Names and Higher-Order Functions

Ian David Bede Stark
Queens' College

A dissertation submitted to the University of Cambridge
towards the degree of Doctor of Philosophy

December 1994

University of Cambridge Computer Laboratory
Technical Report No. 363

April 1995

Abstract

Many functional programming languages rely on the elimination of ‘impure’ features: assignment to variables, exceptions and even input/output. But some of these are genuinely useful, and it is of real interest to establish how they can be reintroducted in a controlled way. This dissertation looks in detail at one example of this: the addition to a functional language of dynamically generated *names*. Names are created fresh, they can be compared with each other and passed around, but that is all. As a very basic example of *state*, they capture the graduation between private and public, local and global, by their interaction with higher-order functions.

The vehicle for this study is the *nu-calculus*, an extension of the simply-typed lambda-calculus. The nu-calculus is equivalent to a certain fragment of Standard ML, omitting side-effects, exceptions, datatypes and recursion. Even without all these features, the interaction of name creation with higher-order functions can be complex and subtle.

Various operational and denotational methods for reasoning about the nu-calculus are developed. These include a computational metalanguage in the style of Moggi, which distinguishes in the type system between values and computations. This leads to categorical models that use a strong monad, and examples are devised based on functor categories.

The idea of *logical relations* is used to derive powerful reasoning methods that capture some of the distinction between private and public names. These techniques are shown to be complete for establishing contextual equivalence between first-order expressions; they are also used to construct a correspondingly abstract categorical model.

All the work with the nu-calculus extends cleanly to *Reduced ML*, a larger language that introduces integer *references*: mutable storage cells that are dynamically allocated. It turns out that the step up is quite simple, and both the computational metalanguage and the sample categorical models can be reused.

Preface

I would like to thank my supervisor, Andrew Pitts, for his continuing encouragement, support, guidance and instruction. I am very grateful to the numerous friends, both within and without the Computer Laboratory, who have brightened these last three years in Cambridge. Extraordinary thanks are due to my wife, Judith, for everything.

I would also like to acknowledge the financial support of the Science and Engineering Research Council, the EC Basic Research Action 'Categorical Logic in Computer Science', the EC SCIENCE project 'Programming Language Semantics and Program Logics', the Computer Laboratory, and Queens' College.

Except where otherwise stated, all the work described in this dissertation is the product of my own original research. Some of the results presented have also appeared in the conference papers [91] and [92] (joint with Andrew Pitts), and in the journal article [120].

This document was prepared using LaTeX 2_ε, with Michael Barr's diagram macros. Thanks to some code originating with Andy Gordon, each reference in the bibliography is followed by a list of the pages on which it is cited.

Preface to the Technical Report

I wish to thank my examiners, Samson Abramsky and Robin Milner, for their comments, suggestions, and discussion of the work in this dissertation. This technical report edition contains alterations suggested by them, together with a number of minor textual corrections. It is otherwise identical to the version originally submitted.

Contents

List of Figures

Chapter 1

Introduction

Functional languages are good for writing programs that work. Clear and well-defined semantics mean that there is agreement between a programmer's expectation and what actually happens, while simplicity in design allows practical methods for reasoning and program verification. What you intend to write is what I read and what the computer executes.

The route to achieving this often includes the elimination of language features judged to be 'impure': direct access to memory and other physical devices, expressions with concealed side-effects, or unstructured control methods (don't even think of $goto$).* Out with these go assignment to variables or arrays, exceptions, and even input/output. By way of compensation, functional languages provide a high level of abstraction and the powerful techniques of recursion and higher-order functions.

But many of these 'impure' features are genuinely useful, and their absence is given by some as a reason to avoid functional programming. So it is of real interest to establish how they can be reintroduced in a controlled way, without losing the advantages of a purely functional language. A. Gordon, for example, has addressed this problem for input/output [25]. For each other language feature, we can ask: why not give it another chance?

This dissertation looks in detail at one particular example: the addition of state to a functional programming language, in the form of dynamically generated *names*. Names are created fresh, they can be compared with each other and passed around, but that is all. As a very basic example of state, they capture the graduation between private and public, local and global, by their interaction with higher-order functions. We examine their behaviour, develop methods for reasoning about it, and construct categorical models to capture the meaning of names. As a more substantial example of state, we then extend all this to a functional language with integer *references*: mutable storage cells that are dynamically allocated. It turns out that the step up is quite simple and much of the work on names can be reused.

While the theory of state in functional languages is far from complete, this work shows that the apparently 'impure' feature of dynamically generated names can be introduced in a safe and well-behaved way. In particular there are good, strong reasoning methods that

* "The purpose of Newspeak was not only to provide a medium of expression for the world-view and mental habits proper to the devotees of Ingsoc, but to make all other modes of thought impossible". George Orwell, *1984*

can handle names without removing their usefulness.

1 Background

Purely functional languages are based on Church's lambda-calculus [10], where everything is a function, and the only operation is application of a function to an argument. In principle, a program is executed by rewriting the corresponding lambda-calculus expression according to certain rules: this is called *reduction*. There are two natural strategies for reduction, which give rise to two families of functional languages:

- In a *strict* functional language, the arguments to a function are evaluated before they are passed to the function body. This is *call-by-value* parameter passing; it is efficient and easy to compile. Strict languages include LISP and Standard ML.

- In a *lazy* functional language, arguments are given to a function unevaluated, and are only examined when their value is required. This is *call-by-name* parameter passing, generally implemented as *call-by-need*, where an argument is evaluated when first used and the value saved to avoid recomputation. Lazy languages are tricky to compile well on standard machine architectures, but they do introduce some novel programming techniques, most notably infinite data structures. Ponder, Miranda, Haskell and Gofer are all lazy languages [128, 130, 33, 35].

In practice the division is not always quite so clear, with certain disputes about the exact requirements for laziness; see Riecke [112] or Abramsky [4] for a discussion of this.

These languages remain close to their mathematical roots, and compilers often base optimizations on the behaviour of programs as mathematical objects. For example, this may allow repeated large-scale code transformations, leading to efficient implementation. All this eschews many of the features developed for conventional imperative languages; stepping outside the mathematical purity of the lambda-calculus is carefully avoided. But there are several practical reasons why such 'impure' features are important:

- The real world. Most programs have to interact with the external world, yet the original lambda-calculus quite reasonably made no provision for input/output.

- They might be simply a good idea. The raising and handling of exceptions is a prime example of a sensible and powerful control mechanism that lies outside the purely functional.

- Algorithms require them. Some algorithms rely on particular structures, such as an array that can be updated in place, to execute efficiently [98]. Often there are alternative algorithms, perhaps less well known, that use the data structures of functional languages; queues are a good example of this [86]. But sometimes no such method is known, as is presently the case with various graph traversal algorithms [38].

- They match the machine. Many traditional language features exist because they accurately represent some aspect of computer hardware; by ignoring this, functional languages are inevitably inefficient. Today this argument has lost much of its force, as programs in any high-level language are often drastically transformed under compilation for modern architectures and operating systems.

> References in Standard ML are a good example of this. Mutable storage cells would seem to map well onto real memory; but they cause surprising difficulties for generational garbage collection, as they break the rule that all pointers should be directed at objects older than themselves.

For various reasons then, real functional languages usually incorporate some 'impure' features: for example, LISP has the destructive $rplca$ and $rplcd$, while Haskell has I/O primitives. Sometimes these features are not added to the language itself, but effected by writing programs in a standard way. This preserves the purity of the language, though it may distort programming style; monads and their associated 'plumbing' are a well-known example [133, 134].

The language most relevant to this dissertation is Standard ML. This is a language with a formal definition [62] and a number of implementations. It is strongly typed, with polymorphic types and a sophisticated module system of 'structures' and 'functors'. Evaluation is strict, though there is a variant 'Lazy ML' [8]. Standard ML provides exceptions, references and I/O primitives. Of all these we shall look only at references: these are mutable storage cells, dynamically allocated on a heap and cleared away by a garbage collector.

No-one could reasonably describe ML as a purely functional language. But much of it is functional in style and spirit, and the work presented here is evidence that such a language can still provide the benefits of functional programming, including a well-defined denotational semantics and powerful reasoning methods.

2 Basic Concepts

This dissertation looks at a single 'impure' addition to a functional language: dynamically generated names. These are brought together with higher-order functions in the nu-calculus, a small experimental language. In this section we introduce each of these concepts in turn; the next section describes methods for studying them.

2.1 Names

The idea of a name is one of the most widely used abstractions in programming languages, from the specification of surface syntax, through the formal meaning of programs, to the details of implementation. An alphanumeric identifier in C may name a variable, a value of reference type in ML names a storage cell, and a machine address names a memory location. The basic, and rather simple, property required of a name is that it should be distinct from all others. It is usual to assume also that names are drawn from some infinite supply, so that a fresh one can always be obtained; thus names lie behind many *generative* programming constructions.

Although the abstract concept of 'name' is relevant to many aspects of programming, its presence is not always obvious. Here are a few examples:

- Clearest is the inclusion of names within a language itself: the $gensym$ operation of LISP produces a new symbol every time it is called.

- Some notion of name may be needed in a formal description: in the definition of Standard ML, every structure created is tagged with a distinct name [62, rule 53].

- The implementation of a language may have generative features: a local variable in a procedural language is one created distinct from all others, usually allocated on a stack or heap.

- Weakest of all, a user may simply be expected to manage something as if it were a name: for example, in an exhortation that global identifiers should be chosen distinct.

A more subtle aspect of naming is its connection with privacy. A name cannot be guessed, or adjusted, or manipulated, except to pass it on: names are a first example of an *abstract type*. Though any actual representation of names must have some internal structure, this should be invisible. The most striking example of this in practice is the use of 'capabilities' in the Cambridge CAP computer [138], where such restrictions are enforced by the processor instruction set; more recent memory protection schemes use similar ideas.

Failure to suitably conceal the implementation details of names can cause problems. For example in C the unrestricted use of pointer arithmetic means that privacy is not respected, and it is quite possible to write to memory locations at random; though this is generally considered poor programming practice.

But privacy is not always so clear-cut: a module may export the names of some of its components but not others; one pointer may lead to another in a linked list; a file may be referred to by two different handles, one for reading and one for writing. Names can be used to capture all these different degrees of access and shades of visibility. Names also lie behind 'object identity', an important concept in the design of object-oriented databases [85]. They are well known to be of significance in distributed and other concurrent systems: they are a key idea in Milner's pi-calculus [60, 61], and receive attention in the specification of real distributed systems [131]. This field highlights some of the more complex aspects of privacy, and leads to the area of security, secure communication, authentication and so forth.

In summary then, the idea of a name is a simple one, relevant to a wide variety of concepts used in the design, implementation and use of programming languages. This flexibility has a price: the exact properties of names, and how they are used, can be subtle and difficult to pin down.

2.2 Higher-Order Functions

In a typed programming language, a function maps values of one type to values of another type. It is *higher-order* if either the argument or result is also a function. For example most functional languages provide a map function that takes a function f and a list l as arguments and applies f to every element of l, returning a list l' as result. Higher-order functions implement the idea that functions should be 'first-class citizens' in a programming language: all that can be done with values of ground type (integers, booleans *etc.*) should also be possible with functions.

Higher-order functions are not restricted to functional languages. Reynolds' 'Idealized Algol' [107] and the original Algol 60 [71] are imperative languages in which functions and procedures can be passed as arguments or returned as results. However the treatment of functions as first-class citizens does require the manipulation of *closures*: a function paired with an environment giving values for its free variables. This can cause difficulties for stack-based languages, so for example C, Modula 3 [72] and Algol 68 [48, 132] cannot make full use of higher-order functions.

Operations like map increase generality, but the direct manipulation of functions can also be used to build procedures 'on the fly' and then execute them. A simple example is a function that takes two functions f, g and returns their composition $(f \circ g)$. Much more sophisticated are the combinators to construct parsers described by Hutton [34], or the use of continuations to manage the flow of control during execution [7].

This facility is like the LISP $eval$ operator, but more closely integrated into the language: instead of manipulating list representations of procedures and then evaluating them, we work with the procedures themselves. In a sense, higher-order functions provide the versatility of run-time code generation without the danger. There remains the issue of how efficiently any particular compiler manages this technique; even so it seems likely that its full power has yet to be exploited.

The treatment of functions on a level with other values is also good for the abstraction of data handling away from the details of representation. For example, a 'dictionary' datatype is best seen as a function from keys to value, even though it may be implemented by a binary tree, hash table or whatever. If treated purely as a function, any particular dictionary can choose whichever representation is most efficient without affecting the code that uses it. As another example, Matthews' language Poly represents assignable variables as a pair of procedures, one that extracts the current value and one that changes it [56]. The same is possible in Reynolds' Gedanken, and is also seen in his semantics for Algol, where a variable is an acceptor paired with an expression [103, 107]. Again, this leaves the way open for whatever implementation is most appropriate.

An extreme example of the power of higher-order functions is that they can be used to encode all the other usual datatypes: products, sums, integers, lists and so forth. This can be done immediately in the untyped lambda-calculus; in a typed setting we need the second-order lambda-calculus of Girard's System F [23], discovered independently by Reynolds [104]. The technique is presented by Girard in [24, Chapter 11] and extended by Wraith in [139], while Abadi and Plotkin's paper [97] gives methods for reasoning about such constructions. Ingenious though it is, the use of higher-order functions to encode other datatypes usually attracts only theoretical interest; though Fairbairn does argue for it as a practical implementation method in the design of the language Ponder [17, 18].

2.3 The Nu-Calculus

The nu-calculus is a simple language providing higher-order functions and the dynamic creation of names. It was identified by Pitts as a sensible subset of ML, and is close to Stoughton's 'identity calculus'. In all respects the nu-calculus is chosen to be as simple as possible: the only ground types are booleans and names, there is no recursion, and all evaluation is deterministic and terminating. Even so, one of the abiding lessons of computer science is that small, simple systems may yet have complex and subtle behaviour, and we shall see that the nu-calculus is no exception.

The chief use of names in the nu-calculus is to look at questions of visibility. If a certain name is known to a function, then it can be handled specially; if the name is unknown, then it must be treated the same as any freshly created name. Or perhaps a function may not have access to a name itself, but only a test for it: such a function cannot generate the name, but can recognise it as an argument. Even more complex, a function might only give out a private name as result if it is given some other particular name as an argument; this is the case with the function F_p on page 25. The possibilities are endless, and all capture some subtlety of names, privacy and scope.

The purpose of the nu-calculus then is not to provide a practical programming language, but rather to bring out a particular aspect of larger languages, so that it can be examined for itself. As it turns out, the interaction between dynamically generated names and higher-order functions is worthy of the attention.

3 Methods

Given a language, the nu-calculus, that combines names with higher-order functions, we want to study its behaviour and find methods for reasoning about it. To this end we use contextual equivalence to describe the properties of the language, categorical models to capture its meaning, and logical relations to refine our reasoning. This section describes these techniques and outlines the motivation behind them.

3.1 Contextual Equivalence

The benchmark relation for describing the operational behaviour of expressions in a functional language is *contextual equivalence*, originating with Morris [70] and used by Milner [59] and Plotkin [94, 95]. Two expressions are judged equivalent if they can be freely exchanged in any program; there is no way in the language itself to distinguish between them.

A simple use of contextual equivalence is to explain the properties of particular language features: for example in ML the equivalence

$$\mathsf{let\,val\,r} = \mathsf{ref}\,i\,\mathsf{in\,!r\,end} \;\approx\; i \qquad i \in \mathbb{Z}$$

illustrates the initialisation of reference cells. This approach is conveniently self-regulating in that details which properly belong to the implementation simply cannot be expressed; in this case for example, it does not matter what strategy the compiler uses for heap allocation.

More formally, such equivalences can be used to verify code transformations made during compilation. This applies to small, even trivial, manipulations as much as it does to complex and ingenious optimizations: in all cases contextual equivalence is the correct notion to check against. At a higher level, contextual equivalence is the right way to show that a programmer can use one algorithm instead of another.

This approach differs considerably from a traditional logic of programs such as Hoare logic, where assertions are made about machine state before, during and after the execution of a program [28, 15]. Nor do we have any distinct notion of a specification that some program must meet, beyond simple type-checking. Nevertheless contextual equivalence can serve in both these rôles. Assertions of a program logic can be replaced by tests within the language: for example, the equivalence above captures the following proposition in Hoare logic:

$$\{T\}\ \mathsf{val\,r} = \mathsf{ref}\,i\ \{contents(\mathsf{r}) = i\}.$$

Similarly a specification can be replaced either by a test expressed in the programming language, or a requirement that a program should be equivalent to some clear example.

For a sorting routine these methods might give

$$\mathsf{xs} : \mathsf{int\ list} \vdash \quad \mathsf{sorted}(\mathsf{clever\_sort}\ \mathsf{xs}) \;\approx\; \mathsf{true}$$

and

$$\mathsf{xs} : \mathsf{int\ list} \vdash \quad \mathsf{clever\_sort}\ \mathsf{xs} \;\approx\; \mathsf{insertion\_sort}\ \mathsf{xs}$$

respectively. In all these cases it is a clear advantage that we work entirely within the programming language itself; in essence, this gives assertion and specification languages with just the right level of abstraction.

Useful though contextual equivalence is, it can be rather a difficult relation to prove in specific instances. It is convenient therefore to identify other relations that imply contextual equivalence but are simpler to demonstrate. For example, an equivalence relation between expressions is a *congruence* if it is preserved by all constructions of the language; and it is usually not hard to show that any congruence which respects the operational semantics of the language, and distinguishes $true$ from $false$, implies contextual equivalence. The complementary relation is rather simpler: to show that two expressions are not contextually equivalent it is enough to demonstrate some program context that distinguishes them.

3.2 Categorical Models

Methods for reasoning about programming languages can be divided broadly into the operational and the denotational. Operational methods work explicitly with expressions of the language and their reduction or evaluation to canonical form. This has an appealing directness, and many of the relations described in this dissertation are described operationally. Denotational methods on the other hand first interpret the language in some mathematical setting, and then work within this *model*. The intention is to abstract away from particular details of a programming language and capture its essential 'meaning': for example, a function might be translated from program text into a map between sets. Such a translation is *adequate* if equality in the model implies contextual equivalence; it is *fully abstract* if this can be used to prove all contextual equivalences. The method is flexible in that different models can be developed to demonstrate particular equivalences.

Denotational semantics is not just for proving equivalences: a good model will illustrate how the features of a programming language fit together, and can be an aid to further language design. Two examples of this approach applied to functional languages are Milner on the typed lambda-calculus [59] and Plotkin on PCF [95]. A more general background on denotational semantics for programming languages is given by Stoy in [122].

Category theory is a general theory of mathematical structures, and provides a suitable setting for models of functional languages. Mac Lane's book [51] is the standard introductory text on categories; Mac Lane and Moerdijk [52] on topos theory is considerably more comprehensive. Two important examples of how categories assist with denotational semantics are the interpretation of the simply-typed lambda-calculus in any cartesian closed category [42], and the solution of recursive domain equations in O-categories [119]. The first of these forms the basis of all the categorical models used in this dissertation. We shall also use the fact that any category has its own *internal language*: this provides a logic to carry out equational reasoning about the structure of the category, with an excellent correspondence between categorical properties and constructions in the language [52, §VI.5].

Moggi observed that various aspects of computation in a programming language could be captured by the categorical concept of a *strong monad*; this has turned out to be a powerful abstraction, unifying several disparate language features [66, 67, 68]. The corresponding internal language is known as the *computational lambda-calculus* and is notable for its type system which separates values from computations. Various rules of the language describe how to reason correctly about computations, and these are enhanced in Pitts' *evaluation logic* by the addition of certain computation modalities [90].

These ideas lead to a particular denotational approach that falls naturally into two stages: we first interpret the nu-calculus in a metalanguage based on the computational lambda-calculus, and then interpret this within a category;

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

The metalanguage can be used to reason about contextual equivalence in the nu-calculus, and is also the internal language of the categorical model. This division into two translation steps allows us to build more than one model on the same foundations, and this framework is reused when we extend the nu-calculus to a language with store.

3.3 Logical Relations

While the methods described above provide a solid basis for reasoning about the behaviour of the nu-calculus, some extra ingredient is necessary if we are to prove results about privacy and the visibility of names; if we want to move from the merely correct to the genuinely informative. The techniques we introduce are based on *logical relations*.

The discussion of any typed language inevitably involves a number of type-indexed collections: whether of expressions, elements in semantic domains or morphisms in a category. A type-indexed relation between such collections is said to be *logical* when elements of function type are related if and only if they take related arguments to related results. Typically this means that a logical relation is fixed by its value at ground types, with the remaining type hierarchy built on top. Clearly this description is rather loose, and the idea of a logical relation can be reinterpreted in many settings.

The concept was introduced by Plotkin, at the suggestion of M. Gordon, to reason about definable elements in models of the simply-typed lambda-calculus [96]. There are similarities with Reynolds' idea of relational parametricity [109, 50], and with some constructions of Statman [121]. Its importance is that in general all expressions definable using the original language are related to themselves; this is the 'fundamental theorem of logical relations' and can usually be proved by induction on the structure of expressions. For example, Sieber has used this to describe a notion of 'sequentiality' in models of the language PCF; by factoring out non-sequential (and hence unused) elements of models, he obtains a model that is fully abstract up to third-order types [115].

Logical relations can be of any arity: we use only unary and binary relations. Traditionally they have been defined over set-based models of languages, in a denotational approach. However for the nu-calculus we begin by developing logical relations at the operational level, in a manner similar to Abramsky's applicative bisimulation [4]. We then go on to cover the denotational side, constructing models that use categories with relations, as introduced by O'Hearn and Tennent [82, 83].

This turns out to be a powerful reasoning method for the nu-calculus, as relations between sets of names capture something of how different expressions use their own private names. Probably the strongest result of the dissertation is that logical relations, whether operational or denotational, are complete for reasoning about contextual equivalence in the nu-calculus up to first-order function types.

4 Outline of Dissertation

The four central chapters of this dissertation are of roughly equal size: Chapter 2 introduces the nu-calculus, Chapter 3 constructs categorical models, Chapter 4 applies logical relations, and Chapter 5 extends all this to a language with store. Chapter 6 concludes. The descriptions below cover the contents of each chapter in more detail.

Chapter 2: The nu-calculus Here we present the nu-calculus, illustrate its behaviour and develop a basic operational reasoning method. We begin with syntax and type structure, and go on to give an operational semantics based on that for Standard ML. We present this in a 'big step' evaluation style and also a 'small step' reduction style, with a proof that the two are equivalent.

We define contextual equivalence for the nu-calculus, and give a collection of examples that illustrate the interaction between higher-order functions and name creation. Contextual equivalence is hard to show directly, and we give a context lemma that simplifies the process; proof of this involves a close analysis of the behaviour of nu-calculus expressions under reduction.

Finally, we describe applicative equivalence, a simpler relation defined by induction over types, and show that it implies contextual equivalence. This provides an operational method for reasoning about expressions of the nu-calculus that is straightforward but not especially powerful.

Chapter 3: Categorical models This chapter falls into two parts: in the first half we describe a metalanguage that captures the properties needed to model the nu-calculus, and in the second we turn these into requirements for a category and give two specific example.

The metalanguage extends Moggi's computational lambda-calculus by adding names and suitable rules for reasoning about them; this allows us to interpret the nu-calculus in a way that respects its operational semantics. We show that the metalanguage can be used to reason about contextual equivalence, with power similar to that of applicative equivalence.

We then detail the construction of a categorical model for the nu-calculus, and explain how this works in two particular cases. The first is a functor category $Set^{\mathcal{I}}$, the second a category $\mathbf{B}G$ of continuous G-sets for a certain group G. We investigate which contextual equivalences these models can and cannot verify: up to second-order types these are the same as for applicative equivalence.

Chapter 4: Logical relations This chapter refines the operational methods of Chapter 2, and the denotational methods of Chapter 3, by the introduction of logical relations that capture how expressions use local names. We define operational logical relations first, and show that they can be used to reason about contextual equivalence through the more general notion of contextual relations. Most importantly, we show that this method is complete up to types of first order; this is a significant improvement over the methods described earlier.

We also present a denotational analogue, using categories with relations to construct a model $\mathcal{P}$ of the nu-calculus that is fully abstract up to first-order types. Just one equivalence from the examples of Chapter 2 remains unverified, and to prove this we develop predicated logical relations, which provide an even finer description of how expressions use their local names.

Chapter 5: A language with store In this chapter we show how the techniques developed for the nu-calculus can be applied to generative features within a larger programming language. The example that we choose is integer references, and we devise a language 'Reduced ML' that combines these with higher-order functions. As the name suggests, Reduced ML is a proper subset of Standard ML, and it has the same operational semantics.

We recapitulate all that was done with the nu-calculus, beginning with a definition of contextual equivalence and an assortment of examples. We look at operational reasoning methods: applicative equivalence and various logical relations. We describe a metalanguage for store that is simply an extension of that for names, and set out the properties required for a categorical model. Remarkably, all the categories built to model the nu-calculus can also be used to interpret Reduced ML.

Although many of the details are omitted, this chapter does illustrate how a thorough understanding of dynamically generated names can make a significant contribution to reasoning about references in Standard ML.

Chapter 6: Conclusion We summarise the results of the dissertation, discuss its relation to other work in this area, and suggest directions for further research.

Chapter 2

The Nu-Calculus

In this chapter we introduce the *nu-calculus*, a small language designed to show the interaction between dynamically generated names and higher-order functions. We give its syntax and describe an operational semantics based on that for Standard ML [62]. We go on to define a notion of equivalence for expressions of the language, based on their observable behaviour, and present some ways to prove examples of this.

The nu-calculus is a typed call-by-value lambda-calculus extended with the notion of a *name*; names have their own type ν, they can be created fresh, passed around and tested for equality. Higher-order functions and booleans are also available, but to ensure termination functions cannot be defined recursively. New names are created in expressions of the form $\nu n.M$, which binds a fresh name to the identifier n and then evaluates M. The use of a call-by-value semantics means that although only the expression M can refer to n explicitly, the new name itself may escape from this scope. For example, $(\lambda x{:}\nu.x = x)(\nu n.n)$ evaluates to $true$, with x bound to a name rather than to $\nu n.n$ itself.

The nu-calculus is equivalent to a fragment of Standard ML. In particular, names correspond to values of type $\mathsf{unit\,ref}$, cells that can only contain the value $()$. The form $\nu n.M$ corresponds to

$$\mathsf{let\ val\ n = ref\ ()\ in\ M\ end}.$$

The operational semantics of the nu-calculus is based on that of ML, with call-by-value (strict) function application and left-to-right evaluation order. In fact the evaluation order turns out to be irrelevant for the nu-calculus, as there are no side-effects. It is still worth taking care over this though, because the storage of values, to be considered in Chapter 5, is sensitive to the order of evaluation.

We present the operational semantics of the nu-calculus in two forms; primarily in a 'big step' evaluation style, but also in a 'small step' reduction style. We prove that the two forms are entirely equivalent, and that the evaluation of expressions always terminates. This is the reason for omitting recursion from the nu-calculus: non-termination would complicate the language without helping us to understand the behaviour of names.

Expressions of the nu-calculus are judged to be equivalent if they can be freely exchanged in any program; we use this as the basis of a notion of *contextual equivalence* for the language. We give a number of examples of expressions that are equivalent or inequivalent. These demonstrate the subtle and complex behaviour that can arise from the combination of names with higher-order functions.

Contextual equivalence is hard to prove directly, and we give a *context lemma* that simplifies the process. The proof of this result requires a close analysis of the behaviour of nu-calculus expressions under reduction.

An alternative approach is to define other relations between expressions and show that they entail contextual equivalence. We describe *applicative equivalence* and *logical equivalence*, both defined by induction over types, and show that they have certain useful properties. In fact they turn out to be equivalent for the nu-calculus, both implying contextual equivalence; in general the reverse implication does not hold, so this is not a complete proof method.

1 Syntax

The syntax of the nu-calculus is based on the simply-typed lambda-calculus. Types are built up from ground types o of *booleans* and ν of *names* by formation of *function types* $\sigma \to \sigma'$. We frequently omit parentheses in types, with $\to$ associating to the right. Each type has an *order*, given by

$$\begin{aligned} \text{Order}(o) \;=\; \text{Order}(\nu) &= 0 \\ \text{Order}(\sigma \to \sigma') &= \max(\text{Order}(\sigma)+1, \text{Order}(\sigma')). \end{aligned}$$

So for example first-order types are all of the form $\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$ where $n > 1$ and $\sigma_i \in \{o, \nu\}$ for $i = 1, \ldots, n$. We shall use σ, τ and decorated variants to range over types.

Expressions of the nu-calculus have the form:

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & | & n & \text{name} \\ & | & \mathit{true} \quad | \quad \mathit{false} & \text{truth values} \\ & | & \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M & \text{conditional} \\ & | & M = M & \text{compare names} \\ & | & \nu n.M & \text{create new name} \\ & | & \lambda x{:}\sigma.M & \text{function abstraction} \\ & | & MM & \text{function application.} \end{array}$$

There are separate infinite supplies of typed variables and names. Function abstraction $\lambda x{:}\sigma.M$ binds the variable x of type σ, and name creation $\nu n.M$ binds the name n. We implicitly identify expressions which only differ in their choice of bound variables and names (α-conversion). An expression is *closed* if it has no free variables; a closed expression may still have free names.

We shall use M to represent general expressions, and B, N, F to suggest expressions of boolean, name and function type respectively. Variables are usually taken from x, y, z, with n for names and variants of s for finite sets of names. A useful abbreviation is new for $\nu n.n$; this is the expression that generates a new name and then immediately returns it.

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

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

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

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

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

Figure 2.1: Rules for assigning types to expressions of the nu-calculus

α-converting M. Simultaneous substitution is also to be capture avoiding; we write this as $M[M_1/x_1, \ldots, M_n/x_n]$, often abbreviated to $M[\vec{M}/\vec{x}]$.

Expressions are given types according to the rules in Figure 2.1. 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.

From now on we shall consider only well-typed expressions. The assignment of types behaves much as we might expect:

Lemma 2.1 *If $s,\Gamma \vdash M : \sigma$ holds then the type σ is unique. Further, if the expression M has free names in s and free variables in Γ, then*

$$s,\Gamma \vdash M : \sigma \quad \iff \quad s \oplus s',\Gamma \oplus \Gamma' \vdash M : \sigma$$

for any s' and Γ'.

Proof Both follow by induction on the structure of M. □

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. 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 at any type σ and for any finite sets s,Γ of names and typed variables.

2 Evaluation Semantics

The operational semantics of the nu-calculus is specified by the inductively defined *evaluation relation* given in Figure 2.2. 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.

The sets s and $s \oplus s'$ can be seen as initial and final *states* of the computation. The arrangement of these states shows the left to right order of evaluation. For example, with the expression $N = N'$, the rules (EQ1) and (EQ2) both evaluate N before N'. The call-by-value nature of the nu-calculus is captured by the choice of a strict (APP) rule. Here the argument M is evaluated to canonical form C before being substituted in the body of the abstraction $\lambda x{:}\sigma.M'$.

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

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

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

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

(NEW) $$\frac{}{s \vdash \mathit{new} \Downarrow_\nu (\{n\})n} \quad n \notin s.$$

The side condition confirms that the name generated by new is fresh, and corresponds precisely to the disjoint union $s \oplus \{n\}$ in the premise of the (LOCAL) rule for evaluating $\nu n.M$. Indeed the rules (LOCAL) and (NEW) are entirely equivalent, and we could formulate the nu-calculus with new as primitive and $\nu n.M$ an abbreviation for $(\lambda n{:}\nu.M)\mathit{new}$. This then makes precise the connection between the relabelling of names bound by ν and of variables bound by λ, mentioned above as necessary to avoid capture. In a setting with new primitive, they are the same thing.

Unfortunately both of the forms $\nu n.M$ and $(\lambda n{:}\nu.M)\mathit{new}$ tend to blur the distinctions between a name, a location bound to a name, and a variable of type ν. Rather than resort to heavy meta-syntactic machinery for a solution, we shall simply choose whichever of new and $\nu n.M$ seems appropriate.

The evaluation of a nu-calculus expression is independent of any unused names:

(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 2.2: Rules for evaluating expressions of the nu-calculus

Lemma 2.2 *For any $M \in \mathrm{Exp}_\sigma(s)$,*

$$s \vdash M \Downarrow_\sigma (s')C \quad \iff \quad s \oplus s'' \vdash M \Downarrow_\sigma (s')C$$

whenever s'' is disjoint from s'.

Proof By induction on the structure of the derivation of the evaluation judgement. □

Evaluation always terminates, and is deterministic up to choice of new names. This might seem obvious, as the nu-calculus has no explicit construction for recursively defined functions. Nevertheless there are many surprising ways to encode recursion in other language features; for example, we shall see later that a mutable store of functions can do this. So it is as well to have a formal proof of termination in the nu-calculus. The proof is our first use of a (unary) logical relation, as described in the introductory chapter. We define the two predicates

$$P_\sigma(s) \subseteq \mathrm{Can}_\sigma(s) \qquad \text{and} \qquad \overline{P}_\sigma(s) \subseteq \mathrm{Exp}_\sigma(s)$$

according to

$$\begin{aligned} P_o(s) &= \{\mathit{true}, \mathit{false}\} \\ P_\nu(s) &= s \\ P_{\sigma\to\tau}(s) &= \{\lambda x{:}\sigma.M \in \mathrm{Can}_{\sigma\to\tau}(s) \mid \forall s', C \in P_\sigma(s\oplus s') \ . \ M[C/x] \in \overline{P}_\tau(s \oplus s')\} \end{aligned}$$

and

$M \in \overline{P}_\sigma(s) \iff$ There are s' and $C \in P_\sigma(s\oplus s')$ such that $s \vdash M \Downarrow_\sigma (s')C$, and these are unique up to renaming the elements of s' and α-conversion of C.

The idea now is to show that P and $\overline{P}$ are both total, and then use the fact that $\overline{P}$ implies termination. To make the induction work, we use an intermediate result on open expressions.

Lemma 2.3 *If $M \in \mathrm{Exp}_\sigma(s,\Gamma)$, where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, and $C_i \in P_{\sigma_i}(s \oplus s')$ for $i = 1, \ldots, n$ and some s', then $M[\vec{C}/\vec{x}] \in \overline{P}_\sigma(s \oplus s')$.*

Proof By induction over the structure of the derivation of $s, \Gamma \vdash M : \sigma$. We consider three example cases:

- Function abstraction. Suppose that the type derivation ends with

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

  and that the C_i are as above. By the induction hypothesis, if $C \in P_\sigma(s \oplus s' \oplus s'')$ for some s'' then $M[\vec{C}/\vec{x}, C/x] \in \overline{P}_\sigma(s \oplus s' \oplus s'')$. But this is precisely the condition for $(\lambda x{:}\sigma.M)[\vec{C}/\vec{x}] \in P_{\sigma\to\sigma'}(s \oplus s')$, from which it follows trivially that $(\lambda x{:}\sigma.M)[\vec{C}/\vec{x}] \in \overline{P}_{\sigma\to\sigma'}(s \oplus s')$.

- Function application. Suppose that the type derivation ends with

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

and that the C_i are as before. Then by the induction hypothesis:

$$s \oplus s' \vdash F[\vec{C}/\vec{x}] \Downarrow_{\sigma \to \sigma'} (s_1)\lambda x{:}\sigma.M'$$

with $\lambda x{:}\sigma.M' \in P_{\sigma \to \sigma'}(s \oplus s' \oplus s_1)$, and

$$s \oplus s' \oplus s_1 \vdash M[\vec{C}/\vec{x}] \Downarrow_{\sigma} (s_2)C$$

with $C \in P_\sigma(s \oplus s' \oplus s_1 \oplus s_2)$. This then means that

$$s \oplus s' \oplus s_1 \oplus s_2 \vdash M'[C/x] \Downarrow_{\sigma'} (s_3)C'$$

with $C' \in P_{\sigma'}(s \oplus s' \oplus s_1 \oplus s_2 \oplus s_3)$. Combining these with the (APP) rule gives

$$s \oplus s' \vdash FM[\vec{C}/\vec{x}] \Downarrow_{\sigma'} (s_1 \oplus s_2 \oplus s_3)C'.$$

Moreover all these evaluations are unique up to renaming and α-conversion, so $FM[\vec{C}/\vec{x}] \in \overline{P}_{\sigma'}(s \oplus s')$ as required.

- Name abstraction. Suppose that the type derivation ends with

$$\frac{s \oplus \{n\}, \Gamma \vdash M : \sigma}{s, \Gamma \vdash \nu n.M : \sigma}$$

and that $C_i \in P_\sigma(s \oplus s')$. Applying the induction hypothesis gives

$$s \oplus \{n\} \oplus s' \vdash M[\vec{C}/\vec{x}] \Downarrow_{\sigma} (s'')C$$

with $C \in P_\sigma(s \oplus \{n\} \oplus s' \oplus s'')$. We can then apply the (LOCAL) rule to obtain

$$s \oplus s' \vdash (\nu n.M)[\vec{C}/\vec{x}] \Downarrow_{\sigma} (\{n\} \oplus s'')C.$$

Again these evaluations are unique up to renaming and α-conversion, so as required $(\nu n.M)[\vec{C}/\vec{x}] \in \overline{P}_\sigma(s \oplus s')$.

The remaining cases are similar. □

Theorem 2.4 (Termination) *If $M \in \mathrm{Exp}_\sigma(s)$ then there are s', C with $s \vdash M \Downarrow_\sigma (s')C$. Moreover, these are unique up to relabelling the elements of s' and α-conversion of C.*

Proof This is Lemma 2.3 applied to the type assertion $s \vdash M : \sigma$. □

Reduction expressions:

$$\begin{array}{rcl} E\langle - \rangle & ::= & \langle - \rangle M \quad | \quad (\lambda x{:}\sigma.M)\langle - \rangle \\ & | & \textit{if } \langle - \rangle \textit{ then } M \textit{ else } M' \quad | \quad \langle - \rangle = N \quad | \quad n = \langle - \rangle \end{array}$$

Redexes:

$$\begin{array}{lll} \textit{if true then } M \textit{ else } M' \to_\sigma M & n = n \ \to_\nu \textit{true} & \\ \textit{if false then } M \textit{ else } M' \to_\sigma M' & n = n' \to_\nu \textit{false} & n \neq n' \end{array}$$

$$(\lambda x{:}\sigma.M)C \to_{\sigma'} M[C/x]$$

Rules:

(REDEXP) $$\frac{M \to_\sigma M'}{E\langle M \rangle \to_{\sigma'} E\langle M' \rangle}$$

(ABSTRACT) $$\frac{M \to_\sigma M'}{\nu n.M \to_\sigma \nu n.M'}$$

(EXTRUDE) $$\frac{}{E\langle \nu n.M \rangle \to_{\sigma'} \nu n.E\langle M \rangle} \quad n \notin \mathit{fn}(E\langle - \rangle)$$

Derived forms:

$$M \to_\sigma^0 M \qquad \frac{M \to_\sigma^n M' \qquad M' \to_\sigma M''}{M \to_\sigma^{n+1} M''}$$

$$M \to_\sigma^* M' \quad \text{if there is some } n \text{ with } M \to_\sigma^n M'$$

Figure 2.3: Step reduction for the nu-calculus

3 Reduction Semantics

The evaluation relation gives a 'big step' operational semantics for the nu-calculus, taking expressions directly to canonical form. We can also give a 'small step' semantics which illustrates how it happens. This uses a *reduction relation* whose elements are of the form

$$M \to_\sigma M'$$

and indicate that the expression M of type σ reduces in a single step to the expression M'. Here $M, M' \in \mathrm{Exp}_\sigma(s)$ for some set of names s. The relation corresponds to Plotkin's *left reduction* $M \underset{V}{\to} N$ for the λ_V-calculus [94].

