useful abbreviation is new for $\nu n.n$.

We denote by $M[M'/x]$ (respectively $M[M'/n]$) the result of substituting the expression M' for free occurrences of the variable x (respectively, the name n) in the expression M. The substitution is *capture avoiding*; the free names and variables of M' should be disjoint from the bound names and variables of M. This can always be arranged by α-converting M.

Expressions are given types according to the rules in Figure 1. The type assertion

$$s, \Gamma \vdash M : \sigma$$

says that in the presence of s, Γ, the expression M has type σ. Here s is a finite set of names, Γ is a finite set of typed variables, and M is an expression with free names in s and free variables in Γ. The symbol $\oplus$ represents disjoint union, as in $s \oplus \{n\}$ and $\Gamma \oplus [x : \sigma]$.

$$\frac{}{s,\Gamma \vdash x : \sigma}\ (x : \sigma \in \Gamma) \qquad \frac{}{s,\Gamma \vdash n : \nu}\ (n \in s) \qquad \frac{}{s,\Gamma \vdash b : o}\ (b = \mathit{true}, \mathit{false})$$

$$\frac{s,\Gamma \vdash B : o \qquad s,\Gamma \vdash M : \sigma \qquad s,\Gamma \vdash M' : \sigma}{s,\Gamma \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' : \sigma}$$

$$\frac{s,\Gamma \vdash N : \nu \qquad s,\Gamma \vdash N' : \nu}{s,\Gamma \vdash (N = N') : o} \qquad \frac{s \oplus \{n\},\Gamma \vdash M : \sigma}{s,\Gamma \vdash \nu n.M : \sigma}$$

$$\frac{s,\Gamma \oplus [x : \sigma] \vdash M : \sigma'}{s,\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \sigma'} \qquad \frac{s,\Gamma \vdash F : \sigma \to \sigma' \qquad s,\Gamma \vdash M : \sigma}{s,\Gamma \vdash FM : \sigma'}$$

Figure 1: Rules for assigning types to expressions of the nu-calculus

An expression is in *canonical form* if it is either a name, a variable, one of the boolean constants true or false, or a function abstraction. These are to be the *values* of the nu-calculus, and correspond to weak head normal form in the lambda-calculus. An expression is *closed* if it has no free variables; a closed expression may still have free names.

It is immediate that if $s,\Gamma \vdash M : \sigma$ holds then the type σ is unique. We define the sets

$$\begin{aligned} \mathrm{Exp}_\sigma(s) &= \{M \mid s,\emptyset \vdash M : \sigma\} \\ \mathrm{Can}_\sigma(s) &= \{C \mid C \in \mathrm{Exp}_\sigma(s),\ C \text{ canonical}\} \end{aligned}$$

of closed expressions and closed canonical expressions respectively, at any type σ and for any finite set s of names.

2.2 Operational Semantics

The operational semantics of the nu-calculus is specified by the inductively defined evaluation relation given in Figure 2. Elements of the relation take the form

$$s \vdash M \Downarrow_\sigma (s')C$$

where s and s' are disjoint finite sets of names, $M \in \mathrm{Exp}_\sigma(s)$ and $C \in \mathrm{Can}_\sigma(s \oplus s')$. This means that in the presence of names s, the expression M of type σ evaluates to canonical form C and creates fresh names s'.

The definition of Standard ML would suggest a relation of the form $s_1, M \Downarrow_\sigma s_2, C$, meaning that in state s_1, expression M evaluates to value C in state s_2. We choose instead the form above to highlight the fact that the only possible change of 'state' is the creation of additional names.

The form of the rules shows the left to right order of evaluation. For example, with the expression $N = N'$, the rules (EQ1) and (EQ2) both evaluate N before N'. The call-by-value nature of the nu-calculus is captured by the choice of a strict (APP) rule; here the argument M is evaluated to canonical form C before being substituted in the body of the abstraction $\lambda x{:}\sigma.M'$.

(CAN) $$\frac{}{s \vdash C \Downarrow_\sigma C}$$

(COND1) $$\frac{s \vdash B \Downarrow_o (s_1)true \qquad s \oplus s_1 \vdash M \Downarrow_\sigma (s_2)C}{s \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' \Downarrow_\sigma (s_1 \oplus s_2)C}$$

(COND2) $$\frac{s \vdash B \Downarrow_o (s_1)false \qquad s \oplus s_1 \vdash M' \Downarrow_\sigma (s_2)C'}{s \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' \Downarrow_\sigma (s_1 \oplus s_2)C'}$$

(EQ1) $$\frac{s \vdash N \Downarrow_\nu (s_1)n \qquad s \oplus s_1 \vdash N' \Downarrow_\nu (s_2)n}{s \vdash (N = N') \Downarrow_o (s_1 \oplus s_2)true} \qquad n \in s$$

(EQ2) $$\frac{s \vdash N \Downarrow_\nu (s_1)n \qquad s \oplus s_1 \vdash N' \Downarrow_\nu (s_2)n'}{s \vdash (N = N') \Downarrow_o (s_1 \oplus s_2)false} \qquad n,\ n' \text{ distinct}$$

(LOCAL) $$\frac{s \oplus \{n\} \vdash M \Downarrow_\sigma (s_1)C}{s \vdash \nu n.M \Downarrow_\sigma (\{n\} \oplus s_1)C} \qquad n \notin s$$

(APP) $$\frac{\begin{array}{c} s \vdash F \Downarrow_{\sigma\to\sigma'} (s_1)\lambda x{:}\sigma.M' \qquad s \oplus s_1 \vdash M \Downarrow_\sigma (s_2)C \\ s \oplus s_1 \oplus s_2 \vdash M'[C/x] \Downarrow_{\sigma'} (s_3)C' \end{array}}{s \vdash FM \Downarrow_{\sigma'} (s_1 \oplus s_2 \oplus s_3)C'}$$

Figure 2: Rules for evaluating expressions of the nu-calculus

Occasionally, in applying these rules it is necessary to relabel bound names. For example, to evaluate $(\nu n.n = \nu n.n)$ we do not use $\vdash \nu n.n = \nu n.n \Downarrow_o (n,n)true$ because it is not well formed; new names have to be distinct, and this is enforced by the disjoint union $\oplus$ in the (LOCAL) rule. Instead we relabel one of the n's in order to obtain $\vdash \nu n.n = \nu n'.n' \Downarrow_o (n,n')false$. As we have previously identified expressions up to α-conversion, this is quite legitimate, but perhaps surprising. The phenomenon is identical to the reduction of a term such as $(\lambda y.\lambda x.yx)x$ in the traditional lambda-calculus, where the bound occurrence of x has to be relabelled to allow

$$(\lambda y.\lambda z.yz)x \quad \longrightarrow \quad \lambda z.xz.$$

In principle, these difficulties can be resolved by using de Bruijn indices, but at the cost of a considerably loss of clarity. In practice we simply avoid the problem wherever we can by choosing sensible bindings to begin with.

The abbreviation new for $\nu n.n$ was introduced earlier. This has the derived evaluation rule:

(NEW) $$\frac{}{s \vdash new \Downarrow_\nu (\{n\})n} \qquad n \notin s$$

The rules (LOCAL) and (NEW) are entirely equivalent, and we could formulate the nu-calculus with new as primitive and $\nu n.M$ an abbreviation for $(\lambda n{:}\nu.M)new$. Unfortunately both of the forms $\nu n.M$ and $(\lambda n{:}\nu.M)new$ can blur the distinctions between a name, a label bound to a name, and a variable of type ν. Rather than resort to heavy meta-syntactic machinery for a solution, we simply choose whichever of new and $\nu n.M$ seems appropriate.

Evaluation of nu-calculus expressions always terminates, and is deterministic up to choice of new names (see [28], Lemma 2.1 and Theorem 3.3).

2.3 Observational Equivalence

The operational semantics gives rise to a notion of equivalence between nu-calculus expressions, based on their behaviour when used in larger expressions. Informally, two expressions are equivalent if they can be freely exchanged; there is no way in the language itself to distinguish between them.

Define a *program* to be a closed expression of boolean type. All that we can observe of a program is whether it evaluates to $true$ or $false$; the creation of new names is not directly observable. A *program context* $P[-]$ is a program with zero or more occurrences of a hole $[-]$. If M is some closed expression then $P[M]$ is the program obtained by substituting M for every occurrence of this hole.

Definition (Observational equivalence). For $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$, the assertion

$$s \vdash M_1 \approx_\sigma M_2$$

means that for all program contexts $P[-]$ and $b \in \{true, false\}$,

$$\exists s_1\,.\,(s \vdash P[M_1] \Downarrow_o (s_1)b) \quad \Longleftrightarrow \quad \exists s_2\,.\,(s \vdash P[M_2] \Downarrow_o (s_2)b).$$

When this holds, we say that M_1 and M_2 are *observationally equivalent*.

