A reformatted version of this article appears in:
Fundamenta Informaticae 33 (1998) 369–396. IOS Press

Names, Equations, Relations: Practical Ways to Reason about *new*

Ian Stark
Department of Computer Science
The University of Edinburgh
Scotland
Ian.Stark@ed.ac.uk

Abstract. The nu-calculus of Pitts and Stark is a typed lambda-calculus, extended with state in the form of dynamically-generated *names*. These names can be created locally, passed around, and compared with one another. Through the interaction between names and functions, the language can capture notions of scope, visibility and sharing. Originally motivated by the study of references in Standard ML, the nu-calculus has connections to local declarations in general; to the mobile processes of the π-calculus; and to security protocols in the spi-calculus.

This paper introduces a logic of equations and relations which allows one to reason about expressions of the nu-calculus: this uses a simple representation of the private and public scope of names, and allows straightforward proofs of contextual equivalence (also known as observational, or observable, equivalence). The logic is based on earlier operational techniques, providing the same power but in a much more accessible form. In particular it allows intuitive and direct proofs of all contextual equivalences between first-order functions with local names.

Keywords: Names, nu-calculus, generativity; contextual equivalence, equational reasoning, operational reasoning, logical relations.

1. Introduction

Many convenient features of programming languages today involve some notion of *generativity*: the idea that an entity may be freshly created, distinct from all others. This is clearly central to object-oriented programming, with the dynamic creation of new objects as instances of a class, and the issue of object identity. In the study of concurrency, the π-calculus [20] uses dynamically-generated names to describe the behaviour of mobile processes, whose communication topology may change over time. The *spi-calculus* of Abadi and Gordon [1] uses generative names to model cryptographic keys in the verification of security protocols. In functional programming, the language Standard ML [21] extends typed lambda-calculus with a number of features, of which mutable reference cells, exceptions and user-declared datatypes are all generative; so are the structures and functors of the module system. More broadly, the concept of lexical scope rests on the idea that local identifiers should always be treated as fresh, distinct from any already declared.

Such dynamic creation occurs at a variety of levels, from the run-time behaviour of Lisp's *gensym* to resolving questions of scope during program linking. Generally, the intention is that its use should be intuitive or even transparent to the programmer. Nevertheless, for correct implementation and sound design it is essential to develop an appropriate abstract understanding of what it means to be *new*.

The *nu-calculus* was devised to explore this common property of generativity, by adding *names* to the simply-typed lambda-calculus. Names may be created locally, passed around, and compared with one another, but that is all. The language is reviewed in Section 2; a full description is given by Pitts and Stark in [28, 29], with its operational and denotational semantics studied at some length in [35, 36]. Central to the nu-calculus is the use of *name abstraction*: the expression $\nu n.M$ represents the creation of a fresh name, which is then bound to n within the body of M. So, for example, the expression

$$\nu n.\nu n'.(n = n')$$

generates two new names, bound to n and n', and compares them, finally returning the answer *false*. Functions may have local names that remain private and persist from one use of the function to the next; alternatively, names may be passed out of their original scope and can even outlive their creator. It is precisely this mobility of names that allows the nu-calculus to model issues of locality, privacy and non-interference.

Two expressions of the nu-calculus are *contextually equivalent*[1] if they can be freely exchanged in any program: there is no way in the language itself to distinguish them. Contextual equivalence is an excellent property in principle, but in practice often hard to work with because of the need to consider all possible programs. As a consequence a number of authors have made considerable effort, in various language settings, to develop convenient methods for demonstrating contextual equivalence.

Milner's context lemma [19], Gordon's 'experiments' [9], and the 'ciu' theorems of Mason and Talcott [15, 37], provide one such approach. These show that instead of all program contexts, it is sufficient to consider only those in some particular form. For the nu-calculus, a suitable context lemma is indeed available [35, §2.6] and states that one need only consider so-called 'argument contexts'. However even this reduced collection of contexts is still inconveniently large, a problem arising from the imperative nature of name creation.

Alternatively, one can look for relations that imply contextual equivalence but are easier to work with. One possibility is to define such relations directly from the operational semantics of the language, as with the *applicative bisimilarity* variously used by Abramsky [2], Howe [13], Gordon [9], and others. Denotational semantics provides another route: if two expressions have equal interpretation in some adequate model, then they are contextually equivalent. For the nu-calculus, such operational methods are developed and refined in [28, 29], while categorical models are presented in [36]. Both approaches are treated at length in [35].

In principle, methods such as these do give techniques for proving contextual equivalences. In practice however, they are often awkward and can require rather detailed mathematical knowledge. The contribution of this paper is to take two existing operational techniques, and extract from them a straightforward logic that allows simple and direct reasoning about contextual equivalence in the nu-calculus.

The first operational technique, *applicative equivalence*, gives rise to an equational logic with assertions of the form

$$s, \Gamma \vdash M_1 =_\sigma M_2 \ .$$

If such an assertion can be proved using the rules of the logic, then it is certain that expressions M_1 and M_2 are contextually equivalent (here s and Γ list the free names and variables respectively). This equational scheme is simple, but not particularly complete: it is good for reasoning in the presence of names, but not so good at reasoning about names themselves.

[1]The same property is variously known as {operational/observational/observable} {equivalence/congruence}.

The technique of *operational logical relations* refines this by considering just how different expressions make use of their local names. The corresponding logic is one of relational reasoning, with assertions of the form

$$\Gamma \vdash M_1 \; R_\sigma \; M_2 \; .$$

Here R is a relation between the free names of M_1 and M_2 that records information on their privacy and visibility. This logic includes the equational one, and is considerably more powerful: it is sufficient to prove all contextual equivalences between expressions of first-order function type.

It is significant that these schemes both build on existing methods; all the proofs of soundness and completeness work by transferring corresponding properties from the earlier operational techniques. For the completeness results in particular this is a considerable saving in proof effort. Such incremental development continues a form of 'technology transfer' from the abstract to the concrete: these same operational techniques were in turn guided by a denotational semantics for the nu-calculus based on categorical monads.

The layout of the paper is as follows: Section 2 reviews the nu-calculus and gives some representative examples of contextual equivalence; Section 3 describes the techniques of applicative equivalence and operational logical relations; Section 4 explains the new logic for equational reasoning; Section 5 extends this to a logic for relational reasoning; and Section 6 concludes.

Related Work

The general issue of adding effects to functional languages has received considerable attention over time, and there is a substantial body of work concerning operational and denotational methods for proving contextual equivalence. A selection of references can be found in [30, 37], for example. However, not so many practical systems have emerged for reasoning about expressions and proving actual examples of contextual equivalence.

Felleisen and Hieb [6] present a calculus for equational reasoning about state and control features. This extends β_v-interconvertibility and is similar to the equational reasoning of this paper, in that it is correct and convenient for proving contextual equivalence, but not particularly complete.

Mason and Talcott's logic for reasoning about destructive update in Lisp [16] is again similar in power to our equational reasoning. Moreover, our underlying operational notion of applicative equivalence corresponds quite closely to Mason's 'strong isomorphism' [14]. Further work [17] adds some particular reasoning principles that resemble aspects of our relational reasoning, but can only be applied to first-order functions; by contrast, our techniques remain valid at all higher function types. In a similar vein, the 'variable typed logic of effects' (VTLoE) of Honsell, Mason, Smith and Talcott [12] is an operationally-based scheme for proving certain assertions about functions with state.

The 'computational metalanguage' of Moggi [22] provides a general method for equational reasoning about additions to functional languages. Its application to the nu-calculus is discussed in [35, §3.3], where it is shown to correspond closely to applicative equivalence. Related to this is 'evaluation logic', a variety of modal logic that can express the possibility or certainty of certain computational effects [23, 27]. Moggi has shown how a variety of program logics, including VTLoE, can be expressed within evaluation logic [24].

Although the nu-calculus may appear simpler than the languages considered in the work cited, the notion of generativity it highlights is still of real significance. Moreover, the relational logic presented here goes beyond all of the above in the variety of contextual equivalences it can prove: we properly capture the subtle interaction between local declarations and higher-order functions.

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

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

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

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

2. The Nu-Calculus

A full description of the nu-calculus can be found in [35, 36]; this section gives just a brief overview. The language is based on the simply-typed lambda-calculus, with a hierarchy of function types $\sigma \to \sigma'$ built over ground types o of *booleans* and ν of *names*. Expressions have the form

$$\begin{aligned} M \ ::= \ & x \mid n \mid \mathit{true} \mid \mathit{false} \mid \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M \\ \mid \ & M = M \mid \nu n.M \mid \lambda x{:}\sigma.M \mid MM\ . \end{aligned}$$

Here x and n are typed variables and names respectively, taken from separate infinite supplies. The expression '$M = M'$' tests for equality between two names. Name abstraction $\nu n.M$ creates a fresh name bound to n within the body M; during evaluation, names may outlive their creator and escape from their original scope. We implicitly identify expressions which only differ in their choice of bound variables and names (α-conversion). A useful abbreviation is *new* for $\nu n.n$; this is the expression that generates a new name and then immediately returns it.