The rules specifying the reduction relation are in Figure 2.3. This also defines the derived relations

$$M \to^n_\sigma M' \qquad \text{and} \qquad M \to^*_\sigma M'$$

of n-step and finite step reduction respectively. For brevity we use *reduction expressions* to describe some of the rules: these are expressions with a single typed *hole* $\langle - \rangle$, taking one of the forms

$$\begin{array}{rcl} E\langle - \rangle & ::= & \langle - \rangle M \quad | \quad (\lambda x{:}\sigma.M)\langle - \rangle \\ & | & \mathit{if}\ \langle - \rangle\ \mathit{then}\ M\ \mathit{else}\ M' \quad | \quad \langle - \rangle = N \quad | \quad n = \langle - \rangle . \end{array}$$

We write $\langle -{:}\sigma \rangle$ if we wish to make explicit the type of the hole. If M is an expression of the appropriate type then $E\langle M \rangle$ denotes the expression $E\langle - \rangle$ with M replacing the single occurrence of the hole. There is no possibility of name or variable capture. The free names $\mathit{fn}(E\langle - \rangle)$ of a reduction expression are the free names of its subexpressions. It is significant that we use reduction expressions simply as an abbreviation; they could be avoided by replacing (REDEXP) and (EXTRUDE) with ten more specific rules.

A more complex notion is that of a *reduction context*. This is also an expression with a single typed hole, fitting the description

$$\begin{array}{rcl} R\langle - \rangle & ::= & \langle - \rangle \quad | \quad R\langle - \rangle M \quad | \quad (\lambda x{:}\sigma.M)R\langle - \rangle \\ & | & \mathit{if}\ R\langle - \rangle\ \mathit{then}\ M\ \mathit{else}\ M' \quad | \quad R\langle - \rangle = N \quad | \quad n = R\langle - \rangle . \end{array}$$

It is clear that a reduction context is just a nested sequence of reduction expressions. Again there can be no name or variable capture by the reduction context: the hole may be filled by an open expression, but the free variables of such M remain free in $R\langle M \rangle$. For example, we write $\lambda x{:}\sigma.R\langle x \rangle$ for the abstraction with variable x replacing the hole.

Reduction contexts were introduced by Felleisen and Friedman in [19]; their purpose is to identify where in an expression the first reduction lies. Section 6 examines this in detail, so for the moment we observe only that the rules

(RCONTEXT) $$\frac{M \to_\sigma M'}{R\langle M \rangle \to_{\sigma'} R\langle M' \rangle}$$

and

(EXTRUDE$*$) $$\frac{}{R\langle \nu n.M \rangle \to^*_{\sigma'} \nu n.R\langle M \rangle} \quad n \notin \mathit{fn}(R\langle - \rangle)$$

can be derived for any reduction context $R\langle -{:}\sigma\rangle$.

Unlike the evaluation relation, the reduction semantics is not entirely deterministic, as some expressions have more than one reduction available. However reduction is still *confluent*, and this choice makes no difference to the eventual outcome. In fact the relation $\rightarrow$ obeys the *diamond property*:

Lemma 2.5 (Diamond) *If $M \rightarrow_\sigma M_1$ and $M \rightarrow_\sigma M_2$ are valid reductions, then there is some expression M' such that $M_1 \rightarrow_\sigma M'$ and $M_2 \rightarrow_\sigma M'$.*

Proof The base case where an expression has two distinct reductions is

$$\cfrac{\cfrac{M \rightarrow_\sigma M'}{\nu n.M \rightarrow_\sigma \nu n.M'}}{E\langle \nu n.M\rangle \rightarrow_{\sigma'} E\langle \nu n.M'\rangle} \qquad \text{and} \qquad E\langle \nu n.M\rangle \rightarrow_{\sigma'} \nu n.E\langle M\rangle,$$

where $M, M' \in \mathrm{Exp}_\sigma(s)$ and $s \vdash E\langle -{:}\sigma\rangle : \sigma'$ is a reduction expression. These two choices can immediately be reconciled by

$$E\langle \nu n.M'\rangle \rightarrow_{\sigma'} \nu n.E\langle M'\rangle \qquad \text{and} \qquad \cfrac{\cfrac{M \rightarrow_\sigma M'}{E\langle M\rangle \rightarrow_{\sigma'} E\langle M'\rangle}}{\nu n.E\langle M\rangle \rightarrow_{\sigma'} \nu n.E\langle M'\rangle.}$$

All other cases are instances of this within a series of reduction expressions and name abstractions. □

We can avoid this indeterminacy by only allowing reduction under name abstraction at the top level. By always choosing the (EXTRUDE) rule over (ABSTRACT), we obtain a deterministic reduction for any expression; we call it the *standard* reduction. It is possible to enforce this, if we introduce another reduction relation '$\rightarrow\!\!\triangleright$' and replace the (ABSTRACT) rule with

$$\frac{M \rightarrow_\sigma M'}{M \rightarrow\!\!\triangleright_\sigma M'} \qquad \text{and} \qquad \frac{M \rightarrow\!\!\triangleright_\sigma M'}{\nu n.M \rightarrow\!\!\triangleright_\sigma \nu n.M'}.$$

The relation '$\rightarrow\!\!\triangleright$' now follows only the standard reduction sequence.

There is an exact correspondence between the reduction and evaluation semantics, if we consider taking an expression to canonical form. This is the analogue of Plotkin's result relating the evaluation and reduction semantics for λ_V [94, Section 4, Theorem 4]. Define $M \rightarrow^*_\sigma \nu s'.C$ to mean that there is an ordering on the names in $s' = \{n'_1, \dots, n'_k\}$ such that $M \rightarrow^*_\sigma \nu n'_1 \dots \nu n'_k.C$.

Theorem 2.6 *For any closed expressions $M \in \mathrm{Exp}_\sigma(s)$ and $C \in \mathrm{Can}_\sigma(s)$,*

$$s \vdash M \Downarrow_\sigma (s')C \quad \Longleftrightarrow \quad M \rightarrow^*_\sigma \nu s'.C\ .$$

Proof The forward direction follows by induction on the structure of the proof of the evaluation judgement $s \vdash M \Downarrow_\sigma (s')C$. This requires a proof step for each of the rules of Figure 2.2. We give a couple of examples; the others follow a similar format.

- (CAN) The evaluation of a canonical $s \vdash C \Downarrow_\sigma C$ is axiomatic; but so is the reduction $C \rightarrow^0_\sigma C$.

- (COND1) Suppose that $s \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' \Downarrow_\sigma (s')C$, with the last rule of the proof being (COND1). Then we must have $s' = s_1 \oplus s_2$ with

$$s \vdash B \Downarrow_o (s_1)\mathit{true} \qquad \text{and} \qquad s \oplus s_1 \vdash M \Downarrow_\sigma (s_2)C\ .$$

By the induction hypothesis the first of these gives $B \to_o^* \nu s_1.\mathit{true}$ and so

$$\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' \ \to_\sigma^*\ \nu s_1.(\mathit{if}\ \mathit{true}\ \mathit{then}\ M\ \mathit{else}\ M').$$

We have

$$\mathit{if}\ \mathit{true}\ \mathit{then}\ M\ \mathit{else}\ M' \ \to_\sigma\ M$$

and from the second evaluation above, by the induction hypothesis

$$M \ \to_\sigma^*\ \nu s_2.C\ .$$

Putting these together gives

$$\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M' \ \to_\sigma^*\ \nu s_1.\nu s_2.C$$

as required.

For the reverse direction we start by showing that

$$M \to_\sigma M' \quad \& \quad s \vdash M' \Downarrow_\sigma (s')C \quad \Longrightarrow \quad s \vdash M \Downarrow_\sigma (s')C.$$

We do this by induction on the structure of the proof of $M \to_\sigma M'$; again, a couple of cases are enough to show the method.

- Consider the judgements $\mathit{if}\ \mathit{true}\ \mathit{then}\ M\ \mathit{else}\ M' \to_\sigma M$ and $s \vdash M \Downarrow_\sigma (s')C$. We can use the evaluation rules (CAN) and (COND1) to obtain

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

as required.

- Suppose that $FM \to_{\sigma'} F'M$ and $s \vdash F'M \Downarrow_\sigma (s')C'$. The last rules in the proofs of these must have been

$$\frac{F \to_{\sigma\to\sigma'} F'}{FM \to_{\sigma'} F'M}$$

and

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

where $s' = s_1 \oplus s_2 \oplus s_3$. By the induction hypothesis we obtain $s \vdash F \Downarrow_{\sigma\to\sigma'} (s_1)\lambda x{:}\sigma.M'$ and then build

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

as desired.

It now follows that

$$M \rightarrow^n_\sigma M' \quad \& \quad s \vdash M' \Downarrow_\sigma (s')C \quad \Longrightarrow \quad s \vdash M \Downarrow_\sigma (s')C.$$

by induction on the number of steps n. The particular case when the expression M' is of the form $\nu s'.C$, for some ordering of s', gives

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

which is the required result. □

4 Contextual Equivalence

We construct a notion of equivalence for expressions of the nu-calculus, based on their behaviour when used in larger expressions. Informally, two expressions are equivalent if they can be freely exchanged; there is no way in the language itself to distinguish between them. To capture this formally requires a little preliminary work.

Define a *program* to be a closed expression of boolean type. All that we can observe of a program is whether it evaluates to $true$ or $false$; the creation of new names is not directly observable. In calculi with the possibility of non-termination, it is common to use termination as the basic observable. We cannot do this for the nu-calculus, as all expressions evaluate to some canonical form; in particular, no observation at all can be made of expressions of function type without using them in some larger expression.

A *program context* $P\langle\!\langle - \rangle\!\rangle$ is a program with zero or more occurrences of a hole $\langle\!\langle - \rangle\!\rangle$. If M is some expression then $P\langle\!\langle M \rangle\!\rangle$ is the program obtained by substituting M for every occurrence of this hole. Often the notion of a program context is left at that, with the possibility of type clashes and variable capture. We shall be rather more precise and annotate holes to track types and the use of free variables.

A *hole of arity* $\sigma_1, \dots, \sigma_n \rightarrow \sigma$ is an n-place operator added to the nu-calculus, whose arguments must be in canonical form. We have the rule

$$\frac{s, \Gamma \vdash C_1 : \sigma_1 \quad \cdots \quad s, \Gamma \vdash C_n : \sigma_n}{s, \Gamma \vdash \langle\!\langle - : \sigma_1, \dots, \sigma_n \!\rightarrow\! \sigma \rangle\!\rangle (C_1, \dots, C_n) : \sigma} \qquad C_1, \dots, C_n \text{ canonical}$$

where the annotation $\langle\!\langle - : \sigma_1, \dots, \sigma_n \!\rightarrow\! \sigma \rangle\!\rangle$ indicates the arity of the hole. We write $M\langle\!\langle - \rangle\!\rangle$ to denote an expression formed with zero or more occurrences of some hole. Such a hole can be filled by any open expression of type σ whose free variables have types $\sigma_1, \dots, \sigma_n$. That is:

$$\frac{s, \{x_1 : \sigma_1, \dots, x_n : \sigma_n\} \vdash M : \sigma \qquad s, \Gamma \vdash M'\langle\!\langle - : \sigma_1, \dots, \sigma_n \!\rightarrow\! \sigma \rangle\!\rangle : \sigma'}{s, \Gamma \vdash M'\langle\!\langle (x_1{:}\sigma_1, \dots, x_n{:}\sigma_n)M \rangle\!\rangle : \sigma'}$$

where $(x_1{:}\sigma_1, \dots, x_n{:}\sigma_n)M$ is a 'meta-abstraction' that makes explicit the free variables of M. The filled holes are then removed by replacing

$$\langle\!\langle (x_1{:}\sigma_1, \dots, x_n{:}\sigma_n)M \rangle\!\rangle (C_1, \dots, C_n) \quad \text{with} \quad M[C_1/x_1, \dots, C_n/x_n],$$

where the substitution on the right hand side is capture avoiding as usual. It is possible that some of the C_i will themselves contain uses of $\langle\!\langle - \rangle\!\rangle$, and these must be substituted first. Despite this complication, it is still a finite and well behaved procedure.

For brevity we shall generally omit the arity $\sigma_1, \dots, \sigma_n \to \sigma$ of a hole, and abbreviate the substitution $M'\langle\!\langle (x_1{:}\sigma_1, \dots, x_n{:}\sigma_n)M \rangle\!\rangle$ to $M'\langle\!\langle (\vec{x})M \rangle\!\rangle$ or even $M'\langle\!\langle M \rangle\!\rangle$ when M is closed. The arguments $(C_1, \dots, C_n)$ we shall write as $(\vec{C})$ and the *instantiation* $M[C_1/x_1, \dots, C_n/x_n]$ as $M[\vec{C}/\vec{x}]$.

With this machinery the substitution $M'\langle\!\langle (\vec{x})M \rangle\!\rangle$ is certain to be a well-typed expression, without capture of free variables. We can safely relabel bound and even free variables of M and M' without worry. In particular we implicitly identify program contexts $P\langle\!\langle - \rangle\!\rangle$ up to α-conversion. The usual arrangement, allowing the capture of free variables from M, can be simulated by forbidding α-conversion and always using the hole in the form $\langle\!\langle - \rangle\!\rangle(x_1, \dots, x_n)$, where the x_i are the free variables of M.

Now that we have fixed how we substitute an open expression M in a program context $P\langle\!\langle - \rangle\!\rangle$, we can use this to describe the behaviour of M. In calculi where non-termination is possible, it is convenient to do this through a preorder. Here we go directly to an equivalence relation:

Definition 2.7 (Contextual Equivalence) If $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ then the assertion

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

means that for all suitably typed program contexts $P\langle\!\langle - \rangle\!\rangle$ defined over s, and each boolean value $b \in \{\mathit{true}, \mathit{false}\}$,

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

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. When this holds, we say that M_1 and M_2 are *contextually equivalent*. If both s and Γ are empty then we write simply $M_1 \approx_\sigma M_2$.

This is the most general notion of equivalence for nu-calculus expressions that we shall consider, and the one that most interests us; as discussed in the introductory chapter, it combines powerful applications with a sensible and intuitive meaning. Other equivalences are useful if they imply contextual equivalence, and are easier to compute.

5 Examples

Much of the work of this dissertation is concerned with methods for proving contextual equivalence or inequivalence for particular expressions of the nu-calculus. This section gives a summary of results, in increasing order of difficulty of proof.

It is straightforward to show from the definition that contextual equivalence is a congruence:

1. $$s, \Gamma \vdash M_1 \approx_\sigma M_2 \implies s \oplus s', \Gamma \oplus \Gamma' \vdash M'[M_1/x] \approx_{\sigma'} M'[M_2/x]$$

where $M' \in \mathrm{Exp}_{\sigma'}(s \oplus s', \Gamma \oplus \Gamma' \oplus \{x : \sigma\})$.

Unused names are irrelevant, as is the order in which names are generated:

2. $$s, \Gamma \vdash \nu n.M \approx_\sigma M \qquad n \notin \mathit{fn}(M)$$
3. $$s, \Gamma \vdash \nu n.\nu n'.M \approx_\sigma \nu n'.\nu n.M\ .$$

For any finite set of names s we define $\nu s.M$ to be $\nu n_1 \dots \nu n_k.M$ when $s = \{n_1, \dots, n_k\}$ and M when s is empty. By the above, this is unambiguous up to contextual equivalence even though s is unordered.

Evaluation and reduction both respect contextual equivalence:

4. $$s \vdash M \Downarrow_\sigma (s')C \quad \Longrightarrow \quad s \vdash M \approx_\sigma \nu s'.C$$

5. $$M \rightarrow_\sigma M' \quad \Longrightarrow \quad s \vdash M \approx_\sigma M'$$

where $M, M' \in \mathrm{Exp}_\sigma(s)$ and $C \in \mathrm{Can}_\sigma(s \oplus s')$.

In Section 3 we defined the notion of a reduction context $R\langle - \rangle$, an expression with a single typed hole in a position where reduction can occur. We can use this to state some equivalences that rearrange expressions without materially affecting the evaluation order. Suppose that $s, \Gamma \vdash R\langle -{:}\sigma \rangle : \sigma'$ is a reduction context, then:

6. $$s, \Gamma \vdash \quad (\lambda x{:}\sigma.R\langle x \rangle)M \;\approx_{\sigma'}\; R\langle M \rangle$$

7. $$s, \Gamma \vdash \quad R\langle \nu n.M \rangle \;\approx_{\sigma'}\; \nu n.R\langle M \rangle$$

8. $$s, \Gamma \vdash \quad R\langle (\lambda x{:}\sigma''.M)M' \rangle \;\approx_{\sigma'}\; (\lambda x{:}\sigma''.R\langle M \rangle)M'$$

where M and M' have the appropriate types. The first of these is a weak form of β-equivalence; the last is the β_{lift} rule from Sabry and Felleisen's axiomatization of the call-by-value lambda-calculus [114]. Some simple instances of these are:

$$\begin{aligned} s, \Gamma \vdash \quad & M \;\approx_\sigma\; (\lambda x{:}\sigma.x)M \\ s, \Gamma \vdash \quad & \mathit{if}\ (\nu n.B)\ \mathit{then}\ M\ \mathit{else}\ M' \;\approx_\sigma\; \nu n.(\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ M') \\ s, \Gamma \vdash \quad & (\lambda y{:}\tau.Q)((\lambda x{:}\sigma.P)M) \;\approx_{\sigma''}\; (\lambda x{:}\sigma.(\lambda y{:}\tau.Q)P)M. \end{aligned}$$

Function application also satisfies Plotkin's β_v-equivalence [94]: if $C \in \mathrm{Can}_\sigma(s, \Gamma)$ and $M \in \mathrm{Exp}_{\sigma'}(s, \Gamma \oplus \{x : \sigma\})$ then

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

However, general β-equivalence fails because the nu-calculus is call-by-value:

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

The left-hand side here reduces to true and the right-hand side to false. Local name declaration and function abstraction do not in general commute:

11. $$\nu n.\lambda x{:}o.n \;\not\approx_{o \to \nu}\; \lambda x{:}o.\nu n.n\ .$$

These can be distinguished by the context $(\lambda f : o \to \nu\ .\ (f\mathit{true} = f\mathit{true}))\,\langle\!\langle - \rangle\!\rangle$, giving true and false respectively.

Expressions may be contextually equivalent if they differ only in their use of 'private' names:

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

13. $$\nu n.\nu n'.\lambda f{:}\nu{\to}o.(fn = fn') \;\approx_{(\nu \to o) \to o}\; \lambda f{:}\nu{\to}o.\mathit{true}\ .$$

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

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

The idea in (12) is that no external context can supply the private name n. Similarly in (13) no externally produced function can distinguish the private names n and n'.

It is however extraordinarily hard to make precise this notion of privacy, particularly where higher-order functions are involved. The next case shows two expressions which look at first sight to be contextually equivalent for the same reason as those in (13):

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

These are distinguished by the context

$$(\lambda F : (\nu \to o) \to o \ .\ F(\lambda x{:}\nu.F(\lambda y{:}\nu.x = y)))\,\langle\!\langle - \rangle\!\rangle.$$

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

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

15. $$\begin{array}{ll} & \nu n.\nu n'.\lambda f : \nu \to o\ .\ \mathit{if}\ fn = fn' \\ & \qquad \mathit{then}\ (\lambda x{:}\nu.\ \mathit{if}\ x = n\ \ \mathit{then}\ \mathit{true} \\ & \qquad\qquad \mathit{else}\ \mathit{if}\ x = n'\ \mathit{then}\ \mathit{false} \\ & \qquad\qquad \mathit{else}\ fx) \\ & \qquad \mathit{else}\ (\lambda x{:}\nu.\mathit{true}) \\ \not\approx_{(\nu\to o)\to(\nu\to o)} & \lambda f : \nu \to o\ .\ f\ . \end{array}$$

These are distinguished by

$$(\lambda F : (\nu \to o) \to (\nu \to o)\ .\ F(F(\lambda x{:}\nu.\mathit{false}))\mathit{new})\,\langle\!\langle - \rangle\!\rangle.$$

What is happening here is that the two functions differ noticeably only on arguments of type $(\nu \to o)$ that can distinguish n from n'. As both names are private, we cannot construct such a function directly. However when we pass the first expression an argument that we can construct, such as $(\lambda x{:}\nu.\mathit{false})$, we get back one that does distinguish n from n', in this case $(\lambda x{:}\nu.(x = n))$. Although this is externally no different from what we started with (see equivalence (12) above), it is a suitable argument to separate the two original expressions.

Up to contextual equivalence, the only closed expressions of type o are true and false, and the only closed expressions of type ν are the names in the name context and new. Higher types are more complicated, and there are infinitely many operationally distinct closed expressions of type $(\nu \to \nu)$. Consider for each $p \geq 1$ the expression which creates $(p+1)$ local names $n_0, \ldots, n_p$ and then acts as the function which cyclically permutes them, and maps any other name to n_0:

$$\begin{array}{rcl} F_p & = & \nu n_0 \ldots \nu n_p.\lambda x{:}\nu.\ \mathit{if}\ x = n_0\ \mathit{then}\ n_1 \\ & & \qquad \mathit{else}\ \mathit{if}\ x = n_1\ \mathit{then}\ n_2 \\ & & \qquad\qquad \vdots \\ & & \qquad \mathit{else}\ \mathit{if}\ x = n_p\ \mathit{then}\ n_0\ \mathit{else}\ n_0\ . \end{array}$$

Take the test function

$$B_q \;=\; \lambda f : \nu \to \nu\ .\ \nu n\ .\ (f^{(q+2)}(n) = f(n))$$

where $f^{(q+2)}$ is an abbreviation for f iterated $(q+2)$ times. This satisfies

$$\begin{array}{ll} \vdash B_q F_p \Downarrow_o (n_0, \ldots, n_p) \mathit{false} & q \in \{1, \ldots, p-1\} \\ \vdash B_p F_p \Downarrow_o (n_0, \ldots, n_p) \mathit{true} & \end{array}$$

and so can be used to distinguish the various F_p, giving

16. $$F_p \not\approx_{\nu\to\nu} F_{p'} \qquad \text{whenever } p \neq p'.$$

The F_p can be regarded as numerals; we can even define addition by taking

$$\begin{array}{rcl} A & = & \lambda f : \nu \to \nu \, . \, \lambda g : \nu \to \nu \, . \, \nu n \, . \, \lambda x : \nu \, . \, \mathit{if} \; f(fx) = fn \; \mathit{then} \; gx \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{else} \; \mathit{if} \; gx = gn \; \mathit{then} \; fx \; \mathit{else} \; gx \end{array}$$

of type $(\nu \to \nu) \to (\nu \to \nu) \to (\nu \to \nu)$, which satisfies

17. $$AF_pF_q \approx_{\nu\to\nu} F_{p+q}.$$

For example, suppose that F_p cycles the private names $n_0, \ldots, n_p$ and F_q the names $n'_0, \ldots, n'_q$, then AF_pF_q does the same thing over the names $n_0, \ldots, n_{p-1}, n'_0, \ldots, n'_q$. The trick here is that the test $(f(fx) = fn)$ is only true when x is n_{p-1}, while $(gx = gn)$ is true when x is any of $n_0, \ldots, n_{p-1}, n'_q$ or some unknown name. These are then used to select which name to return.

These examples show that the nu-calculus has all the properties we might expect for a language of names, and for a call-by-value lambda-calculus. They also illustrate the subtle and perhaps surprising behaviour that can arise from the interaction between names and higher-order functions.

6 A Context Lemma

While contextual equivalence is the relation on expressions of the nu-calculus that we would like to work with, there are certain problems in doing so. The most obvious is that program contexts $P\langle\!\langle - \rangle\!\rangle$ are far too numerous and varied; a direct proof that two expressions behave similarly in all program contexts is generally too unwieldy to be attempted. In this section we show that a smaller collection of contexts is sufficient. Later we consider other equivalences that are simpler to demonstrate yet imply contextual equivalence

A result showing that it is not necessary to consider all program contexts, but only those having a certain form, is called a *context lemma*, after Milner's result for the simply-typed lambda-calculus [59]. The equivalent result for our system would be that it is only necessary to consider *applicative contexts*, which take the form

$$\langle\!\langle - \rangle\!\rangle (\vec{C}) M_1 \ldots M_n$$

where an expression is instantiated and then applied to some arguments. Milner's context lemma is particularly useful because the types of the M_i are all structurally simpler than that of the hole. This means that various results on contextual equivalence can be proved by induction on the structure of types.

Unfortunately the context lemma in this form is not valid for the nu-calculus. The simplest counter-example is the inequivalence (11) of Section 5 above:

$$\nu n.\lambda x{:}o.n \;\not\approx_{o\to\nu}\; \lambda x{:}o.\nu n.n\ .$$

Both of these expressions, when applied to either $true$ or $false$, return a fresh name. Only by using a context such as $(\lambda f{:}o \to \nu.(f\,true = f\,true))\langle\!\langle - \rangle\!\rangle$ can we detect that the first always returns the same name, while the second repeatedly generates new ones. The inequivalences (15) and (16) also provide counterexamples. However we can give a weaker result that is suitable for the nu-calculus:

Theorem 2.8 (Context Lemma) *Two expressions are contextually equivalent*

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

if and only if for all name sets s', functions $\lambda x{:}\sigma.B \in \mathrm{Can}_{\sigma\to o}(s \oplus s')$, instantiations $[\vec{C}/\vec{x}]$ defined over $(s \oplus s')$, and boolean values $b \in \{true, false\}$,

$$\begin{aligned}&(\,\exists s_1\ .\ s \oplus s' \vdash (\lambda x{:}\sigma.B)M_1[\vec{C}/\vec{x}] \Downarrow_o (s_1)b\,)\\ &\iff\\ &(\,\exists s_2\ .\ s \oplus s' \vdash (\lambda x{:}\sigma.B)M_2[\vec{C}/\vec{x}] \Downarrow_o (s_2)b\,).\end{aligned}$$

We call a context of the form $\nu s'.(\lambda x{:}\sigma.B)(\langle\!\langle - \rangle\!\rangle(\vec{C}))$ an *argument context*, and $(\lambda x{:}\sigma.B)$ a *test function*. Although less dramatic than the usual result, this still narrows down the contexts that we have to consider. In the first place, because the nu-calculus is call-by-value, an argument context only evaluates the contents of its hole to canonical form once. Additionally, in a general context the arguments to the hole can have free variables, or even contain holes themselves. In an argument context the $(\vec{C})$ are closed and have no holes. Unfortunately the technique mentioned above, using a context lemma to prove other results by induction on the structure of types, fails to go through as the type of $(\lambda x{:}\sigma.B)$ is larger than the type of the hole.

Results similar to Theorem 2.8 have been found for other calculi. A. Gordon in his thesis shows that 'experimental order' coincides with 'contextual order' for the language $\mu\nu ML$ [25, Lemma 4.4.7]. Honsell, Mason, Smith and Talcott consider an untyped lambda-calculus extended with storage cells and show that to establish 'operational equivalence' it is enough to consider all 'closed instantiations of use' (*ciu*) of an expression [30, §2.3.2].

For expressions with no free variables, we can further simplify the range of contexts required:

Corollary 2.9 *Two closed expressions are contextually equivalent $s \vdash M_1 \approx_\sigma M_2$ if and only if for all test functions $\lambda x{:}\sigma.B \in \mathrm{Can}_{\sigma\to o}(s)$ and all $b \in \{true, false\}$,*

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

Proof Observe that for any test function $(\lambda x{:}\sigma.B) \in \mathrm{Can}_{\sigma\to o}(s \oplus s')$ applied to a closed expression $M \in \mathrm{Exp}_\sigma(s)$,

$$s \oplus s' \vdash (\lambda x{:}\sigma.B)M \Downarrow_o (s'')b \quad\iff\quad s \vdash (\lambda x{:}\sigma.\nu s'.B)M \Downarrow_o (s' \oplus s'')b.$$

So with no free variables to instantiate, the argument context $\nu s'.(\lambda x{:}\sigma.B)\langle\!\langle - \rangle\!\rangle$ from Theorem 2.8 can be replace by the test function $(\lambda x{:}\sigma.\nu s'.B)\langle\!\langle - \rangle\!\rangle$. □

Before we can prove the context lemma, we have to examine more closely the process of reduction. In Section 3 we defined reduction expressions $E\langle - \rangle$ and reduction contexts $R\langle - \rangle$ to be expressions with a single typed hole, taking the following forms:

$$\begin{array}{rcl} E\langle - \rangle & ::= & \langle - \rangle M \quad | \quad (\lambda x{:}\sigma.M)\langle - \rangle \\ & | & \mathit{if}\ \langle - \rangle\ \mathit{then}\ M\ \mathit{else}\ M' \quad | \quad \langle - \rangle = N \quad | \quad n = \langle - \rangle \\ R\langle - \rangle & ::= & \langle - \rangle \quad | \quad R\langle - \rangle M \quad | \quad (\lambda x{:}\sigma.M)R\langle - \rangle \\ & | & \mathit{if}\ R\langle - \rangle\ \mathit{then}\ M\ \mathit{else}\ M' \quad | \quad R\langle - \rangle = N \quad | \quad n = R\langle - \rangle. \end{array}$$

Any reduction context can be represented uniquely as a nested sequence of reduction expressions. A *redex* is an expression taking one of the forms

$$\begin{array}{c} (\lambda x{:}\sigma.M)C \quad | \quad \mathit{if\ true\ then}\ M\ \mathit{else}\ M' \quad | \quad \mathit{if\ false\ then}\ M\ \mathit{else}\ M' \\ | \quad n = n' \quad | \quad n = n. \end{array}$$

As shown in Figure 2.3, all of these have immediate reductions, regardless of the nature of the subexpressions M, M' or C. We can use this classification to break down closed expressions of the nu-calculus and identify their first reduction step.

Lemma 2.10

1. *Each closed expression of the nu-calculus is just one of the following:*
   - *in canonical form*
   - *a redex*
   - *a name abstraction*
   - *a reduction expression with the hole $\langle - \rangle$ filled by some non-canonical expression.*
2. *Any $M \in \mathrm{Exp}_\sigma(s)$ decomposes uniquely as one of the following:*

$$\begin{array}{ll} \nu s'.C & C\ \textit{canonical} \\ \nu s'.R\langle E\langle \nu n.M' \rangle \rangle & E\langle - \rangle\ \textit{a reduction expression} \\ \nu s'.R\langle M' \rangle & M'\ \textit{a redex},\ M' \to_\sigma M'' \end{array}$$

   where s' is some ordered set of names, possibly empty, and $R\langle - \rangle$ is a reduction context.

Proof Item (1) is straightforward, giving consideration to each of the forms of nu-calculus expressions. For example:

- A lambda abstraction $(\lambda x{:}\sigma.M)$ is always in canonical form.
- An expression of the form FM is a redex if F and M are both in canonical form. Otherwise, FM is a reduction expression with a non-canonical in $\langle - \rangle$, as either $F\langle M \rangle$ or $\langle F \rangle M$ according as F is in canonical form or not, respectively.

To obtain (2) we apply (1) repeatedly, using the fact that a succession of nested reduction expressions is precisely a reduction context. □

For any $M \in \mathrm{Exp}_\sigma(s)$ this result determines its unique standard reduction as:

$$\begin{array}{rcl} \nu s'.C & & \text{none} \\ \nu s'.R\langle E\langle \nu n.M'\rangle\rangle & \to_\sigma & \nu s'.R\langle \nu n.E\langle M'\rangle\rangle \\ \nu s'.R\langle M'\rangle & \to_\sigma & \nu s'.R\langle M''\rangle. \end{array}$$

The next lemma is a weak β-rule that we need later. It is related to equivalence (6) from Section 5 above.

Lemma 2.11 *Suppose that $s \vdash R\langle -{:}\sigma\rangle : \sigma'$ is a reduction context and $M \in \mathrm{Exp}_\sigma(s)$.*

1. *If $M \to_\sigma^* \nu s'.C$ where C is in canonical form, then*

$$R\langle M\rangle \ \to_{\sigma'}^* \ \nu s'.R\langle C\rangle \qquad \textit{and} \qquad (\lambda x{:}\sigma.R\langle x\rangle)M \ \to_{\sigma'}^* \ \nu s'.R\langle C\rangle \ .$$

2. *For canonical form C,*

$$s \vdash R\langle M\rangle \Downarrow_{\sigma'} (s')C \quad \Longleftrightarrow \quad s \vdash (\lambda x{:}\sigma.R\langle x\rangle)M \Downarrow_{\sigma'} (s')C.$$

Proof The first part of (1) is derived by applying the (REDEXP) and (EXTRUDE) rules repeatedly, while for the second part we combine

$$(\lambda x{:}\sigma.R\langle x\rangle)M \ \to_{\sigma'}^* \ \nu s'.((\lambda x{:}\sigma.R\langle x\rangle)C)$$

with

$$(\lambda x{:}\sigma.R\langle x\rangle)C \ \to_{\sigma'} \ R\langle C\rangle.$$

For (2), suppose that $s \vdash M \Downarrow_\sigma (s')C'$. Then by (1), both $R\langle M\rangle$ and $(\lambda x{:}\sigma.R\langle x\rangle)M$ reduce to $\nu s'.R\langle C'\rangle$, from which the result follows. □

To understand the behaviour of expressions in context, we must go a stage further. Define an *extended expression* $\bar{M}$ to be what we have previously termed a context: an expression of the nu-calculus with a hole $\langle\!\langle - : \sigma_1, \ldots, \sigma_n \to \sigma\rangle\!\rangle$ appearing zero or more times. We also need extended versions of canonical form, reduction expression, reduction context and redex; the details are given in Figure 2.4.

To determine the complexity of an extended expression, we use a measure for holes that takes account of their nesting in arguments for other holes. Define the *hole count* for an extended expression to be a list of the number of holes at each depth of nesting, deepest first. For example the extended expression

$$\textit{if}\ \langle\!\langle - : o, \nu \to o\rangle\!\rangle(C_1, C_2)\ \textit{then true else}\ \langle\!\langle -\rangle\!\rangle(\langle\!\langle -\rangle\!\rangle(C_3, C_4), C_5)$$

has a hole count $[1, 2]$. Hole counts are ordered by list length and then lexicographically, for example:

$$[4] \sqsubseteq [1, 3] \sqsubseteq [3, 1] \sqsubseteq [1, 1, 1].$$

The set of hole counts is well ordered: there are no strictly decreasing infinite sequences. Another way to look at this ordering is to attach weights to holes, with nested holes being infinitely heavier.

We can break down extended expressions as we did ordinary expressions, using an extended form of Lemma 2.10.

Extended expressions:

$$
\begin{array}{rcl}
\bar{M} & ::= & \langle\!\langle - \rangle\!\rangle \text{ hole} \quad | \quad x \text{ variable} \quad | \quad n \text{ name} \\
& | & \mathit{true} \quad | \quad \mathit{false} \quad | \quad \mathit{if}\ \bar{M}\ \mathit{then}\ \bar{M}\ \mathit{else}\ \bar{M} \\
& | & \nu n.\bar{M} \quad | \quad \bar{M} = \bar{M} \\
& | & \lambda x{:}\sigma.\bar{M} \quad | \quad \bar{M}\bar{M}
\end{array}
$$

Extended canonical form:

$$
\bar{C} ::= x \quad | \quad n \quad | \quad \mathit{true} \quad | \quad \mathit{false} \quad | \quad \lambda x{:}\sigma.\bar{M}
$$

Extended reduction expressions:

$$
\begin{array}{rcl}
\bar{E}\langle - \rangle & ::= & \langle - \rangle \bar{M} \quad | \quad (\lambda x{:}\sigma.\bar{M})\langle - \rangle \\
& | & \mathit{if}\ \langle - \rangle\ \mathit{then}\ \bar{M}\ \mathit{else}\ \bar{M}' \quad | \quad \langle - \rangle = \bar{N} \quad | \quad n = \langle - \rangle
\end{array}
$$

Extended reduction contexts:

$$
\begin{array}{rcl}
\bar{R}\langle - \rangle & ::= & \langle - \rangle \quad | \quad \bar{R}\langle - \rangle \bar{M} \quad | \quad (\lambda x{:}\sigma.\bar{M})\bar{R}\langle - \rangle \\
& | & \mathit{if}\ \bar{R}\langle - \rangle\ \mathit{then}\ \bar{M}\ \mathit{else}\ \bar{M}' \quad | \quad \bar{R}\langle - \rangle = \bar{N} \quad | \quad n = \bar{R}\langle - \rangle
\end{array}
$$

Extended redexes:

$$
\begin{array}{lll}
\mathit{if}\ \mathit{true}\ \mathit{then}\ \bar{M}\ \mathit{else}\ \bar{M}' \to_\sigma \bar{M} & n = n \to_\nu \mathit{true} & \\
\mathit{if}\ \mathit{false}\ \mathit{then}\ \bar{M}\ \mathit{else}\ \bar{M}' \to_\sigma \bar{M}' & n = n' \to_\nu \mathit{false} & n \neq n'
\end{array}
$$

$$
(\lambda x{:}\sigma.\bar{M})\bar{C} \to_{\sigma'} \bar{M}[\bar{C}/x]
$$

Figure 2.4: The extended nu-calculus

Lemma 2.12

1. *Each closed extended expression is just one of the following:*
   - *an application of the hole* $\langle\!\langle - \rangle\!\rangle(\bar{C}_1, \ldots, \bar{C}_n)$
   - *in extended canonical form*
   - *an extended redex*
   - *a name abstraction of some extended expression*
   - *an extended reduction expression with the hole* $\langle - \rangle$ *filled by some non-canonical extended expression.*

2. *If* $s \vdash \bar{M} : \sigma$ *is some closed extended expression, then it takes exactly one of the following forms:*

$$\begin{array}{ll} \nu s'.\bar{C} & \bar{C} \text{ an extended canonical} \\ \nu s'.\bar{R}\langle\langle\!\langle - \rangle\!\rangle(\vec{\bar{C}})\rangle & \\ \nu s'.\bar{R}\langle\bar{E}\langle\nu n.\bar{M}'\rangle\rangle & \bar{E}\langle - \rangle \text{ an extended reduction expression} \\ \nu s'.\bar{R}\langle\bar{M}'\rangle & \bar{M}' \text{ an extended redex, } \bar{M}' \to_\sigma \bar{M}'' \end{array}$$

   where s' *is some ordered set of names and* $\bar{R}\langle - \rangle$ *is an extended reduction context.*

Proof As for Lemma 2.10, part (1) follows by consideration of the structure of the extended expression, while (2) comes by repeated application of (1). □

In part (2) above, suppose that $M_0 \in \mathrm{Exp}_\sigma(s, \Gamma)$ is some open expression suitable to fill the hole in $\bar{M}$. Then in the last two cases the standard reduction for $\bar{M}\langle\!\langle(\vec{x})M_0\rangle\!\rangle$ is independent of M_0:

$$\begin{array}{rcl} \nu s'.\bar{R}\langle\bar{E}\langle\nu n.\bar{M}'\rangle\rangle\langle\!\langle(\vec{x})M_0\rangle\!\rangle & \to_\sigma & \nu s'.\bar{R}\langle\nu n.\bar{E}\langle\bar{M}'\rangle\rangle\langle\!\langle(\vec{x})M_0\rangle\!\rangle \\ \nu s'.\bar{R}\langle\bar{M}'\rangle\langle\!\langle(\vec{x})M_0\rangle\!\rangle & \to_\sigma & \nu s'.\bar{R}\langle\bar{M}''\rangle\langle\!\langle(\vec{x})M_0\rangle\!\rangle. \end{array}$$

This pinpoints the nature of reduction sufficiently for us to prove the context lemma:

Proof of Theorem 2.8 The 'only if' direction is immediate as an argument context

$$\nu s'.(\lambda x{:}\sigma.B)(\langle\!\langle - \rangle\!\rangle(\vec{C}))$$

is clearly a restricted form of context. For the 'if' direction, suppose that we have two expressions $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ satisfying

$$\begin{array}{l} (\,\exists s_1 \,.\, s \oplus s' \vdash (\lambda x{:}\sigma.B)M_1[\vec{C}/\vec{x}] \Downarrow_o (s_1)b\,) \\ \iff \\ (\,\exists s_2 \,.\, s \oplus s' \vdash (\lambda x{:}\sigma.B)M_2[\vec{C}/\vec{x}] \Downarrow_o (s_2)b\,) \end{array}$$

for all suitable s', $(\lambda x{:}\sigma.B)$, $(\vec{C})$ and b. By symmetry it is enough to show that for any extended program $s \vdash \bar{P} : o$ and boolean b that

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

We prove this by a double induction over the length of the left-hand reduction, under the semantics of Section 3, and the hole count of $\bar{P}$.