In fact it is not necessary to consider all possible contexts $P[-]$; we need only pass expressions to a boolean *test function* of the form $(\lambda x{:}\sigma.B)$. This is Lemma 2.3 from [28], and is a weak form of Milner's 'Context Lemma':

Theorem 1 (Context Lemma) *Two closed expressions are observationally equivalent* $s \vdash M_1 \approx_\sigma M_2$ *if and only if for all* $b \in \{true, false\}$ *and all* $\lambda x{:}\sigma.B \in \mathrm{Can}_{\sigma\to o}(s)$*:*

$$\exists s_1\,.\,(s \vdash (\lambda x{:}\sigma.B)M_1 \Downarrow_o (s_1)b) \quad \Longleftrightarrow \quad \exists s_2\,.\,(s \vdash (\lambda x{:}\sigma.B)M_2 \Downarrow_o (s_2)b).$$

Similar results have been found for other calculi. Gordon in his thesis shows that experimental order coincides with contextual order for the language $\mu\nu ML$ [5, Section 4.4]. Mason and Talcott [12] consider an untyped lambda-calculus extended with storage cells and show that to establish operational equivalence it is enough to consider all 'closed instantiations of use' (ciu) of an expression.

Examples.

1. Observational equivalence is a congruence: if $s \vdash M_1 \approx_\sigma M_2$ and $s \oplus s', [x:\sigma] \vdash M' : \sigma'$ then $s \oplus s' \vdash M'[M_1/x] \approx_{\sigma'} M'[M_2/x]$.

2. If $M \in \mathrm{Exp}_\sigma(s)$ and $n \notin s$ then $s \vdash \nu n.M \approx_\sigma M$.

3. If $M \in \mathrm{Exp}_\sigma(s \oplus \{n, n'\})$ then $s \vdash \nu n.\nu n'.M \approx_\sigma \nu n'.\nu n.M$.

4. If $s \vdash M \Downarrow_\sigma (n_1, \ldots, n_k)C$ then $s \vdash M \approx_\sigma \nu n_1 \ldots \nu n_k.C$.

5. If $s, [x:\sigma] \vdash M : \sigma'$ and $C \in \mathrm{Can}_\sigma(s)$ then $s \vdash (\lambda x{:}\sigma.M)C \approx_{\sigma'} M[C/x]$.

The first of these follows from the definition of observational equivalence, and the remainder from Theorem 1. The last is Plotkin's call-by-value β_v-equivalence [29]; the nu-calculus does not satisfy general β-equivalence, for example:

6. $\vdash (\lambda x{:}\nu.x = x)new \not\approx_o (new = new)$. The left hand side reduces to $true$ and the right hand side to $false$.

Local name declaration and function abstraction do not in general commute, as is shown by:

7. $\vdash \nu n.\lambda x{:}o.n \not\approx_{o\to\nu} \lambda x{:}o.\nu n.n$. These can be distinguished by the test function $\lambda f{:}o \to \nu.(f true = f true)$.

Some more sophisticated examples of observational equivalence are:

8. $\vdash \nu n.\lambda x{:}\nu.(x = n) \approx_{\nu\to o} \lambda x{:}\nu.false$.

9. $\vdash \nu n.\nu n'.\lambda f{:}\nu \to o.(fn = fn') \approx_{(\nu\to o)\to o} \lambda f{:}\nu \to o.true$.

In this last example the boolean test $fn = fn'$ is an abbreviation for

$$if\ fn\ then\ fn'\ else\ (if\ fn'\ then\ false\ else\ true).$$

The idea in (8) is that no external context can supply the private name n. Similarly in (9) no externally produced function can distinguish the private names n and n'.

It is however extraordinarily hard to make precise this notion of privacy, particularly where higher-order functions are involved. The next case shows two expressions which at first sight look to be observationally equivalent for the same reason as those in (9):

10. $\vdash \nu n.\lambda f{:}\nu \to o.\nu n'.(fn = fn') \not\approx_{(\nu\to o)\to o} \lambda f{:}\nu \to o.true$

These are distinguished by $\lambda F{:}(\nu \to o) \to o.F(\lambda x{:}\nu.F(\lambda y{:}\nu.x = y))$.

The problem here is that although the name bound to n remains private, the function $\lambda x{:}\nu.F(\lambda y{:}\nu.x = y)$ is able to distinguish n from the fresh names successively bound to n'.

Another tricky example shows that it may be necessary to apply functions repeatedly in order to distinguish them, without any private names being revealed.

11. $\vdash\ \nu n.\nu n'.\lambda f : \nu \to o\,.\,if\ fn = fn'$
$then\ (\lambda x{:}\nu.\ if\ x = n\ \ then\ true$
$else\ if\ x = n'\ then\ false$
$else\ fx)$
$else\ (\lambda x{:}\nu.true)$
$\not\approx_{(\nu\to o)\to(\nu\to o)} \lambda f : \nu \to o\,.\,f$

These are distinguished by

$$\lambda F{:}(\nu \to o) \to (\nu \to o).F(F(\lambda x{:}\nu.false))new.$$

What is happening here is that the two functions differ noticeably only on arguments of type $(\nu \to o)$ that can distinguish n from n'. As both names are private, we cannot construct such a function directly. However when we pass the first expression an argument that we can construct, such as $(\lambda x{:}\nu.false)$, we get back one that does distinguish n from n', in this case $(\lambda x{:}\nu.x = n)$. Although this is externally no different from what we started with (see equivalence (8) above), it is a suitable argument to separate the two original expressions.

Up to observational equivalence, the only closed expressions of type o are $true$ and $false$, and the only closed expressions of type ν are the names in the name context and new. Higher types are more complicated; there are infinitely many operationally distinct closed expressions of type $(\nu \to \nu)$, as demonstrated by Example 2.5 of [28].

3 A Computational Metalanguage

As a first step towards a denotational semantics for the nu-calculus, we interpret it in a typed metalanguage based on the *computational lambda-calculus* of Moggi [17, 19, 26].

The most important feature of this is that it distinguishes between values and computations; for the nu-calculus, a computation will create some names and then return a value. This separation makes explicit the order of computation, which in the operational semantics was only implicit. It also allows equational reasoning, with the reintroduction of β and η axioms at function types.

When we construct categorical models, the computational metalanguage will be their internal language. Much of our calculation is done in this, rather than with commuting diagrams.

Over the metalanguage we use an equational Horn clause logic. This matches the internal logic of categorical models, and more abstract models validate more equalities. Alternatively, we could use a variety of evaluation logic as described by Pitts in [26].

The interpretation of the nu-calculus in this metalanguage is both correct and adequate. This means that it is suitable for reasoning about observational equivalence; indeed it is complete at ground types.

3.1 Syntax of the Metalanguage

The types of the metalanguage mirror those of the nu-calculus:

$$\begin{array}{rcll} A & ::= & Bool & \text{booleans} \\ & | & Name & \text{names} \\ & | & A \to A & \text{functions} \\ & | & TA & \text{computations} \end{array}$$

There is a unary type constructor T; if A is a type, then elements of TA are *computations* of type A. In this particular metalanguage, the difference between values and computations is that computations may generate new names before returning a value.

The term-forming operations of the metalanguage are

$$\begin{array}{rcll} a & ::= & x & \text{variable} \\ & | & tt \mid ff & \text{truth values} \\ & | & cond(a,a,a) & \text{conditional} \\ & | & eq(a,a) & \text{compare names} \\ & | & new & \text{generate a new name} \\ & | & \lambda x \in A.a & \text{function abstraction} \\ & | & aa & \text{function application} \\ & | & [a] & \text{value as trivial computation} \\ & | & let\ x \Leftarrow a\ in\ a & \text{sequential computation} \end{array}$$

Function abstraction $\lambda x{:}A.a$ binds the variable x in the term a, and sequential computation $let\ x \Leftarrow e\ in\ e'$ binds the variable x within the term e'. We implicitly identify terms up to α-conversion, which allows us to require substitution $a[a'/x]$ to be capture avoiding. A term is *closed* if it has no free variables; there is no possibility of free names. As in the nu-calculus, the only operation provided on $Name$ is the equality test eq.

$$\frac{}{\Gamma \vdash x : A}\ (x : A \in \Gamma) \qquad \frac{}{\Gamma \vdash \mathit{new} : \mathit{TName}} \qquad \frac{}{\Gamma \vdash \mathit{tt} : \mathit{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{ff} : \mathit{Bool}}$$

$$\frac{\Gamma \vdash n, n' : \mathit{Name}}{\Gamma \vdash \mathit{eq}(n, n') : \mathit{Bool}} \qquad \frac{\Gamma \vdash b : \mathit{Bool} \qquad \Gamma \vdash a, a' : A}{\Gamma \vdash \mathit{cond}(b, a, a') : A}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : TA} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b : A \to B}$$

$$\frac{\Gamma \vdash e : TA \qquad \Gamma, x : A \vdash e' : A'}{\Gamma \vdash \mathit{let}\, x \Leftarrow e\, \mathit{in}\, e' : TA'} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B}$$

Figure 3: Rules for assigning types to terms of the metalanguage

There are three forms of term involving computation. If a is a value, then $[a]$ is the trivial computation which simply returns a. The sequential form $\mathit{let}\, x \Leftarrow e\, \mathit{in}\, e'$ carries out the computation e, binds the result to x and then computes e'. These are both standard constructions of the computational lambda-calculus. We have also the constant new of type TName which denotes the computation that generates a fresh name.