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

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

says that in the presence of s and 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, here in $s \oplus \{n\}$ and $\Gamma \oplus \{x : \sigma\}$. We may omit Γ when it is empty.

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. We define the sets

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

of expressions and canonical expressions, open and closed.

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

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

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

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

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

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

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

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

(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

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 is intended to mean that in the presence of the names s, expression M of type σ evaluates to canonical form C and creates fresh names s'. We may omit s or s' when they are empty.

Evaluation is chosen to be left-to-right and call-by-value, after Standard ML; it can also be shown to be deterministic and terminating [35, Theorem 2.4]. In addition to this 'big step' semantics, there is an equivalent 'small step' version that specifies a reduction relation $M \rightarrow_\sigma M'$. As usual this factors into *redexes* within *reduction contexts*, which highlight the detailed progress of nu-calculus computation [35, §2.3].

As an example of evaluation, consider the judgement

$$\vdash (\lambda x{:}\nu.(x = x))(\nu n.n) \Downarrow_o (n)\mathit{true} \ .$$

First the argument $\nu n.n$ (or *new*) is evaluated, returning a fresh name bound to n. This is in turn bound to the variable x, and the body of the function compares this name to itself, giving the result *true*. Compare this with

$$\vdash (\nu n'.\lambda x{:}\nu.(x = n'))(\nu n.n) \Downarrow_o (n', n)\mathit{false} \ .$$

Here the evaluation of the function itself creates a fresh name, bound to n'; the argument provides another fresh name, and the comparison then returns *false*.

Notice that for the rule (LOCAL) to evaluate a name abstraction $\nu n.M$, the identifier n must not occur elsewhere ($n \notin (s \oplus s_1)$). We can always ensure this through α-conversion, in the same way that we avoid variable capture during substitution $M[C/x]$.

Repeated evaluation of a name abstraction will give different fresh names. Thus the two expressions

$$\nu n.\lambda x{:}o.n \qquad \text{and} \qquad \lambda x{:}o.\nu n.n$$

behave differently: the first evaluates to the function $\lambda x{:}o.n$, with every subsequent application returning the private name bound to n; while the second gives a different fresh name as result each time it is applied. The expressions are distinguished by the program

$$(\lambda f : o \rightarrow \nu \, . \, (f\mathit{true} = f\mathit{true})) \, \langle\!\langle - \rangle\!\rangle$$

which evaluates to *true* or *false* according to how the hole $\langle\!\langle - \rangle\!\rangle$ is filled.

This leads us to the notion of *program context.* A formal definition is given in [35, §2.4]; here we simply note that the form $P\langle\!\langle - \rangle\!\rangle$ represents a program P with some number of holes $\langle\!\langle - \rangle\!\rangle$, and in $P\langle\!\langle (\vec{x})M \rangle\!\rangle$ these are filled by an expression M whose free variables are in the list $\vec{x}$. There is an arrangement to capture these free variables, and the completed program is a closed expression of boolean type.

Definition 2.1. (Contextual Equivalence) If $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ then we say that they are *contextually equivalent*, written

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

if for all closing program contexts $P\langle\!\langle - \rangle\!\rangle$ and boolean values $b \in \{\mathit{true}, \mathit{false}\}$,

$$(\, \exists s_1 \, . \, s \vdash P\langle\!\langle (\vec{x})M_1 \rangle\!\rangle \Downarrow_o (s_1)b \,) \quad \iff \quad (\, \exists s_2 \, . \, s \vdash P\langle\!\langle (\vec{x})M_2 \rangle\!\rangle \Downarrow_o (s_2)b \,).$$

That is, $P\langle\!\langle - \rangle\!\rangle$ always evaluates to the same boolean value, whether the hole is filled by M_1 or M_2. If both s and Γ are empty then we write simply $M_1 \approx_\sigma M_2$.

This is in many ways the right and proper notion of equivalence between nu-calculus expressions. For example to check code transformations, replace algorithms, or match specification to implementation, contextual equivalence is the benchmark for correctness. However the quantification over all programs makes it inconvenient to demonstrate directly; as discussed in the introduction, the purpose of this paper is to present simple methods for reasoning about contextual equivalence without the need to consider contexts or even evaluation.

Examples.

Up to contextual equivalence unused names are irrelevant, as is the order in which names are generated:

$$s, \Gamma \vdash \quad \nu n.M \approx_\sigma M \qquad\qquad n \notin \mathit{fn}(M) \tag{1}$$

$$s, \Gamma \vdash \nu n.\nu n'.M \approx_\sigma \nu n'.\nu n.M\ . \tag{2}$$

Evaluation respects contextual equivalence:

$$s \vdash M \Downarrow_\sigma (s')C \quad \Longrightarrow \quad s \vdash M \approx_\sigma \nu s'.C \tag{3}$$

where $\nu s'.C$ abbreviates multiple name abstractions. A variety of equivalences familiar from the call-by-value lambda-calculus also hold. For instance Plotkin's β_v-rule [31]: if $C \in \mathrm{Can}_\sigma(s, \Gamma)$ and $M \in \mathrm{Exp}_{\sigma'}(s, \Gamma \oplus \{x : \sigma\})$ then

$$s, \Gamma \vdash (\lambda x{:}\sigma.M)C \approx_{\sigma'} M[C/x]. \tag{4}$$

Names can be used to detect that general β-equivalence fails, as with

$$(\lambda x{:}\nu.x = x)\mathit{new} \not\approx_o (\mathit{new} = \mathit{new}) \tag{5}$$

which evaluate to *true* and *false* respectively. More interestingly, distinct expressions may be contextually equivalent if they differ only in their use of 'private' names:

$$\nu n.\lambda x{:}\nu.(x = n) \approx_{\nu \to o} \lambda x{:}\nu.\mathit{false}\ . \tag{6}$$

Here the right-hand expression is the function that always returns false; while the left-hand expression evaluates to a function with a persistent local name n, that it compares against any name supplied as an argument. Although these function bodies are quite different, no external context can supply the private name bound to n that would distinguish between them; hence the original expressions are in fact contextually equivalent.

A range of further examples can be found in earlier work on the nu-calculus [28, 29, 35, 36]. Among other things, these show how expressions of higher type can capture finer graduations of privacy and sharing than that in (6). Analysing such behaviour also demands more complex testing contexts: for example, the function

$$\nu n.\nu n'.\lambda x{:}\nu.(\mathit{if}\ x = n'\ \mathit{then}\ n\ \mathit{else}\ n')$$

must be applied at least twice to extract all the names within it.

Taking this further still, there is a representation of the natural numbers as functions from names to names: a set of expressions $\{F_p : \nu \to \nu \mid p \in \mathbb{N}\}$ where each F_p cycles through $(p + 1)$ local names [35, §2.5]. This even supports an addition operation A with $AF_pF_q \approx F_{p+q}$. To show however that $(p \neq q \Rightarrow F_p \not\approx F_q)$ requires a context that applies the functions at least $\min(p, q)$ times, passing the result of each application in again as the argument of the next. This is in sharp contrast to PCF or other pure simply-typed lambda-calculi, where Milner's context lemma proves that to distinguish two terms of type $\sigma \to \sigma'$ it is only necessary to apply them once, to some term of the structurally simpler type σ [19].

3. Operational Reasoning

This section describes two operational techniques for demonstrating contextual equivalences in the nu-calculus. *Applicative equivalence* captures much of the general behaviour of higher-order functions and their evaluation, while the more sophisticated *operational logical relations* highlight the particular properties of name privacy and visibility. Both are discussed in more detail in [28] and [35], which also give proofs of the results below.