By Lemma 2.12(2) there are four possible forms for $\bar{P}$. In the simplest case, it is a name abstraction of some extended canonical, necessarily *true* or *false*, and the result is immediate. If $\bar{P} = \nu s'.\bar{R}\langle\bar{E}\langle\nu n.\bar{M}\rangle\rangle$ then as we saw above the first reduction step is the same however the hole $\langle\!\langle - \rangle\!\rangle$ is filled. So if we set $\bar{P}' = \nu s'.\bar{R}\langle\nu n.\bar{E}\langle\bar{M}'\rangle\rangle$ then

$$\bar{P}\langle\!\langle(\vec{x})M_1\rangle\!\rangle \to_o \bar{P}'\langle\!\langle(\vec{x})M_1\rangle\!\rangle \qquad \text{and} \qquad \bar{P}\langle\!\langle(\vec{x})M_2\rangle\!\rangle \to_o \bar{P}'\langle\!\langle(\vec{x})M_2\rangle\!\rangle.$$

Now $\bar{P}'\langle\!\langle(\vec{x})M_1\rangle\!\rangle$ will have a shorter reduction to b, and the result follows by the induction hypothesis. A similar approach applies if $\bar{P} = \nu s'.\bar{R}\langle\bar{M}'\rangle$ where $\bar{M}'$ is an extended redex.

The final possibility is that $\bar{P} = \nu s'.\bar{R}\langle\langle\!\langle - \rangle\!\rangle(\vec{\bar{C}})\rangle$. In this case we fill this particular occurrence of the hole $\langle\!\langle - \rangle\!\rangle$ and consider the extended expression $\nu s'.\bar{R}\langle M_1[\vec{\bar{C}}/\vec{x}]\rangle$, which has a lower hole count than $\bar{P}$. We have

$$\bar{P}\langle\!\langle(\vec{x})M_1\rangle\!\rangle \;=\; \nu s'.\bar{R}\langle M_1[\vec{\bar{C}}/\vec{x}]\rangle\langle\!\langle(\vec{x})M_1\rangle\!\rangle$$

and use this to reason as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\exists s_1 \,.\, s \vdash \bar{P}\langle\!\langle(\vec{x})M_1\rangle\!\rangle \Downarrow_o (s_1)b} \\
\iff & \exists s_1 \,.\, s \vdash \nu s'.\bar{R}\langle M_1[\vec{\bar{C}}/\vec{x}]\rangle\langle\!\langle(\vec{x})M_1\rangle\!\rangle \Downarrow_o (s_1)b \\
\implies & \exists s_3 \,.\, s \vdash \nu s'.\bar{R}\langle M_1[\vec{\bar{C}}/\vec{x}]\rangle\langle\!\langle(\vec{x})M_2\rangle\!\rangle \Downarrow_o (s_3)b \\
\iff & \exists s_3 \,.\, s \oplus s' \vdash ((\lambda x{:}\sigma.\bar{R}\langle x\rangle)(M_1[\vec{\bar{C}}/\vec{x}]))\langle\!\langle(\vec{x})M_2\rangle\!\rangle \Downarrow_o (s_3)b \\
\implies & \exists s_2 \,.\, s \oplus s' \vdash ((\lambda x{:}\sigma.\bar{R}\langle x\rangle)(M_2[\vec{\bar{C}}/\vec{x}]))\langle\!\langle(\vec{x})M_2\rangle\!\rangle \Downarrow_o (s_2)b \\
\iff & \exists s_2 \,.\, s \vdash \nu s'.\bar{R}\langle M_2[\vec{\bar{C}}/\vec{x}]\rangle\langle\!\langle(\vec{x})M_2\rangle\!\rangle \Downarrow_o (s_2)b \\
\iff & \exists s_2 \,.\, s \vdash \bar{P}\langle\!\langle(\vec{x})M_2\rangle\!\rangle \Downarrow_o (s_2)b.
\end{array}$$

Here we use in turn induction on the hole count, Lemma 2.11(2), the original hypothesis of equality under test functions, and Lemma 2.11(2) again. □

7 Applicative Equivalence

In this section we define two notions of equivalence for expressions of the nu-calculus, each with particular good properties. We then show that these are in fact the same relation, and that they imply contextual equivalence. This gives a technique sufficient to prove all but two of the examples in Section 5.

Abramsky introduced the notion of *applicative bisimulation* for the untyped lambda-calculus, based on bisimulation of labelled transition systems [4]. Loosely, one lambda-expression simulates another if it is no less defined and behaves similarly at all arguments. Howe's relation $\leq$ is a similar construction for lazy computation systems [31]. A. Gordon's *applicative similarity* is a version of this for a typed lambda-calculus [25, §4.5]. As usual, because there is no non-termination in the nu-calculus, we work with an equivalence, rather than a preorder.

Definition 2.13 (Applicative Equivalence) We define the 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:

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

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

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*.* We can extend the relation to open expressions: if $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ then we define

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

Lemma 2.14 *If* $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ *then*

$$s, \Gamma \vdash M_1 \sim_\sigma M_2 \quad \iff \quad s \oplus s', \Gamma \oplus \Gamma' \vdash M_1 \sim_\sigma M_2$$

for any s', Γ'.

Proof Induction on the structure of the type σ gives the result for closed expressions, and the general result follows. □

Lemma 2.15 *Applicative equivalence is an equivalence relation.*

Proof Also by induction on the structure of types. □

Applicative equivalence is only a useful relation if we can show that it implies contextual equivalence; however, it is well known that a direct proof is problematic. Abramsky uses the technique of *domain logic* to solve this [3]. Howe defines an auxiliary relation, roughly the congruence closure of applicative simulation, and shows that this satisfies the same defining properties; this is also the approach taken by Gordon [31, 25].

Because the nu-calculus is simply typed, we can use an original and much simpler method based on logical relations. As described in Chapter 1, these were introduced by Plotkin as a tool to show the undefinability of certain elements in models of the simply-typed lambda-calculus [96]. Our logical relation remains entirely in the syntax of the nu-calculus; nevertheless it keeps the basic idea that functions are related if they take related arguments to related results.

*This is a different relation to the applicative equivalence of [91, Definition 13] and [92, Definition 3.4] which (rather unfortunately) turns out not to be an equivalence at all.

Definition 2.16 (Logical Equivalence) We define the two relations $s \vdash C_1 \simeq^{can}_{\sigma} C_2$ for $C_1, C_2 \in \mathrm{Can}_{\sigma}(s)$ and $s \vdash M_1 \simeq^{exp}_{\sigma} M_2$ for $M_1, M_2 \in \mathrm{Exp}_{\sigma}(s)$ inductively over the structure of the type σ, according to:

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

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

Again $\simeq^{exp}_{\sigma}$ and $\simeq^{can}_{\sigma}$ coincide on canonicals and we write $\simeq_{\sigma}$ indiscriminately, calling the relation *logical equivalence*. We extend the relation to open expressions; for $M_1, M_2 \in \mathrm{Exp}_{\sigma}(s, \Gamma)$ with $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ we take

$$
\begin{aligned}
s, \Gamma \vdash M_1 \simeq_{\sigma} M_2 \quad \iff \quad &\forall s', C_{ij} \in \mathrm{Can}_{\sigma_j}(s \oplus s') \quad i = 1,2 \quad j = 1, \ldots, n \ . \\
&(\&_{j=1}^{n} \ . \ s \oplus s' \vdash C_{1j} \simeq_{\sigma_j} C_{2j}) \\
&\implies s \oplus s' \vdash M_1[\vec{C_1}/\vec{x}] \simeq_{\sigma} M_2[\vec{C_2}/\vec{x}].
\end{aligned}
$$

Lemma 2.17 *If $M_1, M_2 \in \mathrm{Exp}_{\sigma}(s, \Gamma)$ then*

$$
s, \Gamma \vdash M_1 \simeq_{\sigma} M_2 \quad \iff \quad s \oplus s', \Gamma \oplus \Gamma' \vdash M_1 \simeq_{\sigma} M_2
$$

for any s', Γ'.

Proof Induction on the structure of the type σ gives the result for closed expressions, and the general result follows. An irritating detail is the necessity to show that there are at least some related canonical expressions of each type, which is also shown by induction over types. □

Unlike applicative equivalence, it is not immediately obvious that logical equivalence is reflexive, transitive or symmetric. However it is a *congruence*; it is preserved by all the rules for forming expressions of the nu-calculus. From this it follows that logical equivalence is reflexive and that it implies contextual equivalence.

Proposition 2.18 *Logical equivalence is a congruence.*

Proof Definition 2.16 and the evaluation rules of Figure 2.2 are enough to show that each of the expression-forming rules of Figure 2.1 preserves logical equivalence. □

Proposition 2.19 *Logical equivalence is reflexive:*

$$
s, \Gamma \vdash M \simeq_{\sigma} M \qquad M \in \mathrm{Exp}_{\sigma}(s, \Gamma).
$$

Proof By induction on the structure of the expression M, using Proposition 2.18. □

Proposition 2.20 *Logical equivalence implies contextual equivalence:*

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

Proof By Proposition 2.19, any test function is related to itself $s \oplus s' \vdash \lambda x{:}\sigma.B \simeq_{\sigma \to o} \lambda x{:}\sigma.B$. The result then follows from the Context Lemma (Theorem 2.8) and the definition of logical equivalence at function types and booleans. □

Applicative and logical equivalence are clearly very close, with the only difference being in the treatment of values of function type. Logical equivalence places the stronger constraint here, so it is not too surprising that it implies applicative equivalence; less expected is the result that the reverse implication also holds.

Lemma 2.21 *If* $s, \Gamma \vdash M_1 \simeq_\sigma M_2$ *then* $s, \Gamma \vdash M_1 \sim_\sigma M_2$.

Proof We show each of the following:

1. $s \vdash C_1 \simeq_\sigma^{can} C_2 \implies s \vdash C_1 \sim_\sigma^{can} C_2$.
2. $s \vdash M_1 \simeq_\sigma^{exp} M_2 \implies s \vdash M_1 \sim_\sigma^{exp} M_2$.
3. $s, \Gamma \vdash M_1 \simeq_\sigma M_2 \implies s, \Gamma \vdash M_2 \sim_\sigma M_2$.

The first two are proved by mutual induction over the structure of σ. Case (1) at ground types is immediate. For function types, suppose that $s \vdash \lambda x{:}\sigma.M_1 \simeq_{\sigma \to \sigma'}^{can} \lambda x{:}\sigma.M_2$ and $C \in \text{Can}_\sigma(s \oplus s')$. By Proposition 2.19 above we have $s \oplus s' \vdash C \simeq_\sigma C$, and so $s \oplus s' \vdash M_1[C/x] \simeq_{\sigma'}^{exp} M_2[C/x]$. Applying the induction hypothesis gives $s \oplus s' \vdash M_1[C/x] \sim_{\sigma'}^{exp} M_2[C/x]$ which confirms that $s \vdash \lambda x{:}\sigma.M_1 \sim_{\sigma \to \sigma'}^{can} \lambda x{:}\sigma.M_2$.

For (2) suppose that $s \vdash M_1 \simeq_\sigma^{exp} M_2$. We must have $s \vdash M_1 \Downarrow_\sigma (s_1)C_1$ and $s \vdash M_2 \Downarrow_\sigma (s_2)C_2$ with $s \oplus (s_1 \cup s_2) \vdash C_1 \simeq_\sigma^{can} C_2$. By (1) this gives $s \oplus (s_1 \cup s_2) \vdash C_1 \sim_\sigma^{can} C_2$ and so $s \vdash M_1 \sim_\sigma^{exp} M_2$ as required.

Finally, for (3) suppose that $s, \Gamma \vdash M_1 \simeq_\sigma M_2$ and $C_i \in \text{Can}_{\sigma_i}(s \oplus s')$ for some s' and $i = 1, \ldots, n$. By Proposition 2.19 each $s \vdash C_i \simeq_{\sigma_i} C_i$ and so $s \oplus s' \vdash M_1[\vec{C}/\vec{x}] \simeq_\sigma M_2[\vec{C}/\vec{x}]$. By (2) then $s \oplus s' \vdash M_1[\vec{C}/\vec{x}] \sim_\sigma M_2[\vec{C}/\vec{x}]$ and hence $s, \Gamma \vdash M_1 \sim_\sigma M_2$ as desired. □

Lemma 2.22 *If* $s, \Gamma \vdash M_1 \sim_\sigma M_2$ *then* $s, \Gamma \vdash M_1 \simeq_\sigma M_2$.

Proof We show each of the following:

1. $s \vdash C_1 \sim_\sigma^{can} C_2 \implies s \vdash C_1 \simeq_\sigma^{can} C_2$.
2. $s \vdash M_1 \sim_\sigma^{exp} M_2 \implies s \vdash M_1 \simeq_\sigma^{exp} M_2$.
3. $s, \Gamma \vdash M_1 \sim_\sigma M_2 \implies s, \Gamma \vdash M_2 \simeq_\sigma M_2$.

Again the first two are proved by mutual induction over the structure of σ. Implication (1) at ground types is immediate. Suppose now that $s \vdash \lambda x{:}\sigma.M_1 \sim_{\sigma \to \sigma'}^{can} \lambda x{:}\sigma.M_2$ and that we have some $C_1, C_2 \in \text{Can}_\sigma(s \oplus s')$ with $s \oplus s' \vdash C_1 \simeq_\sigma^{can} C_2$. The definition of applicative equivalence at function types gives

$$s \oplus s' \vdash M_1[C_1/x] \sim_{\sigma'}^{exp} M_2[C_1/x]$$

while Proposition 2.19 for $s \oplus s', \{x : \sigma\} \vdash M_2 : \sigma'$ gives

$$s \oplus s' \vdash M_2[C_1/x] \simeq^{exp}_{\sigma'} M_2[C_2/x]$$

which by Lemma 2.21 implies

$$s \oplus s' \vdash M_2[C_1/x] \sim^{exp}_{\sigma'} M_2[C_2/x].$$

Now $\sim^{exp}_{\sigma'}$ is transitive, so

$$s \oplus s' \vdash M_1[C_1/x] \sim^{exp}_{\sigma'} M_2[C_2/x]$$

and the induction hypothesis gives $s \oplus s' \vdash M_1[C_1/x] \simeq^{exp}_{\sigma'} M_2[C_2/x]$, from which we obtain $s \vdash \lambda x{:}\sigma.M_1 \simeq^{can}_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2$ as required.

For (2) suppose that $s \vdash M_1 \sim^{exp}_{\sigma} M_2$. We must have $s \vdash M_1 \Downarrow_\sigma (s_1)C_1$ and $s \vdash M_2 \Downarrow_\sigma (s_2)C_2$ with $s \oplus (s_1 \cup s_2) \vdash C_1 \sim^{can}_{\sigma} C_2$. By (1) then $s \oplus (s_1 \cup s_2) \vdash C_1 \simeq^{can}_{\sigma} C_2$ and so $s \vdash M_1 \simeq^{exp}_{\sigma} M_2$.

Finally, for (3) suppose $s, \Gamma \vdash M_1 \sim_\sigma M_2$ and that we have $C_{ij} \in \mathrm{Can}_{\sigma_j}(s \oplus s')$ with $s \oplus s' \vdash C_{1j} \simeq_{\sigma_j} C_{2j}$ for $j = 1, \ldots, n$ and some s'. Then

$$s \oplus s' \vdash M_1[\vec{C_1}/\vec{x}] \sim^{exp}_{\sigma} M_2[\vec{C_1}/\vec{x}]$$

and by Proposition 2.19, $s, \Gamma \vdash M_2 \simeq_\sigma M_2$ from which

$$s \oplus s' \vdash M_2[\vec{C_1}/\vec{x}] \simeq^{exp}_{\sigma} M_2[\vec{C_2}/\vec{x}],$$

giving

$$s \oplus s' \vdash M_2[\vec{C_1}/\vec{x}] \sim^{exp}_{\sigma} M_2[\vec{C_2}/\vec{x}],$$

by Lemma 2.21. Transitivity of $\sim^{exp}_{\sigma}$ gives

$$s \oplus s' \vdash M_1[\vec{C_1}/\vec{x}] \sim^{exp}_{\sigma} M_2[\vec{C_2}/\vec{x}]$$

which by (2) implies $s \oplus s' \vdash M_1[\vec{C_1}/\vec{x}] \simeq^{exp}_{\sigma} M_2[\vec{C_2}/\vec{x}]$. Thus $s, \Gamma \vdash M_1 \simeq_\sigma M_2$ as required. □

Theorem 2.23 *Applicative and logical equivalence are the same relation; it is an equivalence, a congruence and implies contextual equivalence.*

Proof We simply combine Lemma 2.21, Lemma 2.22, Lemma 2.15, Proposition 2.18 and Proposition 2.20. □

Applicative equivalence verifies examples (2)–(9) of Section 5: these concern unused names, order of name generation, evaluation, reduction, rearrangement around reduction contexts and β_v-equivalence. It also confirms the equivalence $AF_pF_q \approx_{\nu\to\nu} F_{p+q}$ that represents addition in example (17). However it is unable to capture the notion of 'private' names, and does not prove examples (12) or (13).

The difficulty is that the definition of applicative equivalence at function types quantifies over every possible argument, and an external context may be unable to construct all of these. The refinements of logical equivalence described in Chapter 4 begin to tackle this problem.

Chapter 3

Categorical Models

Consideration of the nu-calculus has so far been entirely operational; we now develop a denotational semantics for the language, using category theory. This provides abstract models for the behaviour of nu-calculus expressions, and further methods for reasoning about contextual equivalence. A particular feature is that we follow Moggi in using a categorical *strong monad* to encapsulate the notion of 'computation' [67].

This chapter falls into two parts. In the first three sections we describe a *metalanguage*, based on Moggi's computational lambda-calculus [66, 68, 90]. This captures the general properties required for a model of the nu-calculus. In the last four sections we turn these into conditions for a categorical model, and give two specific examples. This two stage technique of

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

makes the construction simpler, and allows us to build more than one categorical model on the same foundations. Both parts are significant; the metalanguage serves as the internal language of the category, while the existence of categorical models proves the consistency of the metalanguage.

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

We give an interpretation of the nu-calculus in the metalanguage, and show that it respects the operational semantics. The translation is adequate, so we can use the metalanguage to reason about contextual equivalence; it is also fully abstract for expressions of ground type. Further, if two expressions of first-order type are applicatively equivalent, then their translations can be proved equal in the metalanguage.

Section 4 explains the conditions for a category to model the metalanguage, and hence the nu-calculus. The interpretation is in many ways quite standard, with all the usual machinery of products, exponentials and so forth; only the monad is affected by names and their equality test. All the results on reasoning in the metalanguage immediately carry over to the categorical setting. The final sections describe two particular models: the functor category $Set^{\mathcal{I}}$, where $\mathcal{I}$ is the category of finite sets and injections, and the category $\mathbf{B}G$ of continuous G-sets, where G is a certain topological group.

Although these categories provide an adequate denotational semantics for the nu-calculus, they are not fully abstract. We still cannot prove equivalences (12) or (13) of the last chapter, concerning private names. However the methods used here are quite general, and in the next chapter we shall see how more abstract models can be built on the same framework.

1 A Computational Metalanguage

The types of the metalanguage are given by:

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

There is a unary type constructor T: if A is a type, then elements of TA are *computations* of type A. In this particular metalanguage the difference between a value and a computation is that a computation may generate new names before returning a value. We shall use A, B and their variants to range over the types of the metalanguage.

The term forming operations of the metalanguage are as follows:

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

There is an infinite supply of typed variables, where $x : A$ indicates that variable x is of type A. Function abstraction $\lambda x{:}A.a$ binds the variable x in the term a, and sequential computation $let\, x{\Leftarrow}e\; in\; e'$ binds the variable x within the term e'. We implicitly identify terms up to α-conversion, which allows us to require substitution, written $a[a'/x]$ and $a[a_1/x_1, \ldots, a_n/x_n]$, to be capture avoiding. A term is *closed* if it has no free variables; there is no possibility of free names. We shall generally denote terms by variants on a, b, c, and take variables from x, y, z. Terms of computation type are usually represented by variants of e, terms of function type by f, g and terms of type $Name$ by n.

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

Type judgements of the metalanguage are of the form

$$\Gamma \vdash a : A$$

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

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

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

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

Figure 3.1: Rules for assigning types to terms of the metalanguage

which asserts that in the presence of Γ, term a has type A. Here Γ is a finite set of typed variables; unlike the nu-calculus, there is no set of free names. The rules for forming type judgements are given in Figure 3.1, where we abbreviate $\Gamma \oplus \{x : A\}$ by $\Gamma, x : A$ and write $\Gamma \vdash a_1, \ldots, a_n : A$ to indicate that all of the judgements $\Gamma \vdash a_1 : A, \ldots, \Gamma \vdash a_n : A$ hold. From now on we shall consider only well-typed terms.

Lemma 3.1 *If $\Gamma \vdash a : A$ holds then the type A is unique. Moreover, if the term a has free variables in Γ, then*

$$\Gamma \vdash a : A \quad \iff \quad \Gamma \oplus \Gamma' \vdash a : A$$

for any Γ'.

Proof Both results follow by induction on the structure of a. □

We reason about terms of the metalanguage with an equational logic of Horn clauses. This could be extended to a full *evaluation logic* with modalities [90], but we shall manage without this sophistication. If the type judgements $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$ are valid then

$$\Gamma \vdash a = a' : A$$

is an *equation in context* Γ; we usually omit the type and write $\Gamma \vdash a = a'$. A *sequent* is a judgement

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

where Γ is a finite set of typed variables, Φ is a finite set of equations in context Γ and ϕ is a single equation in context Γ. We may omit Γ or Φ when empty, and write $\Phi, a = a'$ for $\Phi \oplus \{a = a' : A\}$. The free variables $\mathit{fv}(\Phi)$ of Φ are the free variables of its component terms.

Figures 3.2 and 3.3 detail the rules for deriving sequents. Figure 3.2 gives the usual rules for Horn clauses and equational logic, congruence rules for all the term forming operations and β, η axioms at function types. Figure 3.3 contains rules particular to this metalanguage. The rules for computations are those described by Moggi for any computational lambda-calculus, and include the (MONO) rule, that the operation $[-]$ taking values to computations is an inclusion. The rules for boolean values and the comparison of names are straightforward; a derived property is that $\mathit{eq}(-,-)$ is an equivalence relation.

Horn clauses:

$$\frac{}{\Gamma;\Phi \vdash \phi}\ (\phi \in \Phi) \qquad \frac{\Gamma;\Phi \vdash \phi \qquad \Gamma;\Psi,\phi \vdash \psi}{\Gamma;\Phi \cup \Psi \vdash \psi}$$

Equality:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a = a} \qquad \frac{\Gamma \vdash a, a' : A}{\Gamma; a = a' \vdash a' = a} \qquad \frac{\Gamma \vdash a_1, a_2, a_3 : A}{\Gamma; a_1 = a_2, a_2 = a_3 \vdash a_1 = a_3}$$

Congruence:

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

$$\frac{\Gamma \vdash b, b' : \mathit{Bool} \qquad \Gamma \vdash a_1, a_1', a_2, a_2' : A}{\Gamma; b = b', a_1 = a_1', a_2 = a_2' \vdash \mathit{cond}(b, a_1, a_2) = \mathit{cond}(b', a_1', a_2')}$$

$$\frac{\Gamma \vdash n_1, n_1', n_2, n_2' : \mathit{Name}}{\Gamma; n_1 = n_1', n_2 = n_2' \vdash \mathit{eq}(n_1, n_2) = \mathit{eq}(n_1', n_2')}$$

$$\frac{\Gamma \vdash f, f' : A \to B \qquad \Gamma \vdash a, a' : A}{\Gamma; f = f', a = a' \vdash fa = f'a'}$$

$$\frac{\Gamma \vdash a, a' : A}{\Gamma; a = a' \vdash [a] = [a']} \qquad \frac{\Gamma, x : A; \Phi \vdash b = b' : B}{\Gamma; \Phi \vdash \lambda x{:}A.b = \lambda x{:}A.b'}\quad (x \notin \mathit{fv}(\Phi))$$

$$\frac{\Gamma \vdash e_1, e_1' : TA \qquad \Gamma, x : A; \Phi \vdash e_2 = e_2' : TB}{\Gamma; \Phi, e_1 = e_1' \vdash \mathit{let}\, x \Leftarrow e_1\ \mathit{in}\ e_2 = \mathit{let}\, x \Leftarrow e_1'\ \mathit{in}\ e_2'}\quad (x \notin \mathit{fv}(\Phi))$$

Functions:

$$\beta\quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x{:}A.b)a = b[a/x]} \qquad \eta\quad \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f = \lambda x{:}A.fx}$$

Figure 3.2: Rules for equational reasoning in the metalanguage (I)

Computations:

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

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

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

Booleans:

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

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

Testing names:

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

Generating names:

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

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

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

Figure 3.3: Rules for equational reasoning in the metalanguage (II)

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

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

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

These say that any computation whose value is unused may be discarded and that all computations can be reordered. They are not essential to model the nu-calculus, and would be false in a metalanguage extended to handle store or exceptions, for example. Nevertheless, all the categorical models to follow satisfy them.

An equivalent formulation of the last rule (FRESH) is

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

The alternative candidate

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

can be derived, but appears to be strictly weaker. In particular it is not strong enough to complete the proof of Lemma 3.2 and hence Proposition 3.6, that the metalanguage correctly interprets the nu-calculus.

As it stands the rule (FRESH) only allows for comparison of a new name with one other. This is no restriction, as the following derived rule shows:

Lemma 3.2 *For any finite set of terms $\{n_1, \ldots, n_k\}$ of type Name the rule*

$$\frac{\Gamma \vdash n_1, \ldots, n_k : \mathit{Name} \qquad \Gamma, n : \mathit{Name}; \Phi, \mathit{eq}(n_1, n) = \mathit{ff}, \ldots, \mathit{eq}(n_k, n) = \mathit{ff} \vdash e = e'}{\Gamma; \Phi \vdash \mathit{let}\, n \Leftarrow \mathit{new} \,\mathit{in}\, e = \mathit{let}\, n \Leftarrow \mathit{new} \,\mathit{in}\, e'}$$

is derivable.

Proof We proceed by induction on k. The case $k = 0$ follows from the congruence rules for new and let. The rule (FRESH) is the case $k = 1$. Suppose then that we have derived the rule for k and wish to prove the it for $(k + 1)$. Simple equational reasoning gives

$$\frac{\Gamma, n : \mathit{Name}; \Phi, \mathit{eq}(n_1, n) = \mathit{ff}, \ldots, \mathit{eq}(n_{k+1}, n) = \mathit{ff} \vdash e = e'}{\Gamma, n : \mathit{Name}; \Phi, \mathit{eq}(n_1, n) = \mathit{ff}, \ldots, \mathit{eq}(n_{k+1}, n) = \mathit{ff} \vdash \mathit{cond}(\mathit{eq}(n_{k+1}, n), e, e') = e}$$

and also

$$\begin{aligned} \Gamma, n : \mathit{Name}; \Phi, \mathit{eq}(n_1, n) = \mathit{ff}, \ldots, \mathit{eq}(n_k, n) = \mathit{ff}, & \\ \mathit{eq}(n_{k+1}, n) = \mathit{tt} \vdash \mathit{cond}(\mathit{eq}(n_{k+1}, n), e, e') = e & \end{aligned}$$

from which we can eliminate $eq(n_{k+1}, n)$ and obtain

$$\frac{\Gamma, n : \mathit{Name}; \Phi, eq(n_1, n) = \mathit{ff}, \ldots, eq(n_{k+1}, n) = \mathit{ff} \vdash e = e'}{\Gamma, n : \mathit{Name}; \Phi, eq(n_1, n) = \mathit{ff}, \ldots, eq(n_k, n) = \mathit{ff} \vdash \mathit{cond}(eq(n_{k+1}, n), e, e') = e}.$$

The induction hypothesis then provides the rule

$$\frac{\Gamma, n : \mathit{Name}; \Phi, eq(n_1, n) = \mathit{ff}, \ldots, eq(n_k, n) = \mathit{ff} \vdash \mathit{cond}(eq(n_{k+1}, n), e, e') = e}{\Gamma; \Phi \vdash \mathit{let}\, n \Leftarrow \mathit{new\ in\ cond}(eq(n_{k+1}, n), e, e') = \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e}.$$

It is straightforward that

$$\Gamma, n : \mathit{Name}; \Phi, eq(n_{k+1}, n) = \mathit{ff} \vdash \mathit{cond}(eq(n_{k+1}, n), e, e') = e'$$

which by the (FRESH) rule gives

$$\Gamma; \Phi \vdash \mathit{let}\, n \Leftarrow \mathit{new\ in\ cond}(eq(n_{k+1}, n), e, e') = \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e'.$$

Combining all these we have

$$\frac{\Gamma, n : \mathit{Name}; \Phi, eq(n_1, n) = \mathit{ff}, \ldots, eq(n_{k+1}, n) = \mathit{ff} \vdash e = e'}{\Gamma; \Phi \vdash \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e = \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e'}$$

as required, and the proof is complete. □

The rule

$$\frac{\Gamma \vdash n_1, \ldots, n_k : \mathit{Name} \qquad \Gamma, b_1, \ldots, b_k : \mathit{Bool}, n : \mathit{Name} \vdash e : TA}{\begin{aligned}\Gamma \vdash \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e[eq(n_1, n)/b_1, \ldots, eq(n_k, n)/b_k] \\ = \mathit{let}\, n \Leftarrow \mathit{new\ in}\ e[\mathit{ff}/b_1, \ldots, \mathit{ff}/b_k]\end{aligned}}$$

extending (FRESH′) is also valid.

2 Interpreting the Nu-Calculus

Now that we have a metalanguage for reasoning about names and higher-order functions, we describe a suitable interpretation of the nu-calculus. This must be chosen to respect the operational semantics, in particular call-by-value function application and left-to-right evaluation order. We use an extension of Moggi's call-by-value interpretation of the simply-typed lambda-calculus in the computational lambda-calculus [68]. The translation is correct with respect to the operational semantics of the nu-calculus, in both evaluation and reduction forms.

Types are translated from the nu-calculus to the metalanguage according to:

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

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

There are two mutually defined schemes that translate nu-calculus expressions into terms of the metalanguage. Figure 3.4 describes $|-|$ for expressions in canonical form and $[\![-]\!]$ for general expressions, name and variable contexts. The interpretation respects types and substitution of values; it also treats reduction expressions and contexts in a uniform way. The following results make this precise.

Canonical forms:

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

Expressions:

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

Contexts:

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

Figure 3.4: Interpretation of the nu-calculus in the computational metalanguage

Lemma 3.3 *For any well-typed nu-calculus expression M, or expression C in canonical form:*

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

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

Lemma 3.4 *If $M \in \mathrm{Exp}_\sigma(s, \Gamma \oplus \{x : \sigma\})$ and $C \in \mathrm{Can}_\sigma(s, \Gamma)$ then*

$$[\![s, \Gamma]\!] \vdash [\![M[C/x]]\!] = [\![M]\!][|C|/x]$$

in the metalanguage.

Proof By induction on the structure of M, using only the fact that equality in the metalanguage is a congruence. □

Lemma 3.5 *If $M \in \mathrm{Exp}_\sigma(s, \Gamma)$ and $s, \Gamma \vdash E\langle - {:} \sigma\rangle : \sigma'$ is a reduction expression then*

$$[\![s, \Gamma]\!] \vdash [\![E\langle M\rangle]\!] = \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; [\![E\langle m\rangle]\!]$$

and for any reduction context $s, \Gamma \vdash R\langle - {:} \sigma\rangle : \sigma'$,

$$[\![s, \Gamma]\!] \vdash [\![R\langle M\rangle]\!] = \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; [\![R\langle m\rangle]\!].$$

Proof Consider for example the reduction expression $s, \Gamma \vdash \langle - \rangle M : \sigma'$, and suppose that $F \in \mathrm{Exp}_{\sigma \to \sigma'}(s, \Gamma)$. Then, using the equalities of the computational lambda-calculus,

$$\begin{array}{rcl} [\![s, \Gamma]\!] \vdash \;\; \mathit{let}\, f \Leftarrow [\![F]\!] \;\mathit{in}\; [\![fM]\!] & = & \mathit{let}\, f \Leftarrow [\![F]\!] \;\mathit{in}\; \mathit{let}\, f' \Leftarrow [f] \;\mathit{in}\; \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; f'm \\ & = & \mathit{let}\, f \Leftarrow [\![F]\!] \;\mathit{in}\; \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; fm \\ & = & [\![FM]\!]. \end{array}$$

The other reduction expressions are similar. For the second result, we recall that a reduction context is a nested series of reduction expressions, and proceed by induction on the length of this series. Suppose that we already have the result for the reduction context $s, \Gamma \vdash R\langle - {:} \sigma\rangle : \sigma'$, and that $s, \Gamma \vdash E\langle - {:} \sigma'\rangle : \sigma''$ is a reduction expression. Then we can reason:

$$\begin{array}{rcl} [\![s, \Gamma]\!] \vdash \;\; [\![E\langle R\langle M\rangle\rangle]\!] & = & \mathit{let}\, r \Leftarrow [\![R\langle M\rangle]\!] \;\mathit{in}\; [\![E\langle r\rangle]\!] \\ & = & \mathit{let}\, r \Leftarrow (\mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; [\![R\langle M\rangle]\!]) \;\mathit{in}\; [\![E\langle r\rangle]\!] \\ & = & \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; \mathit{let}\, r \Leftarrow [\![R\langle m\rangle]\!] \;\mathit{in}\; [\![E\langle r\rangle]\!] \\ & = & \mathit{let}\, m \Leftarrow [\![M]\!] \;\mathit{in}\; [\![E\langle R\langle m\rangle\rangle]\!] \end{array}$$

which is the result for the reduction context $s, \Gamma \vdash E\langle R\langle - {:} \sigma\rangle\rangle : \sigma''$. □

The interpretation of the nu-calculus in the metalanguage is *correct* with respect to its operational semantics: if $s \vdash M \Downarrow_\sigma (s')C$ then the terms $[\![M]\!]$ and $[\![\nu s'.C]\!]$ can be proved equal in the metalanguage, under the assumption that all the names in s are distinct. A similar result holds for the reduction relation $M \to_\sigma M'$. To make this formal we need two abbreviations:

$$(\neq s) \;=\; \{eq(n_i, n_j) = \mathit{ff} \mid 1 \leq i < j \leq k\}$$

for the assertion that all the names in $s = \{n_1, \ldots, n_k\}$ are distinct, and

$$\mathit{let}\, s' \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\, e \;=\; \mathit{let}\, n'_1 \Leftarrow \mathit{new}\ \mathit{in} \ldots \mathit{let}\, n'_{k'} \Leftarrow \mathit{new}\ \mathit{in}\, e$$

which is the expression that assigns new names to all of $s' = \{n'_1, \ldots, n'_{k'}\}$ and then computes e. The ordering of names from s and s' does not matter, up to provable equality in the metalanguage. A particular consequence of Lemma 3.2 is then the derived rule:

$$\frac{[\![s \oplus s', \Gamma]\!]; (\neq s \oplus s') \vdash e = e' : TA}{[\![s, \Gamma]\!]; (\neq s) \vdash \mathit{let}\, s' \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\, e = \mathit{let}\, s' \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\, e'}.$$

We can now demonstrate:

Proposition 3.6 (Correctness of Translation) *If $s \vdash M \Downarrow_\sigma (s')C$ is a valid evaluation judgement then*

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

is provable in the metalanguage. Further, if $M, M' \in \mathrm{Exp}_\sigma(s)$ and $M \to^*_\sigma M'$ is a valid reduction then

$$[\![s]\!]; (\neq s) \vdash [\![M]\!] = [\![M']\!]$$

can be proved in the metalanguage.

Proof By induction over the derivation of $s \vdash M \Downarrow_\sigma (s')C$ or $M \to^*_\sigma M'$ respectively. We need to confirm that every rule of Figures 2.2 and 2.3 translates to a derivation that is provable in the metalanguage. We give some example cases:

- (LOCAL) The translation of the rule

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

  is

  $$\frac{[\![s]\!], n : \mathit{Name}; (\neq s \oplus \{n\}) \vdash [\![M]\!] = \mathit{let}\, s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\, [\![C]\!]}{[\![s]\!]; (\neq s) \vdash \mathit{let}\, n \Leftarrow \mathit{new}\ \mathit{in}\, [\![M]\!] = \mathit{let}\, n \Leftarrow \mathit{new}\ \mathit{in}\ \mathit{let}\, s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\, [\![C]\!]}$$

  which is an instance of the Lemma 3.2.