Type judgements of the metalanguage take the form

$$\Gamma \vdash a : A$$

which asserts that in the presence of Γ, term a has type A. Here Γ is a finite set of typed variables; unlike the nu-calculus, there is no set of free names. The rules for forming valid type judgements are given in Figure 3. We write $\Gamma, x : A$ for the disjoint union $\Gamma \oplus [x : A]$, and the abbreviation $\Gamma \vdash a_1, \ldots, a_n : A$ indicates that all of the judgements $\Gamma \vdash a_1 : A, \ldots, \Gamma \vdash a_n : A$ hold.

3.2 A Logic for the Metalanguage

We reason about terms of the metalanguage with an equational logic of Horn clauses. This could be extended to a full *evaluation logic* with modalities, but we shall manage without this sophistication. If the type judgements $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$ are valid then

$$\Gamma \vdash a = a'$$

is an *equation in context* Γ. A *sequent* is

$$\Gamma; \Phi \vdash \phi$$

where Γ is a finite set of typed variables, Φ is a finite set of equations in context Γ and ϕ is a single equation in context Γ. We derive sequents using the rules in Figure 4 together with the usual rules for Horn clauses and equational logic, congruence rules for all term-forming operations and β, η axioms for functions.

Computations:

$$\frac{\Gamma \vdash e : TA}{\Gamma \vdash \mathit{let}\, x \Leftarrow e\, \mathit{in}\, [x] = e} \qquad \text{(MONO)} \qquad \frac{\Gamma \vdash a, a' : A}{\Gamma \vdash ; [a] = [a'] \vdash a = a'}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash e : TB}{\Gamma \vdash \mathit{let}\, x \Leftarrow [a]\, \mathit{in}\, e = e[a/x]}$$

$$\frac{\Gamma \vdash e : TA \qquad \Gamma, x : A \vdash e' : TA' \qquad \Gamma, x' : A' \vdash e'' : TA''}{\Gamma \vdash \mathit{let}\, x' \Leftarrow (\mathit{let}\, x \Leftarrow e\, \mathit{in}\, e')\, \mathit{in}\, e'' = \mathit{let}\, x \Leftarrow e\, \mathit{in}\, (\mathit{let}\, x' \Leftarrow e'\, \mathit{in}\, e'')}$$

Booleans:

$$\frac{\Gamma; \Phi, b = tt \vdash \phi \qquad \Gamma; \Phi, b = ff \vdash \phi}{\Gamma; \Phi \vdash \phi} \qquad \frac{\Gamma; \Phi \vdash tt = ff}{\Gamma; \Phi \vdash \phi}$$

$$\frac{\Gamma \vdash a, a' : A}{\Gamma \vdash \mathit{cond}(tt, a, a') = a} \qquad \frac{\Gamma \vdash a, a' : A}{\Gamma \vdash \mathit{cond}(ff, a, a') = a'}$$

Testing names:

$$\frac{\Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{eq}(n, n) = tt} \qquad \frac{\Gamma \vdash n, n' : \mathit{Name}}{\Gamma; \mathit{eq}(n, n') = tt \vdash n = n'}$$

Generating names:

(DROP) $$\frac{\Gamma \vdash e : TA}{\Gamma \vdash e = \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, e} \quad (n : \mathit{Name} \notin \Gamma)$$

(SWAP) $$\frac{\Gamma, n, n' : \mathit{Name} \vdash e : TA}{\Gamma \vdash \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, e = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, e}$$

(FRESH) $$\frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma, n' : \mathit{Name}; \Phi, \mathit{eq}(n, n') = ff \vdash e = e'}{\Gamma; \Phi \vdash \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, e = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, e'}$$

Figure 4: Rules for reasoning in the metalanguage

The rules in Figure 4 for computations are those described by Moggi for any computational lambda-calculus, and include the (MONO) rule, that the operation $[-]$ taking values to computations is an inclusion. The rules for boolean values and the comparison of names are straightforward; a derived property is that $eq(-,-)$ is an equivalence relation.

The final three rules describe the behaviour of the computation new; asserting that unused names are ignored, the order of generating names is irrelevant, and new names are distinct from all others. The choice of these particular rules is rather *ad hoc*; in their favour, we argue that they are sufficient to carry through the interpretation of the nu-calculus, and there are models to validate them. Stronger versions of the first two are

$$\text{(DROP}^{+}\text{)} \qquad \frac{\Gamma \vdash e : TA \qquad \Gamma \vdash e' : TA'}{\Gamma \vdash e = \mathit{let}\, x \Leftarrow e'\, \mathit{in}\, e} \quad (x : A' \notin \Gamma)$$

$$\text{(SWAP}^{+}\text{)} \qquad \frac{\Gamma \vdash e : TA \qquad \Gamma \vdash e' : TA' \qquad \Gamma, x : TA, x' : TA' \vdash e'' : TA''}{\Gamma \vdash \mathit{let}\, x \Leftarrow e\, \mathit{in}\, \mathit{let}\, x' \Leftarrow e'\, \mathit{in}\, e'' = \mathit{let}\, x' \Leftarrow e'\, \mathit{in}\, \mathit{let}\, x \Leftarrow e\, \mathit{in}\, e''}.$$

These are not essential to model the nu-calculus, and would be false in a metalanguage extended to handle store or exceptions, for example. Nevertheless, all the categorical models to follow satisfy them.

An equivalent formulation of the last rule (FRESH) is

$$\text{(FRESH}'\text{)} \qquad \frac{\Gamma, b : \mathit{Bool}, n' : \mathit{Name} \vdash e : TA \qquad \Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, e[eq(n, n')/b] = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, e[\mathit{ff}/b]}.$$

The alternative candidate

$$\frac{\Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, [eq(n, n')] = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, [\mathit{ff}]}$$

can be derived, but appears to be strictly weaker. In particular it is not strong enough to complete the proof of Proposition 1, that the metalanguage correctly interprets the nu-calculus.

3.3 Interpretation of the Nu-Calculus

We extend Moggi's interpretation of the simply-typed call-by-value lambda-calculus in the computational lambda-calculus [17]. Translation of types is:

$$\begin{array}{rcl} [\![o]\!] & = & \mathit{Bool} \\ [\![\nu]\!] & = & \mathit{Name} \\ [\![\sigma \to \sigma']\!] & = & [\![\sigma]\!] \to T[\![\sigma']\!] \end{array}$$

Function types use the constructor T; the application of a function may result in a computation, that is, the generation of new names.

There are two mutually defined schemes that translate nu-calculus expressions into terms of the metalanguage. Figure 5 describes $[\![-]\!]$ for general expressions, $|-|$ for expressions in canonical form, and $[\![-]\!]$ for name and variable contexts. The interpretation respects types:

Canonical forms:

$$\begin{array}{rcl} |x| & = & x \\ |n| & = & n \\ |true| & = & tt \\ |false| & = & ff \\ |\lambda x{:}\sigma.M| & = & \lambda x{:}[\![\sigma]\!].[\![M]\!] \end{array}$$

Expressions:

$$\begin{array}{rcl} [\![C]\!] & = & [|C|] \\ [\![\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M']\!] & = & \mathit{let}\ b \Leftarrow [\![B]\!]\ \mathit{in}\ \mathit{cond}(b, [\![M]\!], [\![M']\!]) \\ [\![N = N']\!] & = & \mathit{let}\ n \Leftarrow [\![N]\!]\ \mathit{in}\ \mathit{let}\ n' \Leftarrow [\![N']\!]\ \mathit{in}\ [\mathit{eq}(n, n')] \\ [\![\nu n.M]\!] & = & \mathit{let}\ n \Leftarrow \mathit{new}\ \mathit{in}\ [\![M]\!] \\ [\![FM]\!] & = & \mathit{let}\ f \Leftarrow [\![F]\!]\ \mathit{in}\ \mathit{let}\ m \Leftarrow [\![M]\!]\ \mathit{in}\ fm \end{array}$$

Contexts:

$$\begin{array}{rcl} [\![s, \Gamma]\!] & = & n_1, \ldots, n_k : \mathit{Name},\ x_1 : [\![\sigma_1]\!], \ldots, x_l : [\![\sigma_l]\!] \\ \text{where } s & = & \{n_1, \ldots, n_k\} \\ \Gamma & = & [x_1 : \sigma_1, \ldots, x_l : \sigma_l] \end{array}$$

Figure 5: Interpretation of the nu-calculus in the computational metalanguage

Lemma 1 *For any nu-calculus expression M, or expression C in canonical form:*

$$\begin{array}{rcl} s, \Gamma \vdash M : \sigma & \iff & [\![s, \Gamma]\!] \vdash [\![M]\!] : T[\![\sigma]\!] \\ s, \Gamma \vdash C : \sigma & \iff & [\![s, \Gamma]\!] \vdash |C| : [\![\sigma]\!] \end{array}$$