Definition 3.1. (Applicative Equivalence) We define two relations, one between canonical forms and another for general expressions:

$$\begin{array}{ll} s \vdash C_1 \sim^{can}_{\sigma} C_2 & \text{for } C_1, C_2 \in \mathrm{Can}_\sigma(s), \text{ and} \\ s \vdash M_1 \sim^{exp}_{\sigma} M_2 & \text{for } M_1, M_2 \in \mathrm{Exp}_\sigma(s). \end{array}$$

These are given together by induction over the structure of the type σ, according to:

$$\begin{array}{rcl} s \vdash b_1 \sim^{can}_{o} b_2 & \iff & b_1 = b_2 \\ s \vdash n_1 \sim^{can}_{\nu} n_2 & \iff & n_1 = n_2 \\ s \vdash \lambda x{:}\sigma.M_1 \sim^{can}_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2 & \iff & \forall s',\, C \in \mathrm{Can}_\sigma(s \oplus s') \;. \\ & & \qquad s \oplus s' \vdash M_1[C/x] \sim^{exp}_{\sigma'} M_2[C/x] \end{array}$$

$$\begin{array}{rcl} s \vdash M_1 \sim^{exp}_{\sigma} M_2 & \iff & \exists s_1, s_2,\, C_1 \in \mathrm{Can}_\sigma(s \oplus s_1),\, C_2 \in \mathrm{Can}_\sigma(s \oplus s_2) \;. \\ & & \quad s \vdash M_1 \Downarrow_\sigma (s_1)C_1 \;\;\&\;\; s \vdash M_2 \Downarrow_\sigma (s_2)C_2 \\ & & \quad \& \;\; s \oplus (s_1 \cup s_2) \vdash C_1 \sim^{can}_{\sigma} C_2. \end{array}$$

The intuition behind this definition is as follows.

- Functions are equivalent if they give equivalent results at all possible arguments. This includes ones that use additional fresh names, hence the use of $C \in \mathrm{Can}_\sigma(s \oplus s')$.
- Expressions in general are equivalent if they evaluate to equivalent canonical forms. The use of $(s_1 \cup s_2)$ on the last line means that unused fresh names are discounted ('garbage collection').

It is immediate that $\sim^{exp}_{\sigma}$ coincides with $\sim^{can}_{\sigma}$ on canonical forms; we write them indiscriminately as $\sim_\sigma$ and call the relation *applicative equivalence.*[2] We can extend the relation to open expressions: if $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ then we define

$$\begin{array}{rcl} s, \Gamma \vdash M_1 \sim_\sigma M_2 & \iff & \forall s',\, C_i \in \mathrm{Can}_{\sigma_i}(s \oplus s') \quad i = 1, \ldots, n \;. \\ & & \qquad s \oplus s' \vdash M_1[\vec{C}/\vec{x}] \sim_\sigma M_2[\vec{C}/\vec{x}] \end{array}$$

where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. This says that the open expressions M_1 and M_2 are applicative equivalent if replacing their variables by any closed canonical forms, possibly ones that use some extra names s', gives applicative equivalent closed expressions $M_1[\vec{C}/\vec{x}]$ and $M_2[\vec{C}/\vec{x}]$.

Applicative equivalence is based on similar 'bisimulation' relations of Abramsky [2] and Howe [13] for untyped lambda-calculus, and Gordon [10] for typed lambda-calculus. It is well behaved and suffices to prove contextual equivalence:

Theorem 3.1. Applicative equivalence is an equivalence relation and a congruence; it therefore implies contextual equivalence:

$$s, \Gamma \vdash M_1 \sim_\sigma M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \approx_\sigma M_2.$$

[2]This is a different relation to the applicative equivalence of [28, Def. 13] and [29, Def. 3.4] which (rather unfortunately) turns out not to be an equivalence at all.

The proof of the theorem centres on the demonstration that applicative equivalence is a congruence, *i.e.* it is preserved by all the rules for forming expressions of the nu-calculus. It is well known that a direct proof of this fails; the most popular way around is known as 'Howe's method' [13]. Because the nu-calculus is simply typed though, we can instead use an original and much simpler method that proceeds via an intermediate relation of 'logical equivalence'. The details are in [35, §2.7].

Applicative equivalence verifies examples (1)–(4) above, and numerous others: a range of contextual equivalences familiar from the standard typed lambda-calculus, and all others that make straightforward use of names. What it cannot capture is the notion of privacy that lies behind example (6); where equivalence relies on a particular name remaining secret.

To address the distinction between private and public uses of names, we introduce the idea of a *span* between name sets. A span $R : s_1 \rightleftharpoons s_2$ is an injective partial map from s_1 to s_2; this is equivalent to a pair of injections $s_1 \leftarrowtail R \rightarrowtail s_2$, or a relation such that

$$(n_1, n_2) \in R \;\;\&\;\; (n'_1, n'_2) \in R \quad\Longrightarrow\quad (n_1 = n'_1) \Leftrightarrow (n_2 = n'_2)$$

for $n_1, n'_1 \in s_1$ and $n_2, n'_2 \in s_2$. The idea is that for any span R the bijection between $\mathrm{dom}(R) \subseteq s_1$ and $\mathrm{cod}(R) \subseteq s_2$ represents matching use of 'visible' names, while the remaining elements not in the graph of R are 'unseen' names. The identity relation $id_s : s \rightleftharpoons s$ is clearly a span; and if $R : s_1 \rightleftharpoons s_2$ and $R' : s'_1 \rightleftharpoons s'_2$ are spans on distinct name sets, then their disjoint union $R \oplus R' : s_1 \oplus s'_1 \rightleftharpoons s_2 \oplus s'_2$ is also a span. Starting from spans, we now build up a collection of relations between expressions of higher types.

Definition 3.2. (Logical Relations) If $R : s_1 \rightleftharpoons s_2$ is a span then we define relations

$$\begin{aligned} R^{can}_\sigma &\subseteq \mathrm{Can}_\sigma(s_1) \times \mathrm{Can}_\sigma(s_2) \\ R^{exp}_\sigma &\subseteq \mathrm{Exp}_\sigma(s_1) \times \mathrm{Exp}_\sigma(s_2) \end{aligned}$$

by induction over the structure of the type σ, according to:

$$b_1 \; R^{can}_o \; b_2 \quad\Longleftrightarrow\quad b_1 = b_2$$

$$n_1 \; R^{can}_\nu \; n_2 \quad\Longleftrightarrow\quad (n_1, n_2) \in R$$

$$\begin{aligned} &(\lambda x{:}\sigma.M_1) \; R^{can}_{\sigma\to\sigma'} \; (\lambda x{:}\sigma.M_2) \quad\Longleftrightarrow \\ &\qquad \forall R' : s'_1 \rightleftharpoons s'_2,\ C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s'_1),\ C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s'_2)\ . \\ &\qquad\quad C_1 \; (R \oplus R')^{can}_\sigma \; C_2 \implies M_1[C_1/x] \; (R \oplus R')^{exp}_{\sigma'} \; M_2[C_2/x] \end{aligned}$$

$$\begin{aligned} &M_1 \; R^{exp}_\sigma \; M_2 \iff \\ &\qquad \exists R' : s'_1 \rightleftharpoons s'_2,\ C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s'_1),\ C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s'_2)\ . \\ &\qquad\quad s_1 \vdash M_1 \Downarrow_\sigma (s'_1)C_1 \;\;\&\;\; s_2 \vdash M_2 \Downarrow_\sigma (s'_2)C_2 \;\;\&\;\; C_1 \; (R \oplus R')^{can}_\sigma \; C_2. \end{aligned}$$

The intuition here differs somewhat from that for applicative equivalence: in general because we now have spans to consider, and at function types in particular because this is a 'logical' rather than an 'applicative' relation.

- Functions are related if they take related arguments to related results. This is the 'logical' aspect. Dynamic name creation requires that we consider arguments using fresh names, hence the extra span $R' : s'_1 \rightleftharpoons s'_2$.
- Expressions are related if some span can be found between their local names that will relate their canonical forms.

The operational relations R^{can}_σ and R^{exp}_σ coincide on canonical forms, and we may write them as R^{opn}_σ indiscriminately. We can extend the relations to open expressions: for any

$M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$ define

$$\begin{aligned} \Gamma \vdash M_1 \; R^{opn}_\sigma \; M_2 \quad \iff \quad & \forall R' : s_1' \rightleftharpoons s_2', \\ & C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s_i') \quad i = 1,2 \quad j = 1, \dots, n \; . \\ & \quad (\&_{j=1}^{n} \; . \; C_{1j} \; (R \oplus R')^{can}_{\sigma_j} \; C_{2j} \,) \\ & \quad \implies M_1[\vec{C_1}/\vec{x}] \; (R \oplus R')^{exp}_\sigma \; M_2[\vec{C_2}/\vec{x}] \end{aligned}$$

where $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$. Again the 'logical' form directs this definition: related open expressions are those that give related closed expressions under every instantiation of free variables by related canonical forms.

Overall, the intuition is that if $\Gamma \vdash M_1 \; R^{opn}_\sigma \; M_2$ for some $R : s_1 \rightleftharpoons s_2$ then the names in s_1 and s_2 related by R are public and must be treated similarly by M_1 and M_2, while those names not mentioned in R are private and must remain so. When R is the identity relation $id_s : s \rightleftharpoons s$ then all names are public and we can compare logical relations to contextual equivalence.