- (EQ2) This is the rule

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

We combine

$$\frac{[\![s \oplus s_1 \oplus s_2]\!]; (\neq s \oplus s_1 \oplus s_2) \vdash [eq(n, n')] = [\mathit{ff}]}{[\![s \oplus s_1]\!]; (\neq s \oplus s_1) \vdash \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [eq(n, n')] = \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\mathit{ff}]}$$

with

$$\frac{[\![s \oplus s_1]\!]; (\neq s \oplus s_1) \vdash [\![N']\!] = \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [n']}{[\![s \oplus s_1]\!]; (\neq s \oplus s_1) \vdash \mathit{let}\ a' \Leftarrow [\![N']\!]\ \mathit{in}\ [eq(n, a')] = \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [eq(n, n')]}$$

and obtain

$$[\![s]\!]; (\neq s) \vdash \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ \mathit{let}\ a' \Leftarrow [\![N']\!]\ \mathit{in}\ [eq(n, a')] = \mathit{let}\ (s_1 \oplus s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\mathit{ff}].$$

The addition of

$$\frac{[\![s]\!]; (\neq s) \vdash [\![N]\!] = \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [n]}{\begin{array}{rl}[\![s]\!]; (\neq s) \vdash & \mathit{let}\ a \Leftarrow [\![N]\!]\ \mathit{in}\ \mathit{let}\ a' \Leftarrow [\![N']\!]\ \mathit{in}\ [eq(a, a')] \\ = & \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ \mathit{let}\ a' \Leftarrow [\![N']\!]\ \mathit{in}\ [eq(n, a')]\end{array}}$$

gives the rule

$$\frac{\begin{array}{c}[\![s]\!]; (\neq s) \vdash [\![N]\!] = \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [n] \\ [\![s \oplus s_1]\!]; (\neq s \oplus s_1) \vdash [\![N']\!] = \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [n']\end{array}}{[\![s]\!]; (\neq s) \vdash [\![N = N']\!] = \mathit{let}\ (s_1 \oplus s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![\mathit{false}]\!]}$$

which as required is the translation of (EQ2).

- Application redex. Take the reduction

$$(\lambda x{:}\sigma.M)C \to_{\sigma'} M[C/x]$$

where $C \in \mathrm{Can}_\sigma(s)$ and $M \in \mathrm{Exp}_{\sigma'}(s, \{x : \sigma\})$. We can reason in the metalanguage thus:

$$\begin{array}{rcl}[\![s]\!]; (\neq s) \vdash\ [\![(\lambda x{:}\sigma.M)C]\!] & = & \mathit{let}\ f \Leftarrow [\lambda x{:}[\![\sigma]\!].[\![M]\!]]\ \mathit{in}\ \mathit{let}\ a \Leftarrow [|C|]\ \mathit{in}\ fa \\ & = & (\lambda x{:}[\![\sigma]\!].[\![M]\!])|C| \\ & = & [\![M]\!][|C|/x] \\ & = & [\![M[C/x]]\!]\end{array}$$

using the β rule and Lemma 3.4 on substitution.

- Reduction expressions. Consider the rule scheme

$$\frac{M \to_\sigma M'}{E\langle M\rangle \to_{\sigma'} E\langle M'\rangle}$$

where $E\langle - \rangle$ is one of the five forms of reduction expression. The premise translates to

$$[\![s]\!]; (\neq s) \vdash [\![M]\!] = [\![M']\!]$$

and we can use Lemma 3.5 to deduce

$$\begin{array}{rcl}[\![s]\!]; (\neq s) \vdash\ [\![E\langle M\rangle]\!] & = & \mathit{let}\ m \Leftarrow [\![M]\!]\ \mathit{in}\ [\![E\langle m\rangle]\!] \\ & = & \mathit{let}\ m \Leftarrow [\![M']\!]\ \mathit{in}\ [\![E\langle m\rangle]\!] \\ & = & [\![E\langle M'\rangle]\!]\end{array}$$

which is the interpretation in the metalanguage of the rule consequence.

The other cases are all similar to these. □

3 Reasoning in the Metalanguage

Proposition 3.6 above shows that the interpretation of the nu-calculus in the metalanguage is correct with respect to its operational semantics. A consequence of this is that we can use the metalanguage to reason about contextual equivalence; if two expressions of the nu-calculus are interpreted by terms that are provably equal in the metalanguage, then those expressions are contextually equivalent. Moreover, equational reasoning in the metalanguage is complete for contextual equivalence at ground types. It is also complete with respect to applicative equivalence at first-order types; if two expressions of first-order type are applicatively equivalent, then their translations can be proved equal in the metalanguage.

These properties all rely the metalanguage being consistent: the equation $\vdash tt = \mathit{ff}$ is not provable. This is justified by the categorical models of Sections 5 and 7 where tt and ff are distinct.

The first result is that the metalanguage is suitable for reasoning about contextual equivalence:

Proposition 3.7 (Adequacy of Translation) *Suppose that $M_1, M_2 \in \mathrm{Exp}_\sigma(s,\Gamma)$ and that we can derive*

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

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

Proof Suppose that $P\langle\!\langle - \rangle\!\rangle$ is some program context defined over s. By the compositionality of the translation $[\![-]\!]$, and Lemma 3.4 on the substitution of values,

$$[\![s]\!];(\neq s) \vdash [\![P\langle\!\langle(\vec{x})M_1\rangle\!\rangle]\!] = [\![P\langle\!\langle(\vec{x})M_2\rangle\!\rangle]\!].$$

From Theorem 2.4, evaluation in the nu-calculus is deterministic and terminating, so there are evaluation judgements

$$s \vdash P\langle\!\langle(\vec{x})M_i\rangle\!\rangle \Downarrow_o (s_i)b_i \qquad i = 1,2$$

for some name sets s_1, s_2 and unique choice of booleans b_1, b_2. The correctness result above gives

$$[\![s]\!];(\neq s) \vdash [\![P\langle\!\langle(\vec{x})M_i\rangle\!\rangle]\!] = \mathit{let}\ s_i \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [|b_i|] \qquad i = 1,2$$

and hence

$$[\![s]\!];(\neq s) \vdash \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [|b_1|] = \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [|b_2|].$$

Using the rules (DROP) and (MONO) we obtain

$$[\![s]\!];(\neq s) \vdash |b_1| = |b_2| : \mathit{Bool}$$

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

For closed boolean and name expressions the converse to this holds and any contextual equivalence can be proved in the metalanguage:

Theorem 3.8 (Completeness at Ground Types) *If $\sigma \in \{o, \nu\}$ and $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ then*

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

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

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

for suitable sets s_1, s_2 of names. By Proposition 3.6 and repeated use of the (DROP) rule we can reason thus:

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

which is the desired equality. If $\sigma = \nu$ then there are two possibilities:

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

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

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

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

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

  Correctness and (DROP) then give:

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

In all cases, contextual equivalence implies provable equality in the metalogic. □

Using the metalanguage to reason about contextual equivalence gives results broadly similar to the applicative equivalence of the last chapter. In particular we can confirm examples (2)–(9) and (17) from Section 5 of Chapter 2, but not examples (12) or (13) concerning private names. The correspondence with applicative equivalence can be made precise up to first-order types:

Theorem 3.9 *If σ is a ground or first-order type of the nu-calculus, Γ is a set of variables of ground type, and $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ for some set of names s, then*

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

Proof We show first that the result holds for closed expressions, by induction on the structure of the type σ. The proof follows the form of Definition 2.13 describing applicative equivalence; in particular, we distinguish the case when both expressions are in canonical form.

The base case is immediate:

$$s \vdash C_1 \sim_\sigma^{can} C_2 \quad \Longrightarrow \quad [\![s]\!]; (\neq s) \vdash [\![C_1]\!] = [\![C_2]\!] \qquad \sigma \in \{o, \nu\}.$$

Suppose that for some σ we have the result at $\sim_\sigma^{can}$, and wish to show it at $\sim_\sigma^{exp}$. Now

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

By hypothesis then

$$[\![s \oplus (s_1 \cup s_2)]\!]; (\neq s \oplus (s_1 \cup s_2)) \vdash [\![C_1]\!] = [\![C_2]\!]$$

and so

$$[\![s]\!]; (\neq s) \vdash \mathit{let}\ (s_1 \cup s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_1]\!] = \mathit{let}\ (s_1 \cup s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_2]\!].$$

We can now reason as follows:

$$\begin{array}{rcl} [\![s]\!]; (\neq s) \vdash\ [\![M_1]\!] & = & \mathit{let}\ s_1 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_1]\!] \\ & = & \mathit{let}\ (s_1 \cup s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_1]\!] \\ & = & \mathit{let}\ (s_1 \cup s_2) \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_2]\!] \\ & = & \mathit{let}\ s_2 \Leftarrow \overrightarrow{\mathit{new}}\ \mathit{in}\ [\![C_2]\!] \\ & = & [\![M_2]\!] \end{array}$$

which is the required result.

For function types, we assume the result at $\sim_\sigma^{exp}$ and consider $\sim_{o\to\sigma}^{can}$ and $\sim_{\nu\to\sigma}^{can}$. Firstly

$$\begin{array}{lcl} s \vdash \lambda x{:}o.M_1 \sim_{o\to\sigma}^{can} \lambda x{:}o.M_2 & \iff & s \vdash M_1[\mathit{true}/x] \sim_\sigma^{exp} M_2[\mathit{true}/x]\ \ \& \\ & & s \vdash M_1[\mathit{false}/x] \sim_\sigma^{exp} M_2[\mathit{false}/x] \end{array}$$

which by the induction hypothesis gives

$$\begin{array}{rcl} [\![s]\!],\, x{:}\mathit{Bool};\ (\neq s), x = \mathit{tt} & \vdash & [\![M_1]\!] = [\![M_2]\!] \\ [\![s]\!],\, x{:}\mathit{Bool};\ (\neq s), x = \mathit{ff} & \vdash & [\![M_1]\!] = [\![M_2]\!] \end{array}$$

from which we can deduce in turn

$$\begin{array}{rcl} [\![s]\!],\, x{:}\mathit{Bool};\ (\neq s) & \vdash & [\![M_1]\!] = [\![M_2]\!] \\ [\![s]\!];\ (\neq s) & \vdash & \lambda x{:}\mathit{Bool}.[\![M_1]\!] = \lambda x{:}\mathit{Bool}.[\![M_2]\!] \\ [\![s]\!];\ (\neq s) & \vdash & [\lambda x{:}[\![o]\!].[\![M_1]\!]] = [\lambda x{:}[\![o]\!].[\![M_2]\!]] \\ [\![s]\!];\ (\neq s) & \vdash & [\![\lambda x{:}o.M_1]\!] = [\![\lambda x{:}o.M_2]\!] \end{array}$$

the last of which is the desired result.

The proof for functions of type $(\nu \to \sigma)$ is slightly tricky. We have

$$s \vdash \lambda x{:}\nu.M_1 \sim^{can}_{\nu\to\sigma} \lambda x{:}\nu.M_2 \iff \begin{array}{l} \forall n \in s.\, s \vdash M_1[n/x] \sim^{exp}_{\sigma} M_2[n/x] \\ \&\;\; s \oplus \{n\} \vdash M_1[n/x] \sim^{exp}_{\sigma} M_2[n/x]. \end{array}$$

Suppose that $s = \{n_1, \ldots, n_k\}$, then by the induction hypothesis we obtain all of

$$\begin{array}{ll}
[\![s]\!],\, x{:}Name;\, (\neq s), eq(x,n_1) = f\!f, \ldots, eq(x,n_{k-1}) = f\!f, eq(x,n_k) = f\!f & \vdash [\![M_1]\!] = [\![M_2]\!] \\
[\![s]\!],\, x{:}Name;\, (\neq s), eq(x,n_1) = f\!f, \ldots, eq(x,n_{k-1}) = f\!f, eq(x,n_k) = tt & \vdash [\![M_1]\!] = [\![M_2]\!] \\
[\![s]\!],\, x{:}Name;\, (\neq s), eq(x,n_1) = f\!f, \ldots, eq(x,n_{k-1}) = tt & \vdash [\![M_1]\!] = [\![M_2]\!] \\
 & \vdots \\
[\![s]\!],\, x{:}Name;\, (\neq s), eq(x,n_1) = tt & \vdash [\![M_1]\!] = [\![M_2]\!].
\end{array}$$

Eliminating the $eq(x, n_i)$ in turn gives

$$[\![s]\!],\, x{:}Name; (\neq s) \vdash [\![M_1]\!] = [\![M_2]\!]$$

from which follows as before:

$$[\![s]\!]; (\neq s) \vdash [\![\lambda x{:}\nu.M_1]\!] = [\![\lambda x{:}\nu.M_2]\!].$$

This completes the proof for closed expressions. For open expressions we proceed by induction on the number of free variables. From the definition

$$s, \Gamma \oplus \{x : \sigma\} \vdash M_1 \sim_{\sigma'} M_2 \iff s, \Gamma \vdash \lambda x{:}\sigma.M_1 \sim_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2.$$

As σ is a ground type, the induction hypothesis applies and we can reason:

$$\begin{array}{lrl}
 & [\![s,\Gamma]\!];\, (\neq s) & \vdash\; [\lambda x{:}[\![\sigma]\!].[\![M_1]\!]] = [\lambda x{:}[\![\sigma]\!].[\![M_2]\!]] \\
\text{(MONO)} & [\![s,\Gamma]\!];\, (\neq s) & \vdash\; \lambda x{:}[\![\sigma]\!].[\![M_1]\!] = \lambda x{:}[\![\sigma]\!].[\![M_2]\!] \\
(\beta) & [\![s,\Gamma \oplus \{x : \sigma\}]\!];\, (\neq s) & \vdash\; [\![M_1]\!] = [\![M_2]\!]
\end{array}$$

which is the desired result. □

At second or higher orders this fails: applicative equivalence does not necessarily imply provable equality in the metalanguage. This is because terms in the metalanguage of function type do not necessarily denote expressions of the nu-calculus.

4 Constructing Categorical Models

It is standard that the simply-typed lambda-calculus can be modelled in any cartesian closed category, with objects for types and morphisms for terms [42]. Moggi extends this to a model of the computational lambda-calculus in any cartesian closed category equipped with a *strong monad* T [68]. We now specialise to the particular case of the computational metalanguage for names, as described above.

The method is that if a category $\mathcal{C}$ satisfies certain requirements then its *internal language* will include the metalanguage of Section 1. The translation of Section 2 then extends to a model of the nu-calculus in $\mathcal{C}$ that is sound with respect to the operational semantics.

If $\mathcal{C}$ is not degenerate then the translation is also adequate, and the category can be used to reason about contextual equivalence. This will be at least as powerful as the basic metalanguage. In particular reasoning in such $\mathcal{C}$ is complete for contextual equivalence at ground types, and for applicative equivalence up to first-order types. Of course the intention is that a suitable choice of category might prove more than the metalanguage alone.

A category $\mathcal{C}$ is suitable to model the metalanguage if the following conditions hold:

- It is cartesian closed: that is, it has finite limits and exponentials. In particular, we use products to interpret contexts, and exponentials for function types.

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

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

  whenever $f : A \times B \to TC$ and $g : A \times C \to TD$. These correspond precisely to the computation rules for let in Figure 3.3. The lift operation for a strong monad is a generalisation of that for a Kleisli triple, a particular presentation of an ordinary categorical monad. The generalisation is necessary to carry around contexts of let-expressions, as Moggi explains in [68, Remark 3.1].

  A strong monad can also be presented as a monad (T, η, μ) together with natural maps $t_{A,B} : A \times TB \to T(A \times B)$. Here $\eta : 1 \to T$ and $\mu : T^2 \to T$ are the usual unit and multiplication natural transformations for a monad, and the *strength* $t_{A,B}$ must satisfy certain equations. Moggi gives a detailed explanation of this, and further comments on the characterisation of strong monads in [68, Definition 3.2 *et seq.*].

- The monad T satisfies the *mono requirement*, that all $\eta_A : A \to TA$ are monic. This corresponds to the (MONO) rule of Figure 3.3.

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

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \arrow[d, "ff"] \\
1 \arrow[r, "tt"'] & 1+1
\end{tikzcd}$$

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

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

  to interpret the conditional.

- There is a distinguished object N, used to interpret the type of names. This must be *decidable*, which requires a morphism $eq : N \times N \to 1 + 1$ such that

$$\begin{tikzcd} N \arrow[r, "\Delta"] \arrow[d] & N \times N \arrow[d, "eq"] \\ 1 \arrow[r, "tt"'] & 1+1 \end{tikzcd}$$

  is a pullback square, where Δ is the diagonal map. The morphism eq interprets the equality test on names. In the internal language of $\mathcal{C}$, the pullback condition corresponds to the rules for testing names in Figure 3.3.

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

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

  It is clear that these are simply the rules for generating names of Figure 3.3, with the alternate form of (FRESH) as given on page 42. We could express these equations by commutative diagrams asserting the equality of certain morphisms in $\mathcal{C}$, but their essence becomes lost in a mass of variable manipulation.

  The first two of these equations hold automatically if the monad T is respectively *affine* and *commutative*. These are notions due to Kock [39, 40]. A strong monad is affine if

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

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

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

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

In summary, a category $\mathcal{C}$ is suitable to model the metalanguage, and hence the nu-calculus, if the following hold:

- It is cartesian closed.
- It has a strong monad T satisfying the mono requirement.
- It has a disjoint coproduct $1 + 1$.
- There is a distinguished decidable object N.

- There is a distinguished morphism $new : 1 \to TN$ satisfying certain equations.

Given such a category, the embedding of the metalanguage of Section 1 in its internal language is quite standard. Types are interpreted by objects of the category: $Bool$ by $1 + 1$, $Name$ by N, function types by exponentials and computation types using the strong monad T. A context $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$ is interpreted by the product $\Gamma = A_1 \times \cdots \times A_n$. A term in context is interpreted by a morphism:

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

The derivation of such a morphism uses the rules on the right of Figure 3.5, which match those on the left for terms of the metalanguage. An equation in context is then interpreted by equality of morphisms:

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

The sequent

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

is interpreted by equality of the morphisms

$$\begin{tikzcd} E \arrow[r, "e"] & \Gamma \arrow[r, shift left, "a"] \arrow[r, shift right, "a'"'] & A \end{tikzcd} \qquad a \circ e = a' \circ e$$

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

$$\begin{tikzcd} & & A_1 \\ E \arrow[r, "e"] & \Gamma \arrow[ur, shift left, "a_1"] \arrow[ur, shift right, "a_1'"'] \arrow[dr, shift right, "a_n'"'] \arrow[dr, shift left, "a_n'"] & \vdots \\ & & A_n \end{tikzcd}$$

Under this embedding the conditions on $\mathcal{C}$ correspond exactly to the rules of Figures 3.2 and 3.3 for reasoning in the metalanguage, so any equation provable in the metalanguage will also hold in $\mathcal{C}$. As a result, any non-degenerate model demonstrates that the metalanguage is consistent.

The translation of Section 2 now becomes an interpretation of the nu-calculus in the category $\mathcal{C}$. For each valid type assertion $s, \Gamma \vdash M : \sigma$ there is a morphism

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

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

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

Here $N^{|s|}$ is the object of $|s|$-tuples of names. We define the subobject $(\neq s) \rightarrowtail N^{|s|}$ of distinct $|s|$-tuples as the simultaneous equaliser of all the pairs

$$\begin{tikzcd} N^{|s|} \arrow[r, shift left, "eq \circ \langle \pi_i, \pi_j \rangle"] \arrow[r, shift right, "ff \circ !"'] & 1 + 1 \end{tikzcd} \qquad 1 \leq i < j \leq |s|.$$

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

$$\frac{}{\Gamma \vdash \mathit{new} : T\mathit{Name}} \quad \longmapsto \quad \frac{}{\Gamma \xrightarrow{!} 1 \xrightarrow{\mathit{new}} TN}$$

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

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

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

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

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

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

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

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

Figure 3.5: Rules for constructing morphisms to interpret terms of the metalanguage

In the internal language, this corresponds to the conjunction

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

We then define the composite morphisms:

$$\begin{array}{rcll}
[\![M]\!]_{\Gamma,\neq s} & = & \left((\neq s) \times [\![\Gamma]\!] \rightarrowtail N^{|s|} \times [\![\Gamma]\!] \xrightarrow{[\![M]\!]} T[\![\sigma]\!]\right) & M \in \mathrm{Exp}_\sigma(s, \Gamma) \\
|C|_{\Gamma,\neq s} & = & \left((\neq s) \times [\![\Gamma]\!] \rightarrowtail N^{|s|} \times [\![\Gamma]\!] \xrightarrow{|C|} [\![\sigma]\!]\right) & C \in \mathrm{Can}_\sigma(s, \Gamma) \\
[\![M]\!]_{\neq s} & = & \left((\neq s) \rightarrowtail N^{|s|} \xrightarrow{[\![M]\!]} T[\![\sigma]\!]\right) & M \in \mathrm{Exp}_\sigma(s) \\
|C|_{\neq s} & = & \left((\neq s) \rightarrowtail N^{|s|} \xrightarrow{|C|} [\![\sigma]\!]\right) & C \in \mathrm{Can}_\sigma(s)
\end{array}$$

The results of Section 2 and 3 now carry over to the categorical model:

Proposition 3.10 (Correctness) *If $s \vdash M \Downarrow_\sigma (s')C$ is a valid evaluation judgement then*

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

Further, if $M, M' \in \mathrm{Exp}_\sigma(s)$ and $M \to_\sigma^* M'$ is a valid reduction then

$$[\![M]\!]_{\neq s} \;=\; [\![M']\!]_{\neq s}$$

Proof Follows from Proposition 3.6. □

Proposition 3.11 (Adequacy) *Suppose that the category $\mathcal{C}$ is* non-degenerate *in that the objects* 0 *and* 1 *are not isomorphic. Then for all $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$:*

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

Proof Exactly as for Proposition 3.7. □

Theorem 3.12 (Completeness at Ground Types) *If $\sigma \in \{o, \nu\}$ and $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ then*

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

Proof Follows from Theorem 3.8. □

Theorem 3.13 *If σ is a ground or first order type of the nu-calculus, Γ is a set of variables of ground type, and $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ for some set of names s, then*

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

Proof Follows from Theorem 3.9. □

So a non-degenerate categorical model can be used to prove contextual equivalences of the nu-calculus. The more that can be shown, the more *abstract* a model is. It is *fully abstract* if the result of Theorem 3.12 holds at all types σ. More modestly, a model may be fully abstract for some restricted set of types or expressions. As with reasoning in the metalanguage, any adequate categorical model will validate at least the equivalences (2)–(9) and (17) from Section 5 of Chapter 2.

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

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

In this section we describe our first example of a category suitable to model the nu-calculus. Although it is not particularly abstract, the existence of the category does prove that the metalanguage is consistent, and justifies using it to reason about contextual equivalence (Proposition 3.7 above). The construction is based on Moggi's model of dynamic allocation [67, §4.1.4]; it is related to the 'possible worlds' models of Oles, Reynolds, Tennent and O'Hearn [107, 88, 81], and also to Mitchell and Moggi's Kripke-style models [63, 64]. After describing the model, we show that reasoning in it has the same power as applicative equivalence, up to second-order types.

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

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

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

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

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

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

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

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

Explicitly, on objects it is the quotient

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

where $\langle s_1, a_1\rangle \sim \langle s_2, a_2\rangle$ if and only if for some s_0 there are injective functions $f_1 : s_1 \to s_0$ and $f_2 : s_2 \to s_0$ with $A(id_s + f_1)a_1 = A(id_s + f_2)a_2$ in $A(s + s_0)$. We write $[s', a']$ to represent the equivalence class of $\langle s', a'\rangle$; this element is the computation 'create the new names s' and return value a'', and quotienting by the relation '$\sim$' ensures that the (DROP) and (SWAP) rules for names hold true. For any constant functor $C : \mathcal{I} \to Set$ the monad is the identity: $TC = TC$. This is the case for the interpretation of any nu-calculus type that does not involve ν, for example.

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

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

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

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

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

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

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

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

where

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

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

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

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

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

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

which satisfies the necessary equations. The obect TN is isomorphic to $N + 1$, thanks to the quotient in the definition of T.

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

$$\begin{array}{rlrl}
[\![M']\!]_{\Gamma,\neq s} : (\neq s) \times [\![\Gamma]\!] \to T[\![\sigma]\!] & & M' \in \mathrm{Exp}_\sigma(s,\Gamma) \\
|C'|_{\Gamma,\neq s} : (\neq s) \times [\![\Gamma]\!] \to [\![\sigma]\!] & & C' \in \mathrm{Can}_\sigma(s,\Gamma) \\
[\![M]\!]_{\neq s} : (\neq s) \to T[\![\sigma]\!] & & M \in \mathrm{Exp}_\sigma(s) \\
|C|_{\neq s} : (\neq s) \to [\![\sigma]\!] & & C \in \mathrm{Can}_\sigma(s)
\end{array}$$

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

$$\begin{array}{cc}
[\![M']\!]_{\Gamma,\neq s} \in (T[\![\sigma]\!])^{[\![\Gamma]\!]}s & |C'|_{\Gamma,\neq s} \in [\![\sigma]\!]^{[\![\Gamma]\!]}s \\
[\![M]\!]_{\neq s} \in T[\![\sigma]\!]s & |C|_{\neq s} \in [\![\sigma]\!]s.
\end{array}$$

These are generally easier to work with, and the results of Section 4 still hold when stated in terms of elements rather than morphisms.

The interpretation of the nu-calculus in $Set^{\mathcal{I}}$ only makes use of pullback-preserving functors; in particular, if A preserves pullbacks, then so does TA. So we could instead consider just the full subcategory of pullback-preserving functors from $\mathcal{I}$ to Set. This category $\mathcal{A}$ is a *topos*; the topos of sheaves for the atomic topology on $\mathcal{I}^{\mathrm{op}}$, sometimes called the Schanuel topos. More details can be found in Mac Lane and Moerdijk [52, Chap. III; pp. 115, 155, 158]. The model in $\mathcal{A}$ is no more abstract than that in $Set^{\mathcal{I}}$, but it does allow a better interpretation of Moggi's higher-order metalanguage [69, Example 4.12]. Also, it is known that $\mathcal{A}$ is the classifying topos for the geometric theory of an infinite decidable object [52, Chap. VIII Ex. 9, Chap. X Ex. 6; pp. 468, 570]. This seems to fit with the abstract properties that we seek for an object of names, but the exact connection is unclear.

6 Properties of the Model in $Set^{\mathcal{I}}$

Regarding the sample contextual equivalences of Chapter 2, Section 5, the category $Set^{\mathcal{I}}$ validates only those shown by any adequate model of the nu-calculus, and not examples (12) or (13) involving private names. This is because the model makes no provision to identify functions that differ only on arguments which no external context could supply. Applicative equivalence is of similar power, and the two theorems in this section make the connection quite precise.

Theorem 3.14 *For all $M_1, M_2 \in \mathrm{Exp}_\sigma(s,\Gamma)$, equality in $Set^{\mathcal{I}}$ implies applicative equivalence:*

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

Proof We show first that the result holds for closed expressions:

$$\begin{array}{lll}
|C_1|_{\neq s} = |C_2|_{\neq s} & \Longrightarrow\ s \vdash C_1 \sim_\sigma^{can} C_2 & C_1, C_2 \in \mathrm{Can}_\sigma(s) \\
[\![M_1]\!]_{\neq s} = [\![M_2]\!]_{\neq s} & \Longrightarrow\ s \vdash M_1 \sim_\sigma^{exp} M_2 & M_1, M_2 \in \mathrm{Exp}_\sigma(s).
\end{array}$$

The proof is by induction on the structure of the type σ, and follows the form of Definition 2.13 describing applicative equivalence. For boolean and name expressions in canonical form, the result is immediate. For lambda abstractions we recall the definition:

$$s \vdash \lambda x{:}\sigma.M_1 \sim^{can}_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2 \iff \begin{array}{l}\forall s',\ C \in \mathrm{Can}_\sigma(s \oplus s') \ . \\ \quad s \oplus s' \vdash M_1[C/x] \sim^{exp}_{\sigma'} M_2[C/x].\end{array}$$

Suppose then that $|\lambda x{:}\sigma.M_1|_{\neq s} = |\lambda x{:}\sigma.M_2|_{\neq s}$ and $C \in \mathrm{Can}_\sigma(s \oplus s')$ for some set of names s'. We set

$$f_i = |\lambda x{:}\sigma.M_i|_{\neq s} \ \in \ [\![\sigma \to \sigma']\!]s = \mathit{Set}^{\mathcal{I}}(\mathcal{I}(s,-) \times [\![\sigma]\!], T[\![\sigma']\!]) \qquad i = 1, 2$$

and use Lemma 3.4 on substitution to reason

$$\begin{array}{rcl}[\![M_1[C/x]]\!]_{\neq s\oplus s'} &=& f_1 s' \langle inl_{s,s'}, |C|_{\neq s\oplus s'}\rangle \\ &=& f_2 s' \langle inl_{s,s'}, |C|_{\neq s\oplus s'}\rangle \\ &=& [\![M_2[C/x]]\!]_{\neq s\oplus s'}.\end{array}$$

By the induction hypothesis we deduce that $s \oplus s' \vdash M_1[C/x] \sim^{exp}_{\sigma'} M_2[C/x]$ and so $s \vdash \lambda x{:}\sigma.M_1 \sim^{can}_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2$ as required.

For general expressions, applicative equivalence is defined by

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

Suppose then that $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$ with $[\![M_1]\!]_{\neq s} = [\![M_2]\!]_{\neq s}$. Theorem 2.4 tells us that there are evaluation judgements

$$s \vdash M_1 \Downarrow_\sigma (s_1)C_1 \qquad \text{and} \qquad s \vdash M_2 \Downarrow_\sigma (s_2)C_2,$$

from which

$$[\![M_i]\!]_{\neq s} = [\![\nu s_i.C_i]\!]_{\neq s} = [s_i, |C_i|_{\neq s\oplus s_i}] \ \in \ T[\![\sigma]\!]s \qquad i = 1, 2.$$

But then $[s_1, |C_1|_{\neq s\oplus s_1}] = [s_2, |C_2|_{\neq s\oplus s_2}]$, and the definition of the monad gives injections $f_i : s_i \rightarrowtail s'$ for $i = 1, 2$ such that

$$[\![\sigma]\!](id_s + f_1)|C_1|_{\neq s\oplus s_1} \ = \ [\![\sigma]\!](id_s + f_2)|C_2|_{\neq s\oplus s_2}.$$

With suitable relabelling we can assume that the f_i are inclusions and so

$$[\![\sigma]\!](id_s + f_i)|C_i|_{\neq s\oplus s_i} \ = \ |C_i|_{\neq s\oplus s'} \qquad i = 1, 2.$$

We then have $|C_1|_{\neq s\oplus s'} = |C_2|_{\neq s\oplus s'}$, and by the induction hypothesis $s \oplus s' \vdash C_1 \sim^{can}_\sigma C_2$. This is not quite as required, but Lemma 2.14 gives $s \oplus (s_1 \cup s_2) \vdash C_1 \sim^{can}_\sigma C_2$ and hence $s \vdash M_1 \sim^{exp}_\sigma M_2$.

This completes the proof for closed expressions; for open expressions we proceed by induction on the number of free variables. Suppose that $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma \oplus \{x : \sigma\})$ and

$$[\![M_1]\!]_{\Gamma\oplus\{x:\sigma\},\neq s} = [\![M_2]\!]_{\Gamma\oplus\{x:\sigma\},\neq s} = f \ \in \ \left((T[\![\sigma']\!])^{[\![\Gamma]\!]\times[\![\sigma]\!]}\right) s.$$

There is an isomorphism

$$(T[\![\sigma']\!])^{[\![\Gamma]\!]\times[\![\sigma]\!]} \;\cong\; \Big((T[\![\sigma']\!])^{[\![\sigma]\!]}\Big)^{[\![\Gamma]\!]} \;=\; [\![\sigma \to \sigma']\!]^{[\![\Gamma]\!]},$$

across which the element f can be reinterpreted:

$$f \;=\; |\lambda x{:}\sigma.M_i|_{\Gamma,\neq s} \;\in\; [\![\sigma \to \sigma']\!]^{[\![\Gamma]\!]}s \qquad i = 1, 2.$$

This leads to

$$[\![\lambda x{:}\sigma.M_1]\!]_{\Gamma,\neq s} \;=\; [f] \;=\; [\![\lambda x{:}\sigma.M_2]\!]_{\Gamma,\neq s}$$

and from the induction hypothesis $s, \Gamma \vdash \lambda x{:}\sigma.M_1 \sim_{\sigma\to\sigma'} \lambda x{:}\sigma.M_2$. By Definition 2.13 this is equivalent to $s, \Gamma \oplus \{x : \sigma\} \vdash M_1 \sim_{\sigma'} M_2$, which is the desired result. □

Before we can show a result in the other direction, we need to know which elements in the model denote expressions of the nu-calculus.

Lemma 3.15 (Definability) *Suppose that σ is a ground or first-order type of the nu-calculus and s is some set of names. If $a \in [\![\sigma]\!]s$ and $e \in T[\![\sigma]\!]s$, then there are expressions $C \in \mathrm{Can}_\sigma(s)$ and $M \in \mathrm{Exp}_\sigma(s)$ such that $a = |C|_{\neq s}$ and $e = [\![M]\!]_{\neq s}$.*

Proof We begin by showing that the result for values implies that for computations. Suppose that $e \in T[\![\sigma]\!]s$ has a representative $e = [s', a']$ where $a' \in [\![\sigma]\!](s + s')$. By assumption there is $C \in \mathrm{Can}_\sigma(s \oplus s')$ such that $a' = |C|_{\neq s\oplus s'}$, and then

$$e \;=\; [s', a'] \;=\; [\![\nu s'.C]\!]_{\neq s}$$

as required.

We now apply induction over the structure of the type σ. For elements of $[\![o]\!]s = 1 + 1$ or $[\![\nu]\!]s = s$ the result is immediate. Suppose that $f \in [\![o \to \sigma]\!]s$, that is

$$f \;\in\; Set^{\mathcal{I}}(\mathcal{I}(s, -) \times (1 + 1), T[\![\sigma]\!]).$$

Then f is entirely determined by the elements $f s\langle id_s, tt\rangle$ and $f s\langle id_s, f\!f\rangle$ in $T[\![\sigma]\!]s$. From the induction hypothesis these are definable, say by $M_1, M_2 \in \mathrm{Exp}_\sigma(s)$, and then f is too:

$$f \;=\; |\,\lambda b{:}o.\mathit{if}\ b\ \mathit{then}\ M_1\ \mathit{else}\ M_2\,|_{\neq s}.$$

A value $g \in [\![\nu \to \sigma]\!]s = Set^{\mathcal{I}}(\mathcal{I}(s, -) \times N, T[\![\sigma]\!])$ is similarly determined by

$$gs\langle id_s, n_1\rangle, \ldots, gs\langle id_s, n_k\rangle \;\in\; T[\![\sigma]\!]s$$

and

$$g(s+1)\langle inl_{s,1}, inr_{s,1}\rangle \;\in\; T[\![\sigma]\!](s+1),$$

where $s = \{n_1, \ldots, n_k\}$. Suppose that these are definable using $M_1, \ldots, M_k \in \mathrm{Exp}_\sigma(s)$ and $M' \in \mathrm{Exp}_\sigma(s \oplus \{n\})$. Then g is definable too:

$$\begin{aligned} g \;=\; |\,\lambda x{:}\nu.\ &\mathit{if}\ x = n_1\ \mathit{then}\ M_1 \\ &\mathit{else\ if}\ x = n_2\ \mathit{then}\ M_2 \\ &\quad\vdots \\ &\mathit{else\ if}\ x = n_k\ \mathit{then}\ M_k\ \mathit{else}\ M'[x/n]\,|_{\neq s}, \end{aligned}$$

and the result holds for all first-order σ, by induction. □

Theorem 3.16 *Take σ to be a nu-calculus type of ground, first or second order, and Γ a set of variables of ground or first-order type. If $M_1, M_2 \in \mathrm{Exp}_\sigma(s, \Gamma)$ for some set of names s, then*

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

If σ is third-order then the implication fails.

Proof Exactly as for Theorem 3.9, with the addition of an induction step for expressions of second-order type in canonical form. Given that σ is a ground or first order type, and that the result holds for $\sim^{exp}_{\sigma'}$, we show that it is true for $\sim^{can}_{\sigma \to \sigma'}$. Recall that

$$s \vdash \lambda x{:}\sigma.M_1 \sim^{can}_{\sigma \to \sigma'} \lambda x{:}\sigma.M_2 \quad \iff \quad \begin{array}{l} \forall s',\, C \in \mathrm{Can}_\sigma(s \oplus s') \,. \\ \qquad s \oplus s' \vdash M_1[C/x] \sim^{exp}_{\sigma'} M_2[C/x] \end{array}$$

and let

$$f_i = |\lambda x{:}\sigma.M_i|_{\neq s} \ \in \ [\![\sigma \to \sigma']\!] s = \mathit{Set}^{\mathcal{I}}(\mathcal{I}(s, -) \times [\![\sigma]\!], T[\![\sigma']\!]) \qquad i = 1, 2.$$

We seek to prove that $f_1 = f_2$. To do this we must have, for all name sets s', and all elements $a \in [\![\sigma]\!](s + s')$, that

$$f_1 s' \langle \mathit{inl}_{s,s'}, a\rangle = f_2 s' \langle \mathit{inl}_{s,s'}, a\rangle \ \in \ T[\![\sigma']\!](s + s').$$

From Lemma 3.15 there is some $C \in \mathrm{Can}_\sigma(s \oplus s')$ such that $a = |C|_{\neq s \oplus s'}$, and so

$$f_i s' \langle \mathit{inl}_{s,s'}, a\rangle \ = \ [\![M_i[C/x]]\!]_{\neq s \oplus s'} \qquad i = 1, 2.$$

By assumption, $s \oplus s' \vdash M_1[C/x] \sim^{exp}_{\sigma'} M_2[C/x]$ and the induction hypothesis gives

$$[\![M_1[C/x]]\!]_{\neq s \oplus s'} \ = \ [\![M_2[C/x]]\!]_{\neq s \oplus s'},$$

from which $f_1 = f_2$ as required.