Proof By induction over the structure of the type judgement in the nu-calculus, using uniqueness of types in the metalanguage. □

The translation is *correct* with respect to the operational semantics of the nu-calculus: if $s \vdash M \Downarrow_\sigma (s')C$ then the terms $[\![M]\!]$ and $[\![\nu s'.C]\!]$ can be proved equal in the metalanguage, under the assumption that all the names in s are distinct. We first define the abbreviations

$$\begin{array}{rcl} (\neq s) & = & \{eq(n_i, n_j) = \mathit{ff} \mid 1 \leq i < j \leq k\} \\ \mathit{let}\ s' \Leftarrow \overrightarrow{new}\ \mathit{in}\ e & = & \mathit{let}\ n'_1 \Leftarrow \mathit{new}\ \mathit{in} \ldots \mathit{let}\ n'_\ell \Leftarrow \mathit{new}\ \mathit{in}\ e \end{array}$$

where $s = \{n_1, \ldots, n_k\}$ and $s' = \{n'_1, \ldots, n'_\ell\}$. The ordering of names from s and s' does not matter, up to provable equality in the metalanguage. We can now state:

Proposition 1 (Correctness) *If $s \vdash M \Downarrow_\sigma (s')C$ is a valid evaluation judgement, then*

$$[\![s]\!]; (\neq s) \ \vdash\ [\![M]\!] = \mathit{let}\ s' \Leftarrow \overrightarrow{new}\ \mathit{in}\ [\![C]\!]$$

is provable in the metalanguage.

Proof Proceeds by induction over the structure of the derivation of the evaluation judgement $s \vdash M \Downarrow_\sigma (s')C$. We need to confirm that every rule in Figure 2 translates into a derivation that is provable in the metalanguage. The details are routine and are omitted. □

3.4 Reasoning in the Metalanguage

The interpretation of the nu-calculus in the metalanguage is *adequate*, which means that we can use it to reason about observational equivalence. It is strong enough to validate equivalences (2)–(5) of Section 2.3, and is complete at ground types. However, it is less useful at higher types, and fails to confirm examples (8) or (9).

These results rely on the metalanguage being consistent: the equation $\vdash tt = \mathit{ff}$ is not provable. This is justified by the categorical models of Sections 5 and 6 where tt and ff are distinct.

To show that the metalanguage is adequate for reasoning about observational equivalence, we use the characterisation given by Theorem 1, and the fact that evaluation in the nu-calculus is deterministic and terminating.

Proposition 2 (Adequacy) *Suppose that $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ are expressions of the nu-calculus and that we can derive*

$$[\![s]\!]; (\neq s) \ \vdash\ [\![M_1]\!] = [\![M_2]\!]$$

in the metalanguage. Then the expressions are observationally equivalent $s \vdash M_1 \approx_\sigma M_2$.

Proof Suppose that $\lambda x{:}\sigma.B \in \mathrm{Can}_{\sigma\to o}(s)$ is some test function. Then there are evaluation judgements

$$s \vdash (\lambda x{:}\sigma.B)M_i \Downarrow_\sigma (s_i)b_i \qquad i = 1, 2$$

for some name sets s_1, s_2 and booleans b_1, b_2. By Proposition 1 and the compositionality of the translation $[\![-]\!]$, we can reason

$$\begin{aligned}
[\![s]\!]; (\neq s) \vdash [|b_1|] &= \mathit{let}\ s_1 \Leftarrow \overrightarrow{new}\ \mathit{in}\ [|b_1|] \\
&= [\![(\lambda x{:}\sigma.B)M_1]\!] \\
&= [\![(\lambda x{:}\sigma.B)M_2]\!] \\
&= \mathit{let}\ s_2 \Leftarrow \overrightarrow{new}\ \mathit{in}\ [|b_2|] \\
&= [|b_2|].
\end{aligned}$$

By the (MONO) rule

$$\vdash |b_1| = |b_2| : \mathit{Bool}$$

from which $b_1 = b_2$ and hence $s \vdash M_1 \approx_\sigma M_2$ as required. □

For closed boolean and name expressions the converse also holds; any observational equivalence can be proved in the metalanguage:

Theorem 2 (Completeness at Ground Types) *If σ is one of the ground types $\{o, \nu\}$ of the nu-calculus and $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ are two expressions, then*

$$s \vdash M_1 \approx_\sigma M_2 \quad \Longrightarrow \quad [\![s]\!]; (\neq s) \vdash [\![M_1]\!] = [\![M_2]\!].$$

Proof We take each type in turn. Suppose that $\sigma = o$, then there must be some $b \in \{\mathit{true}, \mathit{false}\}$ such that

$$s \vdash M_i \Downarrow_o (s_i)b \qquad i = 1, 2$$

for suitable sets s_1, s_2 of names. By Proposition 1, and repeated use of the (DROP) rule, we can reason

$$\begin{aligned}
[\![s]\!]; (\neq s) \vdash [\![M_1]\!] &= \mathit{let}\ s_1 \Leftarrow \overrightarrow{new}\ \mathit{in}\ [|b|] \\
&= [|b|] \\
&= \mathit{let}\ s_2 \Leftarrow \overrightarrow{new}\ \mathit{in}\ [|b|] \\
&= [\![M_2]\!]
\end{aligned}$$

which gives the desired equality.

If $\sigma = \nu$ then there are two possibilities:

- There is some name $n \in s$ such that

$$s \vdash M_i \Downarrow_\nu (s_i)n \qquad i = 1, 2$$

for suitable sets s_1, s_2 of names. The reasoning is then exactly as in the boolean case.

- There is some name $n \notin s$ and name sets s_1, s_2 such that

$$s \vdash M_i \Downarrow_\nu (\{n\} \oplus s_i)n \qquad i = 1, 2.$$

Using Proposition 1 and (DROP) we derive

$$\begin{aligned} [\![s]\!]; (\neq s) \vdash [\![M_1]\!] &= \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, \mathit{let}\, s_1 \Leftarrow \overrightarrow{\mathit{new}}\, \mathit{in}\, [n] \\ &= \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, [n] \\ &= \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, \mathit{let}\, s_2 \Leftarrow \overrightarrow{\mathit{new}}\, \mathit{in}\, [n] \\ &= [\![M_2]\!]. \end{aligned}$$

In all cases, observational equivalence implies provable equality in the metalanguage. □

4 Modelling the Nu-Calculus Categorically

It is standard that the simply-typed lambda-calculus can be modelled in any cartesian closed category, with objects for types and morphisms for terms [9]. Moggi extends this to a model of the computational lambda-calculus in any cartesian closed category equipped with a strong monad T [19]. We specialise to the particular case of the computational metalanguage for the nu-calculus, using the technique outlined in Section 4 of [28].

The method is that if a category $\mathcal{C}$ satisfies certain requirements then its *internal language* will include the metalanguage, and so provide a model of the nu-calculus which is sound with respect to the operational semantics. If $\mathcal{C}$ is not degenerate then the translation is also adequate, and the category can be used to reason about observational equivalence. This will be at least as powerful as the basic metalanguage; in particular reasoning in such $\mathcal{C}$ is complete for observational equivalence at ground types. Of course the intention is that a suitable choice of category might prove more than the metalanguage alone.

This is the *monadic* approach to denotational semantics. Like the computational metalanguage, its chief advantage is the separation of values from computations. The monad T encapsulates the notion of computation; in the case of the nu-calculus, this is the action of generating new names. The remainder of the model can then be constructed without special regard to the nature of computation. In particular, the usual products and exponentials are sufficient to model contexts and functions respectively.

4.1 Requirements for a Categorical Model

A category $\mathcal{C}$ is suitable to model the metalanguage of Section 3 if the following conditions are satisfied:

- It is cartesian closed. This gives products for contexts and exponentials for function types.

- It has a *strong monad* T, used to interpret the computation types. This can be described as a endofunctor $T : \mathcal{C} \to \mathcal{C}$ together with a *unit* natural transformation $\eta : 1 \to T$ and a *lift* operation taking a morphism $f : A \times B \to TC$ to $f^* : A \times TB \to C$. The lift operation must be natural and satisfy

$$\begin{array}{rcl} (\eta_B \circ snd_{A,B})^* & = & snd_{A,TB} \\ f^* \circ (id_A \times \eta_B) & = & f \\ g^* \circ \langle fst_{A,TB}, f^* \rangle & = & (g^* \circ \langle fst_{A,B}, f \rangle)^* \end{array}$$

  whenever $f : A \times B \to TC$ and $g : A \times C \to TD$. These correspond precisely to the computation rules for let in Figure 4. The lift operation for a strong monad is a generalisation of that for a Kleisli triple, a particular presentation of an ordinary categorical monad. The generalisation is necessary to carry around contexts of let-expressions, as Moggi explains in [19, Remark 3.1].