Theorem 3.2. For any expressions $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$:

$$\Gamma \vdash M_1 \; (id_s)^{opn}_\sigma \; M_2 \quad \implies \quad s, \Gamma \vdash M_1 \approx_\sigma M_2. \tag{7}$$

If σ is a ground or first-order type of the nu-calculus and Γ is a set of variables of ground type, then the converse also holds:

$$s, \Gamma \vdash M_1 \approx_\sigma M_2 \quad \implies \quad \Gamma \vdash M_1 \; (id_s)^{opn}_\sigma \; M_2. \tag{8}$$

Implication (7) says that logical relations are a *sound* method for proving contextual equivalence, while implication (8) says that they are also *complete* to first order.

The first step in the proof of soundness is to show that logical relations are preserved by all the operations of the nu-calculus. For applicative equivalence this was somewhat delicate: here the 'logical' quality makes it a fairly straightforward rule induction. Moving from this congruence property to soundness itself requires a mild generalisation of contextual equivalence to *contextual R-relations* R^{cxt}_σ, where in particular $(id_s)^{cxt}_\sigma = (\approx_\sigma)$. See [35, §4.1] for details.

Showing first-order completeness on the other hand is difficult and requires some ingenuity. The proof of implication (8) is set out in [35, §4.2]: its essence is that we must exhibit enough testing contexts to show that contextual equivalence is at least as discriminating as the logical relations.

The following proposition collects various useful results about logical relations. The main proof method is rule induction following the structure of Definition 3.2.

Proposition 3.1.

1. Logical relations are preserved by all the rules for forming expressions of the nu-calculus.
2. The identity logical relation is reflexive: $\Gamma \vdash M \; (id_s)_\sigma \; M$. This is the appropriate *Fundamental Theorem* for these logical relations.
3. There is a certain transitivity between applicative equivalence and the logical relations: for any span $R : s_1 \rightleftharpoons s_2$,

   $$s_1, \Gamma \vdash M_1 \sim_\sigma M_2 \;\; \& \;\; \Gamma \vdash M_2 \; R^{opn}_\sigma \; M_3 \;\; \& \;\; s_2, \Gamma \vdash M_3 \sim_\sigma M_4 \implies \Gamma \vdash M_1 \; R^{opn}_\sigma \; M_4.$$

4. Logical relations subsume applicative equivalence: whenever we have $s, \Gamma \vdash M_1 \sim_\sigma M_2$ then also $\Gamma \vdash M_1 \; (id_s)^{opn}_\sigma \; M_2$.

Thus logical relations can be used to demonstrate contextual equivalence, extending and significantly improving on applicative equivalence. They are not quite sufficient to handle all contextual equivalences (see [35, §4.6]), but they are complete up to first-order functions; in particular they prove every example in Section 2 above, and any others where there is a clear distinction between private and public use of names.

4. Equational Reasoning

Applicative equivalence is generally much simpler to demonstrate than contextual equivalence, and thus it provides a useful proof technique in itself. However, it is still quite fiddly to apply, and at higher types it involves checking that functions agree on an infinite collection of possible arguments. In this section we present an equational logic that is of similar power but much simpler to use in actual proofs.

Assertions in the logic take the form

$$s, \Gamma \vdash M_1 =_\sigma M_2$$

for open expressions $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$. Valid assertions are derived inductively using the rules of Figure 3. To simplify the presentation we use here a notion of non-binding *univalent context* $U\langle - \rangle$, given by

$$\begin{aligned} U\langle - \rangle \;::=\; & \langle - \rangle M \mid F\langle - \rangle \mid N = \langle - \rangle \mid \langle - \rangle = N' \\ \mid\; & \mathit{if}\ \langle - \rangle\ \mathit{then}\ M\ \mathit{else}\ M' \\ \mid\; & \mathit{if}\ B\ \mathit{then}\ \langle - \rangle\ \mathit{else}\ M' \mid \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ \langle - \rangle . \end{aligned}$$

Thus M is always an immediate subterm of $U\langle M \rangle$, though it may not be the first to be evaluated. This abbreviation appears in the rules for congruence, functions and new names.

The first two sets of rules, for equality and congruence, ensure that we have an equivalence relation, closed under all the operations of the nu-calculus. The univalent contexts $U\langle - \rangle$ are simply a convenience; through transitivity we can easily derive familiar congruence rules like this one for application:

$$\frac{s, \Gamma \vdash F_1 =_{\sigma \to \sigma'} F_2 \qquad s, \Gamma \vdash M_1 =_\sigma M_2}{s, \Gamma \vdash F_1 M_1 =_{\sigma'} F_2 M_2} \;.$$

The rules for functions are more delicate as general β and η-equivalences do not hold for a call-by-value system such as the nu-calculus. Even so, the four rules β_v, η_v, β_{id} and β_U given here still allow considerable scope for function manipulation. In particular the β_U-rule lifts $U\langle - \rangle$ contexts through function application; this is a generalisation of Sabry and Felleisen's β_{lift} [33, Fig. 1].

The rules for booleans precisely capture the properties of *true* and *false*, including the fact that they are the only possible values of type o.

The most interesting rules of the logic are those concerned with names and name creation. Two expressions with a free variable of type ν are equal if they are equal after instantiation with any existing name, and with a single representative fresh one. Unused names can be garbage collected, and name abstractions $\nu n.(-)$ may be moved past each other. They can also move through contexts $U\langle - \rangle$, providing that name capture is avoided.

Many of these rules are simplified because evaluation in the nu-calculus always terminates. For a language with divergence we could follow the same general scheme but with some finer distinctions: equality $s, \Gamma \vdash M_1 =_\sigma M_2$ should split into three mutually dependent assertions:

$$\text{order } s, \Gamma \vdash M_1 \leq_\sigma M_2, \qquad \text{convergence } s, \Gamma \vdash M \downarrow \qquad \text{and divergence } s, \Gamma \vdash M \uparrow;$$

while univalent contexts $U\langle - \rangle$ must be distinguished according to whether or not they evaluate their hole $\langle - \rangle$. A guideline here is Riecke's proof system for value PCF [32].

Proposition 4.1. This equational theory respects evaluation:

$$s \vdash M \Downarrow_\sigma (s')C \quad \Longrightarrow \quad s \vdash M =_\sigma \nu s'.C \;.$$

Equality:

$$\frac{}{s,\Gamma \vdash M =_\sigma M} \qquad \frac{s,\Gamma \vdash M_1 =_\sigma M_2}{s,\Gamma \vdash M_2 =_\sigma M_1} \qquad \frac{s,\Gamma \vdash M_1 =_\sigma M_2 \quad s,\Gamma \vdash M_2 =_\sigma M_3}{s,\Gamma \vdash M_1 =_\sigma M_3}$$

Congruence:

$$\frac{s,\Gamma \vdash M_1 =_\sigma M_2}{s,\Gamma \vdash U\langle M_1\rangle =_{\sigma'} U\langle M_2\rangle} \qquad \frac{s,\Gamma \oplus \{x:\sigma\} \vdash M_1 =_{\sigma'} M_2}{s,\Gamma \vdash \lambda x{:}\sigma.M_1 =_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2} \qquad \frac{s \oplus \{n\},\Gamma \vdash M_1 =_\sigma M_2}{s,\Gamma \vdash \nu n.M_1 =_\sigma \nu n.M_2}$$

Functions:

$$\beta_v \qquad \frac{}{s,\Gamma \vdash (\lambda x{:}\sigma.M)C =_{\sigma'} M[C/x]} \qquad C \text{ canonical}$$

$$\eta_v \qquad \frac{}{s,\Gamma \vdash C =_{\sigma\to\sigma'} \lambda x{:}\sigma.Cx} \qquad C \text{ canonical}$$

$$\beta_{id} \qquad \frac{}{s,\Gamma \vdash (\lambda x{:}\sigma.x)M =_\sigma M}$$

$$\beta_U \qquad \frac{}{s,\Gamma \vdash (\lambda x{:}\sigma.U\langle M\rangle)M' =_{\sigma'} U\langle(\lambda x{:}\sigma.M)M'\rangle}\ (x \notin fv(U\langle-\rangle))$$

Booleans:

$$\frac{}{s,\Gamma \vdash (\textit{if true then } M \textit{ else } M') =_\sigma M} \qquad \frac{}{s,\Gamma \vdash (\textit{if false then } M \textit{ else } M') =_\sigma M'}$$

$$\frac{s,\Gamma \vdash M_1[\textit{true}/b] =_\sigma M_2[\textit{true}/b] \qquad s,\Gamma \vdash M_1[\textit{false}/b] =_\sigma M_2[\textit{false}/b]}{s,\Gamma \oplus \{b:o\} \vdash M_1 =_\sigma M_2}$$

Names:

$$\frac{}{s,\Gamma \vdash (n = n) =_o \textit{true}}\ (n \in s) \qquad \frac{}{s,\Gamma \vdash (n = n') =_o \textit{false}}\ (n,n' \in s \text{ distinct})$$

$$\frac{\begin{array}{ll} s,\Gamma \vdash M_1[n/x] =_\sigma M_2[n/x] & \text{each } n \in s \\ s \oplus \{n'\},\Gamma \vdash M_1[n'/x] =_\sigma M_2[n'/x] & \text{some fresh } n' \end{array}}{s,\Gamma \oplus \{x:\nu\} \vdash M_1 =_\sigma M_2}$$

New names:

$$\frac{}{s,\Gamma \vdash M =_\sigma \nu n.M}\ (n \notin fn(M)) \qquad \frac{}{s,\Gamma \vdash \nu n.\nu n'.M =_\sigma \nu n'.\nu n.M}$$

$$\frac{}{s,\Gamma \vdash U\langle \nu n.M\rangle =_\sigma \nu n.U\langle M\rangle}\ (n \notin fn(U\langle-\rangle))$$

Figure 3. Rules for deriving equational assertions.