To show failure at third-order types we start with the first-order example (12) from Chapter 2, Section 5:

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

The model in $\mathit{Set}^{\mathcal{I}}$ cannot confirm this equivalence; if we set $g_1 = \nu n.\lambda x{:}\nu.(x = n)$ and $g_2 = \lambda x{:}\nu.\mathit{false}$, then their denotations are not equal:

$$[\![g_1]\!] \neq [\![g_2]\!] \ \in \ T[\![\nu \to o]\!]0.$$

For example, they are distinguished by the element

$$\begin{array}{rcl} h & \in & ((T(1+1))^N \to T(1+1))\,0 \\ hs\langle 0, g\rangle & = & \begin{cases} [\mathit{tt}] & \text{if } \forall i : s \to s',\, n \in s' \,.\, gs'\langle i, n\rangle = [\mathit{tt}] \\ [\mathit{ff}] & \text{otherwise} \end{cases} \end{array}$$

where $0 : 0 \to s$ and $g \in (T(1+1))^N s$. This h tests whether a function g returns $[\mathit{tt}]$ at every name, private or public. The fact that $g_1 \approx_{\nu \to o} g_2$ and yet

$$h^* 0\langle \mathit{id}_0, [\![g_1]\!]\rangle = [\mathit{ff}] \ \neq \ [\mathit{tt}] = h^* 0\langle \mathit{id}_0, [\![g_2]\!]\rangle,$$

tells us that h cannot denote any expression of the nu-calculus. Proceeding to higher types, we consider

$$\begin{aligned} F_1 &= \lambda f : (\nu \to o) \to o \, . \, f(\nu n.\lambda x{:}\nu.(x = n)) \\ F_2 &= \lambda f : (\nu \to o) \to o \, . \, f(\lambda x{:}\nu.\mathit{false}). \end{aligned}$$

Now $[\![F_1]\!]$ and $[\![F_2]\!]$ are distinct in the model, differing at arguments such as h that can separate $[\![g_1]\!]$ from $[\![g_2]\!]$. But, as argued above, none of these can be defined in the nu-calculus, so F_1 and F_2 are applicatively equivalent. We then have the desired counterexample:

$$\vdash F_1 \sim_{((\nu \to o) \to o) \to o} F_2 \quad \text{but} \quad [\![F_1]\!] \neq [\![F_2]\!].$$

It also follows that F_1 and F_2 are contextually equivalent. □

So applicative equivalence and equality in $Set^{\mathcal{I}}$ prove exactly the same contextual equivalences up to second-order types, and at higher types applicative equivalence is more powerful. However, this extra strength is quite illusory; to demonstrate it, we had to use the equivalence

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

which neither method is able to prove.

7 Continuous G-sets

Our next example of a category suitable to model the nu-calculus is in fact equivalent to the Schanuel topos $\mathcal{A}$ mentioned in Section 5. Mac Lane and Moerdijk explain the connection in [52, §III.9; pp. 150–155]. A consequence is that this model is no more abstract than the last; however it does provide a quite different presentation of the category.

We consider the topological group G of automorphisms of $\mathbb{N}$, the natural numbers, with topology inherited from the product topology on $\mathbb{N}^{\mathbb{N}}$. If k is a finite subset of $\mathbb{N}$ then the stabiliser subgroups

$$G_k = \mathit{Stab}_G(k) \leq G$$

form a basis of neighbourhoods of the identity. A *continuous G-set* A is a set equipped with an action

$$\begin{aligned} G \times A &\to A \\ \langle g, a \rangle &\mapsto g \cdot a \end{aligned}$$

which is continuous when A is given the discrete topology [52, §I.1(xi)]. Equivalently, the stabilisers of elements of A must all be open. We write $|A|$ for the set underlying A. A morphism of continuous G-sets is a function on the underlying sets which respects the action.

The category $\mathbf{B}G$ of continuous G-sets is cartesian closed [52, Chap. I Ex. 6]. Finite limits and colimits correspond to those in Set, while the exponential B^A is the set of all the functions from $|A|$ to $|B|$ whose stabilisers are open in G, according to the action

$$(g \cdot p)a \;=\; g \cdot (p(g^{-1} \cdot a)) \qquad g \in G, \quad p : A \to B, \quad a \in A.$$

The interpretation of a nu-calculus type is a set of values at all possible stages, with the group action describing how values change under name substitution. If k is a finite subset of $\mathbb{N}$ and A an object of $\mathbf{B}G$, then we define

$$A_k = \mathit{Fix}_A(G_k) \subseteq A,$$

those elements of A which use only the names in k. For any $a \in A$ the smallest $k \subseteq \mathbb{N}$ such that $a \in A_k$ is the *support* of a, the names which it actually needs. The condition that a G-set be continuous is equivalent to requiring every element to have finite support.

To interpret the ground types, the object of booleans $1 + 1$ is the two-element set with trivial G-action, and the object of names N is $\mathbb{N}$ with the evident G-action. The morphism $\mathit{eq} : N \times N \to 1 + 1$ is just the equality test on $\mathbb{N}$.

For a continuous G-set A, to construct the object TA we must make some preliminary definitions. If $f : \mathbb{N} \rightarrowtail \mathbb{N}$ is an injection, then it induces a map $f^A : |A| \to |A|$ as follows: for $a \in A$, take any finite $k \subseteq \mathbb{N}$ such that $a \in A_k$, choose $g \in G$ with $g|_k = f|_k$ and define $f^A a = g \cdot a$. This is well defined: suppose that $a \in A_{k'}$ too, and $g' \in G$ with $g'|_{k'} = f|_{k'}$. Then $a \in A_l$ where $l = k \cap k'$, also $g|_l = g'|_l = f|_l$ and so $(g^{-1}g')|_l = id|_l$, that is $g^{-1}g' \in G_l$. Thus

$$g \cdot a = g \cdot ((g^{-1}g') \cdot a) = g' \cdot a$$

and the choice of k and g does not matter.

There is an alternative action of G on the destination $|A|$ which makes this induced map a morphism in $\mathbf{B}G$. For $g \in G$, we define $f^*g \in G$ by

$$(f^*g)n \;=\; \begin{cases} n & \text{if } n \notin \mathit{Im}(f) \\ f(g(f^{-1}n)) & \text{if } n \in \mathit{Im}(f). \end{cases}$$

In particular $(f^*g) \circ f = f \circ g$. We take fA to be the G-set with underlying set $|A|$ and action

$$g \cdot_{fA} a \;=\; (f^*g) \cdot_A a.$$

Then $f^A : A \to fA$ is a morphism in $\mathbf{B}G$, and every commuting trangle of injections

$$\begin{tikzcd} \mathbb{N} \arrow[r, tail, "f"] \arrow[dr, tail, "f'"'] & \mathbb{N} \arrow[d, tail, "h"] \\ & \mathbb{N} \end{tikzcd}$$

induces a commuting triangle in $\mathbf{B}G$

$$\begin{tikzcd} A \arrow[r, "f^A"] \arrow[dr, "f'^A"'] & fA \arrow[d, "h^{fA}"] \\ & f'A \end{tikzcd}$$

by the action of f, f' and h on the underlying set $|A|$.

The object TA is the colimit of the diagram comprising all such triangles from A. That is, we consider in turn the one object category $\mathcal{N}$ of $\mathbb{N}$ and all injections into itself, the coslice category $\mathcal{N} \setminus \mathbb{N}$, the diagram this induces in $\mathbf{B}G \setminus A$ and its colimit $\eta_A : A \to TA$. In fact the injection $f : \mathbb{N} \rightarrowtail \mathbb{N}$ induces an endofunctor $f_{\mathbf{B}G} : \mathbf{B}G \to \mathbf{B}G$, and the commuting triangle induces one of natural transformations between endofunctors:

$$\begin{tikzcd} id_{\mathbf{B}G} \arrow[r, "f"] \arrow[dr, "f'"'] & f_{\mathbf{B}G} \arrow[d, "h"] \\ & f'_{\mathbf{B}G} \end{tikzcd}.$$

The colimit of all such triangles is the endofunctor $T : \mathbf{B}G \to \mathbf{B}G$.

We could use a much smaller diagram for the colimit. For example, we might take for f only injections of the form $i \mapsto i + t$, and for h all morphisms mediating between these. Or for f just the injection $i \mapsto 2i$ and for h all maps that permute the odd numbers and fix the even numbers. Both of these give countable diagrams with the same colimit as before.

We represent elements of TA as $[f, a]$ where $f : \mathbb{N} \rightarrowtail \mathbb{N}$ is an injection and $a \in fA$. The natural transformations that make T a strong monad are:

$$\begin{array}{rcl} \eta_A : A & \to & TA \\ a & \mapsto & [id_{\mathbb{N}}, a] \end{array} \qquad \begin{array}{rcl} \mu_A : T^2 A & \to & TA \\ [f, [f', a]] & \mapsto & [f' \circ f, a] \end{array}$$

$$\begin{array}{rcl} t_{A,B} : A \times TB & \to & T(A \times B) \\ \langle a, [f, b] \rangle & \mapsto & [f, \langle f^A a, b \rangle] \end{array}$$

If $p : A \times B \to TC$ is a morphism in $\mathbf{B}G$ then its lift $p^* : A \times TB \to TC$ is given by

$$p^* \langle a, [f, b] \rangle = [f' \circ f, c] \quad \text{where} \quad p \langle f^A a, b \rangle = [f', c].$$

For any G-set A with trivial G-action, the monad is the identity: $TA = A$. This happens with the interpretation of any nu-calculus type that does not use ν. The following are more interesting examples:

$$\begin{array}{rcl} TN & = & N + 1 \qquad new = inr : 1 \to N + 1 \\ T(N \times N) & = & ((N + 1) \times (N + 1)) + 1 \\ T(N \to T(1 + 1)) & = & ((N + N) \to (1 + 1)) / \sim \\ T(N \to TN) & = & ((N + N) \to (N + N + 1)) / \sim \end{array}$$

For $T(N \times N)$ the action of G is on both copies of N at once. In the last two cases, the relation '$\sim$' quotients by the action of G on the second N in each $(N + N)$, while the actual action of G is on the first.

Now that the foundations are laid, the method of Section 4 constructs a model for the nu-calculus, with a continuous G-set $[\![\sigma]\!]$ for each type σ, and morphisms for expressions. Conveniently, morphisms $(\neq s) \to A$ in $\mathbf{B}G$ are in bijection with the elements of $|A|$, given some ordering on s, so expressions are also interpreted by elements:

$$\begin{array}{ll} [\![M']\!]_{\Gamma,\neq s} \in (T[\![\sigma]\!])^{[\![\Gamma]\!]} & M' \in \mathrm{Exp}_\sigma(s,\Gamma) \\ |C'|_{\Gamma,\neq s} \in [\![\sigma]\!]^{[\![\Gamma]\!]} & C' \in \mathrm{Can}_\sigma(s,\Gamma) \\ [\![M]\!]_{\neq s} \in T[\![\sigma]\!] & M \in \mathrm{Exp}_\sigma(s) \\ |C|_{\neq s} \in [\![\sigma]\!] & C \in \mathrm{Can}_\sigma(s). \end{array}$$

All the results of Sections 4 and 6 carry across to this model in $\mathbf{B}G$: it correctly interprets the operational semantics; it is adequate for reasoning about contextual equivalence, with completeness at ground types; it validates examples (2)–(9) and (17) of Chapter 2, Section 5; and it agrees with applicative equivalence up to second order, but is weaker at higher types.

Chapter 4

Logical Relations

The previous two chapters have provided various methods, both operational and denotational, for proving that two expressions of the nu-calculus are contextually equivalent. These are enough to confirm most of the example equivalences given in Section 5 of Chapter 2. Two however remain:

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

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

Both of these equivalences rely on local names remaining private; for example in (12) the function $(\lambda x{:}\nu.(x = n))$ would return the answer true, if applied to the name n, but no external context can detect this. Similarly in (13) no externally produced function can distinguish the private names n and n'.

In this chapter we develop methods that can prove both of these equivalences, by identifying the different uses an expression may make of its local names. We show that these methods are strong enough to prove all contextual equivalences between expressions of first-order type, and construct a categorical model that is fully abstract at ground and first-order types.

The techniques that can do this all use some kind of *logical relation*. We encountered these earlier, in the context of Definition 2.16 of logical equivalence. The basic idea is to construct (binary) relations between nu-calculus expressions by induction on the structure of their type, with functions related if they take related arguments to related results. How successful this is depends on careful choice of the relations used at ground types, and the treatment of expressions not in canonical form.

Section 1 describes an operational form of logical relations for the nu-calculus, and shows that these can be used to prove contextual equivalences, in particular example (12) above. A significant intermediate step is the extension of contextual equivalence to *contextual relations* between nu-calculus expressions. Section 2 gives a proof that the method of logical relations is complete for first-order types; this is probably the most powerful result of the thesis.

Sections 3, 4 and 5 develop a denotational analogue of this work. This uses the notion of a *category with relations*, and extends the usual categorical idea of naturality with *parametricity*, where relations too must be preserved and respected. The result is a category $\mathcal{P}$, based on the functor category $Set^{\mathcal{I}}$ of Chapter 3, but incorporating a relational structure. We prove various results connecting this with operational logical relations, and show that $\mathcal{P}$ provides a model of the nu-calculus that is fully abstract up to first-order function types.

The final section tackles the remaining equivalence (13), between two functions of second-order type. This is not validated by ordinary logical relations, and we construct more sophisticated *predicated logical relations* which provide an even finer description of how expressions use their local names. While this successfully proves equivalence (13), it is a technique very much targeted at this one example, and no completeness result is proved. There are however some possibilities for further generalisation, which may yet lead to a fully abstract categorical model for the nu-calculus.

1 Operational Logical Relations

This section describes a system of binary logical relations between nu-calculus expressions, based on their operational behaviour. These take a similar form to the logical equivalence of Chapter 2, Section 7, but are extended to allow relations other than the identity at ground types. We show that they have certain good properties, and in particular can be used to demonstrate contextual equivalence. This then gives a proof of the problematic example (12) involving two first-order functions with private names.

We begin by identifying a certain kind of relation between sets of names:

Definition 4.1 (Spans) If s_1 and s_2 are sets of names, then a *span*, or *partial bijection* $R : s_1 \rightleftharpoons s_2$ is an injective partial map from s_1 to s_2. That is, the graph $R \subseteq s_1 \times s_2$ satisfies

$$n_1 \; R \, n_2 \quad \& \quad n_1' \; R \, n_2' \quad \Longrightarrow \quad (\, n_1 = n_1' \iff n_2 = n_2' \,).$$

A span can also be represented as a pair of injections $s_1 \leftarrowtail R \rightarrowtail s_2$.

If $R' : s_1' \rightleftharpoons s_2'$ is another span, with s_1' and s_2' disjoint from s_1 and s_2 respectively, then the disjoint union of R and R' is also a span:

$$R \oplus R' : s_1 \oplus s_1' \rightleftharpoons s_2 \oplus s_2'.$$

The *identity* span $id_s : s \rightleftharpoons s$ has

$$n \; id_s \, n' \quad \iff \quad n = n'.$$

The domain and codomain of definition for $R : s_1 \rightleftharpoons s_2$ are defined by:

$$\begin{aligned} \mathrm{dom}(R) &= \{\, n_1 \in s_1 \mid \exists n_2 \in s_2 \,.\, n_1 \; R \, n_2 \,\} \\ \mathrm{cod}(R) &= \{\, n_2 \in s_2 \mid \exists n_1 \in s_1 \,.\, n_1 \; R \, n_2 \,\}. \end{aligned}$$

We generally represent spans by variants of R and S.

The intuition behind spans is that they should capture the use that nu-calculus expressions make of public and private names. So if $R : s_1 \rightleftharpoons s_2$, then 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. In this spirit, Definition 2.16 of logical equivalence now extends to a method that takes a span on names to a set of relations between expressions of all types:

Definition 4.2 (Logical Relations) If $R : s_1 \rightleftharpoons s_2$ is a span then the relations

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

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

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

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

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

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

The relations R^{can}_{σ} and R^{exp}_{σ} coincide on canonical forms, and we may write them as R_{σ} indiscriminately. We can extend the relations to open expressions: if $M_1 \in \mathrm{Exp}_{\sigma}(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_{\sigma}(s_2, \Gamma)$ where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ then define

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

We use the symbol '$\neg$' to negate relations; so $\Gamma \vdash M_1 \neg R_{\sigma} \; M_2$ holds if and only if $\Gamma \vdash M_1 \; R_{\sigma} \; M_2$ does not.

Lemma 4.3 *If $R : s_1 \rightleftharpoons s_2$ and $M_1 \in \mathrm{Exp}_{\sigma}(s_1, \Gamma)$, $M_2 \in \mathrm{Exp}_{\sigma}(s_2, \Gamma)$ then*

$$\Gamma \vdash M_1 \; R_{\sigma} \; M_2 \iff \Gamma \oplus \Gamma' \vdash M_1 \; (R \oplus R')_{\sigma} \; M_2$$

for any Γ' and $R' : s'_1 \rightleftharpoons s'_2$.

Proof The result for closed expressions is shown by induction on the structure of the type σ, and the general result follows. A mild complication is the need to prove that there are some related expressions at every type:

$$\begin{array}{l} \forall \sigma.\, \forall R : s_1 \rightleftharpoons s_2\,.\, \exists R' : s'_1 \rightleftharpoons s'_2\,, \\ \qquad C_1 \in \mathrm{Can}_{\sigma}(s_1 \oplus s'_1),\; C_2 \in \mathrm{Can}_{\sigma}(s_2 \oplus s'_2)\,.\, C_1 \; (R \oplus R')_{\sigma} \; C_2, \end{array}$$

which can also be done by induction over types. □

Lemma 4.4 *Logical relations are respected by all the rules for forming expressions of the nu-calculus.*

Proof It is straightforward to show from the definition of logical relations that they are preserved by each of the expression-forming rules in Figure 2.1. □

This last result explains why the choice of relations at ground types is restricted to the identity for booleans and partial bijections for names; more general relations would not be preserved by conditionals and the equality test on names.

The definition of the relation at function types involves quantification over all further spans. This is because functions may be applied to arguments that use additional names; two functions are related only if they take related arguments to related results at all possible later stages of computation.

The use of existential quantification in the definition of R_σ^{exp} is crucial: two expressions are related if there is some way to span their locally-defined names, such that the computed values are related. This gives an essential flexibility to how expressions may use their local names. There is here a similarity to Plotkin and Abadi's work on relational parametricity for polymorphism in System F, and their definition of relations between terms of existential type [97]. There, two terms are related if there is some suitable way to relate their hidden types such that their values are related.

Note that the logical relations for closed expressions of ground type are comparatively straightforward:

- $B_1\ R_o\ B_2$ if both evaluate to $true$, or both to $false$.
- $N_1\ R_\nu\ B_2$ if they evaluate as $s_1 \vdash N_1 \Downarrow_\nu (s_1')n_1$ and $s_2 \vdash N_2 \Downarrow_\nu (s_2')n_2$ with either $n_1\ R\ n_2$, or $n_1 \in s_1'$ and $n_2 \in s_2'$.

Extension to logical relations of other arities seems possible, though it is not clear that it would be helpful (but see Abadi and Plotkin [2] on nullary and unary relations for System F).

Before we can state useful results for logical relations, we must define some other families of relations. These are also based on spans, extending syntactic identity and contextual equivalence respectively.

Definition 4.5 (Syntactic Relations) For any span $R : s_1 \rightleftharpoons s_2$ the relation

$$R_\sigma^{syn} \ \subseteq\ \mathrm{Exp}_\sigma(s_1,\Gamma) \times \mathrm{Exp}_\sigma(s_2,\Gamma)$$

is defined by

$$\Gamma \vdash M_1\ R_\sigma^{syn}\ M_2 \quad \iff \quad \begin{array}{l} M_1 \in \mathrm{Exp}_\sigma(\mathrm{dom}(R),\Gamma)\ \ \&\ \ M_2 \in \mathrm{Exp}_\sigma(\mathrm{cod}(R),\Gamma) \\ \&\ \ M_2 = M_1[n_2/n_1 \mid n_1\ R\ n_2]. \end{array}$$

When this holds we say that M_1 and M_2 are *syntactically R-related.* In particular, $(id_s)_\sigma^{syn}$ is syntactic identity.

Definition 4.6 (Contextual Relations) For any span $R : s_1 \rightleftharpoons s_2$ the relation

$$R_\sigma^{cxt} \ \subseteq\ \mathrm{Exp}_\sigma(s_1,\Gamma) \times \mathrm{Exp}_\sigma(s_2,\Gamma)$$

is to hold between expressions that behave similarly in all syntactically R-related contexts. So the expressions M_1 and M_2 are *contextually R-related* if

$$P_1\langle\!\langle - \rangle\!\rangle\ R_\tau^{syn}\ P_2\langle\!\langle - \rangle\!\rangle \quad \Longrightarrow \quad P_1\langle\!\langle (\Gamma)M_1 \rangle\!\rangle\ R_\tau\ P_2\langle\!\langle (\Gamma)M_2 \rangle\!\rangle$$

where $P_1\langle\!\langle-\rangle\!\rangle$ and $P_2\langle\!\langle-\rangle\!\rangle$ are suitable closed contexts of some type $\tau \in \{o,\nu\}$. In the light of the Context Lemma (Theorem 2.8), we actually define the relation using only argument contexts:

$$\begin{array}{l}\Gamma \vdash M_1\ R^{cxt}_\sigma\ M_2 \iff \\ \qquad \forall R' : s'_1 \rightleftharpoons s'_2,\ \tau \in \{o,\nu\}, \\ \qquad \lambda x{:}\sigma.T_i \in \mathrm{Can}_{\sigma\to\tau}(s_i \oplus s'_i),\ C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s'_i) \quad i=1,2 \quad j=1,\dots,n\,. \\ \qquad\quad (\ \&_{j=1}^n\,.\, C_{1j}\ (R\oplus R')^{syn}_{\sigma_j}\ C_{2j}\ )\ \ \&\ \ (\lambda x{:}\sigma.T_1)\ (R\oplus R')^{syn}_{\sigma\to\tau}\ (\lambda x{:}\sigma.T_2) \\ \qquad\qquad \implies (\lambda x{:}\sigma.T_1)M_1[\vec{C_1}/\vec{x}]\ (R\oplus R')_\tau\ (\lambda x{:}\sigma.T_2)M_2[\vec{C_2}/\vec{x}].\end{array}$$

So two expressions are contextually related if they agree under all syntactically related instantiations and test functions.

This definition is rather more sophisticated than that for contextual equivalence, with contexts of name as well as boolean type, and a logical relation $(R\oplus R')_\tau$ on the right hand side. Nevertheless it does specialise to give:

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

With these definitions we can extend some of the results of Chapter 2 on logical equivalence to ones about logical relations. We have that syntactically related expressions are logically related, and logically related ones are contextually related:

Proposition 4.7 *If $\Gamma \vdash M_1\ R^{syn}_\sigma\ M_2$ then $\Gamma \vdash M_1\ R_\sigma\ M_2$. In particular the relation $(id_s)_\sigma$ is reflexive: $\Gamma \vdash M\ (id_s)_\sigma\ M$ for any $M \in \mathrm{Exp}_\sigma(s,\Gamma)$.*

Proof From the definition of syntactic equivalence, M_1 and M_2 have the same structure, and induction over this using Lemma 4.4 shows that they are logically related. □

Proposition 4.8 *If $\Gamma \vdash M_1\ R_\sigma\ M_2$ then $\Gamma \vdash M_1\ R^{cxt}_\sigma\ M_2$. In particular (id_s)-related expressions are contextually equivalent:*

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

Proof By Proposition 4.7, any syntactically related test functions and instantiations of Γ are logically related. The result then follows from the definitions of contextual and logical relations. □

The following results show that the power of logical relations includes that of applicative equivalence:

Theorem 4.9 *If $R : s_1 \rightleftharpoons s_2$ with $M_1 \in \mathrm{Exp}_\sigma(s_1,\Gamma)$ and $M_2, M_3 \in \mathrm{Exp}_\sigma(s_2,\Gamma)$ then*

$$\Gamma \vdash M_1\ R_\sigma\ M_2 \quad\&\quad s_2,\Gamma \vdash M_2 \sim_\sigma M_3 \quad\implies\quad \Gamma \vdash M_1\ R_\sigma\ M_3.$$

Proof We show each of the following:

1. $C_1\ R^{can}_\sigma\ C_2\ \ \&\ \ s_2 \vdash C_2 \sim^{can}_\sigma C_3 \implies C_1\ R^{can}_\sigma\ C_3$.
2. $M_1\ R^{exp}_\sigma\ M_2\ \ \&\ \ s_2 \vdash M_2 \sim^{exp}_\sigma M_3 \implies M_1\ R^{exp}_\sigma\ M_3$.

3. $\Gamma \vdash M_1\ R_\sigma\ M_2\ \ \&\ \ s_2, \Gamma \vdash M_2 \sim_\sigma M_3 \implies M_1\ R_\sigma\ M_3$.

The first two are proved by mutual induction over the structure of σ. Case (1) is immediate at ground types. For function types, suppose that $(\lambda x{:}\sigma.M_1)\ R^{can}_{\sigma\to\sigma'}\ (\lambda x{:}\sigma.M_2)$ and $s_1 \vdash \lambda x{:}\sigma.M_2 \sim^{can}_{\sigma\to\sigma'} \lambda x{:}\sigma.M_3$, with $C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s_1')$, $C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s_2')$ such that $C_1\ (R \oplus R')^{can}_\sigma\ C_2$ for some $R' : s_1' \rightleftharpoons s_2'$. Then by definition $M_1[C_1/x]\ (R \oplus R')^{exp}_{\sigma'}$ $M_2[C_2/x]$ and $s_2 \oplus s_2' \vdash M_2[C_2/x] \sim^{exp}_{\sigma'} M_3[C_2/x]$. Applying the induction hypothesis gives $M_1[C_1/x]\ (R \oplus R')^{exp}_{\sigma'}\ M_3[C_2/x]$ and so confirms $(\lambda x{:}\sigma.M_1)\ R^{can}_{\sigma\to\sigma'}\ (\lambda x{:}\sigma.M_3)$.

For (2) suppose that we have $M_1\ R^{exp}_\sigma\ M_2$ and $s_2 \vdash M_2 \sim^{exp}_\sigma M_3$. That is:

$$s_1 \vdash M_1 \Downarrow_\sigma (s_1')C_1, \qquad s_2 \vdash M_2 \Downarrow_\sigma (s_2')C_2 \qquad \text{and} \qquad s_2 \vdash M_3 \Downarrow_\sigma (s_3')C_3,$$

with $C_1\ (R \oplus R')^{can}_\sigma\ C_2$ for some $R' : s_1' \rightleftharpoons s_2'$, and $s_2 \oplus (s_2' \cup s_3') \vdash C_2 \sim^{can}_\sigma C_3$. Using Lemma 4.3 and (1) we obtain $C_1\ (R \oplus S)^{can}_\sigma\ C_3$, where $S : s_1' \rightleftharpoons s_3'$ is defined as $S = R' \cap (s_1' \times s_3')$. Thus $M_1\ R^{exp}_\sigma\ M_3$ as required.

Finally, for (3) suppose that $\Gamma \vdash M_1\ R_\sigma\ M_2$, that $s_2, \Gamma \vdash M_2 \sim_\sigma M_3$, and with some $R' : s_1' \rightleftharpoons s_2'$ we have $C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s_i')$ for $i = 1, 2$ and $j = 1, \ldots, n$ such that each $C_{1j}\ (R \oplus R')^{can}_{\sigma_j}\ C_{2j}$. Then by definition $M_1[\vec{C_1}/\vec{x}]\ (R \oplus R')^{exp}_\sigma\ M_2[\vec{C_2}/\vec{x}]$ and $s_2 \vdash M_2[\vec{C_2}/\vec{x}] \sim_\sigma M_3[\vec{C_2}/\vec{x}]$. Applying (2) gives us $M_1[\vec{C_1}/\vec{x}]\ (R \oplus R')^{exp}_\sigma\ M_3[\vec{C_2}/\vec{x}]$ and hence $\Gamma \vdash M_1\ R_\sigma\ M_3$ as desired. □

Corollary 4.10 *If* $s, \Gamma \vdash M_1 \sim_\sigma M_2$ *then* $\Gamma \vdash M_1\ (id_s)_\sigma\ M_2$.

Proof By Proposition 4.7, $\Gamma \vdash M_1\ (id_s)_\sigma\ M_1$ and the result follows as an instance of Theorem 4.9 above. □

So logical relations can be used to prove contextual equivalence, and are at least as strong as applicative equivalence. This at once tells us that they are enough to prove the basic examples (2)–(9) and (17) of Chapter 2. More importantly though, the relations describe private names to a degree, and can prove results like example (12). This concerns the equivalence

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

Now the only possible span $R : \{n\} \rightleftharpoons \{\}$ is the empty relation, and with this we have that

$$(\lambda x{:}\nu.(x = n))\ \ R^{can}_{\nu\to o}\ \ (\lambda x{:}\nu.\mathit{false})$$

because both functions take R-related names to false. But then

$$(\nu n.\lambda x{:}\nu.(x = n))\ \ (id_\emptyset)^{exp}_{\nu\to o}\ \ (\lambda x{:}\nu.\mathit{false})$$

with the above relation R satisfying the existential quantifier, and applying Proposition 4.8 shows that the expressions are contextually equivalent as claimed.

In the next section we shall see that logical relations can prove all contextual equivalences, up to first-order function types. So the notion of a span $R : s_1 \rightleftharpoons s_2$ completely captures the privacy and visibility of names as passed between functions of the nu-calculus. At higher types difficulties can arise, with names being only partially revealed, as in example (15) from page 25. As a consequence, logical relations are not enough to prove the equivalence (13):

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

Again the only span $R : \{n, n'\} \rightleftharpoons \{\}$ is the empty one, and the first-order functions

$$(\lambda x{:}\nu.(x = n)) \;\; R^{can}_{\nu\to o} \;\; (\lambda x{:}\nu.\mathit{false})$$

are related because they take R-related names to false. But the two second-order functions above differ at these arguments, so they are not related:

$$(\lambda f{:}\nu{\to}o.(fn = fn')) \;\; \neg\, R^{can}_{(\nu\to o)\to o} \;\; (\lambda f{:}\nu{\to}o.\mathit{true}),$$

and logical relations cannot prove the stated contextual equivalence. Later we shall see how the method of relations can be enhanced to prove even this example.

A final curiosity is that the relation $(id_s)_\sigma$ is not necessarily transitive at higher-order types. Indeed there are expressions of second-order type which are not (id_s)-related, but can still be shown contextually equivalent through a chain of other expressions, each logically related to the next. This makes it hard to gauge the full power of the technique; however even such a roundabout method does not seem to work for equivalence (13) above.

2 Completeness at First-Order Types

In the previous section we saw that logical relations can be used to show contextual equivalence in the nu-calculus; we now demonstrate that this technique is complete up to types of first order. This requires various results on contextual relations, beginning with the observation that for closed expressions we need not explicitly consider tests at later stages:

Proposition 4.11 *For any span $R : s_1 \rightleftharpoons s_2$ and closed expressions $M_1 \in \mathrm{Exp}_\sigma(s_1)$, $M_2 \in \mathrm{Exp}_\sigma(s_2)$:*

$$\begin{aligned} &M_1 \; R^{cxt}_\sigma \; M_2 \iff \\ &\qquad \forall \tau \in \{o, \nu\},\ \lambda x{:}\sigma.T_1 \in \mathrm{Can}_{\sigma\to\tau}(s_1),\ \lambda x{:}\sigma.T_2 \in \mathrm{Can}_{\sigma\to\tau}(s_2)\ . \\ &\qquad\quad (\lambda x{:}\sigma.T_1)\; R^{syn}_{\sigma\to\tau}\; (\lambda x{:}\sigma.T_2) \implies (\lambda x{:}\sigma.T_1)M_1 \; R_\tau \; (\lambda x{:}\sigma.T_2)M_2. \end{aligned}$$

Proof The forward direction is immediate. For the reverse, suppose that $M_1 \neg R^{cxt}_\sigma M_2$ and that they are distinguished by the test functions

$$(\lambda x{:}\sigma.T_1) \;\; (R \oplus R')^{syn}_{\sigma\to\tau} \;\; (\lambda x{:}\sigma.T_2)$$

where $R' : s'_1 \rightleftharpoons s'_2$. If we take $t_1 = \mathrm{dom}(R') \subseteq s'_1$ and $t_2 = \mathrm{cod}(R') \subseteq s'_2$, then the functions

$$(\lambda x{:}\sigma.\nu t_1.T_1) \;\; R^{syn}_{\sigma\to\tau} \;\; (\lambda x{:}\sigma.\nu t_2.T_2)$$

will generally serve instead, given that

$$\begin{aligned} s_1 \oplus s'_1 \vdash (\lambda x{:}\sigma.T_1)M_1 \Downarrow_\tau (s''_1)C_1 &\iff s_1 \vdash (\lambda x{:}\sigma.\nu t_1.T_1)M_1 \Downarrow_\tau (t_1 \oplus s''_1)C_1 \\ s_2 \oplus s'_2 \vdash (\lambda x{:}\sigma.T_2)M_2 \Downarrow_\tau (s''_2)C_2 &\iff s_2 \vdash (\lambda x{:}\sigma.\nu t_2.T_2)M_2 \Downarrow_\tau (t_2 \oplus s''_2)C_2 \end{aligned}$$

and the fairly simple nature of the logical relation $(R \oplus R')_\tau$ at ground τ. The only exception is when $\tau = \nu$ and

$$s_i \oplus s'_i \vdash (\lambda x{:}\sigma.T_i)M_i \Downarrow_\nu (s''_i)n_i \qquad i = 1, 2$$

with $n_1 \in t_1$ or $n_2 \in t_2$ but $(n_1, n_2) \notin R'$. Without loss of generality we take $n_1 \in t_1$ and pick $n_2' \in t_2$ with $n_1 \; R' \; n_2'$, so $n_2 \neq n_2'$. Then the tests

$$(\lambda x{:}\sigma.\nu t_1.(T_1 = n_1)) \;\; R^{syn}_{\sigma\to\tau} \;\; (\lambda x{:}\sigma.\nu t_2.(T_2 = n_2'))$$

give *true* and *false* when applied to M_1 and M_2 respectively. □

We can now prove a series of lemmas about contextual relations, matching the defining clauses for logical relations:

Lemma 4.12 *For canonical expressions of ground type $\tau \in \{o, \nu\}$ the relations R^{syn}_τ, R^{cxt}_τ and R^{can}_τ all coincide.*

Proof It is immediate from their definitions that R^{syn}_τ and R^{can}_τ are the same; for R^{cxt}_τ, consider the test function $(\lambda x{:}\tau.x)$. □

Lemma 4.13 *Contextually related lambda abstractions, applied to syntactically related values, give contextually related results. Suppose that $R : s_1 \rightleftharpoons s_2$ and $R' : s_1' \rightleftharpoons s_2'$ with*

$$\begin{array}{ll} (\lambda x{:}\sigma.M_1) \in \mathrm{Can}_{\sigma\to\sigma'}(s_1) & C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s_1') \\ (\lambda x{:}\sigma.M_2) \in \mathrm{Can}_{\sigma\to\sigma'}(s_2) & C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s_2'). \end{array}$$

Then

$$\begin{array}{l} (\lambda x{:}\sigma.M_1) \; R^{cxt}_{\sigma\to\sigma'} \; (\lambda x{:}\sigma.M_2) \;\;\&\;\; C_1 \; (R \oplus R')^{syn}_\sigma \; C_2 \\ \qquad\qquad\qquad\qquad \Longrightarrow \; M_1[C_1/x] \; (R \oplus R')^{cxt}_{\sigma'} \; M_2[C_2/x]. \end{array}$$

Proof Suppose that we have test functions

$$(\lambda x{:}\sigma'.T_1) \;\; (R \oplus R')^{syn}_{\sigma'\to\tau} \;\; (\lambda x{:}\sigma'.T_2)$$

to apply to $M_1[C_1/x]$ and $M_2[C_2/x]$. Then these give the same results as the functions

$$(\lambda f{:}\sigma \to \sigma'.(\lambda x{:}\sigma'.T_1)(f C_1)) \;\; (R \oplus R')^{syn}_{(\sigma\to\sigma')\to\tau} \;\; (\lambda f{:}\sigma \to \sigma'.(\lambda x{:}\sigma'.T_2)(f C_2))$$

applied to $(\lambda x{:}\sigma.M_1)$ and $(\lambda x{:}\sigma.M_2)$. □

This next result is central to the proof of completeness, as it shows how the use of spans between name sets really can capture the way different expressions make public their local names.

Lemma 4.14 *Closed expressions are contextually R-related if and only if they evaluate to contextually $(R \oplus R')$-related canonical forms, for some R'. That is, for $R : s_1 \rightleftharpoons s_2$ and $M_1 \in \mathrm{Exp}_\sigma(s_1)$, $M_2 \in \mathrm{Exp}_\sigma(s_2)$:*

$$\begin{array}{lll} M_1 \; R^{cxt}_\sigma \; M_2 & \iff & \exists R' : s_1' \rightleftharpoons s_2',\, C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s_1'),\, C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s_2') \;. \\ & & \quad s_1 \vdash M_1 \Downarrow_\sigma (s_1')C_1 \;\;\&\;\; s_2 \vdash M_2 \Downarrow_\sigma (s_2')C_2 \\ & & \quad \&\;\; C_1 \; (R \oplus R')^{cxt}_\sigma \; C_2. \end{array}$$

Proof We begin with the implication from right to left. Suppose that for $i = 1, 2$ we have the test functions $\lambda x{:}\sigma.T_i \in \mathrm{Can}_{\sigma\to\tau}(s_i)$ with $\tau \in \{o, \nu\}$ and

$$(\lambda x{:}\sigma.T_1)\ \ R^{syn}_{\sigma\to\tau}\ \ (\lambda x{:}\sigma.T_2);$$

then we have by hypothesis that

$$(\lambda x{:}\sigma.T_1)C_1\ \ (R \oplus R')_\tau\ \ (\lambda x{:}\sigma.T_2)C_2.$$

So there must be $R'' : s''_1 \rightleftharpoons s''_2$ and $C'_i \in \mathrm{Can}_\tau(s_i \oplus s'_i \oplus s''_i)$ for $i = 1, 2$ with

$$s_i \oplus s'_i \vdash (\lambda x{:}\sigma.T_i)C_i \Downarrow_\tau (s''_i)C'_i \qquad i = 1, 2$$

and $C'_1\ (R \oplus R' \oplus R'')_\tau\ C'_2$. Combining evaluation judgements we have that

$$s_i \vdash (\lambda x{:}\sigma.T_i)M_i \Downarrow_\tau (s'_i \oplus s''_i)C'_i \qquad i = 1, 2$$

and then

$$(\lambda x{:}\sigma.T_1)M_1\ \ R_\tau\ \ (\lambda x{:}\sigma.T_2)M_2$$

as required.