  A strong monad can also be presented as a monad (T, η, μ) together with natural maps $t_{A,B} : A \times TB \to T(A \times B)$. Here $\eta : 1 \to T$ and $\mu : T^2 \to T$ are the usual natural transformations for a monad, and the *strength* $t_{A,B}$ must satisfy certain equations. Moggi gives a detailed explanation of this, and further comments on the characterisation of strong monads in [19, Definition 3.2 *et seq.*].

- The monad T satisfies the *mono requirement*, that all $\eta_A : A \to TA$ are monic. This corresponds to the (MONO) rule of Figure 4.

- The coproduct $1 + 1$ of the terminal object 1 with itself exists and is *disjoint*, meaning that the square

$$\begin{array}{ccc} 0 & \longrightarrow & 1 \\ \downarrow & & \downarrow{\scriptstyle ff} \\ 1 & \xrightarrow[tt]{} & 1 + 1 \end{array}$$

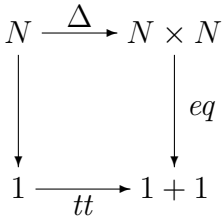
  is a pullback. Here tt and ff are the left and right inclusion maps. This is used to model the type of booleans; given that $\mathcal{C}$ is cartesian closed, we can define for each object A a morphism

$$\begin{array}{rcl} cond_A & = & eval \circ ([curry(fst_{A,A}), curry(snd_{A,A})] \times id_{A \times A}) \\ & : & (1 + 1) \times (A \times A) \longrightarrow A \end{array}$$

  to interpret the conditional.

- There is a distinguished object N, used to interpret the type of names. This must be

decidable, which requires a morphism $eq : N \times N \to 1 + 1$ such that



is a pullback square, where Δ is the diagonal map. The morphism eq interprets the equality test on names. In the internal language of $\mathcal{C}$, the pullback condition corresponds to the rules for testing names in Figure 4.

- There is a distinguished morphism $new : 1 \to TN$ such that for any morphisms $f : A \to TB$, $g : A \times N \times N \to TB$ and $h : A \times (1+1) \times N \times N \to TB$ the following equations in the internal language of $\mathcal{C}$ are satisfied:

$$\begin{aligned} a : A &\vdash \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, f(a) = f(a) \\ a : A &\vdash \mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, (\mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, g(a,n,n')) \\ &\qquad = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, (\mathit{let}\, n \Leftarrow \mathit{new}\, \mathit{in}\, g(a,n,n')) \\ a : A, n : N &\vdash \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, h(a, \mathit{eq}(n,n'), n, n') \\ &\qquad = \mathit{let}\, n' \Leftarrow \mathit{new}\, \mathit{in}\, h(a, \mathit{ff}, n, n') \end{aligned}$$

  It is clear that these are simply the rules for generating names of Figure 4, with the alternate form (FRESH′). We could express them by commutative diagrams asserting the equality of certain morphisms in $\mathcal{C}$, but their essence becomes lost in a mass of variable manipulation.

  The first two of these equations hold automatically if the monad T is respectively *affine* and *commutative*; these are notions due to Kock [7, 8]. A strong monad is affine if

$$\mathit{fst}_{TA,TB} = \mathit{fst}^*_{TA,B} : TA \times TB \to TA$$

  for all objects A, B; equivalently, if $\eta_1 : 1 \to T1$ is an isomorphism. It is commutative if the two natural maps from $TA \times TB$ to $T(A \times B)$ are equal:

$$(\eta^*_{A\times B} \circ \mathit{tw}_{TB,A})^* \circ \mathit{tw}_{TA,TB} = ((\eta_{A\times B} \circ \mathit{tw}_{B,A})^* \circ \mathit{tw}_{TA,B})^*.$$

  Here $tw_{X,Y} : X \times Y \to Y \times X$ is the twist map. These stronger conditions correspond to the rules (DROP$^+$) and (SWAP$^+$) for the metalanguage.

4.2 Construction of a Categorical Model

Suppose that we have a cartesian closed category $\mathcal{C}$ with a disjoint coproduct $1 + 1$, a strong monad T satisfying the (MONO) condition, a distinguished decidable object N and a distinguished morphism $new : 1 \to TN$ satisfying the necessary equalities. Interpreting the metalanguage, and hence the nu-calculus, in $\mathcal{C}$ is quite standard.

Types of the metalanguage are interpreted by objects of the category: $Bool$ by $1+1$, $Name$ by N, function types by exponentials, and computations using the strong monad T. A context $\Gamma = \{x_1 \in A_1, \ldots, x_n \in A_n\}$ is interpreted by the product $\Gamma = A_1 \times \cdots \times A_n$. A term in context is interpreted by a morphism:

$$\Gamma \vdash a : A \quad \longmapsto \quad a : \Gamma \to A.$$

The derivation of such a morphism uses the rules on the right of Figure 6, which match those on the left for terms of the metalanguage. An equation in context is interpreted by equality of morphisms:

$$\Gamma \vdash a = a' : A \quad \longmapsto \quad a = a' : \Gamma \to A.$$

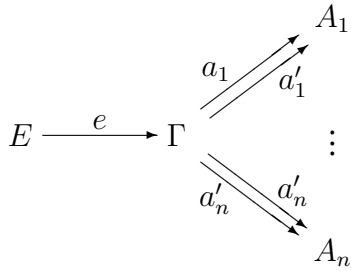
The sequent

$$\Gamma; a_1 = a_1' : A_1, \ldots, a_n = a_n' : A_n \vdash a = a' : A$$

is interpreted by equality of the morphisms

$$E \xrightarrow{\ e\ } \Gamma \underset{a'}{\overset{a}{\rightrightarrows}} A \qquad a \circ e = a' \circ e$$

where $e : E \to \Gamma$ is the simultaneous equalizer of all the equations on the left hand side:



Under this embedding the conditions on $\mathcal{C}$ correspond exactly to the rules of Figure 4 for reasoning in the metalanguage; so any equation provable in the metalanguage will hold in $\mathcal{C}$. As a result, any non-degenerate model demonstrates that the metalanguage is consistent.

Combining the two translations, we obtain a model in $\mathcal{C}$ of the nu-calculus. For each valid typing assertion $s, \Gamma \vdash M : \sigma$ this gives a morphism

$$[\![M]\!] : N^{|s|} \times [\![\Gamma]\!] \to T[\![\sigma]\!] \quad \text{where} \quad [\![\Gamma]\!] = \prod_{x_i : \sigma_i \in \Gamma} [\![\sigma_i]\!].$$

For an expression C in canonical form this morphism factors through $\eta : [\![\sigma]\!] \to T[\![\sigma]\!]$ and there is

$$|C| : N^{|s|} \times [\![\Gamma]\!] \to [\![\sigma]\!] \quad \text{with} \quad [\![C]\!] = \eta_{[\![\sigma]\!]} \circ |C|.$$

Here $N^{|s|}$ is the object of $|s|$-tuples of names. We define the subobject $(\neq s) \to N^{|s|}$ of

$$\frac{}{\Gamma \vdash x : A} \quad\longmapsto\quad \frac{}{\pi_x : \Gamma \to A} \qquad (x : A \in \Gamma)$$

$$\frac{}{\Gamma \vdash new : TName} \quad\longmapsto\quad \frac{}{\Gamma \xrightarrow{!} 1 \xrightarrow{new} TN}$$

$$\frac{}{\Gamma \vdash tt : Bool} \quad\longmapsto\quad \frac{}{\Gamma \xrightarrow{!} 1 \xrightarrow{tt} 1+1}$$

$$\frac{}{\Gamma \vdash ff : Bool} \quad\longmapsto\quad \frac{}{\Gamma \xrightarrow{!} 1 \xrightarrow{ff} 1+1}$$

$$\frac{\Gamma \vdash n, n' : Name}{\Gamma \vdash eq(n,n') : Bool} \quad\longmapsto\quad \frac{n : \Gamma \to N \qquad n' : \Gamma \to N}{\Gamma \xrightarrow{\langle n,n'\rangle} N \times N \xrightarrow{eq} 1+1}$$

$$\frac{\Gamma \vdash b : Bool \qquad \Gamma \vdash a, a' : A}{\Gamma \vdash cond(b,a,a') : A} \quad\longmapsto\quad \frac{b : \Gamma \to 1+1 \quad a : \Gamma \to A \quad a' : \Gamma \to A}{\Gamma \xrightarrow{\langle b,a,a'\rangle} (1+1) \times A \times A \xrightarrow{cond_A} A}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : TA} \quad\longmapsto\quad \frac{a : \Gamma \to A}{\Gamma \xrightarrow{a} A \xrightarrow{\eta_A} TA}$$

$$\frac{\Gamma \vdash e : TA \qquad \Gamma, x : A \vdash e' : A'}{\Gamma \vdash let\, x \Leftarrow e\, in\, e' : TA'} \quad\longmapsto\quad \frac{e : \Gamma \to TA \qquad e' : \Gamma \times A \to TA'}{\Gamma \xrightarrow{\langle 1,e\rangle} \Gamma \times TA \xrightarrow{(e')^*} TA'}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b : A \to B} \quad\longmapsto\quad \frac{b : \Gamma \times A \to B}{curry(b) : \Gamma \to B^A}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B} \quad\longmapsto\quad \frac{f : \Gamma \to B^A \qquad a : \Gamma \to A}{\Gamma \xrightarrow{\langle f,a\rangle} B^A \times A \xrightarrow{eval} B}$$