Proof:
It is not hard to demonstrate, using the equational theory, that every rule for evaluation in Figure 2 preserves the property given. Note that the ambiguity in the multiple name abstraction $\nu s'.C$ is acceptable because the equational logic allows name abstractions to be moved past each other.

The corresponding result for the small-step semantics $M \rightarrow_\sigma M'$ is even simpler to prove: the rules defining $\rightarrow$ are a proper subset of those for $=$ and so

$$M \rightarrow_\sigma M' \quad \Longrightarrow \quad s \vdash M =_\sigma M' \qquad \text{for all } M, M' \in \mathrm{Exp}_\sigma(s)$$

follows immediately.

We now link the equational theory to the operational reasoning methods of Section 3.

Theorem 4.1. (Soundness and Completeness) Equational reasoning can be used to prove applicative equivalence, and hence also contextual equivalence:

$$s, \Gamma \vdash M_1 =_\sigma M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \sim_\sigma M_2 \tag{9}$$
$$s, \Gamma \vdash M_1 =_\sigma M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \approx_\sigma M_2. \tag{10}$$

Moreover, it corresponds exactly to applicative equivalence at first-order types, and to contextual equivalence at ground types:

$$s, \Gamma \vdash M_1 \sim_\sigma M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 =_\sigma M_2 \qquad \sigma \text{ first-order, ground } \Gamma \tag{11}$$
$$s \vdash M_1 \approx_\sigma M_2 \quad \Longrightarrow \quad s \vdash M_1 =_\sigma M_2 \qquad \sigma \in \{o, \nu\}. \tag{12}$$

Proof:
Soundness, implication (9), follows from the fact that every rule of Figure 3 for $=_\sigma$ also holds for $\sim_\sigma$. Theorem 3.1 gives us the rules for equality and congruence; every other rule has to be handled individually by reference to the definition of applicative equivalence. This in turn involves considering how each expression in a rule may evaluate — which is at least helped by the fact that the evaluation relation $s \vdash M \Downarrow_\sigma (s')C$ is both syntax-directed and deterministic. The details are straightforward but long-winded.

From (9) we apply Theorem 3.1, which states that applicative equivalence $\sim_\sigma$ implies contextual equivalence $\approx_\sigma$. This immediately gives result (10), that provable equality entails contextual equivalence.

The completeness results (11) and (12) are a little more involved. For applicative equivalence we follow its definition and work by induction over types, separating canonical forms from general expressions and treating closed expressions before open ones. The order of proof is as follows.

- Base case: closed canonical forms of ground type.

$$s \vdash C_1 \sim_\sigma^{\mathit{can}} C_2 \quad \Longrightarrow \quad s \vdash C_1 =_\sigma C_2 \qquad \sigma \in \{o, \nu\}.$$

  Here C_1 and C_2 are necessarily the same ground constant: *true*, *false* or some name $n \in s$.

- Extension to general expressions. If for some type σ we have

$$s \vdash C_1 \sim_\sigma^{\mathit{can}} C_2 \quad \Longrightarrow \quad s \vdash C_1 =_\sigma C_2 \qquad \text{for all } s, C_1, C_2$$

then the same result holds for general expressions:

$$s \vdash M_1 \sim_\sigma^{\mathit{exp}} M_2 \quad \Longrightarrow \quad s \vdash M_1 =_\sigma M_2 \qquad \text{for all } s, M_1, M_2.$$

  Proving this uses the definition of $\sim_\sigma^{\mathit{exp}}$ from $\sim_\sigma^{\mathit{can}}$, Proposition 4.1 on evaluation, and the equational rules for manipulating new names.

- First-order function types. This is the induction step: assuming the result for $\sim_\sigma^{exp}$ we prove it for $\sim_{o\to\sigma}^{can}$ and $\sim_{\nu\to\sigma}^{can}$. The crucial observation is that the definition of applicative equivalence at these function types provides just the hypotheses needed by the equational rules that introduce free boolean or name variables. For example, suppose that we have

$$s \vdash \lambda b{:}o.M_1 \sim_{o\to\sigma}^{can} \lambda b{:}o.M_2 \ .$$

Expanding the definition of $\sim_{o\to\sigma}^{can}$ gives

$$s \vdash M_1[\mathit{true}/b] \sim_\sigma^{can} M_2[\mathit{true}/b] \quad \& \quad s \vdash M_1[\mathit{false}/b] \sim_\sigma^{can} M_2[\mathit{false}/b]$$

which by the inductive hypothesis entails

$$s \vdash M_1[\mathit{true}/b] =_\sigma M_2[\mathit{true}/b] \quad \& \quad s \vdash M_1[\mathit{false}/b] =_\sigma M_2[\mathit{false}/b] \ .$$

The boolean variable rule supplies

$$s, \{b : o\} \vdash M_1 =_\sigma M_2$$

and congruence provides

$$s \vdash \lambda b{:}o.M_1 =_{o\to\sigma} \lambda b{:}o.M_2$$

as required. The argument for names, $\sim_{\nu\to\sigma}^{can}$, is similar.

- Open expressions. This requires induction on the length of the context Γ. As each variable is of type o or ν, the induction step proceeds exactly as above for first-order functions, without the need to lambda-abstract at the end.

The final result to show is that equational reasoning is complete for proving contextual equivalence at ground types. This follows immediately from Proposition 4.1 on evaluation and the observation that:

- two closed boolean expressions are contextually equivalent if and only if they both evaluate to *true*, or both to *false*; and
- two closed name expressions are contextually equivalent if and only if they both evaluate to the same known name, or both compute some fresh name.

These facts are easily established by the construction of some simple testing contexts, and this completes the proof of Theorem 4.1.

At ground and first-order function types then, equational reasoning is just as powerful as applicative equivalence. At higher types applicative equivalence is in principle stronger; but this advantage is an illusion. In fact the only way to demonstrate it is to use some more sophisticated technique (such as logical relations) to show that expressions with certain properties can never be written in the nu-calculus. In practice, the equational logic is much more direct and convenient for reasoning about higher-order functions.

The sample contextual equivalences (1)–(4) from Section 2 are all confirmed immediately by the equational theory. We expand here on two further examples. First, that full β-reduction can be applied to functions with univalent bodies:

$$\beta_{one} \qquad s, \Gamma \vdash (\lambda x{:}\sigma.U\langle x\rangle)M \approx_{\sigma'} U\langle M\rangle, \tag{13}$$

which we deduce from

$$\begin{aligned} s, \Gamma \vdash (\lambda x{:}\sigma.U\langle x\rangle)M &= U\langle(\lambda x{:}\sigma.x)M\rangle && \text{by } \beta_U \\ &= U\langle M\rangle && \text{by } \beta_{id} \text{ and congruence.} \end{aligned}$$

This extends easily to nested $U\langle-\rangle$ contexts, showing that β-reduction is valid for any function whose bound variable appears just once.

Furthermore, if a function makes no use of its argument at all, then it need not be evaluated:

$$\beta_{zero} \qquad s, \Gamma \vdash (\lambda x{:}\sigma.M)M' \approx_{\sigma'} M \qquad \text{if } x \notin fv(M). \tag{14}$$

In a certain sense then the nu-calculus is free of side-effects. To prove this, we use the univalent context *if true then* M *else* $\langle-\rangle$, which is certain to ignore the contents of its hole. Thus:

$$\begin{aligned} s, \Gamma \vdash (\lambda x{:}\sigma.M)M' &= (\lambda x{:}\sigma.\mathit{if\ true\ then}\ M\ \mathit{else}\ M)M' \\ &= \mathit{if\ true\ then}\ M\ \mathit{else}\ ((\lambda x{:}\sigma.M)M') \qquad \text{by } \beta_U \\ &= M. \end{aligned}$$

Note that both (13) and (14) may include expressions with free variables of any type, and are truly higher-order: it matters not at all what is the order of the final type σ'.