The forward implication is rather more difficult and requires some ingenuity. Suppose that $M_1\ R^{cxt}_\sigma\ M_2$ and $s_i \vdash M_i \Downarrow_\sigma (s'_i)C_i$ for $i = 1, 2$; we need to pick some span $R' : s'_1 \rightleftharpoons s'_2$ such that $C_1\ (R \oplus R')^{cxt}_\sigma\ C_2$.

Take R' to relate those names that can be simultaneously produced from C_1 and C_2 by syntactically R-related expressions of type $(\sigma \to \nu)$. That is, for $(n_1, n_2) \in s'_1 \times s'_2$ we have:

$$\begin{aligned} n_1\ R'\ n_2 \quad \iff \quad & \exists \lambda x{:}\sigma.N_1 \in \mathrm{Can}_{\sigma\to\nu}(s_1),\ \lambda x{:}\sigma.N_2 \in \mathrm{Can}_{\sigma\to\nu}(s_2)\ . \\ & \quad (\lambda x{:}\sigma.N_1)\ R^{syn}_{\sigma\to\nu}\ (\lambda x{:}\sigma.N_2) \\ & \quad \&\ \ s_1 \oplus s'_1 \vdash (\lambda x{:}\sigma.N_1)C_1 \Downarrow_\nu (s''_1)n_1 \\ & \quad \&\ \ s_2 \oplus s'_2 \vdash (\lambda x{:}\sigma.N_2)C_2 \Downarrow_\nu (s''_2)n_2 \end{aligned}$$

The intuition here is that R' should identify local names of M_1 and M_2 that are public, with private names being unrelated. We enumerate the elements of R' as $\{(n_{1j}, n_{2j}) \mid j = 1, \ldots, k\}$ and record witnessing expressions $(\lambda x{:}\sigma.N_{ij})$ for $i = 1, 2$ and $j = 1, \ldots, k$.

To show that R' is a partial bijection, suppose that $n_1\ R'\ n_2$ and $n'_1\ R'\ n'_2$ with witnesses $(\lambda x{:}\sigma.N_i)$ and $(\lambda x{:}\sigma.N'_i)$ respectively. Then we have the syntactically related test functions

$$(\lambda x{:}\sigma.N_1 = N'_1)\ \ R^{syn}_{\sigma\to o}\ \ (\lambda x{:}\sigma.N_2 = N'_2)$$

which, applied to $M_1\ R^{cxt}_\sigma\ M_2$, give

$$((\lambda x{:}\sigma.N_1 = N'_1)M_1)\ \ R_o\ \ ((\lambda x{:}\sigma.N_2 = N'_2)M_2).$$

From this it follows that $n_1 = n'_1$ if and only if $n_2 = n'_2$, according to whether the boolean expressions above evaluate to $true$ or $false$.

We now demonstrate that $C_1\ (R \oplus R')^{cxt}_\sigma\ C_2$. Suppose we have test functions

$$(\lambda x{:}\sigma.T_1)\ \ (R \oplus R')^{syn}_{\sigma\to\tau}\ \ (\lambda x{:}\sigma.T_2) \qquad \tau \in \{o, \nu\},$$

with

$$s_i \oplus s'_i \vdash (\lambda x{:}\sigma.T_i)C_i \Downarrow_\tau (s''_i)C'_i \qquad i = 1, 2.$$

We need to show that $C'_1\ (R \oplus R' \oplus R'')_\tau\ C'_2$ for some $R'' : s''_1 \rightleftharpoons s''_2$. If we construct the test functions

$$\begin{aligned}\lambda x{:}\sigma.U_i \;&=\; \lambda x{:}\sigma.((\lambda y_1{:}\nu \ldots \lambda y_k{:}\nu\ .\ T_i[y_j/n_{ij} \mid j = 1, \ldots, k])N_{i1} \ldots N_{ik}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \in\ \mathrm{Can}_{\sigma\to\tau}(s_i) \qquad i = 1, 2\end{aligned}$$

then we have

$$(\lambda x{:}\sigma.U_1)\ \ R^{syn}_{\sigma\to\tau}\ \ (\lambda x{:}\sigma.U_2)$$

and

$$s_i \vdash (\lambda x{:}\sigma.U_i)M_i \Downarrow_\tau (s'_i \oplus t_i \oplus s''_i)C'_i \qquad i = 1, 2,$$

where the t_i are the additional names created by the evaluation of the N_{ij}. As $M_1\ R^{cxt}_\sigma\ M_2$, we have that

$$C'_1\ (R \oplus S)_\tau\ C'_2 \qquad \text{for some} \qquad S : s'_1 \oplus t_1 \oplus s''_1 \rightleftharpoons s'_2 \oplus t_2 \oplus s''_2.$$

If $\tau = o$ then this at once gives $C'_1 = C'_2$, so $C'_1\ (R \oplus R' \oplus R'')_o\ C'_2$ for any choice of $R'' : s''_1 \rightleftharpoons s''_2$, and $C_1\ (R \oplus R')^{cxt}_\sigma\ C_2$ as required.

When $\tau = \nu$ we need to consider where the names C'_1 and C'_2 might lie:

- If $C'_1 \in s_1$ then $C'_2 \in s_2$ and $C'_1\ R\ C'_2$.
- If $C'_1 \in s'_1$ then $C'_2 \in s'_2 \oplus s''_2$. Taking the syntactically R-related test functions

  $$(\lambda x{:}\sigma.(U_i = U_i))\ \in\ \mathrm{Can}_{\sigma\to o}(s_i) \qquad i = 1, 2,$$

  we have that

  $$s_1 \vdash (\lambda x{:}\sigma.(U_1 = U_1))M_1 \Downarrow_o (s'_1 \oplus t_1 \oplus s''_1 \oplus t_3 \oplus s''_3)\mathit{true}$$

  where t_3, s''_3 are copies of t_1, s''_1. Contextual equivalence then gives

  $$s_2 \vdash (\lambda x{:}\sigma.(U_2 = U_2))M_2 \Downarrow_o (s'_2 \oplus t_2 \oplus s''_2 \oplus t_4 \oplus s''_4)\mathit{true}$$

  with t_4, s''_4 copies of t_2, s''_2, and this tells us that $C'_2 \in s'_2$. So $(C'_1, C'_2) \in s'_1 \times s'_2$ are a pair of names simultaneously produced from C_1 and C_2, and thus $C'_1\ R'\ C'_2$ by the definition of R'.
- By symmetry the only remaining case is when $C'_1 \in s''_1$ and $C'_2 \in s''_2$. Here we may choose any $R'' : s''_1 \rightleftharpoons s''_2$ that contains (C'_1, C'_2).

In all of these cases we obtain $C'_1\ (R \oplus R' \oplus R'')_\nu\ C'_2$ for some $R'' : s''_1 \rightleftharpoons s''_2$, and so $C_1\ (R \oplus R')^{cxt}_\sigma\ C_2$ as required. □

With these results on contextual relations, so close to the defining properties of logical relations, it is not hard to show completeness:

Theorem 4.15 *Suppose that σ is a ground or first-order type of the nu-calculus and Γ is a set of variables of ground type. Then for any span $R : s_1 \rightleftharpoons s_2$ and expressions $M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$,*

$$\Gamma \vdash M_1 \; R^{cxt}_\sigma \; M_2 \quad \Longrightarrow \quad \Gamma \vdash M_1 \; R_\sigma \; M_2.$$

In particular

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

Proof We show first that the result holds for closed expressions, by induction on the structure of the type σ. By Lemma 4.14 we need only consider expressions in canonical form, and Lemma 4.12 gives the result for ground types.

Consider then two lambda abstractions

$$\lambda x{:}\sigma.M_1 \in \mathrm{Can}_{\tau\to\sigma}(s_1) \qquad \text{and} \qquad \lambda x{:}\sigma.M_2 \in \mathrm{Can}_{\tau\to\sigma}(s_2)$$

where $\tau \in \{o, \nu\}$ and $(\lambda x{:}\sigma.M_1) \; R^{cxt}_{\tau\to\sigma} \; (\lambda x{:}\sigma.M_2)$. Suppose that $R' : s_1' \rightleftharpoons s_2'$ is some other span, and $C_1 \in \mathrm{Can}_\tau(s_1 \oplus s_1')$, $C_2 \in \mathrm{Can}_\tau(s_2 \oplus s_2')$ with $C_1 \; (R \oplus R')^{can}_\tau \; C_2$. As τ is a ground type, $(R \oplus R')^{can}_\tau = (R \oplus R')^{syn}_\tau$, and we can apply Lemma 4.13 to obtain

$$M_1[C_1/x] \;\; (R \oplus R')^{cxt}_\sigma \;\; M_2[C_2/x].$$

The induction hypothesis gives

$$M_1[C_1/x] \;\; (R \oplus R')^{exp}_\sigma \;\; M_2[C_2/x]$$

and so

$$(\lambda x{:}\sigma.M_1) \;\; (R \oplus R')^{can}_{\tau\to\sigma} \;\; (\lambda x{:}\sigma.M_2)$$

as desired.

For open expressions, suppose that $\Gamma \vdash M_1 \; R^{cxt}_\sigma \; M_2$, that some $C_{ij} \in \mathrm{Can}_{\tau_j}(s_i \oplus s_i')$ for $i = 1, 2$, $j = 1, \ldots, k$ are suitable instantiations of Γ, and that there is a span $R' : s_1' \rightleftharpoons s_2'$ such that

$$C_{1j} \;\; (R \oplus R')^{can}_{\tau_j} \;\; C_{2j} \qquad j = 1, \ldots, k.$$

All the τ_j are ground types, so $(R \oplus R')^{can}_{\tau_j} = (R \oplus R')^{syn}_{\tau_j}$, and the definition of contextual relations gives

$$M_1[\vec{C_1}/\vec{x}] \;\; (R \oplus R')^{cxt}_\sigma \;\; M_2[\vec{C_2}/\vec{x}].$$

By the result for closed terms:

$$M_1[\vec{C_1}/\vec{x}] \;\; (R \oplus R')^{exp}_\sigma \;\; M_2[\vec{C_2}/\vec{x}]$$

and so $\Gamma \vdash M_1 \; R_\sigma \; M_2$ as required. $\square$

3 Categories with Relations

The logical relations of the previous sections are wholly operational, and work directly with expressions of the nu-calculus. However, there is also a corresponding denotational approach, which incorporates a relational structure into the categorical models of Chapter 3. The idea is that with a correctly chosen category and monad, the standard interpretation of the nu-calculus will automatically make identifications similar to those derived from operational logical relations.

This section describes the extra categorical tools that we need to build such a model, principally the notion of 'categories with relations'. These were introduced by Tennent and O'Hearn to give a semantics for local variables in Algol-like languages; a good outline of this use is found in [82]. The definition below is all that we need to build models for the nu-calculus; following this are two alternative presentations which put the ideas in a wider categorical setting.

Definition 4.16 A *category with relations* is a category with certain additional structure. As well as objects and morphisms, it has a collection of binary *relations* between pairs of objects, represented $R : A \leftrightarrow B$, and *parametric squares* of the form

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & A' \arrow[d, leftrightarrow, "R'"] \\ B \arrow[r, "g"'] & B' \end{tikzcd}$$

where R, R' are relations and f, g are morphisms. Relations, like morphisms, are simply abstract data; they need not stand for set-based relations, though that is obviously the motivating example. Similarly parametric squares need not 'mean' anything particular: like composition of morphisms, they are just part of the specification of the category.

Parametric squares should compose horizontally with identity

$$\begin{tikzcd} A \arrow[r, "id_A"] \arrow[d, leftrightarrow, "R"'] & A \arrow[d, leftrightarrow, "R"] \\ B \arrow[r, "id_A"'] & B \end{tikzcd}$$

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

Categories with relations are a weak form of *double category*, and as such are akin to *2-categories* [37]; parametric squares are more general than 2-cells, but they do not compose vertically.

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

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

$$\begin{tikzcd} FA \arrow[r, "tA"] \arrow[d, leftrightarrow, "FR"'] & GA \arrow[d, leftrightarrow, "GR"] \\ FB \arrow[r, "tB"'] & GB \end{tikzcd}$$

is parametric. In both these cases the parametricity constraints are additional to the usual uniformity conditions of functorality and naturality.

We define $CatR$ to be the 2-category whose objects are small categories with relations, morphisms are parametric functors, and 2-cells are parametric natural transformations.

The use of categories with relations is an instance of a general principle for studying programming languages categorically, as described by Moggi in [68]:

> when studying a complex language the 2-category Cat of small categories, functors and natural transformations may not be adequate; however, one may replace Cat with a different 2-category, whose objects capture better some *fundamental structure* of the language, while *less fundamental structure* can be modelled by *2-categorical concepts*.

In our case $CatR$ is the more sophisticated setting, able to capture a strong notion of uniformity through the additional requirement of parametricity.

We can give another, more abstract description of $CatR$. A *reflexive graph* is a set of vertices and edges between them, where each vertex has a distinguished identity edge to itself. Essentially, this is a category without composition. There is an evident category $ReflGrph$ of small reflexive graphs, with morphisms required to preserve the appropriate structure.

A category with relations is then a category object in $ReflGrph$, and $CatR$ is the internal 2-category of categories, functors and natural transformations. That is, when working over reflexive graphs rather than sets, the ordinary notion of category automatically comes equipped with relations, while functors and natural transformations are suitably parametric.

For example, given a category with relations $\mathcal{C}$ as described above, the object of objects $\mathcal{C}_0$ is the reflexive graph of objects A and relations $R : A \leftrightarrow B$, while the object of morphisms $\mathcal{C}_1$ is the reflexive graph of morphisms $f : A \to A'$ and parametric squares:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & A' \arrow[d, leftrightarrow, "R'"] \\ B \arrow[r, "g"'] & B' \end{tikzcd}.$$

Consider the categories Set of sets and functions, and Cat of small categories and functors. Then a reflexive graph has more structure than a set, but less than a category. In the same way categories with relations (internal to $\mathit{ReflGrph}$) lie between ordinary categories (internal to Set) and double categories (internal to Cat).

Alternatively, categories with relations can be seen as reflexive graphs in Cat. Take the two-object category

$$\mathcal{G}: \quad \begin{tikzcd} v \arrow[r, "\Delta"] & e \arrow[l, bend right, "\delta_0"'] \arrow[l, bend left, "\delta_1"] \end{tikzcd} \qquad \delta_0 \circ \Delta = \delta_1 \circ \Delta = id_v .$$

A reflexive graph is exactly a functor from $\mathcal{G}$ to Set, with image

$$\begin{tikzcd} \text{vertices} \arrow[r, "\text{identity}"] & \text{edges.} \arrow[l, bend right, "\text{domain}"'] \arrow[l, bend left, "\text{codomain}"] \end{tikzcd}$$

Indeed if we extend $\mathit{ReflGrph}$ to be a 2-category, with the appropriate structure-preserving 2-cells, then there is an equality of 2-categories:

$$\mathit{ReflGrph} \;=\; \mathit{Hom}_{2CAT}(\mathcal{G}, Set).$$

Morphisms of graphs are natural transformations between functors into Set, and 2-cells are modifications in the 3-category $2CAT$ of large 2-categories. This is a precise correspondence: the data needed for an item on one side of the equation exactly describes the matching item on the other side.

If we now replace Set by Cat then we obtain a description of categories with relations:

$$\mathit{CatR} \;=\; \mathit{Hom}_{2CAT}(\mathcal{G}, Cat).$$

A functor $\mathcal{C} : \mathcal{G} \to Cat$ is a category with relations, where $\mathcal{C}v$ is the underlying category, and $\mathcal{C}e$ is the category of relations and parametric squares. If $\mathcal{C}$ and $\mathcal{D}$ are two categories with relations, then a parametric functor $F : \mathcal{C} \to \mathcal{D}$ is a natural transformation between the corresponding functors $\mathcal{C}, \mathcal{D} : \mathcal{G} \to Cat$. A parametric natural transformation $t : F \to G$ is a modification between natural transformations F and G into Cat. In all cases the parametricity constraints of CatR match exactly the existing structure of Cat, mediated by the two-object category $\mathcal{G}$.

In practice we shall only use the direct description of Definition 4.16 to handle categories with relations. Nevertheless, the characterisations above do give them decent categorical credentials.

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

In this section we use the machinery of categories with relations to rebuild the model $Set^{\mathcal{I}}$ from Section 5 of Chapter 3 in a parametric setting. This gives a denotational semantics for the nu-calculus that directly validates all the contextual equivalences of Chapter 2, except for (13). In the next section we prove various properties of the model, in particular that it is fully abstract at ground and first-order types.

We take the index category $\mathcal{I}$ of finite sets and injections as before, and extend it to a category with relations. A relation $R : s_1 \leftrightarrow s_2$ on $\mathcal{I}$ consists of a finite set R and a pair of injections $s_1 \leftarrowtail R \rightarrowtail s_2$. This is exactly the notion of a *span* from Definition 4.1. The operation '+' on $\mathcal{I}$ extends to relations: if $R : s_1 \leftrightarrow s_2$ and $R' : s_1' \leftrightarrow s_2'$ then $R + R' : s_1 + s_1' \leftrightarrow s_2 + s_2'$. A square in $\mathcal{I}$ is parametric

$$\begin{tikzcd} s_1 \arrow[r, "f_1"] \arrow[d, leftrightarrow, "R"'] & s_1' \arrow[d, leftrightarrow, "R'"] \\ s_2 \arrow[r, "f_2"'] & s_2' \end{tikzcd} \quad \text{if and only if both squares in} \quad \begin{tikzcd} s_1 \arrow[r] & s_1' \\ R \arrow[u] \arrow[d] \arrow[r] & R' \arrow[u] \arrow[d] \\ s_2 \arrow[r] & s_2' \end{tikzcd} \quad \text{are pullbacks.}$$

This is a stronger condition than just requiring it to commute as a square of relations. In fact, up to isomorphism, all parametric squares in $\mathcal{I}$ are of the form

$$\begin{tikzcd} s_1 \arrow[r, "inl_{s_1,s_1'}"] \arrow[d, leftrightarrow, "R"'] & s_1 + s_1' \arrow[d, leftrightarrow, "R + R'."] \\ s_2 \arrow[r, "inl_{s_1,s_2'}"'] & s_2 + s_2' \end{tikzcd}$$

Some delicacy is required to manage the connection between the structure of $\mathcal{I}$ and constructions made previously in an operational setting: we shall write $R : s_1 \rightleftharpoons s_2$ for a span between name sets, and $R : s_1 \leftrightarrow s_2$ for the corresponding relation in $\mathcal{I}$; similarly with $s \oplus s'$ and $s + s'$.

For the base category we take Set with ordinary binary relations and squares parametric

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & A' \arrow[d, leftrightarrow, "R'"] \\ B \arrow[r, "g"'] & B' \end{tikzcd} \quad \text{if and only if} \quad \forall a \in A, b \in B \,.\, (a, b) \in R \implies (fa, gb) \in R'.$$

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

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

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

$$\begin{tikzcd}
s_1 \arrow[r, "f_1"] \arrow[d, leftrightarrow, "R"'] & s_1' \arrow[d, leftrightarrow, "R'"] \\
s_2 \arrow[r, "f_2"'] & s_2'
\end{tikzcd}$$

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

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

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

So a function from A to B defined at stage s includes data on how it behaves at all later stages. Both naturality and parametricity place bounds on what this behaviour can be.

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

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

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

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

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

where $e_1 \in TAs_1$ and $e_2 \in TAs_2$. It is significant that the definition of TAR makes implicit use of the quotient by '$\sim$': to see if e_1 and e_2 are related, it is necessary to check all possible representatives $[s_1', a_1']$ and $[s_2', a_2']$. The resulting strong monad T is both affine and commutative.

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

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

which satisfies the necessary equations.

In certain cases, the action of the monad T is quite simple. If $A : \mathcal{I} \to Set$ is a constant parametric functor that takes all relations to the identity, then $TA = A$. This applies to the interpretation of any nu-calculus type that does not use ν; so for example $T[\![o]\!]s = [\![o]\!]s = 1+1$ and $T[\![o]\!]R = id_{1+1}$. For computations of type ν itself:

$$T[\![\nu]\!]s \;=\; TNs \;=\; s+1$$

and

$$T[\![\nu]\!]R \;=\; TNR \;=\; R + id_1 \;:\; s_1 + 1 \leftrightarrow s_2 + 1.$$

The interpretation of Chapter 3, Section 4 takes expressions of the nu-calculus to morphisms in $\mathcal{P}$:

$$\begin{array}{rll} [\![M']\!]_{\Gamma,\neq s} : (\neq s) \times [\![\Gamma]\!] \to T[\![\sigma]\!] & & M' \in \mathrm{Exp}_\sigma(s,\Gamma) \\ |C'|_{\Gamma,\neq s} : (\neq s) \times [\![\Gamma]\!] \to [\![\sigma]\!] & & C' \in \mathrm{Can}_\sigma(s,\Gamma) \\ [\![M]\!]_{\neq s} : (\neq s) \to T[\![\sigma]\!] & & M \in \mathrm{Exp}_\sigma(s) \\ |C|_{\neq s} : (\neq s) \to [\![\sigma]\!] & & C \in \mathrm{Can}_\sigma(s). \end{array}$$

As with $Set^{\mathcal{I}}$, the object $(\neq s)$ is isomorphic to $\mathcal{I}(s,-)$, and we obtain elements

$$\begin{array}{ll} [\![M]\!]_{\neq s} \in T[\![\sigma]\!]s & \qquad |C|_{\neq s} \in [\![\sigma]\!]s \\ [\![M']\!]_{\Gamma,\neq s} \in (T[\![\sigma]\!])^{[\![\Gamma]\!]}s & \qquad |C'|_{\Gamma,\neq s} \in [\![\sigma]\!]^{[\![\Gamma]\!]}s \end{array}$$

using a version of the Yoneda Lemma adapted for categories with relations. By Proposition 3.11 this model in $\mathcal{P}$ is adequate, and equality in the category implies contextual equivalence in the nu-calculus.

5 Properties of the Model in $\mathcal{P}$

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

The main result of this section is that the model in $\mathcal{P}$ is fully abstract at first-order types. This arises from a close connection with operational logical relations, and the fact that relations in the category can be used to show contextual relations between expressions of the nu-calculus.

First though, we fix when expressions of the nu-calculus are related by the model in $\mathcal{P}$:

Definition 4.17 Given some span $R : s_1 \rightleftharpoons s_2$, two expressions $M_1 \in \mathrm{Exp}_\sigma(s_1,\Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2,\Gamma)$ are *$\mathcal{P}$-R-related* if

$$([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \;\in\; (T[\![\sigma]\!])^{[\![\Gamma]\!]}R.$$

Usually the choice of R is clear, and we say that the expressions are *$\mathcal{P}$-related*.

We now need to establish how $\mathcal{P}$-relations compare to the other relations described earlier in this chapter. A few technical lemmas lay the foundations:

Lemma 4.18 *Logical relations and $\mathcal{P}$-relations coincide at ground types. For any span $R : s_1 \rightleftharpoons s_2$ and closed expressions $M_1 \in \mathrm{Exp}_\tau(s_1)$, $M_2 \in \mathrm{Exp}_\tau(s_2)$ of type $\tau \in \{o,\nu\}$:*

$$([\![M_1]\!]_{\neq s_1}, [\![M_2]\!]_{\neq s_2}) \in T[\![\tau]\!]R \quad\iff\quad M_1 \; R_\tau \; M_2.$$

Proof Follows directly from the description of $T[\![o]\!] = 1 + 1$ and $T[\![\nu]\!] = N + 1$ given earlier, and the straightforward nature of logical relations at ground types. □

Lemma 4.19 *Syntactically related expressions are related in $\mathcal{P}$:*

$$\begin{array}{rcl} \Gamma \vdash M_1 \; R^{syn}_\sigma \; M_2 & \Longrightarrow & ([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \in T[\![\sigma]\!]^{[\![\Gamma]\!]}R \\ \Gamma \vdash C_1 \; R^{syn}_\sigma \; C_2 & \Longrightarrow & (|C_1|_{\Gamma,\neq s_1}, |C_2|_{\Gamma,\neq s_2}) \in [\![\sigma]\!]^{[\![\Gamma]\!]}R. \end{array}$$

Proof By induction on the structure of the expressions. □

Lemma 4.20 *Syntactically related substitutions preserve $\mathcal{P}$-relations. Suppose*

$$([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \;\in\; T[\![\sigma]\!]^{[\![\Gamma]\!]}R$$

and that Γ can be instantiated by $C_{1j} \; (R \oplus R')^{syn}_{\sigma_j} \; C_{2j}$ where $j = 1, \ldots, n$, then

$$(\,[\![M_1[\vec{C_1}/\vec{x}]]\!]_{\neq s_1 \oplus s'_1}, \, [\![M_2[\vec{C_2}/\vec{x}]]\!]_{\neq s_2 \oplus s'_2}\,) \;\in\; T[\![\sigma]\!](R + R').$$

Proof For each of $i = 1, 2$ we have

$$[\![M_i]\!]_{\Gamma,\neq s_i} \;\in\; T[\![\sigma]\!]^{[\![\Gamma]\!]}s_i \;=\; \mathcal{P}(\mathcal{I}(s_i, -) \times [\![\Gamma]\!], T[\![\sigma]\!])$$

and, looking back to Lemma 3.4 on substitution,

$$[\![M_i[\vec{C_i}/\vec{x}]]\!]_{\neq s_i \oplus s'_i} \;=\; [\![M_i]\!]_{\Gamma,\neq s_i}(s_i + s'_i)\,\langle inl_{s_i,s'_i}, \langle |C_{i1}|_{\neq s_i \oplus s'_i}, \ldots, |C_{in}|_{\neq s_i \oplus s'_i}\rangle\rangle$$

By Lemma 4.19

$$(|C_{1j}|_{\neq s_1 \oplus s'_1}, |C_{2j}|_{\neq s_2 \oplus s'_2}) \;\in\; [\![\sigma_j]\!](R + R') \qquad j = 1, \ldots, n,$$

and thus

$$(\,\langle |C_{11}|_{\neq s_i \oplus s'_i}, \ldots, |C_{1n}|_{\neq s_i \oplus s'_i}\rangle, \, \langle |C_{21}|_{\neq s_i \oplus s'_i}, \ldots, |C_{2n}|_{\neq s_i \oplus s'_i}\rangle\,) \;\in\; [\![\Gamma]\!](R + R').$$

So from the definition of relations at exponentials in $\mathcal{P}$,

$$([\![M_1[\vec{C_1}/\vec{x}]]\!]_{\neq s_1 \oplus s'_1}, [\![M_2[\vec{C_2}/\vec{x}]]\!]_{\neq s_2 \oplus s'_2}) \;\in\; T[\![\sigma]\!](R + R')$$

as required. □

Lemma 4.21 *Syntactically related lambda abstractions, applied to $\mathcal{P}$-related expressions, give $\mathcal{P}$-related results:*

$$\begin{array}{l} (\lambda x{:}\sigma.M_1) \; R^{syn}_{\sigma \to \sigma'} \; (\lambda x{:}\sigma.M_2) \;\;\&\;\; ([\![M'_1]\!]_{\neq s_1}, [\![M'_2]\!]_{\neq s_2}) \in T[\![\sigma]\!]R \\ \qquad\qquad \Longrightarrow \;\; ([\![(\lambda x{:}\sigma.M_1)M'_1]\!]_{\neq s_1}, [\![(\lambda x{:}\sigma.M_2)M'_2]\!]_{\neq s_2}) \in T[\![\sigma']\!]R. \end{array}$$

Proof Choose representatives $[\![M'_1]\!]_{\neq s_1} = [s'_1, a'_1]$ and $[\![M'_2]\!]_{\neq s_2} = [s_2, a'_2]$ so that there is $R' : s'_1 \rightleftharpoons s'_2$ with $(a'_1, a'_2) \in [\![\sigma]\!](R + R')$. Then for $i = 1, 2$,

$$\begin{array}{rcl} [\![(\lambda x{:}\sigma.M_i)M'_i]\!]_{\neq s_i} & = & let\, m \Leftarrow [\![M'_i]\!]_{\neq s_i} \; in\, (|\lambda x{:}\sigma.M_i|_{\neq s_i} m) \\ & = & [s'_i + s''_i, b''_i] \end{array}$$

where

$$[s_i'', b_i''] \;=\; |\lambda x{:}\sigma.M_i|_{\neq s_i}\, s_i'\, \langle inl_{s_i,s_i'}, a_i'\rangle \qquad b_i'' \in [\![\sigma']\!](s_i + s_i' + s_i'').$$

From Lemma 4.19, $(|\lambda x{:}\sigma.M_1|_{\neq s_1}, |\lambda x{:}\sigma.M_2|_{\neq s_2}) \in [\![\sigma \to \sigma']\!]R$, so there must be some $R'' : s_1'' \rightleftharpoons s_2''$ such that $(b_1'', b_2'') \in [\![\sigma']\!](R + R' + R'')$. Then

$$\begin{aligned}
(\,[\![(\lambda x{:}\sigma.M_1)M_1']\!]_{\neq s_1},\, [\![(\lambda x{:}\sigma.M_2)M_2']\!]_{\neq s_2}\,) \;&=\; (\,[s_1' + s_1'', b_1''],\, [s_2' + s_2'', b_2'']\,)\\
&\in\; T[\![\sigma']\!]R
\end{aligned}$$

as required. □

We can now put these together to show an adequacy result, that $\mathcal{P}$-relations imply contextual relations. This enhances the previous result of Proposition 3.11, that equality in a categorical model implies contextual equivalence.

Proposition 4.22 (Relational Adequacy) *Expressions related in $\mathcal{P}$ are contextually related:*

$$\begin{aligned}
([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \in (T[\![\sigma]\!])^{[\![\Gamma]\!]}R \quad&\Longrightarrow\quad \Gamma \vdash M_1\; R_\sigma^{cxt}\; M_2\\
(|C_1|_{\Gamma,\neq s_1}, |C_2|_{\Gamma,\neq s_2}) \in [\![\sigma]\!]^{[\![\Gamma]\!]}R \quad&\Longrightarrow\quad \Gamma \vdash C_1\; R_\sigma^{cxt}\; C_2.
\end{aligned}$$

Proof The result for closed expressions follows from Lemmas 4.21 and 4.18 above, with Lemma 4.20 extending this to open expressions. □

Moving to a different set of relations, there is clearly a close connection between the clauses of Definition 4.2 for operational logical relations, and the construction of the category $\mathcal{P}$. In fact the categorical model came first, and the form of the relations $B^A R$ and TAR motivated the definition of logical relations at $R^{can}_{\sigma\to\sigma'}$ and R^{exp}_σ respectively. We make the link precise by showing when $\mathcal{P}$-relations imply logical relations, and *vice versa*.

Theorem 4.23 *Suppose that σ is a nu-calculus type of ground or first order, that Γ is a set of variables of ground type, and $R : s_1 \rightleftharpoons s_2$ is some span. Then $\mathcal{P}$-related expressions over these are logically related:*

$$\begin{aligned}
([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \in (T[\![\sigma]\!])^{[\![\Gamma]\!]}R \quad&\Longrightarrow\quad \Gamma \vdash M_1\; R_\sigma\; M_2\\
(|C_1|_{\Gamma,\neq s_1}, |C_2|_{\Gamma,\neq s_2}) \in [\![\sigma]\!]^{[\![\Gamma]\!]}R \quad&\Longrightarrow\quad \Gamma \vdash C_1\; R_\sigma\; C_2.
\end{aligned}$$

Proof Combine relational adequacy of $\mathcal{P}$ (Proposition 4.22) with the completeness of logical relations at first-order types (Theorem 4.15). □

Lemma 4.24 (Definability) *Suppose that σ is a ground or first order type of the nu-calculus, and s is some set of names. If $a \in [\![\sigma]\!]s$ and $e \in T[\![\sigma]\!]s$, then there are expressions $C \in \mathrm{Can}_\sigma(s)$ and $M \in \mathrm{Exp}_\sigma(s)$ such that $a = |C|_{\neq s}$ and $e = [\![M]\!]_{\neq s}$.*

Proof Exactly as in Lemma 3.15 for the category $Set^{\mathcal{I}}$. □

Theorem 4.25 *Suppose that σ is a nu-calculus type of ground, first or second order, that Γ is a set of variables of ground or first order, and that $R : s_1 \rightleftharpoons s_2$ is some span. Then logically related expressions over these are $\mathcal{P}$-related:*

$$\begin{aligned} \Gamma \vdash M_1 \ R_\sigma \ M_2 \quad &\Longrightarrow \quad ([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \in (T[\![\sigma]\!])^{[\![\Gamma]\!]}R \\ \Gamma \vdash C_1 \ R_\sigma \ C_2 \quad &\Longrightarrow \quad (|C_1|_{\Gamma,\neq s_1}, |C_2|_{\Gamma,\neq s_2}) \in [\![\sigma]\!]^{[\![\Gamma]\!]}R. \end{aligned}$$

Proof We show first that the result holds for closed expressions, by induction on the structure of the type σ. Lemma 4.18 deals with expressions of ground type, so we consider $(\sigma \to \sigma')$ where σ is a ground or first-order type. Suppose we have two lambda abstractions

$$(\lambda x{:}\sigma.M_1) \ \ R^{can}_{\sigma\to\sigma'} \ \ (\lambda x{:}\sigma.M_2)$$

and wish to show that

$$(f_1, f_2) \ \in \ [\![\sigma \to \sigma']\!]R,$$

where

$$f_i \ = \ |\lambda x{:}\sigma.M_i|_{\neq s_i} \ \in \ [\![\sigma \to \sigma']\!]s_i \qquad i = 1, 2.$$

Take any $(a_1, a_2) \in [\![\sigma]\!](R + R')$, for some $R' : s_1' \rightleftharpoons s_2'$. Lemma 4.24 gives some $C_i \in \mathrm{Can}_\sigma(s_i \oplus s_i')$ with $a_i \in |C_i|_{\neq s_i \oplus s_i'}$ for $i = 1, 2$, and by Theorem 4.23 these are logically related $C_1 \ (R \oplus R')^{can}_\sigma \ C_2$. Now for each of $i = 1, 2$:

$$\begin{aligned} f_i\,(s_i + s_i')\,\langle inl_{s_i,s_i'}, a_i\rangle \quad &= \quad |\lambda x{:}\sigma.M_i|_{\neq s_i}\,(s_i + s_i')\,\langle inl_{s_i,s_i'}, |C_i|_{\neq s_i \oplus s_i'}\rangle \\ &= \quad |(\lambda x{:}\sigma.M_i)C_i|_{\neq s_i \oplus s_i'}, \end{aligned}$$

while $(\lambda x{:}\sigma.M_1)C_1 \ (R \oplus R')^{exp}_{\sigma'} \ (\lambda x{:}\sigma.M_2)C_2$ from the definition of $R^{can}_{\sigma\to\sigma'}$. Applying the induction hypothesis gives

$$(\, f_1(s_1 + s_1')\langle inl_{s_1,s_1'}, a_1\rangle, \ f_2(s_2 + s_2')\langle inl_{s_2,s_2'}, a_2\rangle\,) \ \in \ [\![\sigma']\!](R + R')$$

and so $(f_1, f_2) \in [\![\sigma \to \sigma']\!]R$ as desired.

For general expressions, suppose that $M_1 \ R^{exp}_\sigma \ M_2$, with $s_i \vdash M_i \Downarrow_\sigma (s_i')C_i$ for $i = 1, 2$ and $R' : s_1' \rightleftharpoons s_2'$ such that $C_1 \ (R \oplus R')^{can}_\sigma \ C_2$. Then

$$[\![M_i]\!]_{\neq s_i} \ = \ [s_i', |C_i|_{\neq s_i \oplus s_i'}],$$

and by hypothesis $(|C_1|_{\neq s_1 \oplus s_1'}, |C_2|_{\neq s_2 \oplus s_2'}) \in [\![\sigma]\!](R + R')$, so

$$([\![M_1]\!]_{\neq s_1}, [\![M_2]\!]_{\neq s_2}) \ \in \ T[\![\sigma]\!]R$$

as required.

This completes the proof for closed expressions; we now extend to open expressions. Given $\Gamma \vdash M_1\ R_\sigma\ M_2$, we wish to show that

$$([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \ \in\ T[\![\sigma]\!]^{[\![\Gamma]\!]}R.$$

Take any $(g_1, g_2) \in [\![\Gamma]\!](R + R')$, for some $R' : s'_1 \rightleftharpoons s'_2$; then

$$g_i \ =\ \langle a_{i1}, \dots, a_{in}\rangle \qquad a_{ij} \in [\![\sigma_j]\!]s_i \qquad i = 1,2 \quad j = 1, \dots, n$$

with each $(a_{1j}, a_{2j}) \in [\![\sigma_j]\!](R + R')$, and by Lemma 4.24 there are $C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s'_i)$ such that $a_{ij} = |C_ij|_{\neq s_i \oplus s'_i}$. From Theorem 4.23, these are related

$$C_{1j}\ \ (R \oplus R')^{can}_{\sigma_j}\ \ C_{2j} \qquad j = 1, \dots, n,$$

and so $M_1[\vec{C}_1/\vec{x}]\ (R \oplus R')^{exp}_\sigma\ M_2[\vec{C}_2/\vec{x}]$. Now for each of $i = 1, 2$:

$$\begin{aligned}
&[\![M_i]\!]_{\Gamma,\neq s_i}(s_i + s'_i)\langle inl_{s_i,s'_i}, g_i\rangle \\
&\qquad = \ [\![M_i]\!]_{\Gamma,\neq s}\,(s_i + s'_i)\,\langle inl_{s_i,s'_i}, \langle |C_{11}|_{\neq s_i \oplus s'_i}, \dots, |C_{1n}|_{\neq s_i \oplus s'_i}\rangle\rangle \\
&\qquad = \ [\![M_i[\vec{C}_i/\vec{x}]]\!]_{\neq s_i \oplus s'_i}
\end{aligned}$$

and by the result for closed expressions

$$(\,[\![M_1[\vec{C}_1/\vec{x}]]\!]_{\neq s_1 \oplus s'_1},\ [\![M_2[\vec{C}_2/\vec{x}]]\!]_{\neq s_2 \oplus s'_2}\,)\ \in\ T[\![\sigma]\!]R.$$

From the definition of the relation $T[\![\sigma]\!]^{[\![\Gamma]\!]}R$ then,

$$([\![M_1]\!]_{\Gamma,\neq s_1}, [\![M_2]\!]_{\Gamma,\neq s_2}) \ \in\ T[\![\sigma]\!]^{[\![\Gamma]\!]}R$$

as required. A similar proof gives the result for open expressions in canonical form. □

In Section 2 we showed that operational logical relations were complete for proving contextual equivalence in the nu-calculus, up to types of first order. We can now carry this result over to $\mathcal{P}$-relations:

Corollary 4.26 (Full Abstraction) *The model of the nu-calculus in the category $\mathcal{P}$ is fully abstract for expressions of ground and first-order type.*

Proof Combine Theorem 4.25 with Theorem 4.15 on the completeness of logical relations at first-order types. □

So the interpretation of the nu-calculus in $\mathcal{P}$ proves as many equivalences as operational logical relations, up to second-order function types. This includes the examples (2)–(9), (12) and (17) of Chapter 2. As with operational logical relations, the equivalence (13) between two second-order functions is not validated, for the reasons outlined at the end of Section 1.