Figure 6: Rules for constructing morphisms to interpret terms of the metalanguage

distinct $|s|$-tuples as the simultaneous equaliser of all the pairs

$$N^{|s|} \underset{ff \circ !}{\overset{eq \circ \langle \pi_i, \pi_j \rangle}{\rightrightarrows}} 1 + 1 \qquad 1 \leq i < j \leq |s|.$$

In the internal language, this corresponds to the conjunction

$$x_1 : N, \ldots, x_{|s|} : N \vdash \bigwedge_{1 \leq i < j \leq |s|} (eq(x_i, x_j) = ff).$$

We then define the composite morphisms:

$$\begin{aligned} [\![M]\!]_{\neq s} &= \Big((\neq s) \to N^{|s|} \xrightarrow{[\![M]\!]} T[\![\sigma]\!] \Big) \quad M \in \mathrm{Exp}_\sigma(s) \\ |C|_{\neq s} &= \Big((\neq s) \to N^{|s|} \xrightarrow{|C|} [\![\sigma]\!] \Big) \qquad C \in \mathrm{Can}_\sigma(s). \end{aligned}$$

The results of Section 3 now carry over to this categorical model:

Proposition 3 (Correctness) *If* $s \vdash M \Downarrow_\sigma (s')C$ *is a valid evaluation judgement then*

$$[\![M]\!]_{\neq s} = [\![\nu s'.C]\!]_{\neq s}.$$

Proof Follows from Proposition 1. □

Proposition 4 (Adequacy) *Suppose that the category* $\mathcal{C}$ *is* non-degenerate, *in that* $0 \ncong 1$. *Then for all* $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$*:*

$$[\![M_1]\!]_{\neq s} = [\![M_2]\!]_{\neq s} \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \approx_\sigma M_2.$$

Proof Exactly as for Proposition 2. □

Theorem 3 (Completeness at Ground Types) *If* $\sigma \in \{o, \nu\}$ *and* $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ *then*

$$s \vdash M_1 \approx_\sigma M_2 \quad \Longrightarrow \quad [\![M_1]\!]_{\neq s} = [\![M_2]\!]_{\neq s}.$$

Proof Follows from Theorem 2. □

So a non-degenerate categorical model can be used to prove observational equivalences of the nu-calculus. The more that can be shown, the more *abstract* a model is. It is *fully abstract* if the result of Theorem 3 holds at all types σ. More modestly, a model may be fully abstract for some restricted set of types or expressions. Any adequate categorical model will validate at least the equivalences (2)–(5) of Section 2.3.

In the case of languages like PCF, the difficulties in finding fully abstract models are to do with characterising sequentiality, and arise through ingenious use of non-termination. Because evaluation in the nu-calculus always terminates, the same problems do not occur. Full abstraction is still a hard problem, but in different ways, to do with the privacy of names.

5 The Functor Category $Set^{\mathcal{I}}$

In this section we describe our first example of a category suitable to model the nu-calculus. Although it is not particularly abstract, the existence of the category does prove that the metalanguage is consistent, and justifies using it to reason about contextual equivalence (Proposition 2 above). The construction is based on Moggi's model of dynamic allocation from [18, §4.1.4], and is also related to the *possible worlds* models of Reynolds, Oles, Tennent and O'Hearn.

We take the category $Set^{\mathcal{I}}$ of functors and natural transformations between $\mathcal{I}$, the category of finite sets and injections, and Set, the category of sets and functions. Objects of $\mathcal{I}$ represent *stages of computation*, that is, what names have been declared. We shall use s and variants to stand for objects of $\mathcal{I}$, and the symbol '$+$' for their coproduct. For a functor $A : \mathcal{I} \to Set$, the set As is composed of values defined over the names in s. Morphisms in $\mathcal{I}$ and their images in Set correspond to name substitutions.

It is standard that this category is cartesian closed. Finite limits and colimits are taken pointwise; for example, the object of booleans $1 + 1$ is the constant functor to a two-element set. If $A, B : \mathcal{I} \to Set$ are functors then their exponent is defined:

$$\begin{array}{rcll} B^A s & = & Set^{\mathcal{I}}(\mathcal{I}(s,-) \times A, B) & s, s', s'' \in \mathcal{I} \\ B^A f p s''\langle i, a''\rangle & = & ps''\langle i \circ f, a''\rangle & f : s \to s' \quad p \in B^A s \\ & & & i : s' \to s'' \quad a'' \in As''. \end{array}$$

As well as this standard construction of exponentials, the particular choice of index category $\mathcal{I}$ means that there is an equivalent and simpler way to compute the object part of the functor:

$$B^A s = Set^{\mathcal{I}}(A(s + \_), B(s + \_)).$$

So a function from A to B defined at stage s includes data on how it behaves at all later stages. Naturality places some bounds on what this behaviour can be.

The monad is a colimit of shape $\mathcal{I}$. We use the coproduct functor $+ : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ and take T to be the composition

$$Set^{\mathcal{I}} \xrightarrow{Set^{+}} Set^{\mathcal{I}\times\mathcal{I}} \xrightarrow{\cong} \left(Set^{\mathcal{I}}\right)^{\mathcal{I}} \xrightarrow{\varinjlim} Set^{\mathcal{I}}.$$

Explicitly, on objects it is the quotient

$$TAs \;=\; \{\langle s', a'\rangle \mid s' \in \mathcal{I}, a' \in A(s + s')\}/\sim$$

where $\langle s_1, a_1\rangle \sim \langle s_2, a_2\rangle$ if and only if for some s_0 there are injective functions $f_1 : s_1 \to s_0$ and $f_2 : s_2 \to s_0$ with $A(id_s + f_1)a_1 = A(id_s + f_2)a_2$ in $A(s + s_0)$. We write $[s', a']$ to represent the equivalence class of $\langle s', a'\rangle$. This element is the computation 'create the new names s' and return value a''', and quotienting by the relation '$\sim$' ensures that the (DROP) and (SWAP) rules for names hold true.

The remaining parts of the monad are as follows. If $f : s \to s''$ in $\mathcal{I}$ then the map $TAf : TAs \to TAs''$ is

$$TAf[s', a'] \;=\; [s', A(f + id_{s'})a'] \qquad a' \in A(s + s').$$

If $p : A \to B$ is a morphism in $Set^{\mathcal{I}}$, then $Tp : TA \to TB$ is the natural transformation with maps $Tps : TAs \to TBs$ given by

$$Tps[s', a'] \;=\; [s, p(s + s')a'].$$

The unit of the monad $\eta_A : A \to TA$ has components $\eta_A s : As \to TAs$ for each $s \in \mathcal{I}$ given by

$$\eta_A sa = [0, a] \qquad a \in As.$$

A morphism $q : A \times B \to TC$ has lift $q^* : A \times TB \to TC$ whose component maps $q^*s : As \times TBs \to TCs$ are

$$q^*s\langle a, [s', b']\rangle \;=\; [s' + s'', c''] \qquad b' \in B(s + s')$$

where

$$[s'', c''] \;=\; q(s + s')\langle A(inl_{s,s'})a, b'\rangle \qquad c'' \in C(s + s' + s'').$$

Finally, the natural transformation $\mu : T^2 \to T$ and strength maps $t_{A,B} : A \times TB \to T(A \times B)$ are described by

$$\begin{array}{rcll} \mu_A s[s', [s'', a'']] & = & [s' + s'', a''] & a'' \in A(s + s' + s'') \\ t_{A,B}s\langle a, [s', b']\rangle & = & [s', \langle A(inl_{s,s'})a, b'\rangle]. & \end{array}$$

These are all well defined regardless of choice of representative, and satisfy the appropriate equalities. In addition the monad T is both affine and commutative.

We take the object of names N to be the inclusion functor $\mathcal{I} \hookrightarrow Set$. The morphism $eq : N \times N \to 1 + 1$ is simple equality at all stages. New names are generated by

$$new\, s \;=\; [1, inr_{s,1}] \in TNs$$

which satisfies the necessary equations.

Thus the category $Set^{\mathcal{I}}$ fulfils all the conditions of the previous section, and the interpretation described there gives morphisms:

$$\begin{array}{rl} [\![M]\!]_{\neq s} : (\neq s) \to T[\![\sigma]\!] & M \in \mathrm{Exp}_\sigma(s) \\ |C|_{\neq s} : (\neq s) \to [\![\sigma]\!] & C \in \mathrm{Can}_\sigma(s) \end{array}$$

It happens that the object $(\neq s)$ in $Set^{\mathcal{I}}$ is isomorphic to $\mathcal{I}(s, -)$, and we may apply the

Yoneda Lemma to obtain elements:

$$[\![M]\!]_{\neq s} \in T[\![\sigma]\!]s \qquad \text{and} \qquad |C|_{\neq s} \in [\![\sigma]\!]s.$$

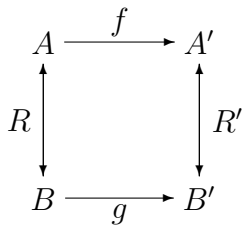
These are generally easier to work with, and the results of Section 4.2 still hold when stated in terms of elements rather than morphisms. Regarding the examples of Section 2.3, this model confirms the basic equivalences (2)–(5), but not (8) or (9) which use private names.

