BRICS RS-96-31 I. Stark: Names, Equations, Relations: Practical Ways to Reason about *new*

BRICS

Basic Research in Computer Science

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

Ian Stark

BRICS Report Series **RS-96-31**

ISSN 0909-0878 **September 1996**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

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

Ian Stark
BRICS[*], Department of Computer Science
University of Aarhus, Denmark

September 1996

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 other kinds of local declaration, and to the mobile processes of the π-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.

[*]**B**asic **R**esearch **i**n **C**omputer **S**cience, a centre of the Danish National Research Foundation.

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 [13] uses dynamically-generated names to describe the behaviour of mobile processes, whose communication topology may change over time. In functional programming, the language Standard ML [14] extends typed lambda-calculus with a number of features, of which mutable reference cells, exceptions and user-declared datatypes are all generative; as 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.

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 [21, 22], with its operational and denotational semantics studied at some length in [25, 26]. 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

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

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 [12], Gordon's 'experiments' [4], and the 'ciu' theorems of Mason and Talcott [9, 27], 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 [25, §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 [1], Howe [7], Gordon [4], 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 [21, 22], while categorical models are presented in [26]. Both approaches are treated at length in [25].

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.

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 [19, 27], for example. However, not so many practical systems have emerged for reasoning about expressions and proving actual examples of contextual equivalence.

Felleisen and Hieb [2] 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 [10] 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' [8]. Further work [11] 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 [6] is an operationally-based scheme for proving certain assertions about functions with state.

The 'computational metalanguage' of Moggi [15] provides a general method for equational reasoning about additions to functional languages. Its application to the nu-calculus is discussed in [25, §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 [16, 20]. Moggi has shown how a variety of program logics, including VTLoE, can be expressed within evaluation logic [17].

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.

2 The Nu-Calculus

A full description of the nu-calculus can be found in [25, 26]; 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 ab-

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

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

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

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

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

breviation 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.

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

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

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

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

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

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

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

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

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 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 [25, Theorem 2.4].

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*.

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

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

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 [25, §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 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. However the 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. Numerous equivalences familiar from the call-by-value lambda-calculus hold, such as Plotkin's β_v-equivalence [23]: 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

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

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 [21, 22, 25, 26].

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 [21] and [25], which also give proofs of the results below.

Definition 2 (Applicative Equivalence). We define a pair of relations $s \vdash C_1 \sim_\sigma^{can} C_2$ for $C_1, C_2 \in \mathrm{Can}_\sigma(s)$ and $s \vdash M_1 \sim_\sigma^{exp} M_2$ for $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ inductively over the structure of the type σ, according to:

$$
\begin{aligned}
s \vdash b_1 \sim_o^{can} b_2 \quad &\iff \quad b_1 = b_2 \\
s \vdash n_1 \sim_\nu^{can} n_2 \quad &\iff \quad n_1 = n_2 \\
s \vdash \lambda x{:}\sigma.M_1 \sim_{\sigma\to\sigma'}^{can} \lambda x{:}\sigma.M_2 \quad &\iff \quad \forall s', C \in \mathrm{Can}_\sigma(s \oplus s')\ . \\
&\qquad\quad s \oplus s' \vdash M_1[C/x] \sim_{\sigma'}^{exp} M_2[C/x] \\
s \vdash M_1 \sim_\sigma^{exp} M_2 \quad &\iff \quad \exists s_1, s_2, C_1 \in \mathrm{Can}_\sigma(s \oplus s_1), C_2 \in \mathrm{Can}_\sigma(s \oplus s_2)\ . \\
&\qquad\quad s \vdash M_1 \Downarrow_\sigma (s_1)C_1 \ \ \&\ \ s \vdash M_2 \Downarrow_\sigma (s_2)C_2 \\
&\qquad\quad \&\ \ s \oplus (s_1 \cup s_2) \vdash C_1 \sim_\sigma^{can} C_2.
\end{aligned}
$$

The intuition here is that functions are equivalent if they give equivalent results at possible arguments; while expressions in general are equivalent if they evaluate to equivalent canonical forms.