At higher types the operational and denotational methods differ, because the category $\mathcal{P}$ contains some elements that are not definable in the nu-calculus. This is similar to the comparison between applicative equivalence and the model in $Set^{\mathcal{I}}$, as discussed in Section 6 of Chapter 3. However in contrast to Theorem 3.16 there, with logical relations neither approach is strictly more powerful: this can be shown at fourth and fifth order types, using examples built from the equivalence (13).

6 Predicated Logical Relations

Section 5 of Chapter 2 presented a number of contextual equivalences and inequivalences, almost all of which have now been validated. The only exception is the equivalence

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

and we now extend the method of operational logical relations to prove this particular example. The technique we use is tailored to fit certain symmetries of equivalence (13), and there is no obvious completeness result. However there are possible generalisations, which may prove more powerful; we shall also look at some rather open connections to other current work on logical relations.

Informally, the two functions in (13) are contextually equivalent because they agree on all arguments that can be defined using them. As pointed out at the end of Section 1, ordinary logical relations fail to show this because they admit too many arguments, including $(\lambda x{:}\nu.(x = n))$ and $(\lambda x{:}\nu.\mathit{false})$ on which the functions differ.

Similarly, the heart of the completeness proof of Section 2 lies in the proof of Lemma 4.14, which shows that spans can capture this 'definability of arguments' at ground types. With arguments of first-order type though, the problem is more difficult (see the inequivalences (14) and (15) on page 25 for example), and spans are not enough.

Definition 4.27 (Augmented Spans) For sets of names s_1 and s_2, an *augmented span* $\widehat{R} : s_1 \rightleftharpoons s_2$ comprises three ordinary spans $R_1 : s_1 \rightleftharpoons s_1$, $R : s_1 \rightleftharpoons s_2$ and $R_2 : s_2 \rightleftharpoons s_2$. If we wish to make these explicit, we write $\widehat{R} : s_1 \rightleftharpoons s_2$ as $(R_1, R, R_2) : s_1 \rightleftharpoons s_2$.

If $\widehat{R}' : s_1' \rightleftharpoons s_2'$ is another augmented span, with s_1' and s_2' disjoint from s_1 and s_2 respectively, then there is a disjoint union of augmented spans:

$$\widehat{R} \oplus \widehat{R}' \ = \ (R_1 \oplus R_1', R \oplus R', R_2 \oplus R_2') \ : \ s_1 \oplus s_1' \rightleftharpoons s_2 \oplus s_2'.$$

There is also the identity augmented span:

$$\widehat{id}_s \ = \ (id_s, id_s, id_s) \ : \ s \rightleftharpoons s.$$

The intuition for enhancing the span R with R_1 and R_2 is that they should capture additional symmetries in how expressions use their own private names. The motivating example is

$$\nu n.\nu n'.\lambda f{:}\nu{\to}o.(fn = fn'),$$

where the rôles of n and n' are entirely interchangeable. We can now define an extended form of logical relation:

Definition 4.28 (Predicated Logical Relations) If $\widehat{R} : s_1 \rightleftharpoons s_2$ is an augmented span then the relations

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

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

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

$$n_1 \ \widehat{R}^{can}_\nu \ n_2 \quad \iff \quad n_1 \ R_1 \ n_1 \ \ \& \ \ n_1 \ R \ n_2 \ \ \& \ \ n_2 \ R_2 \ n_2$$

$$\begin{aligned}
&(\lambda x{:}\sigma.M_1) \ \widehat{R}^{can}_{\sigma\to\sigma'} \ (\lambda x{:}\sigma.M_2) \quad \iff \\
&\qquad (\lambda x{:}\sigma.M_1) \ (R_1)^{can}_{\sigma\to\sigma'} \ (\lambda x{:}\sigma.M_1) \ \ \& \ \ (\lambda x{:}\sigma.M_2) \ (R_2)^{can}_{\sigma\to\sigma'} \ (\lambda x{:}\sigma.M_2) \\
&\qquad \& \ \ \forall \widehat{R}' : s_1' \rightleftharpoons s_2', \ C_1 \in \mathrm{Can}_\sigma(s_1 \oplus s_1'), \ C_2 \in \mathrm{Can}_\sigma(s_2 \oplus s_2') \, . \\
&\qquad\qquad C_1 \ (\widehat{R} \oplus \widehat{R}')^{can}_\sigma \ C_2 \implies M_1[C_1/x] \ (\widehat{R} \oplus \widehat{R}')^{exp}_{\sigma'} \ M_2[C_2/x]
\end{aligned}$$

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

Notice that we use ordinary logical relations R_1 and R_2 in the clause for function types. The relations $\widehat{R}^{can}_\sigma$ and $\widehat{R}^{exp}_\sigma$ coincide on canonical forms, and we may write them as $\widehat{R}_\sigma$ indiscriminately. We can extend the relations to open expressions: if $M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma)$ and $M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$ where $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ then define

$$\begin{aligned}
\Gamma \vdash M_1 \ \widehat{R}_\sigma \ M_2 \quad \iff \quad & \Gamma \vdash M_1 \ (R_1)_\sigma \ M_1 \ \ \& \ \ \Gamma \vdash M_2 \ (R_2)_\sigma \ M_2 \\
& \& \ \ \forall \widehat{R}' : s_1' \rightleftharpoons s_2', \\
& \quad C_{ij} \in \mathrm{Can}_{\sigma_j}(s_i \oplus s_i') \quad i = 1,2 \quad j = 1, \ldots, n \, . \\
& \qquad (\&_{j=1}^n . \, C_{1j} \ (\widehat{R} \oplus \widehat{R}')^{can}_{\sigma_j} \ C_{2j}) \\
& \qquad \implies M_1[\vec{C}_1/\vec{x}] \ (\widehat{R} \oplus \widehat{R}')^{exp}_\sigma \ M_2[\vec{C}_2/\vec{x}].
\end{aligned}$$

Lemma 4.29 *If* $\widehat{R} : s_1 \rightleftharpoons s_2$ *and* $M_1 \in \mathrm{Exp}_\sigma(s_1, \Gamma), M_2 \in \mathrm{Exp}_\sigma(s_2, \Gamma)$ *then*

$$\Gamma \vdash M_1 \ \widehat{R}_\sigma \ M_2 \quad \iff \quad \Gamma \oplus \Gamma' \vdash M_1 \ (\widehat{R} \oplus \widehat{R}')_\sigma \ M_2$$

for any Γ' *and* $\widehat{R}' : s_1' \rightleftharpoons s_2'$.

Proof Exactly as for ordinary logical relations in Lemma 4.3. □

The strength of predicated logical relations lies in the way they combine the ‘logical’ clauses of Definition 4.2, using the structure of the type σ, with additional predicates on either side of the relation. In this particular case, the predicates are the diagonals of the ordinary logical relations $(R_1)_\sigma$ and $(R_2)_\sigma$. So for example the result

$$\Gamma \vdash M_1 \ \widehat{R}_\sigma \ M_2 \quad \implies \quad \Gamma \vdash M_1 \ (R_1)_\sigma \ M_1 \quad \& \quad \Gamma \vdash M_2 \ (R_2)_\sigma \ M_2$$

follows directly from the definition of $\widehat{R}_\sigma$.

The ‘logical’ part of the definition means that we can proceed as in Section 1, and show that predicated logical relations lie between syntactic and contextual ones:

Lemma 4.30 *For any span* $R : s_1 \rightleftharpoons s_2$ *and type* $\tau \in \{o, \nu\}$,

$$(id_{s_1}, R, id_{s_2})_\tau \ = \ R_\tau .$$

Proof Straightforward, given the simple form of R_τ for ground τ. $\square$

Proposition 4.31 *If* $\Gamma \vdash M_1\ R^{syn}_\sigma\ M_2$ *then* $\Gamma \vdash M_1\ (id_{s_1}, R, id_{s_2})_\sigma\ M_2$*. In particular the predicated relation* $(\widehat{id}_s)_\sigma$ *is reflexive:* $\Gamma \vdash M\ (\widehat{id}_s)_\sigma\ M$ *for any* $M \in \mathrm{Exp}_\sigma(s, \Gamma)$.

Proof By induction on the structure of the expressions. $\square$

Proposition 4.32 *If* $\Gamma \vdash M_1\ (id_{s_1}, R, id_{s_2})_\sigma\ M_2$ *then* $\Gamma \vdash M_1\ R^{cxt}_\sigma\ M_2$*. In particular,* $(\widehat{id}_s)$*-related expressions are contextually equivalent:*

$$\Gamma \vdash M_1\ (\widehat{id}_s)_\sigma\ M_2 \quad \Longrightarrow \quad s, \Gamma \vdash M_1 \approx_\sigma M_2 .$$

Proof Combine the definition of contextual and predicated logical relations with Proposition 4.31 and Lemma 4.30. $\square$

So predicated logical relations can be used to prove contextual equivalence. It happens that they validate all the examples from Section 5 of Chapter 2, but we shall concentrate on the most elusive:

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

Consider the augmented span $\widehat{R} : \{n, n'\} \rightleftharpoons \{\}$ where

$$R_1 = \{(n, n'), (n', n)\} \qquad \text{and} \qquad R = R_2 = \{\}.$$

That is, R and R_2 are both the empty span, while $R_1 : \{n, n'\} \rightleftharpoons \{n, n'\}$ is the twist map. Now the only arguments at which the function $(\lambda f{:}\nu{\to}o.(fn = fn'))$ gives the result $false$, are expressions of type $(\nu \to o)$ that distinguish n from n'. But such an expression cannot be R_1-related to itself, and so

$$C_1\ (\widehat{R} \oplus \widehat{R}')^{can}_{\nu\to o}\ C_2 \quad \Longrightarrow \quad s_1 \oplus s_1' \vdash (\lambda f{:}\nu{\to}o.(fn = fn'))C_1 \Downarrow_o (s_1'')true.$$

This is enough to show that

$$(\lambda f{:}\nu{\to}o.(fn = fn'))\ \ \widehat{R}^{can}_{(\nu\to o)\to o}\ \ (\lambda f{:}\nu{\to}o.true)$$

and hence

$$(\nu n.\nu n'.\lambda f{:}\nu{\to}o.(fn = fn'))\ \ (\widehat{id}_\emptyset)^{exp}_{(\nu\to o)\to o}\ \ (\lambda f{:}\nu{\to}o.true)$$

from which the equivalence (13) follows by Proposition 4.32.

We could now give a denotational development of predicated logical relations. This would begin with a three-object category something like

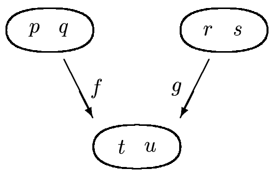
$$\begin{tikzcd} v \arrow[r, "\Delta"] & e \arrow[l, bend right, "\delta_0"'] \arrow[l, bend left, "\delta_1"] & t \arrow[l, "\epsilon_1"'] \arrow[l, bend right, "\epsilon_0"'] \arrow[l, bend left, "\epsilon_2"] \end{tikzcd} \qquad \begin{aligned} \delta_0 \circ \Delta &= \delta_1 \circ \Delta = id_v \\ \delta_0 \circ \epsilon_0 &= \delta_1 \circ \epsilon_0 = \delta_0 \circ \epsilon_1 \\ \delta_0 \circ \epsilon_2 &= \delta_1 \circ \epsilon_2 = \delta_1 \circ \epsilon_1 \end{aligned}$$

and go on to mimic the work of Sections 3, 4 and 5 by defining 'categories with predicated relations' and further extending the $Set^{\mathcal{I}}$ model. However, this is now so far from any standard categorical constructions, and so specific to example (13), that there is little insight to be gained.

Predicated logical relations are a little like the *layered predicates* studied by the Nielsons in [73]. Their Definition 3.3 describes a way to combine two (Kripke-indexed) relations P and Q into a relation $P\&Q$. Again this is a technique for fine-tuning logical relations, while keeping induction on the structure of types.

In defining predicated logical relations, we could make other choices for the predicates. For example, taking the everywhere-true predicate gives ordinary logical relations. Increasing complexity, we could use the diagonals of predicated logical relations themselves, or try different forms of predicate on either side of the relation. Another approach would be to carry out the same ideas for relations of other arities. With no completeness results though, this just adds unwarranted sophistication, and the resulting systems are far too complex for any practical use.

These generalisations lead towards the work of Jung and Tiuryn on logical relations of varying arity [36], which Riecke and O'Hearn have used to give a fully abstract denotational semantics for PCF [80]. In this setting, our predicated logical relations are similar to Kripke logical relations of varying arity over the category



$$fp = fq = t$$
$$gr = gs = u$$

of three objects, each a two-element set. However, it is not yet clear how to make this correspondence exact, and whether or not this leads to a fully abstract model for the nu-calculus.

Chapter 5

A Language with Store

Despite its curious and interesting behaviour, the nu-calculus cannot claim to provide much useful computational power. However, many significant and useful features of programming languages include the generativity that it captures; this chapter shows by example how the techniques developed for the nu-calculus can still be applied. Note that we intend only to sketch an outline of the method, so we do not seek the same level of detail as in previous chapters.

A full programming language like Standard ML contains several features with generative aspects: Core ML has exceptions, references and datatypes, all dynamically created, as are the signatures and structures of the module system. We concentrate on *references*, storage cells created on a run-time heap and removed by a garbage collector. These are an imperative part of ML and various implementations use them as the basis for arrays that can be updated in place.

To study references we use *Reduced ML*, a language that combines higher-order functions with integer references. Sections 1 and 2 describe its syntax, type structure and operational semantics. Whereas the nu-calculus merely resembles a small fragment of ML, Reduced ML is a selected subset of the full language.

Section 3 defines *contextual equivalence* for Reduced ML: this is the relation that holds between two expressions of the language if they are entirely interchangeable. For example, it could show what program manipulations a compiler can safely carry out, or it might demonstrate that an algorithm matches its specification. More immediately, the examples of Section 4 use contextual equivalence to illustrate how references and higher-order functions behave together.

As with the nu-calculus, contextual equivalence in Reduced ML is expressive but difficult to work with. This is where the work of the preceding chapters pays off. All the methods for proving contextual equivalence in the nu-calculus extend smoothly to Reduced ML; the passage from dynamically generated names to dynamically generated store is truly incremental. What is more, because we restrict storage to integers alone the step turns out to be a fairly small one.

The rest of the chapter is then a recapitulation of all that we did with the nu-calculus. Section 5 looks at operational methods for reasoning about contextual equivalence: applicative equivalence and operational logical relations. For a denotational approach, we use the familiar two stage technique:

$$\text{Reduced ML} \longrightarrow \text{computational metalanguage} \longrightarrow \text{category with strong monad.}$$

Each stage builds on the matching construction for the nu-calculus.

Section 6 describes a computational metalanguage suitable for reasoning about store. Conveniently, this is simply the metalanguage of Chapter 3 with additional types, terms and rules. Under this extension, any equality proved in the metalanguage for names also holds in the metalanguage for store. The interpretation of Reduced ML is given in Section 7, which shows that the computational metalanguage can be used to reason about contextual equivalence.

Section 8 sets out the properties that a category must have to model Reduced ML, and how they are used. Section 9 gives examples of this in practice. Remarkably, all the categories used to model the nu-calculus can also model Reduced ML. This emphasises how dynamically created names really do capture the 'difficult' part of ML references; actual value storage is not so hard.

This analysis of Reduced ML does not go into the same detail as we did for the nu-calculus, nor are reasoning methods necessarily pushed to their limits. Also, integer references alone are less versatile than full ML, where cells can hold functions, or references to other cells. Nevertheless, this chapter does illustrate how a thorough understanding of dynamically generated names can make a significant contribution to reasoning about references in Standard ML.

A closely related line of research is that initiated by Reynolds into 'Algol-like' languages. These are command-based imperative languages, strongly typed, with a stack discipline for local variables and a call-by-name semantics for procedure calls. There is a clear separation between commands and expressions: only commands can change the state, and only expressions can return values. An analogue to higher-order functions in ML is that procedures can be declared locally and may have parameters of procedure type. However direct comparison with Standard ML is impossible, as the programming styles appropriate to the two languages are so different: exact translation takes simple Algol-like code to convoluted ML, and *vice versa*. In particular there is no correspondence between the notions of 'first-order' and 'second-order' in each language, as types often go up a couple of orders under translation.

Despite this, local variables and higher-order procedures do raise many of the same problems as ML references, and this chapter contains references to the corresponding work on Algol-like languages. Reynolds' original [107] has been followed up by Oles [87, 88], O'Hearn and Tennent [81, 82, 83, 84, 125, 126, 127], and Lent [46]. Meyer [57] and Sieber [116, 117] also describe models of Algol-like languages, and approaches connected with linear logic have been suggested by Reddy [101, 102] and O'Hearn [77, 78].

1 Syntax of Reduced ML

The types of Reduced ML are built up from various ground types by the formation of function types:

$$\sigma \ ::= \ \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{int\,ref} \mid \sigma \to \sigma.$$

Here unit and bool are types with one and two elements respectively, and int is the type of integers. The type $\mathsf{int\,ref}$ denotes *integer references*, or *locations*: each of these provides storage for a single integer value. The *order* of a type is defined exactly as for the nu-calculus. We again use variants of σ to range over types.

Reduced ML expressions take the forms listed in Figure 5.1. General expressions

$$
\begin{array}{lcllll}
\mathsf{M} & ::= & \mathsf{x} & & & \text{variable} \\
& | & l & & & \text{location} \\
& | & () & & & \text{unit value} \\
& | & i & & & \text{integer constants } i = \ldots, -1, 0, 1, 2, \ldots \\
& | & \mathsf{true} \quad | \quad \mathsf{false} & & & \text{truth values} \\
& | & \mathsf{if\ M\ then\ M\ else\ M} & & & \text{conditional} \\
\\
& | & \mathsf{M} + \mathsf{M} & | \ \mathsf{M} - \mathsf{M} & | \ \mathsf{M} * \mathsf{M} & \text{operations on integers} \\
& | & \mathsf{M} < \mathsf{M} & | \ \mathsf{M} > \mathsf{M} & | \ \mathsf{M} = \mathsf{M} & \text{tests on integers} \\
& | & \mathsf{M} <= \mathsf{M} & | \ \mathsf{M} >= \mathsf{M} & | \ \mathsf{M} <> \mathsf{M} & \\
\\
& | & \mathsf{ref\ M} & & & \text{create a new reference cell} \\
& | & !\mathsf{M} & & & \text{fetch stored value} \\
& | & \mathsf{M} := \mathsf{M} & & & \text{alter stored value} \\
& | & \mathsf{M} = \mathsf{M} & & & \text{compare locations} \\
\\
& | & \mathsf{fn\ x}{:}\sigma \Rightarrow \mathsf{M} & & & \text{function abstraction} \\
& | & \mathsf{M}\mathsf{M} & & & \text{function application}
\end{array}
$$

Figure 5.1: Expressions of Reduced ML

are usually denoted by M, with B, N, R, F suggesting boolean, integer, reference and function expressions respectively. There is an infinite supply of typed variables, generally chosen from $\mathsf{x}, \mathsf{y}, \mathsf{z}$ and variants. Function abstraction $(\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{M})$ binds the variable x of type σ; we identify expressions up to α-conversion of bound variables, and substitution of expressions for free variables is capture avoiding.

Locations, written as variants of l, are always free, taken from some infinite supply. Properly speaking, this is an extension to the syntax: a true program in Reduced ML would have no explicit locations at all. However we also need to consider program fragments, which may have free locations; explicit locations also appear during the evaluation of expressions. Finite sets of locations are represented by variants of u, for *universe*.

A variety of binary operations and tests on integers are assumed; we shall generally take $\mathsf{N} + \mathsf{N}'$ and $\mathsf{N} < \mathsf{N}'$ to stand for all of these.

There are four sorts of expression that act on store. The expression $\mathsf{ref}\,\mathsf{N}$ picks a fresh location, stores the integer value of N there, and returns the location as an $\mathsf{int\,ref}$. The stored value can be retrieved with $!\mathsf{R}$ and changed with $\mathsf{R} := \mathsf{N}'$. Expressions denoting locations can be compared using $\mathsf{R} = \mathsf{R}'$, not to be confused with the equality test for integers.

With no polymorphism or recursion in Reduced ML, we can define let-expressions as syntactic sugar for function application:

$$\begin{aligned}
\mathsf{let\,val\,x} = \mathsf{M\,in\,M'\,end} &= (\mathsf{fn\,x}{:}\sigma{\Rightarrow}\mathsf{M'})\mathsf{M} \\
\mathsf{let\,fun\,f\,x} = \mathsf{M\,in\,M'\,end} &= \mathsf{let\,val\,f} = (\mathsf{fn\,x}{:}\sigma{\Rightarrow}\mathsf{M})\ \mathsf{in\,M'\,end}
\end{aligned}$$

where f is not free in M. This extends to multiple let-expressions:

$$\begin{array}{lcl}
\mathsf{let\,val\,x_1} = \mathsf{M_1} & = & (\mathsf{fn\,x_1}{:}\sigma_1 \Rightarrow \\
\quad \mathsf{val\,x_2} = \mathsf{M_2} & & \quad \mathsf{fn\,x_2}{:}\sigma_2 \Rightarrow \\
\qquad \vdots & & \qquad \vdots \\
\quad \mathsf{val\,x_n} = \mathsf{M_n} & & \qquad \mathsf{fn\,x_n}{:}\sigma_n \Rightarrow \mathsf{M'}) \\
\mathsf{in} & & \mathsf{M_1 M_2 \ldots M_n} \\
\quad \mathsf{M'} & & \\
\mathsf{end} & &
\end{array}$$

and also infix semicolon for sequencing:

$$\mathsf{M}; \mathsf{M'} \quad = \quad \mathsf{let\,val\,x} = \mathsf{M\,in\,M'\,end}$$

where x is chosen not free in $\mathsf{M'}$.

Type judgements of Reduced ML take the form

$$u, \Gamma \vdash \mathsf{M} : \sigma$$

where u is some finite set of locations, and Γ a finite set of typed variables. This says that the expression M, with locations in u and free variables in Γ, is well typed with type σ. Figure 5.2 gives the rules for forming valid type judgements; some of these use the abbreviation $u, \Gamma \vdash \mathsf{M_1}, \ldots, \mathsf{M_n} : \sigma$ to indicate that all of $u, \Gamma \vdash \mathsf{M_1} : \sigma, \ldots, u, \Gamma \vdash \mathsf{M_n} : \sigma$ hold.

This type assignment is suitably well behaved:

$$\frac{}{u, \Gamma \vdash \mathsf{x} : \sigma}\ (\mathsf{x} : \sigma \in \Gamma) \qquad \frac{}{u, \Gamma \vdash () : \mathsf{unit}}$$

$$\frac{}{u, \Gamma \vdash b : \mathsf{bool}}\ (b = \mathsf{true}, \mathsf{false}) \qquad \frac{}{u, \Gamma \vdash i : \mathsf{int}}\ (i \in \mathbb{Z})$$

$$\frac{u, \Gamma \vdash \mathsf{B} : \mathsf{bool} \qquad u, \Gamma \vdash \mathsf{M}, \mathsf{M}' : \sigma}{u, \Gamma \vdash \mathsf{if\ B\ then\ M\ else\ M}' : \sigma}$$

$$\frac{u, \Gamma \vdash \mathsf{N}, \mathsf{N}' : \mathsf{int}}{u, \Gamma \vdash \mathsf{N} + \mathsf{N}' : \mathsf{int}}\ (\text{also } -,\ *)$$

$$\frac{u, \Gamma \vdash \mathsf{N}, \mathsf{N}' : \mathsf{int}}{u, \Gamma \vdash \mathsf{N} < \mathsf{N}' : \mathsf{bool}}\ \begin{array}{l}(\text{also } >,\ =,\ <=,\\ \quad >= \text{ and } <>)\end{array}$$

$$\frac{}{u, \Gamma \vdash l : \mathsf{int\ ref}}\ (l \in u) \qquad \frac{u, \Gamma \vdash \mathsf{R}, \mathsf{R}' : \mathsf{int\ ref}}{u, \Gamma \vdash \mathsf{R} = \mathsf{R}' : \mathsf{bool}}$$

$$\frac{u, \Gamma \vdash \mathsf{N} : \mathsf{int}}{u, \Gamma \vdash \mathsf{ref\ N} : \mathsf{int\ ref}} \qquad \frac{u, \Gamma \vdash \mathsf{R} : \mathsf{int\ ref}}{u, \Gamma \vdash \mathsf{!R} : \mathsf{int}}$$

$$\frac{u, \Gamma \vdash \mathsf{R} : \mathsf{int\ ref} \qquad u, \Gamma \vdash \mathsf{N} : \mathsf{int}}{u, \Gamma \vdash \mathsf{R} := \mathsf{N} : \mathsf{unit}}$$

$$\frac{u, \Gamma \oplus \{\mathsf{x} : \sigma\} \vdash \mathsf{M} : \sigma'}{u, \Gamma \vdash \mathsf{fn\ x}{:}\sigma{\Rightarrow}\mathsf{M} : \sigma \to \sigma'}$$

$$\frac{u, \Gamma \vdash \mathsf{F} : \sigma \to \sigma' \qquad u, \Gamma \vdash \mathsf{M} : \sigma}{u, \Gamma \vdash \mathsf{FM} : \sigma'}$$

Figure 5.2: Rules for assigning types to expressions of Reduced ML

Lemma 5.1 *If $u, \Gamma \vdash \mathsf{M} : \sigma$ then the type σ is unique. Further, if the expression M has locations in u and free variables in Γ, then*

$$u, \Gamma \vdash \mathsf{M} : \sigma \quad \iff \quad u \oplus u', \Gamma \oplus \Gamma' \vdash \mathsf{M} : \sigma$$

for any u' and Γ'.

Proof By induction on the structure of M. □

From now on we consider only well-typed expressions.

An expression is in *canonical form* if it is either a variable, a location, the unit value $()$, one of the boolean constants $\mathtt{true}$ or $\mathtt{false}$, an integer constant or a function abstraction. These are the *values* of Reduced ML at each type. We define the sets

$$\begin{aligned}
\mathsf{Exp}_\sigma(u, \Gamma) &= \{\, \mathsf{M} \mid u, \Gamma \vdash \mathsf{M} : \sigma \,\} \\
\mathsf{Can}_\sigma(u, \Gamma) &= \{\, \mathsf{C} \mid \mathsf{C} \in \mathsf{Exp}_\sigma(u, \Gamma),\ \mathsf{C} \text{ canonical} \,\} \\[6pt]
\mathsf{Exp}_\sigma(u) &= \mathsf{Exp}_\sigma(u, \emptyset) \\
\mathsf{Can}_\sigma(u) &= \mathsf{Can}_\sigma(u, \emptyset)
\end{aligned}$$

of expressions and canonical expressions at any type σ and for any finite sets u, Γ of locations and typed variables.

2 Operational Semantics

With the nu-calculus, evaluation of expressions involves a state s, which is the set of names created so far. Reduced ML expressions also manipulate a state, which consists of locations and the values stored in them. Over any universe u, the possible states are:

$$\mathsf{State}(u) \;=\; \{s \mid s : u \to \mathbb{Z}\}.$$

If $s \in \mathsf{State}(u)$, then $s\{l \mapsto i\} \in \mathsf{State}(u)$ is the same state, but with the value i stored at location $l \in u$. Similarly $s \oplus \{l' \mapsto i'\} \in \mathsf{State}(u \oplus \{l'\})$ extends s with an extra location $l' \notin u$, holding the value i'.

We can combine states defined over disjoint sets of locations:

$$s \in \mathsf{State}(u) \quad \& \quad s' \in \mathsf{State}(u') \quad \Longrightarrow \quad s \oplus s' \in \mathsf{State}(u \oplus u')$$

by taking the disjoint union of their graphs as functions.

Evaluation judgements of Reduced ML have the form

$$\langle u, s \rangle\, \mathsf{M} \Downarrow_\sigma \langle u', s' \rangle\, \mathsf{C}$$

where $u \subseteq u'$ are finite sets of locations, $s \in \mathsf{State}(u)$, $s' \in \mathsf{State}(u')$, $\mathsf{M} \in \mathsf{Exp}_\sigma(u)$ and $\mathsf{C} \in \mathsf{Can}_\sigma(u')$. This means that in state s, over universe u, the expression M evaluates to canonical form C, in state s', over the extended universe u'. This evaluation relation is inductively defined by the rules in Figure 5.3. As most of these do not explicitly involve the state, we adopt a *state convention* from the definition of Standard ML. This says that a rule presented as

$$\frac{\mathsf{M}_1 \Downarrow_{\sigma_1} \mathsf{C}_1 \quad \mathsf{M}_2 \Downarrow_{\sigma_2} \mathsf{C}_2 \quad \dots \quad \mathsf{M}_\mathsf{n} \Downarrow_{\sigma_\mathsf{n}} \mathsf{C}_\mathsf{n}}{\mathsf{M} \Downarrow_\sigma \mathsf{C}}$$

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

(COND1) $$\frac{\mathsf{B} \Downarrow_{\mathsf{bool}} \mathsf{true} \qquad \mathsf{M} \Downarrow_\sigma \mathsf{C}}{\mathsf{if\ B\ then\ M\ else\ M'} \Downarrow_\sigma \mathsf{C}}$$

(COND2) $$\frac{\mathsf{B} \Downarrow_{\mathsf{bool}} \mathsf{false} \qquad \mathsf{M'} \Downarrow_\sigma \mathsf{C}}{\mathsf{if\ B\ then\ M\ else\ M'} \Downarrow_\sigma \mathsf{C}}$$

(INT+) $$\frac{\mathsf{N} \Downarrow_{\mathsf{int}} i \qquad \mathsf{N'} \Downarrow_{\mathsf{int}} i'}{\mathsf{N} + \mathsf{N'} \Downarrow_{\mathsf{int}} i''} \quad (i'' = i + i')$$

(INT<) $$\frac{\mathsf{N} \Downarrow_{\mathsf{int}} i \qquad \mathsf{N'} \Downarrow_{\mathsf{int}} i'}{\mathsf{N} < \mathsf{N'} \Downarrow_{\mathsf{bool}} \mathsf{true}} \quad (i < i')$$

(INT$\not<$) $$\frac{\mathsf{N} \Downarrow_{\mathsf{int}} i \qquad \mathsf{N'} \Downarrow_{\mathsf{int}} i'}{\mathsf{N} < \mathsf{N'} \Downarrow_{\mathsf{bool}} \mathsf{false}} \quad (i \geq i')$$

(EQ1) $$\frac{\mathsf{R} \Downarrow_{\mathsf{int\,ref}} l \qquad \mathsf{R'} \Downarrow_{\mathsf{int\,ref}} l}{\mathsf{R} = \mathsf{R'} \Downarrow_{\mathsf{bool}} \mathsf{true}}$$

(EQ2) $$\frac{\mathsf{R} \Downarrow_{\mathsf{int\,ref}} l \qquad \mathsf{R'} \Downarrow_{\mathsf{int\,ref}} l'}{\mathsf{R} = \mathsf{R'} \Downarrow_{\mathsf{bool}} \mathsf{false}} \quad l, l' \text{ distinct}$$

(CREATE) $$\frac{\langle u, s\rangle\ \mathsf{N} \Downarrow_{\mathsf{int}} \langle u', s'\rangle\ i}{\langle u, s\rangle\ \mathsf{ref\ N} \Downarrow_{\mathsf{int\,ref}} \langle u' \oplus \{l\}, s' \oplus \{l \mapsto i\}\rangle\ l} \quad (l \notin u')$$

(FETCH) $$\frac{\langle u, s\rangle\ \mathsf{R} \Downarrow_{\mathsf{int\,ref}} \langle u', s'\rangle\ l}{\langle u, s\rangle\ !\mathsf{R} \Downarrow_{\mathsf{int}} \langle u', s'\rangle\ i} \quad (s'(l) = i)$$

(ALTER) $$\frac{\langle u, s\rangle\ \mathsf{R} \Downarrow_{\mathsf{int\,ref}} \langle u', s'\rangle\ l \qquad \langle u', s'\rangle\ \mathsf{N} \Downarrow_{\mathsf{int}} \langle u'', s''\rangle\ i}{\langle u, s\rangle\ \mathsf{R} := \mathsf{N} \Downarrow_{\mathsf{unit}} \langle u'', s''\{l \mapsto i\}\rangle\ ()}$$

(APP) $$\frac{\mathsf{F} \Downarrow_{\sigma \to \sigma'} \mathsf{fn\,x}{:}\sigma{\Rightarrow}\mathsf{M'} \qquad \mathsf{M} \Downarrow_\sigma \mathsf{C} \qquad \mathsf{M'[C/x]} \Downarrow_{\sigma'} \mathsf{C'}}{\mathsf{FM} \Downarrow_{\sigma'} \mathsf{C'}}$$

Figure 5.3: Rules for evaluating expressions of Reduced ML

is actually an abbreviation for

$$\frac{\begin{array}{c}\langle u,s\rangle\,\mathsf{M_1} \Downarrow_{\sigma_1} \langle u_1,s_1\rangle\,\mathsf{C_1} \quad \langle u_1,s_1\rangle\,\mathsf{M_2} \Downarrow_{\sigma_2} \langle u_2,s_2\rangle\,\mathsf{C_2} \\ \ldots \quad \langle u_{n-1},s_{n-1}\rangle\,\mathsf{M_n} \Downarrow_{\sigma_n} \langle u_n,s_n\rangle\,\mathsf{C_n}\end{array}}{\langle u,s\rangle\,\mathsf{M} \Downarrow_{\sigma} \langle u_n,s_n\rangle\,\mathsf{C}},$$

where the state is simply handed from one subexpression to the next. Note that this means the ordering of the hypotheses in such rules is significant. Regarding integer arithmetic, the rules (INT+), (INT<) and (INT$\not<$) also stand for all the other operations and tests available.

The evaluation relation for Reduced ML is taken from Standard ML, with left-to-right evaluation and call-by-value function application. However the presentation of the rules, in particular (APP), does differ slightly from the Definition [62]. There, function application is carried out by evaluating a function body in an environment; this helps with ML's sophisticated pattern matching (and also happens to correspond more closely to most implementation methods). Reduced ML, on the other hand, directly substitutes arguments for variables. As the only possible function abstraction is the simple form ($\mathsf{fn\,x{:}\sigma{\Rightarrow}M}$), the two approaches have identical effect.

A similar comment applies to the rules (COND1) and (COND2) for the conditional: these are built into Reduced ML, but in Standard ML they are syntactic sugar for a certain application.

The correspondence between these two styles of presentation is investigated by Ritter and Pitts in [113]: they use *applicative bisimulation* to show that these alternatives are equivalent over a much larger subset of Standard ML, including full pattern matching.

Evaluation in Reduced ML ignores any unreachable store:

Lemma 5.2 *For any* $\mathsf{M} \in \mathsf{Exp}_\sigma(u)$ *and* $s \in \mathsf{State}(u)$*,*

$$\langle u,s\rangle\,\mathsf{M} \Downarrow_\sigma \langle u',s'\rangle\,\mathsf{C} \iff \langle u\oplus u'', s\oplus s''\rangle\,\mathsf{M} \Downarrow_\sigma \langle u'\oplus u'', s'\oplus s''\rangle\,\mathsf{C}$$

whenever u'' *is disjoint from* u'*, and* $s'' \in \mathsf{State}(u'')$.

Proof By induction on the structure of the derivation of the evaluation judgement. □

Evaluation always terminates, and is deterministic up to choice of fresh locations. The proof is a simple extension of that for the nu-calculus on pages 16–17. Define the two predicates

$$P_\sigma(u) \subseteq \mathsf{Can}_\sigma(u) \qquad \text{and} \qquad \overline{P}_\sigma(u) \subseteq \mathsf{Exp}_\sigma(u)$$

according to

$$\begin{array}{rcl} P_{\mathsf{unit}}(u) &=& \{()\} \qquad\qquad P_{\mathsf{int}}(u) \;=\; \mathbb{Z} \\ P_{\mathsf{bool}}(u) &=& \{\mathsf{true},\mathsf{false}\} \qquad P_{\mathsf{int\,ref}}(u) \;=\; u \\ P_{\sigma\to\sigma'}(u) &=& \{\, \mathsf{fn\,x{:}\sigma{\Rightarrow}M} \mid \forall u' \supseteq u,\, \mathsf{C} \in P_\sigma(u')\,.\, \mathsf{M[C/x]} \in \overline{P}_{\sigma'}(u')\,\} \end{array}$$

and

$M \in \overline{P}_\sigma(u) \iff$ For all $s \in \mathsf{State}(u)$ there are $u' \supseteq u$, $\mathsf{C} \in P_\sigma(u')$ and $s' \in \mathsf{State}(u')$ such that $\langle u,s\rangle\,\mathsf{M} \Downarrow_\sigma \langle u',s'\rangle\,\mathsf{C}$, and these are unique up to renaming the locations $(u' \setminus u)$ and α-conversion of C.

These form a unary logical relation over the expressions of Reduced ML. As before, the idea is to show that P and $\overline{P}$ are both total, and then use the fact that $\overline{P}$ implies termination.

Lemma 5.3 *If* $u, \Gamma \vdash \mathsf{M} : \sigma$*, where* $\Gamma = \{\mathsf{x_1} : \sigma_1, \ldots, \mathsf{x_n} : \sigma_\mathsf{n}\}$*, and* $\mathsf{C}_i \in P_{\sigma_i}(u')$ *for* $i = 1, \ldots, n$ *for some* $u' \supseteq u$*, then* $\mathsf{M}[\vec{\mathsf{C}}/\vec{\mathsf{x}}] \in \overline{P}_\sigma(u')$.

Proof By structural induction on the derivation of $u, \Gamma \vdash \mathsf{M} : \sigma$. The details are almost exactly the same as for Lemma 2.3. For the case of (!R), it is significant that the values stored are integers, for these are all, by definition, in $P_{\mathsf{int}}(u)$. □

Theorem 5.4 (Termination) *If* $\mathsf{M} \in \mathsf{Exp}_\sigma(u)$ *and* $s \in \mathsf{State}(u)$*, then there are* $u' \supseteq u$*,* $\mathsf{C} \in \mathsf{Can}_\sigma(u')$ *and* $s' \in \mathsf{State}(u')$*, such that* $\langle u, s\rangle\, \mathsf{M} \Downarrow_\sigma \langle u', s'\rangle\, \mathsf{C}$*. Moreover, these are unique up to choice of the locations* $(u' \setminus u)$ *and* α*-conversion of* C.