6 A Model using Logical Relations

The papers [27] and [28] describe an operational method for proving observational equivalence using binary relations similar to logical relations. We now construct a denotational equivalent, using a notion of categories with relations based on that in [24]. This model is more abstract than the one in $Set^{\mathcal{I}}$, and is fully abstract for ground and first order types.

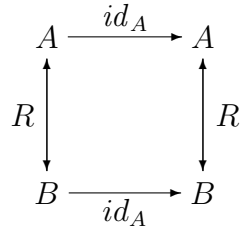
6.1 Categories with Relations

A *category with relations* [24] is a category with certain additional structure. As well as objects and morphisms, it has a collection of binary *relations* between pairs of objects, represented $R : A \leftrightarrow B$, and *parametric squares* of the form



where R, R' are relations and f, g are morphisms. Relations, like morphisms, are simply abstract data; they need not stand for set-based relations, though that is obviously the motivating example.

Parametric squares should compose horizontally with identity



and there should be a distinguished identity relation $id_A : A \leftrightarrow A$ for each object.

Suppose that $\mathcal{C}$ and $\mathcal{D}$ are categories with relations. A *parametric functor* $F : \mathcal{C} \to \mathcal{D}$ is a functor on the underlying categories together with a map on relations. If $R : A \leftrightarrow B$ is a relation in $\mathcal{C}$ then $FR : FA \leftrightarrow FB$ should be a relation in $\mathcal{D}$, with $Fid_A = id_{FA}$, and F must take parametric squares to parametric squares.

Suppose that $F, G : \mathcal{C} \to \mathcal{D}$ are two parametric functors. A *parametric natural transformation* $t : F \to G$ is a natural transformation on the underlying categories such that for any relation $R : A \leftrightarrow B$ of $\mathcal{C}$ the square

$$\begin{array}{ccc} FA & \xrightarrow{tA} & GA \\ {\scriptstyle FR}\updownarrow & & \updownarrow{\scriptstyle GR} \\ FB & \xrightarrow[tB]{} & GB \end{array}$$

is parametric.

6.2 The Parametric Functor Category $\mathcal{P}$

We can now rebuild the model $Set^{\mathcal{I}}$ in a parametric setting. We take the index category $\mathcal{I}$ of finite sets and injections as before. A relation $R : s_1 \leftrightarrow s_2$ on $\mathcal{I}$ consists of a finite set R and a pair of injections $s_1 \leftarrow R \rightarrow s_2$. This is sometimes called a *span* or a *partial bijection*. The coproduct '$+$' on $\mathcal{I}$ extends to relations: if $R : s_1 \leftrightarrow s_2$ and $R' : s_1' \leftrightarrow s_2'$ then $R + R' : s_1 + s_1' \leftrightarrow s_2 + s_2'$. A square in $\mathcal{I}$ is parametric

$$\begin{array}{ccc} s_1 & \xrightarrow{f_1} & s_1' \\ {\scriptstyle R}\updownarrow & & \updownarrow{\scriptstyle R'} \\ s_2 & \xrightarrow[f_2]{} & s_2' \end{array}$$

if and only if both squares in

$$\begin{array}{ccc} s_1 & \longrightarrow & s_1' \\ \uparrow & & \uparrow \\ R & \longrightarrow & R' \\ \downarrow & & \downarrow \\ s_2 & \longrightarrow & s_2' \end{array}$$

are pullbacks.

This is a stronger condition than just requiring it to commute as a square of relations. For the ground category we take Set with ordinary binary relations and squares parametric

$$\begin{array}{ccc} A & \xrightarrow{f} & A' \\ {\scriptstyle R}\updownarrow & & \updownarrow{\scriptstyle R'} \\ B & \xrightarrow[g]{} & B' \end{array}$$

if and only if

$$\forall a \in A, b \in B.(a, b) \in R \quad \Longrightarrow \quad (fa, gb) \in R'.$$

We then take the ordinary category $\mathcal{P}$ of parametric functors and parametric natural transformations from $\mathcal{I}$ to Set. As before objects of $\mathcal{I}$ represent stages of computation, and if $A : \mathcal{I} \to Set$ is a parametric functor then elements of As are values defined over the names in s.

The category $\mathcal{P}$ is cartesian closed. Finite limits and colimits are taken pointwise: the object of booleans is the constant parametric functor to the two-element set $1 + 1$, taking all relations to id_{1+1}. Thanks to a relational version of the Yoneda Lemma, exponentials are

defined by

$$\begin{array}{rcll} B^A s & = & Set^{\mathcal{I}}(\mathcal{I}(s,-)\times A, B) & s,s',s''\in\mathcal{I} \\ B^A f p s''\langle i,a''\rangle & = & p s''\langle i\circ f, a''\rangle & f: s\to s' \quad p\in B^A s \\ & & & i: s'\to s'' \quad a''\in As'' \\ (p_1,p_2)\in B^A R & \iff & & s_1,s_2\in\mathcal{I} \quad p_1\in B^A s_1 \\ & \text{for all parametric squares} & & R: s_1\leftrightarrow s_2 \quad p_2\in B^A s_2 \end{array}$$

$$\begin{array}{ccc} s_1 & \xrightarrow{f_1} & s_1' \\ R\updownarrow & & \updownarrow R' \\ s_2 & \xrightarrow[f_2]{} & s_2' \end{array}$$

and elements $a_1'\in As_1'$, $a_2'\in As_2'$ it is true that
$(a_1',a_2')\in AR' \implies (p_1 s_1'\langle f_1,a_1'\rangle, p_2 s_2'\langle f_2,a_2'\rangle)\in BR'$.

As with the model in $Set^{\mathcal{I}}$, there is a simpler form for the object part of the exponential:

$$B^A s = \mathcal{P}(A(s+\_), B(s+\_)).$$

We define the monad explicitly. On objects it is the quotient

$$TAs \;=\; \{\langle s',a'\rangle \mid s'\in\mathcal{I}, a'\in A(s+s')\}/\sim$$

where $\langle s_1,a_1\rangle\sim\langle s_2,a_2\rangle$ if and only if there is some $R: s_1\leftrightarrow s_2$ such that $(a_1,a_2)\in A(id_s+R)$. The relation $\sim$ is not necessarily transitive. We take $[s',a']$ to represent the equivalence class of $\langle s',a'\rangle$; as before, this denotes the computation 'create the new names s' and return value a''.

The remaining details of the monad are specified exactly as for the $Set^{\mathcal{I}}$ model. The only addition is that if $R: s_1\leftrightarrow s_2$ in $\mathcal{I}$ then the relation $TAR: TAs_1\leftrightarrow TAs_2$ is given by

$$\begin{array}{rcl} (e_1,e_2)\in TAR & \iff & \exists R': s_1'\leftrightarrow s_2',\, a_1'\in A(s_1+s_1'), a_2'\in A(s_2+s_2')\,. \\ & & e_1=[s_1',a_1'] \;\&\; e_2=[s_2',a_2'] \;\&\; (a_1',a_2')\in A(R+R') \end{array}$$

where $e_1\in TAs_1$ and $e_2\in TAs_2$. Again the resulting strong monad T is both affine and commutative.

The object of names N is the parametric inclusion functor $\mathcal{I}\hookrightarrow Set$, and $eq: N\times N\to 1+1$ is equality at all stages. Fresh names are produced by

$$new\; s \;=\; [1, inr_{s,1}]\in TNs.$$

As with the model in the functor category $Set^{\mathcal{I}}$, we can interpret closed expressions $M\in \mathrm{Exp}_\sigma(s)$ and $C\in \mathrm{Can}_\sigma(s)$ either by morphisms

$$[\![M]\!]_{\neq s}: (\neq s)\to T[\![\sigma]\!] \qquad \text{and} \qquad |C|_{\neq s}: (\neq s)\to [\![\sigma]\!]$$

or elements

$$[\![M]\!]_{\neq s}\in T[\![\sigma]\!]s \qquad \text{and} \qquad |C|_{\neq s}\in [\![\sigma]\!]s$$

using a version of the Yoneda Lemma adapted for categories with relations.

Although many of the details above are given precisely as for the model in $Set^{\mathcal{I}}$, the relational structure makes an important difference. Function spaces are smaller, and more computations are identified in TAs, with the consequence that more observational equivalences of the nu-calculus can be proved.