It is immediate that $\sim_\sigma^{exp}$ coincides with $\sim_\sigma^{can}$ 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

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

where $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$.

Applicative equivalence is based on similar 'bisimulation' relations of Abramsky [1] and Howe [7] for untyped lambda-calculus, and Gordon [5] for typed lambda-calculus. It is well behaved and suffices to prove contextual equivalence:

Theorem 3. *Applicative equivalence is an equivalence, a congruence, and implies contextual equivalence.*

The proof of this 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. That it implies contextual equivalence follows from this without difficulty; details are in [25, §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

$$(n_1, n_2) \in R \quad \& \quad (n_1', n_2') \in R \quad \implies \quad (n_1 = n_1') \iff (n_2 = n_2')$$

[2]This is a different relation to the applicative equivalence of [21, Definition 13] and [22, Definition 3.4] which (rather unfortunately) turns out not to be an equivalence at all.

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 4 (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:

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

$$n_1 \; R^{can}_\nu \; n_2 \iff n_1 \; R \; n_2$$

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

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

This definition differs somewhat from that for applicative equivalence. Functions are now related if they take related arguments to related results; and expressions in general are related if some span can be found between their respective 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: if $M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$ then define

$$\begin{aligned} \Gamma \vdash M_1 \; R^{opn}_\sigma \; M_2 \iff\; &\forall R' : s'_1 \rightleftharpoons s'_2, \\ &\quad C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s'_i) \quad i = 1,2 \quad j = 1, \ldots, n\,. \\ &\qquad (\,\&^n_{j=1} \,.\, C_{1j} \; (R \oplus R')^{can}_{\sigma_j} \; C_{2j}\,) \\ &\qquad\; \implies M_1[\vec{C_1}/\vec{x}] \; (R \oplus R')^{exp}_\sigma \; M_2[\vec{C_2}/\vec{x}] \end{aligned}$$

where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$.

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.

Theorem 5. *For any expressions $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$:*

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

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:

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

Proposition 6. *Logical relations subsume applicative equivalence: if $s, \Gamma \vdash M_1 \sim_\sigma M_2$ then $\Gamma \vdash M_1\ (id_s)^{opn}_\sigma\ M_2$.*

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 [25, §4.6], but they are complete up to first-order functions, and in particular they prove all the examples of Section 2 above.

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

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

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:

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

Functions:

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

Booleans:

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

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

Names:

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

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

New names:

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

$$\frac{s \oplus \{n\}, \Gamma \vdash M_1 =_\sigma M_2}{s, \Gamma \vdash \nu n.M_1 =_\sigma \nu n.M_2} \qquad \frac{}{s, \Gamma \vdash U\langle \nu n.M \rangle =_\sigma \nu n.U\langle M \rangle} (n \notin \mathit{fn}(U\langle - \rangle))$$

Figure 3: Rules for deriving equational assertions.

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 and new names.

The 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. Name abstractions $\nu n.(-)$ can be moved past each other, and through contexts $U\langle -\rangle$, providing that name capture is avoided.

There is no η-rule given for functions; the general rule is not true, and the more restrictive

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

follows from the β_v-rule, using the fact that the variable 'x' counts as a value. Other rules are simplified because evaluation in the nu-calculus is terminating; compare this for example with Riecke's axiomatisation of call-by-value PCF [24].