5. Relational Reasoning

The equational logic presented above is fairly simple, and powerful in that it allows correct reasoning in the presence of an unusual language feature. However it is unable to distinguish between private and public names, and thus cannot prove example (6) of Section 2. The same limitation in the operational technique of applicative equivalence is addressed by a move to logical relations; in this section we introduce a correspondingly refined scheme for relational reasoning about the nu-calculus. As with the equational theory, the aim is to provide all the useful power of operational logical relations in a more accessible form.

Assertions now take the form

$$\Gamma \vdash M_1\ R_\sigma\ M_2$$

where $R : s_1 \rightleftharpoons s_2$ is a span such that $M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$. As with operational logical relations, the intuition is that the names in s_1 and s_2 related by R are public and must be treated similarly by M_1 and M_2, while those names not mentioned in R are private and must remain so.

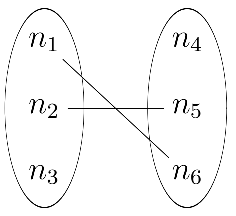
To write these assertions, we first need an explicit language to describe spans between sets of names. We build this up using disjoint sum $R \oplus R' : s_1 \oplus s_1' \rightleftharpoons s_2 \oplus s_2'$ over the following basic spans:

$$\begin{array}{rl} \overrightarrow{n} : \emptyset \rightleftharpoons \{n\} & \qquad \overleftarrow{n} : \{n\} \rightleftharpoons \emptyset \\ \emptyset : \emptyset \rightleftharpoons \emptyset & \qquad n_1\widehat{\ }n_2 : \{n_1\} \rightleftharpoons \{n_2\} \quad \text{nonempty.} \end{array}$$

In particular, we shall use the derived span:

$$\widehat{n} = n\widehat{\ }n : \{n\} \rightleftharpoons \{n\} \quad \text{nonempty.}$$

It is clear that this language is enough to express all finite spans. For example, consider the two three-element name sets $\{n_1, n_2, n_3\}$ and $\{n_4, n_5, n_6\}$. A span between them which we might draw as



is written as $\qquad (n_1\widehat{\ }n_6) \oplus (n_2\widehat{\ }n_5) \oplus \overleftarrow{n_3} \oplus \overrightarrow{n_4}$.

Equational Reasoning:

$$\frac{s_1, \Gamma \vdash M_1 =_\sigma M_2 \quad \Gamma \vdash M_2 \; R_\sigma \; M_3 \quad s_2, \Gamma \vdash M_3 =_\sigma M_4}{\Gamma \vdash M_1 \; R_\sigma \; M_4} \; (R : s_1 \rightleftharpoons s_2)$$

Congruence:

$$\frac{}{\Gamma \vdash x \; R_\sigma \; x} \; (x : \sigma \in \Gamma) \qquad \frac{}{\Gamma \vdash \mathit{true} \; R_o \; \mathit{true}}$$

$$\frac{\Gamma \vdash F_1 \; R_{\sigma \to \sigma'} \; F_2 \quad \Gamma \vdash M_1 \; R_\sigma \; M_2}{\Gamma \vdash (F_1 M_1) \; R_{\sigma'} \; (F_2 M_2)} \qquad \frac{}{\Gamma \vdash \mathit{false} \; R_o \; \mathit{false}}$$

$$\frac{\Gamma \oplus \{x : \sigma\} \vdash M_1 \; R_{\sigma'} \; M_2}{\Gamma \vdash (\lambda x{:}\sigma.M_1) \; R_{\sigma \to \sigma'} \; (\lambda x{:}\sigma.M_2)} \qquad \frac{\Gamma \vdash N_1 \; R_\nu \; N_2 \quad \Gamma \vdash N_1' \; R_\nu \; N_2'}{\Gamma \vdash (N_1 = N_1') \; R_o \; (N_2 = N_2')}$$

$$\frac{\Gamma \vdash B_1 \; R_o \; B_2 \quad \Gamma \vdash M_1 \; R_\sigma \; M_2 \quad \Gamma \vdash M_1' \; R_\sigma \; M_2'}{\Gamma \vdash (\mathit{if}\ B_1\ \mathit{then}\ M_1\ \mathit{else}\ M_1') \; R_\sigma \; (\mathit{if}\ B_2\ \mathit{then}\ M_2\ \mathit{else}\ M_2')}$$

Booleans:

$$\frac{\Gamma \vdash (M_1[\mathit{true}/b]) \; R_\sigma \; (M_2[\mathit{true}/b]) \quad \Gamma \vdash (M_1[\mathit{false}/b]) \; R_\sigma \; (M_2[\mathit{false}/b])}{\Gamma \oplus \{b : o\} \vdash M_1 \; R_\sigma \; M_2}$$

Names:

$$\frac{}{\Gamma \vdash n_1 \; R_\nu \; n_2} \; (\,(n_1, n_2) \in R\,)$$

$$\frac{\begin{array}{ll} \Gamma \vdash (M_1[n/x]) \; (R \oplus \widehat{n})_\sigma \; (M_2[n/x]) & \text{some fresh } n \\ \Gamma \vdash (M_1[n_1/x]) \; R_\sigma \; (M_2[n_2/x]) & \text{each } (n_1, n_2) \in R \end{array}}{\Gamma \oplus \{x : \nu\} \vdash M_1 \; R_\sigma \; M_2}$$

New names:

$$\frac{\Gamma \vdash M_1 \; (R \oplus \overleftarrow{n_1})_\sigma \; M_2}{\Gamma \vdash (\nu n_1.M_1) \; R_\sigma \; M_2} \qquad \frac{\Gamma \vdash M_1 \; (R \oplus \overrightarrow{n_2})_\sigma \; M_2}{\Gamma \vdash M_1 \; R_\sigma \; (\nu n_2.M_2)} \qquad \frac{\Gamma \vdash M_1 \; (R \oplus n_1 {}^\frown n_2)_\sigma \; M_2}{\Gamma \vdash (\nu n_1.M_1) \; R_\sigma \; (\nu n_2.M_2)}$$

Figure 4. Rules for deriving relational assertions.

Note that the domain and codomain of a span can easily be read off from this representation.

The rules for deriving relational assertions are given in Figure 4. The first of these integrates equational results into the logic, so that existing equational reasoning can be reused and we need only consider spans when absolutely necessary. This is followed by straightforward rules for congruence and booleans. Note that a trace of logical relations comes through in the congruence rule for application: related functions applied to related arguments give related results. As usual the most interesting rules are those concerning names.

To introduce a free variable of type ν requires checking its instantiation with all related pairs of names, and one representative fresh name. This is a weaker constraint than the corresponding rule in the equational logic, where every current name had to be considered; and it is precisely this difference that makes relational reasoning more powerful.

The final three rules handle the name restriction operator $\nu n.(-)$, and capture the notion that local names may be private or public. In combination with the equational rules for new names, they are equivalent to the following general rule:

$$\frac{\Gamma \vdash M_1\ (R \oplus S)_\sigma\ M_2}{\Gamma \vdash (\nu s_1.M_1)\ R_\sigma\ (\nu s_2.M_2)} \quad S : s_1 \rightleftharpoons s_2. \tag{15}$$

Thus in order to show that two expressions $(\nu s_1.M_1)$ and $(\nu s_2.M_2)$ are related it is enough to find some span between their local names under which the bodies M_1 and M_2 are related. This matches closely the clause for the operational relation R_σ^{exp} in Definition 3.2.

The search for the right span S here means that relational reasoning demands a little more creativity than the equational logic. To use the new name rules successfully requires some insight into how an expression uses its local names; which if any are ever revealed to a surrounding program.

We now link this relational theory to the operational reasoning methods of Section 3.

Theorem 5.1. (Soundness) Relational reasoning can be used to prove the corresponding operational relations:

$$\Gamma \vdash M_1\ R_\sigma\ M_2 \quad \Longrightarrow \quad \Gamma \vdash M_1\ R_\sigma^{opn}\ M_2\ .$$

By implication (7) of Theorem 3.2, this can then be used to demonstrate contextual equivalence:

$$\Gamma \vdash M_1\ (id_s)_\sigma\ M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \approx_\sigma M_2\ .$$

Proof:
We need to show that the operational logical relations R_σ^{opn} satisfy all the rules of Figure 4. This is reasonably straightforward, by reference to the various clauses of Definition 3.2; in particular the congruence rules follow from the 'logical' character of R_σ^{opn}.

The very first rule is a little different: this integrates relational with equational reasoning, and corresponds to the transitivity result of Proposition 3.1(3) that connects operational logical relations to applicative equivalence. Applying this here depends in turn on Theorem 4.1, that provable equality $=_\sigma$ implies applicative equivalence $\sim_\sigma$.