6.3 Properties of the model in $\mathcal{P}$

The model of the nu-calculus in the category $\mathcal{P}$ is adequate and validates equivalences (2)–(5) and (8), but not (9), of Section 2.3. It is fully abstract for types up to first order:

Theorem 4 (Full Abstraction at First Order Types) *If σ is a ground or first order type and $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ then:*

$$s \vdash M_1 \approx_\sigma M_2 \quad \Longrightarrow \quad [\![M_1]\!]_{\neq s} = [\![M_2]\!]_{\neq s}.$$

Proof Exactly as for Theorem 22 of [27]. Alternatively, we can show a correspondence between the operational relation of [27] and the relational structure of $\mathcal{P}$, with this result as a corollary. □

So the model in $\mathcal{P}$ can be used to validate all observational equivalences involving types up to first order. As a demonstration we consider example (8) of Section 2.3 which concerns the equivalence

$$\vdash \nu n.\lambda x{:}\nu.(x = n) \approx_{\nu\to o} \lambda x{:}\nu.\mathit{false}.$$

The two expressions are interpreted in $\mathcal{P}$ by

$$\left.\begin{array}{rcl} [\![\nu n.\lambda x{:}\nu.(x = n)]\!] & = & [\{n\}, \lambda x{:}N.[\mathit{eq}(x, n)]] \\ [\![\lambda x{:}\nu.\mathit{false}]\!] & = & [\emptyset, \lambda x{:}N.[\mathit{false}]] \end{array}\right\} \in T(N \to T(1+1))\emptyset.$$

If $R : \{n\} \leftrightarrow \emptyset$ is the empty relation then

$$(\lambda x{:}N.[\mathit{eq}(x, n)], \lambda x{:}N.[\mathit{false}]) \in (N \to T(1+1))R$$

and so

$$[\![\nu n.\lambda x{:}\nu.(x = n)]\!] = [\![\lambda x{:}\nu.\mathit{false}]\!] \in T(N \to T(1+1))\emptyset$$

which gives the desired observational equivalence.

7 Related Work

There has been considerable study of 'Algol-like' languages which combine local variables and block structure with higher-order functions in a call-by-name semantics. Meyer and Sieber [14] describe a selection of the difficulties that arise. These differ from the nu-calculus examples of Section 2.3, chiefly because of the contrast between call-by-value and call-by-name styles of parameter passing. For example, a call-by-name nu-calculus would

only need name abstraction at ground types. However, some of the techniques used do carry across: in particular, the use of functor categories comes from the 'possible worlds' model originated by Reynolds [31] and followed up in [25], [23] and [10]. Categories with relations were used for Algol-like languages by O'Hearn and Tennent [24], and Sieber's (non-categorical) models, in [32] and most recently [33], also use relations.

Mason and Talcott have developed operational methods for reasoning about LISP programs, in [12], [13] and, with Honsell and Smith, in [6]. They consider an untyped language with call-by-value semantics and dynamically generated mutable cells. There are substantial example proofs of program equivalences in [11], though the techniques described are generally restricted to the language without higher order functions. Mason's notion of *strong isomorphism* compares with equational reasoning in the metalanguage of Section 3 above.

Felleisen and others have added variable assignment to the call-by-value lambda-calculus, in [3] and [4]. They present a syntactic, equational theory for the lambda-calculus, and show that it can be extended with certain axioms for reasoning about state.

Odersky has developed a theory $\lambda\nu$ that extends the lambda-calculus with a binding construct for local names [20], and proves that the theory of observable equivalence in $\lambda\nu$ is a conservative extension of that for the lambda-calculus. Syntactically this language appears similar to the nu-calculus; differences are that $\lambda\nu$ is untyped and has a call-by-name reduction strategy, with the possibility of 'stuck' terms. So, taking example (6) of Section 2.3, in $\lambda\nu$ the expression $(\lambda x.x == x)(\nu n.n)$ reduces first to $\nu n.n == \nu n.n$ and is then stuck; the equivalent nu-calculus expression evaluates to $true$. Odersky works around the limited scope of names by using a continuation-passing style of programming. Despite these differences, it seems likely that a typed version of $\lambda\nu$ could be interpreted in the metalanguage of Section 3, using $[\![\sigma \to \sigma']\!] = T[\![\sigma]\!] \to T[\![\sigma']\!]$ to capture the call-by-name behaviour at function types.

8 Conclusions and Further Work

Categorical monads can be used to construct sound and adequate models for the nu-calculus, a language that extends the simply-typed lambda-calculus with dynamically generated names. The use of a computational metalanguage as an intermediary helps to separate the general mechanism of constructing a model from the particular difficulties of the nu-calculus.

The model described in Section 5 provides a denotational semantics for the nu-calculus and validates reasoning in the metalanguage. The category $\mathcal{P}$ of Section 6.2, using categories with relations, gives a more abstract denotational model.

A fully-abstract denotational semantics for the nu-calculus still seems a reasonable aim; after all, there are no difficulties with non-termination. O'Hearn and Riecke [22] have been able to use logical relations of varying arity to build a fully-abstract model of PCF, and it may be possible to extend this to the nu-calculus.

A related issue is whether the theory of the metalanguage can be made complete for reasoning about observational equivalence. One route to this would be the construction of a categorical model from the syntax of the nu-calculus itself, in the style of [15]. However there are difficulties, most notably the need for equalizers in the category, and the matter is

not yet settled.

The requirements of Section 4, for a category to model the nu-calculus, say nothing about which features are forced by the others. For example, it may be that given a category $\mathcal{C}$ with a strong monad T, the object of names N and morphism new can be characterised by some universal property. This would be similar to a natural numbers object for a category, which if present is unique up to isomorphism [9]. No such description is known yet.

The definition of the monad T for the parametric functor category $\mathcal{P}$ of Section 6.2 is strikingly similar to the proof rule of [30] for existential types in System F. Indeed they do have a common origin in Reynolds' notion of 'relational parametricity', but a closer connection is given by a direct interpretation of the nu-calculus in System F. Details will appear elsewhere; the method is similar to the translation into a computational metalanguage, with certain existential types replacing the constructor T. This is independent of the work of O'Hearn and Riecke [21], who use polymorphism to interpret similar behaviour in Algol-like languages; nevertheless, it seems likely that some correspondence can be made.

Another direction for future work is to extend the nu-calculus to a 'store calculus' allowing the dynamic creation of typed storage cells. This would more fully describe the use of references in Standard ML. The same scheme of a computational metalanguage might be used, and the existing categorical models could be suitably enhanced.

Acknowledgements

Andy Pitts and Eugenio Moggi gave much advice on the methods used here, and Bob Tennent explained how they relate to his models of Algol-like languages. The anonymous referees prompted several adjustments to make the explanations clearer.

References

1. S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.
2. H.-J. Boehm. *A Logic for the Russell Programming Language*. PhD thesis, Cornell University, Ithaca, New York, February 1984. Also published as Technical Report 84-593.
3. M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.
4. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
5. A. D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, August 1992. Also published as Technical Report 285, University of Cambridge Computer Laboratory.
6. F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. Submitted to *Information and Computation*, 1993.
7. A. Kock. Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, XXI:1–10, 1970.
8. A. Kock. Bilinearity and cartesian closed monads. *Mathematica Scandinavica*, 29:161–174, 1971.
9. J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics 7. Cambridge University Press, 1986.

10. A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. In *SIPL '93* [34], pages 101–119.
11. I. A. Mason. *The Semantics of Destructive Lisp*. PhD thesis, Stanford University, 1986. Also published as CSLI Lecture Notes Number 5, Center for the Study of Language and Information, Stanford University.
12. I. A. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):297–327, July 1991.
13. I. A. Mason and C. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105:167–215, 1992.
14. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203. ACM Press, 1988.
15. R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT press, 1990.
17. E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, 1989.
18. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1990.
19. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
20. M. Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Yale University, May 1993.
21. P. W. O'Hearn and J. G. Riecke. Fully abstract translations and parametric polymorphism. In *Programming Languages and Systems – ESOP '94*, Lecture Notes in Computer Science 788, pages 454–468. Springer-Verlag, 1994.
22. P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. Submitted to *Information and Computation*, June 1994.
23. P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, London Mathematical Society Lecture Note Series 177, pages 217–238. Cambridge University Press, 1992.
24. P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables (preliminary report). In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 171–184. ACM Press, 1993.
25. F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, 1985.
26. A. M. Pitts. Evaluation logic. In *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer-Verlag, 1991. Also published as Technical Report 198, University of Cambridge Computer Laboratory.
27. A. M. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 711, pages 122–141. Springer-Verlag, 1993.
28. A. M. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *SIPL '93* [34], pages 31–45.
29. G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
30. G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 664, pages 361–375. Springer-Verlag, 1993.
31. J. C. Reynolds. The essence of Algol. In *Algorithmic Languages*, pages 345–372. North Holland, Amsterdam, 1981.
32. K. Sieber. New steps towards full abstraction for local variables. In *SIPL '93* [34], pages 88–100.

33. K. Sieber. Full abstraction for the second order subset of an Algol-like language (preliminary report). Technical Report A 01/94, Universität des Saarlandes, Saarbrücken, January 1994.
34. *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark, June 12, 1993*. Research Report YALEU/DCS/RR-968, Yale University Department of Computer Science, 1993.