Proposition 7. *This equational theory respects evaluation:*

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

Proof. It is not hard to demonstrate, using the equational theory, that every rule for evaluation in Figure 2 preserves the property given. □

Theorem 8 (Soundness and Completeness). *Equational reasoning is sufficient to prove applicative equivalence, and hence also contextual equivalence:*

$$\begin{aligned} s, \Gamma \vdash M_1 =_\sigma M_2 &\quad\Longrightarrow\quad s, \Gamma \vdash M_1 \sim_\sigma M_2 \\ s, \Gamma \vdash M_1 =_\sigma M_2 &\quad\Longrightarrow\quad s, \Gamma \vdash M_1 \approx_\sigma M_2. \end{aligned}$$

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

$$\begin{aligned} s, \Gamma \vdash M_1 \sim_\sigma M_2 &\quad\Longrightarrow\quad s, \Gamma \vdash M_1 =_\sigma M_2 \qquad \sigma \textit{ first-order, ground } \Gamma \\ s \vdash M_1 \approx_\sigma M_2 &\quad\Longrightarrow\quad s \vdash M_1 =_\sigma M_2 \qquad \sigma \in \{o, \nu\}. \end{aligned}$$

Proof. Soundness follows from the fact that every rule of Figure 3 for $=_\sigma$ also holds for $\sim_\sigma$. The converse results on completeness are tedious to prove but not especially difficult; however correct use of the rules for introducing free variables of ground type is important for handling first-order functions. □

At higher types applicative equivalence is in principle more powerful than our equational reasoning. However this advantage is illusory: the only way to demonstrate it is to use some more sophisticated technique (such as logical relations) to show that particular functions cannot be expressed 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. As a slightly more complex example, we prove β-equivalence for univalent function bodies:

$$s \vdash (\lambda x{:}\sigma.U\langle x\rangle)M \approx_{\sigma'} U\langle M\rangle \qquad x \notin fv(U\langle - \rangle). \tag{7}$$

By termination and Proposition 7, we have that necessarily $s \vdash M =_\sigma \nu s'.C$ for some names s' and canonical form C. Thus we may reason as follows:

$$\begin{aligned}
s \vdash (\lambda x{:}\sigma.U\langle x\rangle)M &= (\lambda x{:}\sigma.U\langle x\rangle)(\nu s'.C) && \text{congruence}\\
&= \nu s'.((\lambda x{:}\sigma.U\langle x\rangle)C) && \text{pass names through application}\\
&= \nu s'.U\langle C\rangle && \beta_v\\
&= U\langle \nu s'.C\rangle && \text{pass names through } U\langle - \rangle\\
&= U\langle M\rangle && \text{congruence.}
\end{aligned}$$

This is a truly higher-order result: 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 $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$. To write such 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:

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

In particular, we shall use the derived span:

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

It is clear that this language is enough to express all finite spans.

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. The succeeding rules for congruence and booleans are straightforward, and as before the 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 creation 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:

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

To apply such rules successfully requires some insight into how an expression uses its local names; which if any are ever revealed to a surrounding program.

Theorem 9 (Soundness). *Relational reasoning is sufficient 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 Theorem 5, these in turn can be used to demonstrate contextual equivalence:

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

Equational Reasoning:

$$\frac{\Gamma \vdash M_1 =_\sigma M_2 \quad \Gamma \vdash M_2\ R_\sigma\ M_3 \quad \Gamma \vdash M_3 =_\sigma M_4}{\Gamma \vdash M_1\ R_\sigma\ M_4}$$

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:

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

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

Name creation:

$$\frac{\Gamma \vdash M_1\ (R \oplus \overleftarrow{n_1})_\sigma\ M_2}{\Gamma \vdash (\nu n_1.M_1)\ R_\sigma\ M_2} \quad \frac{\Gamma \vdash M_1\ (R \oplus \overrightarrow{n_2})_\sigma\ M_2}{\Gamma \vdash M_1\ R_\sigma\ (\nu n_2.M_2)} \quad \frac{\Gamma \vdash M_1\ (R \oplus n_1\widehat{\ }n_2)_\sigma\ M_2}{\Gamma \vdash (\nu n_1.M_1)\ R_\sigma\ (\nu n_2.M_2)}$$

Figure 4: Rules for deriving relational assertions.