Theorem 5.2. (Completeness) Relational reasoning corresponds exactly to operational logical relations up to first-order types:

$$\Gamma \vdash M_1\ R_\sigma^{opn}\ M_2 \quad \Longrightarrow \quad \Gamma \vdash M_1\ R_\sigma\ M_2 \qquad \sigma \text{ first-order, ground } \Gamma.$$

By implication (8) of Theorem 3.2, the same result holds for contextual equivalence:

$$s, \Gamma \vdash M_1 \approx_\sigma M_2 \quad \Longrightarrow \quad \Gamma \vdash M_1\ (id_s)_\sigma\ M_2 \qquad \sigma \text{ first-order, ground } \Gamma.$$

Proof:
We follow much the same course as we did in proving the matching Theorem 4.1 for the equational logic. Guided by Definition 3.2 of logical relations, the proof is by induction on the size of Γ and the structure of σ; as usual we distinguish canonical forms from general expressions, and open expressions from closed ones.

- Base case: closed canonical forms of ground type.

$$\begin{aligned} b_1 \; R_o^{can} \; b_2 \quad &\Longrightarrow \quad \vdash b_1 \; R_o \; b_2 \\ n_1 \; R_\nu^{can} \; n_2 \quad &\Longrightarrow \quad \vdash n_1 \; R_\nu \; n_2 \end{aligned}$$

  Expanding the definition of R_o^{can} and R_ν^{can} respectively, the left hand sides assert that b_1 and b_2 are the same boolean constant, and that names n_1 and n_2 are related: $(n_1, n_2) \in R$. In both cases the statement on the right is then an axiom of the relational logic.

- Extension to general expressions. If for some type σ we have

$$C_1 \; R_\sigma^{can} \; C_2 \quad \Longrightarrow \quad \vdash C_1 \; R_\sigma \; C_2 \qquad \text{for all } R, C_1, C_2$$

then the same result holds for general expressions:

$$M_1 \; R_\sigma^{exp} \; M_2 \quad \Longrightarrow \quad \vdash M_1 \; R_\sigma \; M_2 \qquad \text{for all } R, M_1, M_2.$$

  To show this, suppose that $R : s_1 \rightleftharpoons s_2$, so $M_i \in \text{Exp}_\sigma(s_i)$ for $i = 1, 2$. Expanding R_σ^{exp} on the left we have that for some $R' : s_1' \rightleftharpoons s_2'$:

$$s_1 \vdash M_1 \Downarrow_\sigma (s_1')C_1, \quad s_2 \vdash M_2 \Downarrow_\sigma (s_2')C_2 \qquad \text{and} \qquad C_1 \; (R \oplus R')_\sigma^{can} \; C_2 \; .$$

  Applying Proposition 4.1 to the two evaluations we obtain the equational assertions

$$s_1 \vdash M_1 =_\sigma (\nu s_1'.C_1) \qquad \text{and} \qquad s_2 \vdash M_2 =_\sigma (\nu s_2'.C_2) \; . \tag{16}$$

  By hypothesis, from $C_1 \; (R \oplus R')_\sigma^{can} \; C_2$ we can deduce that $\vdash C_1 \; (R \oplus R')_\sigma \; C_2$, and applying the restriction rule (15) gives

$$\vdash (\nu s_1'.C_1) \; R_\sigma \; (\nu s_2'.C_2) \; . \tag{17}$$

  Taking (16) with (17) and applying the first rule of Figure 4 then proves

$$\vdash M_1 \; R_\sigma \; M_2$$

  as required.

- First-order function types. This is the induction step: we assume the result at R_σ^{exp} for all R and then prove it for $R_{o\to\sigma}^{can}$ and $R_{\nu\to\sigma}^{can}$. As with applicative equivalence, the key point is that the definition of logical relations at function types provides exactly the hypotheses needed for the rules that introduce free boolean or name variables. For example, suppose that

$$(\lambda x{:}\nu.M_1) \; R_{\nu\to\sigma}^{can} \; (\lambda x{:}\nu.M_2),$$

  i.e. that for all spans $R' : s_1' \rightleftharpoons s_2'$ and names $n_1 \in s_1 \oplus s_1'$, $n_2 \in s_2 \oplus s_2'$ we have

$$(n_1, n_2) \in R \oplus R' \quad \Longrightarrow \quad (M_1[n_1/x]) \; (R \oplus R')_\sigma^{exp} \; (M_2[n_2/x]) \; .$$

  Applying the induction hypothesis gives

$$(n_1, n_2) \in R \oplus R' \quad \Longrightarrow \quad \vdash (M_1[n_1/x]) \; (R \oplus R')_\sigma \; (M_2[n_2/x])$$

and setting R' to be the spans $\emptyset : \emptyset \rightleftharpoons \emptyset$ and $\widehat{n} : \{n\} \rightleftharpoons \{n\}$ for some fresh n gives just the hypotheses for the rule of Figure 4 introducing free name variables. Thus we deduce

$$\{x : \nu\} \vdash M_1 \; R_\sigma \; M_2$$

and by congruence

$$\vdash (\lambda x{:}\nu.M_1) \; R_{\nu\to\sigma} \; (\lambda x{:}\nu.M_2)$$

as required. The argument for booleans, $R_{o\to\sigma}$, is similar.

- Open expressions. Apply induction over the length of the context Γ. Every variable has ground type so the induction step is justified exactly as for first-order functions above.

This completes the proof of completeness with respect to operational logical relations. As indicated, we can extend this to contextual equivalence using the existing result of Theorem 3.2.

Thus relational reasoning provides a further practical method for reasoning about contextual equivalence. Like the equational logic it can be used freely at higher types and for expressions with free variables. Moreover, because spans capture the distinction between private and public names, the relational scheme is significantly more powerful than equational reasoning alone. Indeed it can prove every contextual equivalence between expressions of first-order type, thanks to the corresponding (hard) result for operational logical relations. In particular we obtain a demonstration of the final example (6) from Section 2: the crucial closing steps are

$$\cfrac{\cfrac{x : \nu \vdash (x = n) \; (\overleftarrow{n})_o \; \mathit{false}}{\vdash (\lambda x{:}\nu.(x = n)) \; (\overleftarrow{n})_{\nu\to o} \; (\lambda x{:}\nu.\mathit{false})}}{\vdash (\nu n.\lambda x{:}\nu.(x = n)) \; \emptyset_{\nu\to o} \; (\lambda x{:}\nu.\mathit{false})}$$

from which we deduce

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

as required. The span $(\overleftarrow{n}) : \{n\} \rightleftharpoons \emptyset$ used here captures our intuition that the name bound to n on the left hand side is private, never revealed, and need not be matched in the right hand expression.

6. Conclusions and Further Work

We have looked at the nu-calculus, a language of names and higher-order functions, designed to expose the effect of generativity on program behaviour. Building on operational techniques of applicative equivalence and logical relations, we have derived schemes for equational and relational reasoning; where a collection of inductive rules allow for straightforward proofs of contextual equivalence. We have proved that this approach successfully captures the distinction between private and public names, and is complete up to first-order function types.

Figure 5 summarises the inclusions between the five equivalences that we have considered. For general higher types they are all distinct; at first-order function types the three right-hand equivalences are identified; and at ground types all five are the same. Furthermore, as explained after the proof of Theorem 4.1, the reasoning schemes of this paper in the bottom row are in practice just as powerful as the operational methods above them.

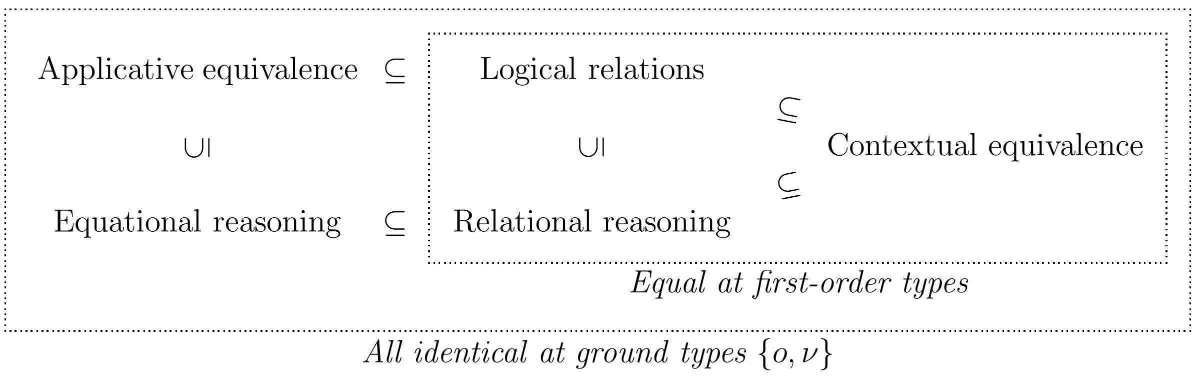

Applicative equivalence ⊆
Logical relations
⊆
⊇
Contextual equivalence
⊇
⊆
Equational reasoning ⊆
Relational reasoning
Equal at first-order types
All identical at ground types {o, ν}

Figure 5. Various equivalences between expressions of the nu-calculus

One direction for future work is to extend the language from names to the dynamically allocated *references* of Standard ML, storage cells that allow imperative update and retrieval. For integer references, appropriate denotational and operational techniques are already available [35, §5]. These use relations between sets of states to indicate how equivalent expressions may make different use of local storage cells. The idea then would be to make a similar step in the logic, from name relations to these state relations. For example, we might replace the span $R : s_1 \rightleftharpoons s_2$ with a more general predicate $\phi \subseteq \mathit{Store}(s_1) \times \mathit{Store}(s_2)$ and construct rules for the relational assertion $\Gamma \vdash M_1\ \phi_\sigma\ M_2$.

Recent joint work with Pitts [30] on a language of integer references has extended previous operational techniques to give logical relations that exactly match contextual equivalence at all types. The innovation is that we define not only R^{can} on canonicals and R^{exp} on expressions, but also R^{cont} on *continuations*, in a three-way mutually inductive definition. The completeness of this enhanced relational scheme is exciting; however to make full use of it we need to distill this extra power into new rules for the relational logic. The aim would be a scheme of rules that precisely characterise contextual equivalence, while still providing a practical basis for reasoning and proof.

A number of calculi for concurrent and distributed systems make use of abstract names to keep track of scope or privacy; it seems likely that the relational logic will adapt to reasoning about some of these. In fact the standard π-calculus has no function types, so equational reasoning is appropriate and sufficient [25]; but second and higher-order calculi like CHOCS [38] and HOπ [34] might benefit from a relational treatment. Other possibilities are the spi-calculus [1] and the ambient calculus [4], both of which rely explicitly on the detailed behaviour of names.

Consider for example the spi-calculus, which uses names as a foundation for reasoning about security protocols. In order to test for authenticity and secrecy one must verify certain contextual equivalences between processes, using insight into the visibility of cryptographic keys as represented by local names. This is exactly the territory over which our relational logic is effective. Thus where the spi-calculus writes $\{M\}_n$ for expression M encrypted under key n, we might approximately interpret

$$\{M\}_n \quad \text{by} \quad \lambda x{:}\nu.\mathit{if}\ x = n\ \mathit{then}\ M\ \mathit{else}\ ()$$

for some suitable null expression (). This is a function that will reveal M only if presented with the correct key n. We can then use relational reasoning to derive

$$s \vdash \{M\}_n\ (\overleftarrow{n} \oplus \overrightarrow{n})_{\nu\to\sigma}\ \{M'\}_n$$

for any expressions M and M', and it follows that these two encrypted expressions behave indistinguishably within any process P which does not know n:

$$s \vdash (\nu n.P[\{M\}_n/x])\ \mathit{id}_s\ (\nu n.P[\{M'\}_n/x]) \qquad \text{if } n \notin \mathit{fn}(P). \tag{18}$$

This confirms the security of the coding, and captures the fact that P cannot decipher $\{M\}_n$ without knowing n. The span $(\overleftarrow{n} \oplus \overrightarrow{n})$ used here matches Abadi and Gordon's *underpinning relation*, while equation (18) corresponds to Proposition 10 of [1] which is essential to their proofs of equivalence between processes.

Leaving aside such extensions, there is also the challenge of mechanising the relational logic within a general automated reasoning system like Isabelle [26] or Coq [3]. For example, Frost and Mason have already begun to do this for a fragment of VTLoE [7]. In our case the task is aided by the fact that all our definitions are inductive, and packages to reason about such constructions are by now fairly common currency among theorem provers. Perhaps the most demanding aspect would be that the nu-calculus uses name abstraction as well as lambda abstraction. Reasoning about binding mechanisms like these is still a delicate area — see [5, 11, 18] for some approaches — and concentrating attention onto pure names may provide some useful insights. Note that we are not concerned here with an implementation of the proof that the reasoning system itself is correct (Theorem 5.1); what might benefit from machine assistance is the demonstration that two particular expressions are id_s-related, and hence contextually equivalent.

Acknowledgements

This work was begun while the author was at the Dipartimento di Informatica within the University of Pisa, Italy, supported by the European Community under the HCM network "Euro-Focs". It was subsequently completed at BRICS (**B**asic **R**esearch **i**n **C**omputer **S**cience, a centre of the Danish National Research Foundation), in the University of Aarhus, Denmark.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997. An extended version is to appear in Information and Computation.

[2] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, editor D. A. Turner, pages 65–117. Addison Wesley, 1990.

[3] B. Barras et al. *The Coq Proof Assistant: Reference Manual.* Version 6.1, INRIA/CNRS, December 1996.

[4] L. Cardelli and A. D. Gordon. Mobile ambients. To appear in *Foundations of Software Science and Computation Structures: Proceedings of FoSSaCS '98*, Lecture Notes in Computer Science, Springer Verlag, 1998. A preprint is available electronically from http://research.microsoft.com/~adg/Publications/fossacs98.ps.

[5] J. Despeyroux, F. Pfenning, and C. Schuermann. Primitive recursion for higher-order abstract syntax. In *Typed Lambda Calculi and Applications: Proceedings of the Third International Conference TLCA '97*, editors P. de Groote and J. R. Hindley. Lecture Notes in Computer Science 1210, pages 147–164. Springer-Verlag, April 1997.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992. Also published as Technical Report 100-89, Rice University.

[7] J. Frost and I. A. Mason. An operational logic of effects. In *Computing: The Australasian Theory Symposium, Proceedings of CATS '96*, pages 147–156. Computer Science Association of Australia, January 1996.

[8] A. Gordon and A. Pitts, editors. *Higher Order Operational Techniques in Semantics.* Cambridge University Press, Publications of the Newton Institute, 1997.

[9] A. D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press Distinguished Dissertations in Computer Science, September 1994.

[10] A. D. Gordon. Bisimilarity as a theory of functional programming. In *Mathematical Foundations of Programming Semantics: Proceedings of the 11th International Conference*, Electronic Notes in Theoretical Computer Science 1. Elsevier, 1995.

[11] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *Theorem Proving in Higher Order Logics: Proceedings of the 9th International Conference TPHOLs '96*, editors J. von Wright, J. Grundy, and J. Harrison. Lecture Notes in Computer Science 1125, pages 173–191. Springer-Verlag, 1996.

[12] F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, May 1995.

[13] D. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, February 1996.

[14] 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.

[15] I. A. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):297–327, July 1991.

[16] I. A. Mason and C. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105:167–215, 1992.

[17] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 186–197. IEEE Computer Society Press, 1992.

[18] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. Submitted for publication, 1997.

[19] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992.

[21] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[23] E. Moggi. A general semantics for evaluation logic. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1994.

[24] E. Moggi. Representing program logics in evaluation logic. Preprint, available electronically from http://www.disi.unige.it/person/MoggiE/publications.html, 1994.

[25] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):172–197, August 1994.

[26] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Lecture Notes in Computer Science 828. Springer-Verlag, 1994.

[27] A. M. Pitts. Evaluation logic. In *IVth Higher Order Workshop, Banff 1990*, pages 162–189. Springer-Verlag Workshops in Computing, 1991. Also published as Technical Report 198, University of Cambridge Computer Laboratory.

[28] 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: Proceedings of the 18th International Symposium MFCS '93*, editors A. M. Borzyszkowski and S. Sokolowski. Lecture Notes in Computer Science 711, pages 122–141. Springer-Verlag, 1993.

[29] A. M. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *Proceedings of the 1993 ACM SIGPLAN Workshop on State in Programming Languages*, Yale University Department of Computer Science, Research Report YALEU/DCS/RR-968, pages 31–45, 1993.

[30] A. M. Pitts and I. Stark. Operational reasoning for functions with local state. In Gordon and Pitts [8], pages 227–273.

[31] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[32] J. G. Riecke. A complete and decidable proof system for call-by-value equalities. In *Automata, Languages and Programming: Proceedings of the 17th International Colloquium*, editor M. S. Paterson. Lecture Notes in Computer Science 443, pages 20–31. Springer-Verlag, 1990.

[33] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358, 1993.

[34] D. Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, December 1996.

[35] I. Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory (email tech-reports@cl.cam.ac.uk).

[36] I. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, February 1996.

[37] C. Talcott. Reasoning about functions with effects. In Gordon and Pitts [8], pages 347–390.

[38] B. Thomsen. Plain CHOCS. *Acta Informatica*, 30:1–59, 1993.