Proof. We can show that the operational logical relations R_σ^{opn} satisfy all the rules of Figure 4; this in turn depends on Theorem 8, that provable equality $=_\sigma$ implies applicative equivalence $\sim_\sigma$. $\square$

Theorem 10 (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 Theorem 5, 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. By induction on the size of Γ and structure of σ; essentially, we work through the defining clause for logical relations (Definition 4). It is significant here that evaluation is respected by the equational logic (Proposition 7), which is in turn incorporated into the relational theory. $\square$

Thus relational reasoning provides a further practical method for reasoning about contextual equivalence. We even have that it can prove all contextual equivalences 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.

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 [19, 25]. 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.

The question of completeness remains open: can these methods be enhanced to prove all contextual equivalences? The operational method of 'predicated logical relations' [25, §4.6] does take things a little further, with an even finer analysis of name use; however the theoretical effort involved seems at present to outweigh the practical returns.

Separately from this, it should not be hard to implement the existing relational logic within a generic theorem prover such as Isabelle [18], or even directly in a logic programming language like Prolog. The only difficulty for proof search lies in the choice of relation between local name sets. Human guidance is one solution here, but even a brute force approach would work as there are only finitely many spans $R : s_1 \rightleftharpoons s_2$ between any two given name sets. Note that we are not concerned here with an implementation of the proof that the reasoning system itself is correct (Theorem 9); what might benefit from machine assistance is the demonstration that two particular expressions are id_s-related, and hence contextually equivalent.

References

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

[2] 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.

[3] A. Gordon and A. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997. To appear.

[4] A. D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, September 1994.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

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

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

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

[16] 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.

[17] E. Moggi. Representing program logics in evaluation logic. Unpublished manuscript, available electronically, 1994.

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

[19] A. Pitts and I. Stark. Operational reasoning for functions with local state. In Gordon and Pitts [3]. To appear.

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

[21] 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*, Lecture Notes in Computer Science 711, pages 122–141. Springer-Verlag, 1993.

[22] 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.

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

[24] 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*, Lecture Notes in Computer Science 443, pages 20–31. Springer-Verlag, 1990.

[25] 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.

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

[27] C. Talcott. Reasoning about functions with effects. In Gordon and Pitts [3]. To appear.

Recent Publications in the BRICS Report Series

RS-96-31 Ian Stark. *Names, Equations, Relations: Practical Ways to Reason about 'new'*. September 1996. ii+22 pp.

RS-96-30 Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. *Fusion Trees can be Implemented with AC^0 Instructions only*. September 1996. 8 pp.

RS-96-29 Lars Arge. *The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*. August 1996. 35 pp. An extended abstract version appears in Staples, Eades, Kato, and Moffat, editors, *Algorithms and Computation: 6th International Symposium*, ISAAC '95 Proceedings, LNCS 1004, 1995, pages 82–91.

RS-96-28 Lars Arge. *The Buffer Tree: A New Technique for Optimal I/O Algorithms*. August 1996. 34 pp. This report is a revised and extended version of the BRICS Report RS-94-16. An extended abstract appears in Akl, Dehne, Sack, and Santoro, editors, *Algorithms and Data Structures: 4th Workshop*, WADS '95 Proceedings, LNCS 955, 1995, pages 334–345.

RS-96-27 Devdatt Dubhashi, Volker Priebe, and Desh Ranjan. *Negative Dependence Through the FKG Inequality*. July 1996. 15 pp.

RS-96-26 Nils Klarlund and Theis Rauhe. *BDD Algortihms and Cache Misses*. July 1996. 15 pp.

RS-96-25 Devdatt Dubhashi and Desh Ranjan. *Balls and Bins: A Study in Negative Dependence*. July 1996. 27 pp.

RS-96-24 Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. *Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL*. July 1996. 20 pp. Presented at *DIMACS Workshop SPIN96 – 2nd International SPIN Verification Workshop on Algorithms, Applications, Tool Use, Theory* (Rutgers University, New Jersey, USA, August 5, 1996).