Proof By Lemma 5.3, $\mathsf{M} \in \overline{P}_\sigma(u)$ and the result follows from the definition of $\overline{P}_\sigma(u)$. □

As noted, for this result it is essential that only values of ground type, such as int, can be kept in reference cells. If functions can be stored, then it is possible to encode recursion and so write non-terminating expressions, such as:

```
let val r = ref(fn x:unit⇒())
    fun f x = (!r)x
in
    r := f; f()
end.
```

3 Contextual Equivalence

As with the nu-calculus, it is of particular interest to determine when one Reduced ML expression can be used to replace another. Section 4 of Chapter 2 develops this into the notion of contextual equivalence for the nu-calculus, and we now adapt this for Reduced ML. The general idea is that two, possibly open, expressions are equivalent if they cannot be distinguished by placing them in a complete program. We make this into a formal definition, and in the next section give a number of examples.

Define a *program* in Reduced ML to be a closed boolean expression. All that can be observed of a program is whether, in some initial state, it evaluates to true or false. The creation of new locations, and changes in the values stored, are not directly observable; though obviously the outcome of the program itself may depend on those parts of the store to which it has access.

A *program context* $\mathsf{P}\langle\!\langle - \rangle\!\rangle$ is a program with zero or more occurrences of a hole $\langle\!\langle - \rangle\!\rangle$ with a certain arity $\sigma_1, \ldots, \sigma_n \to \sigma$. The hole may be filled $\mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M} \rangle\!\rangle$ by some expression M, with free variables in $(\vec{\mathsf{x}})$ of types matching the hole's arity. The details are exactly as on pages 22–23 for the nu-calculus; in particular, we recall that the purpose of holes with arity is to formalise the possible capture of free variables.

Definition 5.5 (Contextual Equivalence) Suppose that $\mathsf{M}_1, \mathsf{M}_2 \in \mathsf{Exp}_\sigma(u, \Gamma)$ are two expressions of Reduced ML. They are *contextually equivalent*

$$u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2$$

if for all suitably typed program contexts $\mathsf{P}\langle\!\langle - \rangle\!\rangle$ defined over u, states $s \in \mathsf{State}(u)$ and boolean values $b \in \{\mathsf{true}, \mathsf{false}\}$,

$$\begin{array}{l}(\,\exists u_1 \supseteq u,\ s_1 \in \mathsf{State}(u_1)\ .\ \langle u, s\rangle\, \mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M}_1 \rangle\!\rangle \Downarrow_{\mathsf{bool}} \langle u_1, s_1\rangle\, b\,)\\ \iff \\ (\,\exists u_2 \supseteq u,\ s_2 \in \mathsf{State}(u_2)\ .\ \langle u, s\rangle\, \mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M}_2 \rangle\!\rangle \Downarrow_{\mathsf{bool}} \langle u_2, s_2\rangle\, b\,).\end{array}$$

That is, $\mathsf{P}\langle\!\langle - \rangle\!\rangle$ always evaluates to the same boolean value, whether the hole is filled by M_1 or M_2. When both u and Γ are empty, we write simply $\mathsf{M}_1 \approx_\sigma \mathsf{M}_2$.

Section 6 of Chapter 2 uses a detailed analysis of the evaluation process to show that, in order to establish equivalence, it is only necessary to use certain kinds of program context. Specifically, those that are just a function abstraction applied to a hole $\nu s'.((\lambda x{:}\sigma.B)\langle\!\langle - \rangle\!\rangle(\vec{C}))$; and if the expressions to be compared are closed, the fresh names s' and the instantiation $(\vec{C})$ can be omitted.

There seems no obvious reason why the same analysis could not be made for Reduced ML:

Conjecture 5.6 (Context Lemma) *Two expressions are contextually equivalent* $u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2$ *if and only if for all* $u' \supseteq u$, $s' \in \mathsf{State}(u')$, *test functions* $(\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B}) \in \mathsf{Can}_{\sigma\to\mathsf{bool}}(u')$, *instantiations* $[\vec{\mathsf{C}}/\vec{\mathsf{x}}]$ *defined over* u', *and each* $b \in \{\mathsf{true}, \mathsf{false}\}$, *it is the case that:*

$$\begin{array}{l}(\,\exists u_1 \supseteq u',\ s_1 \in \mathsf{State}(u_1)\ .\ \langle u', s'\rangle\, (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B})\mathsf{M}_1[\vec{\mathsf{C}}/\vec{\mathsf{x}}] \Downarrow_{\mathsf{bool}} \langle u_1, s_1\rangle\, b\,)\\ \iff \\ (\,\exists u_2 \supseteq u',\ s_2 \in \mathsf{State}(u_2)\ .\ \langle u', s'\rangle\, (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B})\mathsf{M}_2[\vec{\mathsf{C}}/\vec{\mathsf{x}}] \Downarrow_{\mathsf{bool}} \langle u_2, s_2\rangle\, b\,).\end{array}$$

Further, two closed expressions are contextually equivalent $u \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2$ *if and only if for all states* $s \in \mathsf{State}(u)$, *test functions* $(\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B}) \in \mathsf{Can}_{\sigma\to\mathsf{bool}}(u)$ *and each* $b \in \{\mathsf{true}, \mathsf{false}\}$, *it is the case that:*

$$\begin{array}{l}(\,\exists u_1 \supseteq u,\ s_1 \in \mathsf{State}(u_1)\ .\ \langle u, s\rangle\, (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B})\mathsf{M}_1 \Downarrow_{\mathsf{bool}} \langle u_1, s_1\rangle\, b\,)\\ \iff \\ (\,\exists u_2 \supseteq u,\ s_2 \in \mathsf{State}(u_2)\ .\ \langle u, s\rangle\, (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{B})\mathsf{M}_2 \Downarrow_{\mathsf{bool}} \langle u_2, s_2\rangle\, b\,).\end{array}$$

A proof of this requires a suitable reduction semantics for Reduced ML, a notion of reduction context, and counterparts to Lemmas 2.10, 2.11 and 2.12, breaking down the possible structure of expressions and contexts. All of these are concerned with evaluation strategy, rather than the presence of state, so they should work much as for the nu-calculus. Indeed, the (ciu) theorem described by Talcott and Mason is essentially this context lemma, but for an untyped language based on destructive LISP [30, §2.3.2].

4 Examples

We now consider a series of example contextual equivalences. These are important not only because they indicate safe program transformations, but also because they give a good idea of how private, public and shared store can be used in Reduced ML. A couple of inequivalences also show how, as with the nu-calculus, higher-order functions add significant power, but must be handled with care.

As Reduced ML builds on the nu-calculus, many of the contextual equivalences listed on pages 23–26 have analogues in the presence of store. There are also other equivalences, which rely on properties of storage and retrieval of values; often though, these too are really about the visibility of locations, private and public.

Meyer and Sieber, to illustrate their model of store in Algol-like languages, give in [57] a list of example equivalences that have become something of a benchmark in this area; these are reproduced in Appendix A as MS1–MS7. As explained in the introduction, direct comparison with Reduced ML is impossible because the programming styles appropriate to the two languages are so different. Nevertheless, there are some similarities, particularly when higher-order procedures are introduced, and most of the points raised by Meyer and Sieber appear somewhere in these examples.

It is basic from the definition that contextual equivalence is a congruence:

1. $$u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2 \quad \Longrightarrow \quad u \oplus u', \Gamma \oplus \Gamma' \vdash \mathsf{M}'[\mathsf{M}_1/\mathsf{x}] \approx_\sigma \mathsf{M}'[\mathsf{M}_2/\mathsf{x}]$$

where $\mathsf{M}' \in \mathsf{Exp}_{\sigma'}(u \oplus u', \Gamma \oplus \Gamma' \oplus [\mathsf{x} : \sigma])$.

As with names in the nu-calculus, unused locations are ignored:

2. $$u, \Gamma \vdash \quad \mathsf{let\,val\,r = ref}\, i \,\mathsf{in\,M\,end} \;\approx_\sigma\; \mathsf{M} \qquad i \in \mathbb{Z}, \quad \mathsf{r} \notin fv(\mathsf{M})$$

and it does not matter in what order fresh cells are allocated:

3. $$u, \Gamma \vdash \quad \begin{array}{l}\mathsf{let\,val\,r\ = ref}\, i \\ \quad \mathsf{val\,r' = ref}\, i' \\ \mathsf{in} \\ \quad \mathsf{M} \\ \mathsf{end}\end{array} \approx_\sigma \begin{array}{l}\mathsf{let\,val\,r' = ref}\, i' \\ \quad \mathsf{val\,r\ = ref}\, i \\ \mathsf{in} \\ \quad \mathsf{M} \\ \mathsf{end}\end{array} \qquad i, i' \in \mathbb{Z}.$$

These are similar to the Algol-like examples MS1 and MS3.

The intermediate states of a computation are not visible:

4. $$\mathsf{r : int\,ref} \vdash \quad (\mathsf{r := !r + 1; r := !r - 1}) \;\approx_{\mathsf{unit}}\; ().$$

In addition, there is no way to detect whether or not a particular computation is carried out using store:

5. $$\begin{array}{l}\mathsf{fn\,(x : int)} \Rightarrow \\ \quad \mathsf{let\,val\,r = ref\,x\,in\,(!r + !r)\,end}\end{array} \approx_{\mathsf{int \to int}} \mathsf{fn\,(x : int)} \Rightarrow (\mathsf{x + x}).$$

Equivalences (6)–(8) of the nu-calculus concern the rearrangement of expressions around reduction contexts; given a suitable corresponding notion for Reduced ML, it seems likely that these equivalences would still hold. Even without reduction contexts, we can formulate β_v-equivalence: if $\mathsf{M} \in \mathsf{Exp}_{\sigma'}(u, \Gamma \oplus [\mathsf{x} : \sigma])$ and $\mathsf{C} \in \mathsf{Can}_\sigma(u, \Gamma)$ then

6. $$u, \Gamma \vdash \quad (\mathsf{fn\,x{:}\sigma{\Rightarrow}M})\mathsf{C} \;\approx_{\sigma'}\; \mathsf{M}[\mathsf{C}/\mathsf{x}].$$

Example (12) from the nu-calculus has a direct analogue, that a dynamically generated location is private:

7.
$$\begin{array}{l}\mathsf{let\,val\,r} = \mathsf{ref}\,0\\ \mathsf{in}\\ \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{int\,ref}) \Rightarrow (\mathsf{x} = \mathsf{r})\\ \mathsf{end}\end{array} \quad \approx_{\mathsf{int\,ref} \to \mathsf{bool}} \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{int\,ref}) \Rightarrow \mathsf{false}.$$

The expression on the left evaluates to a function that compares its argument to the location r. However this function is used, r remains private, and the result of the test is always false. With functions that may have side-effects, there seems no obvious counterpart to the second-order example (13).

The value stored in a private location cannot be altered by external code:

8.
$$\begin{array}{l}\mathsf{let\,val\,r} = \mathsf{ref}\,i\\ \mathsf{in}\\ \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{unit}) \Rightarrow\, !\mathsf{r}\\ \mathsf{end}\end{array} \quad \approx_{\mathsf{unit} \to \mathsf{int}} \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{unit}) \Rightarrow i.$$

Once the value $i \in \mathbb{Z}$ has been stored in the cell r, if the location is not revealed, and the function itself leaves the contents alone, then it will not change. This is the same principle as in the Algol-like examples MS2 and MS4.

It is possible to release partial access to a storage cell:

9.
$$\begin{array}{l}\mathsf{fn}\,(\mathsf{f} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \Rightarrow\\ \quad \mathsf{let\,val\,r} = \mathsf{ref}\,1\\ \quad\quad \mathsf{fun\,inc\,x} = (\mathsf{r} := !\mathsf{r} + 1)\\ \quad \mathsf{in}\\ \quad\quad \mathsf{f(inc)}; !\mathsf{r} > 0\\ \quad \mathsf{end}\end{array} \quad \approx_{((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{bool}} \quad \begin{array}{l}\mathsf{fn}\,(\mathsf{f} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \Rightarrow\\ \quad \mathsf{let\,fun\,skip\,x} = ()\\ \quad \mathsf{in}\\ \quad\quad \mathsf{f(skip)}; \mathsf{true}\\ \quad \mathsf{end}.\end{array}$$

On the left hand side, the function f is given the facility to increase the value stored in r, but not to decrease it. Consequently, this value remains positive, and the test $(!\mathsf{r} > 0)$ can only return true. Algol-like example MS5 captures a similar idea of partial access; MS6 does too, combined with the privacy of example (7) above.

If a storage cell is private, then it does not matter exactly how it is used:

10.
$$\begin{array}{l}\mathsf{let\,val\,r} = \mathsf{ref}\,0\\ \mathsf{in}\\ \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{r} := !\mathsf{r} + \mathsf{x}; !\mathsf{r})\\ \mathsf{end}\end{array} \quad \approx_{\mathsf{int} \to \mathsf{int}} \quad \begin{array}{l}\mathsf{let\,val\,r} = \mathsf{ref}\,0\\ \mathsf{in}\\ \quad \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{r} := !\mathsf{r} - \mathsf{x}; -!\mathsf{r})\\ \mathsf{end}.\end{array}$$

Both of these expressions give a function that maintains a running total of the integers it is applied to. The second one, perversely, represents this total internally by its negative; externally, the two appear identical. O'Hearn and Tennent give an Algol-like version of this in the introduction to [82]; this 'counter' object is their basic running example.

A practical application of privacy is the silent attachment of profiling code:

$$
\begin{array}{lcl}
\mathsf{profile} & = & \mathsf{fn}\,(\mathsf{f} : \sigma \to \sigma') \Rightarrow \\
 & & \quad \mathsf{let\,val\,r} = \mathsf{ref}\,0 \\
 & & \qquad \mathsf{fun\,f'\,x} = (\mathsf{r} := !\mathsf{r} + 1; \mathsf{f\,x}) \\
 & & \quad \mathsf{in} \\
 & & \qquad \mathsf{f'} \\
 & & \quad \mathsf{end}.
\end{array}
$$

This expression takes a function f, of type $(\sigma \to \sigma')$, and returns an *instrumented* version $\mathsf{f'}$ that increments a counter in r each time it is called. Within the language, $\mathsf{f'}$ appears exactly the same as the original f, and so

11. $$\mathsf{f} : \sigma \to \sigma' \vdash \quad \mathsf{profile\,f} \;\approx_{\sigma \to \sigma'}\; \mathsf{f}.$$

The difference is that an instrumented version of a function might be useful to a profiler or debugger, quite external to the language itself, that could track down r, and examine the counter within. Contextual equivalence guarantees that the attachment of this counter cannot affect the outcome of the program. Algol-like example MS7 embodies a restricted instance of this idea, attaching a private counter to a skip command.

Notice that, thanks to higher-order functions, we can use a single piece of code that will instrument any function f, and show once and for all that its effect is not visible within the language. This is undoubtedly preferable to attaching code to the functions themselves, and checking that this is safe in each case.

An extension of this idea is the production of *memoized* functions, that keep a record of previous calls so as to avoid recomputation where possible. To simplify things, we only consider modifying functions of type $(\mathsf{int} \to \mathsf{int})$ to record their most recent argument and result. Again, a higher-order function is the most general approach:

$$
\begin{array}{lcl}
\mathsf{memo} & = & \mathsf{fn}\,(\mathsf{f} : \mathsf{int} \to \mathsf{int}) \Rightarrow \\
 & & \quad \mathsf{let\,val\,q} = \mathsf{ref}\,0 \\
 & & \qquad \mathsf{val\,a} = \mathsf{ref}\,(\mathsf{f}\,0) \\
 & & \quad \mathsf{in} \\
 & & \qquad \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow \\
 & & \qquad\quad \mathsf{if\,x} = !\mathsf{q}\,\mathsf{then}\,()\,\mathsf{else}\,(\mathsf{q} := \mathsf{x}; \mathsf{a} := \mathsf{f\,x}); \\
 & & \qquad\quad !\mathsf{a} \\
 & & \quad \mathsf{end}.
\end{array}
$$

However, this is rather too general for an imperative language:

12. $$\mathsf{memo} \;\not\approx_{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}\; \mathsf{fn}\,(\mathsf{f} : \mathsf{int} \to \mathsf{int}) \Rightarrow \mathsf{f}.$$

For example, consider memo applied to the accumulator of example (10): there it is vital that the code keeping the running total really is executed each time the function is called.

In some particular cases all is well, and it is safe to memoize functions whose results are repeatable.

13. $$\mathsf{memo}\,(\mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{x} + \mathsf{x})) \;\approx_{\mathsf{int}\to\mathsf{int}}\; \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{x} + \mathsf{x}).$$

This does not mean that a function to be memoized cannot use store. For example, both the doubling functions of example (5) can be safely memoized; as they are contextually equivalent, memo affects them equally.

A final, rather complicated, example shows the perhaps surprising ways in which supposedly private store can interact with higher-order functions. Consider the following function:

$$\begin{array}{rcl} \mathsf{F}_1 & = & \mathsf{let\,val\,r} = \mathsf{ref}\,0 \\ & & \quad \mathsf{val\,a} = \mathsf{ref}\,0 \\ & & \mathsf{in} \\ & & \quad \mathsf{fn}\,(\mathsf{f} : \mathsf{int} \to \mathsf{int}) \Rightarrow \\ & & \quad\quad (\mathsf{r} := !\mathsf{r} + 1; \mathsf{a} := \mathsf{f}(!\mathsf{r}); \mathsf{r} := !\mathsf{r} - 1; !\mathsf{a}) \\ & & \mathsf{end.} \end{array}$$

In a roundabout way, F_1 evaluates to a function that takes an argument f, of type $(\mathsf{int} \to \mathsf{int})$, and seems to apply it to the value 1. To do this, it keeps the value 0 in a private cell r, and increments it to 1 temporarily for the function application. As with example (8), this 0 will be preserved in r between calls to F_1.

Taking then the rather simpler function

$$\mathsf{F}_2 \quad = \quad \mathsf{fn}\,(\mathsf{f} : \mathsf{int} \to \mathsf{int}) \Rightarrow \mathsf{f}(1),$$

it may come as a surprise that

14. $$\mathsf{F}_1 \;\not\approx_{(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}\; \mathsf{F}_2.$$

They are distinguished by the test function

$$\begin{array}{rcl} \mathsf{G} & = & \mathsf{fn}\,(\mathsf{F} : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}) \Rightarrow \\ & & \quad \mathsf{F}(\mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow \mathsf{F}(\mathsf{fn}\,(\mathsf{y} : \mathsf{int}) \Rightarrow \mathsf{y})) \end{array}$$

as GF_1 evaluates to 2 and GF_2 evaluates to 1. The details are as follows:

- For GF_1, executing the outer F_1 increments r and binds 1 to x; the inner F_1 then increments r again and binds 2 to y. The contents of r is reset to 0 on the way out, but the final result is 2, as bound to y.

- For GF_2, the outer and inner occurrences of F_2 bind 1 to x and y respectively; the final result is the 1 from y.

Sceptics may care to try this example in a real implementation of Standard ML. As with the nu-calculus example (14) on page 25, the heart of G lies in the fact that when building a (function) argument to be passed to F_i, we can make use of F_i itself.

Together these examples show that a language such as Reduced ML can provide private, local storage, persisting if desired between function invocations. Expressions can give surrounding code partial access to this; or the use of local store need not be externally visible at all. The combination with higher-order functions can be useful, as in the profiling and memoizing functions, but needs to be treated with care, as shown by the inequivalences (12) and (14) above.

5 Proof Methods

All of the techniques developed for the nu-calculus are also suitable for proving contextual equivalences of Reduced ML. In later sections we look at denotational methods, through a computational metalanguage; for now, we outline some operational methods. The most powerful of these, operational logical relations based on store, is sufficient to prove all the equivalences of the previous section.

Section 7 of Chapter 2 describes applicative equivalence for the nu-calculus, and shows that it implies contextual equivalence. This can be adapted to Reduced ML in two stages. The first is a relation, *strong applicative equivalence*, written '$\cong$', that requires expressions to allocate exactly the same private locations. Closely following Definition 2.13, we have a mutual induction between the relation on canonical forms:

$$
\begin{array}{llll}
u \vdash () \cong^{\mathsf{can}}_{\mathsf{unit}} () & & u \vdash b_1 \cong^{\mathsf{can}}_{\mathsf{bool}} b_2 \iff b_1 = b_2 \\
u \vdash i_1 \cong^{\mathsf{can}}_{\mathsf{int}} i_2 \iff i_1 = i_2 & & u \vdash l_1 \cong^{\mathsf{can}}_{\mathsf{int\,ref}} l_2 \iff l_1 = l_2
\end{array}
$$

$$
\begin{aligned}
u \vdash (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{M}_1) \cong^{\mathsf{can}}_{\sigma\to\sigma'} (\mathsf{fn}\,\mathsf{x}{:}\sigma{\Rightarrow}\mathsf{M}_2) \iff{}& \forall u' \supseteq u,\ \mathsf{C} \in \mathsf{Can}_\sigma(u')\ .\\
& u' \vdash \mathsf{M}_1[\mathsf{C}/\mathsf{x}] \cong^{\mathsf{exp}}_{\sigma'} \mathsf{M}_2[\mathsf{C}/\mathsf{x}]
\end{aligned}
$$

and on general expressions:

$$
\begin{aligned}
u \vdash \mathsf{M}_1 \cong^{\mathsf{exp}}_\sigma \mathsf{M}_2 \iff{}& \forall u' \supseteq u,\ s' \in \mathsf{State}(u')\ .\\
& \exists u'' \supseteq u',\ s'' \in \mathsf{State}(u''),\ \mathsf{C}_1, \mathsf{C}_2 \in \mathsf{Can}_\sigma(u'')\ .\\
& \quad u'' \vdash \mathsf{C}_1 \cong^{\mathsf{can}}_\sigma \mathsf{C}_2\\
& \quad \&\ \langle u', s'\rangle\, \mathsf{M}_1 \Downarrow_\sigma \langle u'', s''\rangle\, \mathsf{C}_1\\
& \quad \&\ \langle u', s'\rangle\, \mathsf{M}_2 \Downarrow_\sigma \langle u'', s''\rangle\, \mathsf{C}_2.
\end{aligned}
$$

Notice that M_1 and M_2 must evaluate to the same final state $\langle u'', s''\rangle$. This then extends to a relation on open expressions:

$$
\begin{aligned}
u, \Gamma \vdash \mathsf{M}_1 \cong_\sigma \mathsf{M}_2 \iff{}& \forall u' \supseteq u,\ \mathsf{C_i} \in \mathsf{Can}_{\sigma_i}(u') \quad i = 1, \ldots, n\ .\\
& u' \vdash \mathsf{M}_1[\vec{\mathsf{C}}/\vec{\mathsf{x}}] \cong^{\mathsf{exp}}_\sigma \mathsf{M}_2[\vec{\mathsf{C}}/\vec{\mathsf{x}}].
\end{aligned}
$$

It remains to show that '$\cong$' is a congruence, and so implies contextual equivalence. Because Reduced ML has no recursively defined types, and only stores integers, we can proceed exactly as for the nu-calculus, by defining a relation of *strong logical equivalence* and proving that the two are the same. Alternatively, we could adapt the more general Howe's method, as used in [31, 25].

Strong applicative equivalence is more or less the same as Ritter and Pitts' applicative bisimulation, from [113]. It is able to prove only equivalences (3), (4) and (6) from Section 4, because it demands a very close correspondence on the use of private locations.

A more liberal relation, *applicative equivalence*, written '$\sim$', relaxes this constraint. The definition is the same, except for expressions:

$$
\begin{aligned}
u \vdash \mathsf{M}_1 \sim^{\mathsf{exp}}_\sigma \mathsf{M}_2 \iff{}& \forall u' \supseteq u,\ s' \in \mathsf{State}(u')\ .\\
& \exists u_1, u_2 \supseteq u',\ s_1 \in \mathsf{State}(u_1),\ s_2 \in \mathsf{State}(u_2),\\
& \qquad \mathsf{C}_1 \in \mathsf{Can}_\sigma(u_1),\ \mathsf{C}_2 \in \mathsf{Can}_\sigma(u_2)\ .\\
& \quad u_1 \cup u_2 \vdash \mathsf{C}_1 \sim^{\mathsf{can}}_\sigma \mathsf{C}_2\\
& \quad \&\ \langle u_1, s_1\rangle \sim \langle u_2, s_2\rangle\\
& \quad \&\ \langle u', s'\rangle\, \mathsf{M}_1 \Downarrow_\sigma \langle u_1, s_1\rangle\, \mathsf{C}_1\\
& \quad \&\ \langle u', s'\rangle\, \mathsf{M}_2 \Downarrow_\sigma \langle u_2, s_2\rangle\, \mathsf{C}_2
\end{aligned}
$$

and a relation between states:

$$\begin{aligned}\langle u_1, s_1\rangle \sim \langle u_2, s_2\rangle \quad &\Longleftrightarrow \quad \forall l \in u_1 \cap u_2 \,.\, u_1 \cap u_2 \vdash s_1(l) \sim^{\mathsf{can}}_{\mathsf{int}} s_2(l) \\ &\Longleftrightarrow \quad \forall l \in u_1 \cap u_2 \,.\, s_1(l) = s_2(l).\end{aligned}$$

It is now only necessary that expressions should agree up to the production of garbage. Mason's notion of *strong isomorphism* for an untyped lambda-calculus with store, as described in [54, §3.2] and [55, §2.4], is roughly this relation restricted to ground types.

Applicative equivalence verifies examples (2)–(6), and any others where dynamically generated store is used only for temporary variables. Consider the case of equivalence (5):

5. $$\begin{array}{l}\mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow \\ \quad \mathsf{let\,val\,r} = \mathsf{ref\,x\,in}\,(!\mathsf{r} + !\mathsf{r})\,\mathsf{end}\end{array} \approx_{\mathsf{int}\to\mathsf{int}} \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{x} + \mathsf{x}).$$

We have that for any universe u and integer i,

$$u \vdash \quad \mathsf{let\,val\,r} = \mathsf{ref}\,i\,\mathsf{in}\,(!\mathsf{r} + !\mathsf{r})\,\mathsf{end} \ \sim^{\mathsf{exp}}_{\mathsf{int}} \ (i + i).$$

This is because for every $u' \supseteq u$ and $s' \in \mathsf{State}(u')$ there are evaluation judgements

$$\langle u', s'\rangle \ \mathsf{let\,val\,r} = \mathsf{ref}\,i\,\mathsf{in}\,(!\mathsf{r} + !\mathsf{r})\,\mathsf{end} \ \Downarrow_{\mathsf{int}} \ \langle u' \oplus \{l\}, s' \oplus \{l \mapsto i\}\rangle \ 2i$$

and

$$\langle u', s'\rangle \ (i + i) \ \Downarrow_{\mathsf{int}} \ \langle u', s'\rangle \ 2i$$

with

$$\langle u' \oplus \{l\}, s' \oplus \{l \mapsto i\}\rangle \ \sim \ \langle u', s'\rangle$$

as the two states agree at their common locations. From this and the definition of applicative equivalence at function types, we deduce that

$$\vdash \quad \begin{array}{l}\mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow \\ \quad \mathsf{let\,val\,r} = \mathsf{ref\,x\,in}\,(!\mathsf{r} + !\mathsf{r})\,\mathsf{end}\end{array} \sim_{\mathsf{int}\to\mathsf{int}} \mathsf{fn}\,(\mathsf{x} : \mathsf{int}) \Rightarrow (\mathsf{x} + \mathsf{x})$$

and the contextual equivalence follows. The same relation, extended to a rather larger subset of Standard ML, is studied by Pitts, Stark and de Paiva in [93].

As with the nu-calculus, for stronger proof methods we can turn to some form of logical relation. Extending the work in Section 1 of Chapter 4 gives operational logical relations based on spans $R : u_1 \rightleftharpoons u_2$ across sets of locations. These subsume applicative equivalence, and also prove equivalences like example (7), that a dynamically generated location is private. This covers the profile function of example (11) too: incrementing a private counter does not affect a function's behaviour.

Such relations successfully capture the fact that an expression can only read certain locations. However they say little about limiting write access to store: for example, logical relations based on locations would allow an expression that reset all store to 0 indiscriminately. This affects all the remaining equivalences of Section 4, each of which relies on some preservation of private store.

The solution is to move the relations from locations to states: replace $R : u_1 \rightleftharpoons u_2$ by $R \subseteq \mathsf{State}(u_1) \times \mathsf{State}(u_2)$ and build a logical relation R_σ over this. At public locations, the relation R has to be the identity, but on private locations there is great flexibility to represent the different ways expressions may use store.

Rather than go into construction details, we consider here which relations between states capture particular aspects of privacy. For example, suppose that the private cell r is at location l. Then in example (8), we want a relation R that keeps the fixed value i at this location:

$$R \;=\; \{\langle s \oplus \{l \mapsto i\}, s\rangle \mid s \in \mathsf{State}(u),\ \text{fixed } i\}.$$

In example (9), we claim that the value at l is always positive, represented by the relation:

$$R \;=\; \{\langle s \oplus \{l \mapsto i\}, s\rangle \mid s \in \mathsf{State}(u),\ i > 0\}.$$

We can then use the fact that inc preserves this property. Such an *invariant* also arises in example (10) where two functions keep the same value in a private cell, but with a change of sign:

$$R \;=\; \{\langle s \oplus \{l \mapsto i\}, s \oplus \{l \mapsto -i\}\rangle \mid s \in \mathsf{State}(u),\ i \in \mathbb{Z}\}.$$

When the memo function can safely be applied, as in example (13), the invariant is that two locations l_q and l_a hold a valid pair of argument and result for the memoized function f:

$$R \;=\; \{\langle s \oplus \{l_q \mapsto i, l_a \mapsto j\}, s\rangle \mid s \in \mathsf{State}(u),\ j = fi,\ \text{fixed } f : \mathbb{Z} \to \mathbb{Z}\}.$$

Logical relations based on store genuinely extend the simpler relations based on locations: given a span $R_l : u_1 \rightleftharpoons u_2$ we can define a relation R_s on states

$$\langle s_1, s_2\rangle \in R_s \quad\iff\quad \forall \langle l_1, l_2\rangle \in R_l \,.\, s_1(l_1) = s_2(l_2)$$

with similar effect. Thus all the equivalences of Section 4 can be handled by a system of operational logical relations based on state.

O'Hearn and Tennent, in [82], and Sieber, in [117], also use relations between states, in denotational methods for reasoning about Algol-like languages. While the languages are not directly comparable, the range of store manipulations that this method can tackle seems much the same in both settings.

6 A Computational Metalanguage for Store

We now move on to denotational methods for reasoning about Reduced ML, beginning with an interpretation in a computational metalanguage. To deal with store, we enlarge the metalanguage of Chapter 3 with additional types, terms and rules.

The extra types are Unit and Int, with terms $()$ and $\{\ldots, -1, 0, 1, 2, \ldots\}$, and operations like $\mathit{plus}(i, i')$ and $\mathit{less}(i, i')$. These behave in an entirely straightforward way, and make no use of computation types.

More interesting are two extra term-forming operations $\mathit{get}(-)$ and $\mathit{set}(-,-)$, with typing rules

$$\frac{\Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{get}(n) : T\mathit{Int}} \qquad \text{and} \qquad \frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma \vdash i : \mathit{Int}}{\Gamma \vdash \mathit{set}(n, i) : T\mathit{Unit}}.$$

The intuition is that $\mathit{set}(n, i)$ associates the integer i to the name n, while $\mathit{get}(n)$ retrieves it; both are computations. There is no explicit store; rather, we add rules to the metalanguage that simulate the visible behaviour of a store. This allows us to use

$$\frac{}{\Gamma \vdash 3 : \mathit{Int}} \qquad \frac{}{\Gamma \vdash () : \mathit{Unit}}$$

$$\frac{\Gamma \vdash i, i' : \mathit{Int}}{\Gamma \vdash \mathit{plus}(i, i') : \mathit{Bool}} \qquad \frac{\Gamma \vdash i, i' : \mathit{Int}}{\Gamma \vdash \mathit{less}(i, i') : \mathit{Bool}}$$

$$\frac{\Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{get}(n) : T\mathit{Int}} \qquad \frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma \vdash i : \mathit{Int}}{\Gamma \vdash \mathit{set}(n, i) : T\,\mathit{Unit}}.$$

Figure 5.4: Some additional typing rules for the metalanguage

static, equational reasoning in the metalanguage, while the computation types correctly handle the dynamic aspects.

Figure 5.4 gives the additional type rules for the metalanguage. To avoid an explosion of uninteresting rules about integers, in all these rules plus, less and occasionally equal, should be taken as exemplarary. Similarly, where numeric constants are involved, we give only a single case, as illustration.

With the introduction of side-effects, a useful abbreviation is

$$(e; e') \;=\; \mathit{let}\, x \Leftarrow e\; \mathit{in}\; e' \qquad x \notin \mathit{fv}(e')$$

for sequential evaluation, with iterated form:

$$(e_1; e_2; \ldots; e_n) \;=\; (e_1; (e_2; \ldots; e_n)).$$

This is associative, and has various other properties:

$$\begin{aligned} \Gamma \vdash\; (e; (e'; e'')) &\;=\; ((e; e'); e'') \\ \Gamma \vdash\; (e; \mathit{let}\, x' \Leftarrow e'\; \mathit{in}\; e'') &\;=\; \mathit{let}\, x' \Leftarrow (e; e')\; \mathit{in}\; e'' \\ \Gamma \vdash\; (\mathit{let}\, x \Leftarrow e\; \mathit{in}\; e'; e'') &\;=\; \mathit{let}\, x \Leftarrow e\; \mathit{in}\; (e'; e''). \end{aligned}$$

All these are easily derived from the standard computation rules.

A number of rules for store extend the equational reasoning of the metalanguage. Figure 5.5 gives congruence rules for all the extra term-forming operations, and standard properties of types Unit and Int; Figure 5.6 gives some rules for reasoning about $get(-)$ and $set(-,-)$. All these are in addition to the rules presented in Figures 3.2 and 3.3, on pages 40 and 41.

The rules of Figure 5.6 are chosen for entirely pragmatic reasons: they appear to be the least necessary to prove Propositions 5.9 and 5.10, that the interpretation of Reduced ML in the metalanguage is both correct and adequate. We can explain them as follows:

- The (MONO$^+$) rule subsumes the standard (MONO) rule of Figure 3.3, and states that side-effects cannot divert simple value computations. In the metalanguage of Chapter 3, it can be derived from the strong (DROP$^+$) rule of page 42; however, that is not valid in the presence of store. The full generality of the (MONO$^+$) rule is not strictly necessary: for example, if exceptions were added then restrictions on the computations e and e' would be in order.

Congruence:

$$\frac{}{\Gamma \vdash () = ()} \qquad \frac{}{\Gamma \vdash 3 = 3}$$

$$\frac{\Gamma \vdash i_1, i_1', i_2, i_2' : \mathit{Int}}{\Gamma; i_1 = i_1', i_2 = i_2' \vdash \mathit{plus}(i_1, i_2) = \mathit{plus}(i_1', i_2')}$$

$$\frac{\Gamma \vdash i_1, i_1', i_2, i_2' : \mathit{Int}}{\Gamma; i_1 = i_1', i_2 = i_2' \vdash \mathit{less}(i_1, i_2) = \mathit{less}(i_1', i_2')}$$

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

$$\frac{\Gamma \vdash n, n' : \mathit{Name} \qquad \Gamma \vdash i, i' : \mathit{Int}}{\Gamma; n = n', i = i' \vdash \mathit{set}(n, i) = \mathit{set}(n', i')}$$

Unit:

$$\frac{\Gamma \vdash u : \mathit{Unit}}{\Gamma \vdash u = ()}$$

Integers:

$$\frac{}{\Gamma \vdash \mathit{plus}(1, 1) = 2} \qquad \frac{}{\Gamma \vdash \mathit{less}(-1, 1) = \mathit{tt}}$$

$$\frac{\Gamma \vdash i : \mathit{Int}}{\Gamma \vdash \mathit{equal}(i, i) = \mathit{tt}} \qquad \frac{\Gamma \vdash i : \mathit{Int}}{\Gamma \vdash \mathit{less}(i, i) = \mathit{ff}}$$

$$\frac{\Gamma \vdash i, i' : \mathit{Int}}{\Gamma; \mathit{equal}(i, i') = \mathit{tt} \vdash i = i'}$$

Figure 5.5: Some extra rules for equational reasoning in the metalanguage (I)

Computations:

(MONO$^+$)
$$\frac{\Gamma \vdash a, a' : A \qquad \Gamma \vdash e, e' : TB}{\Gamma; \mathit{let}\, x \Leftarrow e\ \mathit{in}\, [a] = \mathit{let}\, x \Leftarrow e'\ \mathit{in}\, [a'] \vdash a = a'}\ (x \notin \mathit{fv}(a, a'))$$

Storage:

(READ)
$$\frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma \vdash i : \mathit{Int}}{\Gamma \vdash (\mathit{set}(n, i); \mathit{get}(n)) = (\mathit{set}(n, i); [i])}$$

(WRITE)
$$\frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma \vdash i, i' : \mathit{Int}}{\Gamma \vdash (\mathit{set}(n, i); \mathit{set}(n, i')) = \mathit{set}(n, i')}$$

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

(SWAP$''$)
$$\frac{\Gamma \vdash n, n' : \mathit{Name} \qquad \Gamma \vdash i, i' : \mathit{Int}}{\Gamma; \mathit{eq}(n, n') = \mathit{ff} \vdash (\mathit{set}(n, i); \mathit{set}(n', i')) = (\mathit{set}(n', i'); \mathit{set}(n, i))}$$

Figure 5.6: Some extra rules for equational reasoning in the metalanguage (II)

- The (READ) and (WRITE) rules assert that $get(n)$ does indeed fetch the value associated to the name n, and that $set(n, i)$ overwrites any previous association on n.

- The (SWAP′) and (SWAP″) rules are restricted forms of the (SWAP$^+$) rule of page 42, and state that the ordering of certain computations does not matter. Specifically, $set(-,-)$ and new may be interchanged, as can two $set(-,-)$ commands, if they refer to different names. These augment the existing (SWAP) rule, that fresh names may be generated in any order.

While these are all plausible statements about storage, and indeed there are categorical models that confirm them, it is unfortunate that there is no more systematic method behind this choice of rules.

This enhanced metalanguage provides a general setting for reasoning about computation with dynamically generated storage. In the next section we see how this applies to Reduced ML in particular.

7 Interpretation of Reduced ML

The translation from nu-calculus to computational metalanguage, described in Section 2 of Chapter 3, extends smoothly to Reduced ML. This gives an interpretation that is correct with respect to the operational semantics, and adequate for reasoning about contextual equivalence.

Types are interpreted without complication:

$$\begin{aligned} [\![\mathsf{unit}]\!] &= \mathit{Unit} & [\![\mathsf{int}]\!] &= \mathit{Int} \\ [\![\mathsf{bool}]\!] &= \mathit{Bool} & [\![\mathsf{int\ ref}]\!] &= \mathit{Name} \\ & & [\![\sigma \to \sigma']\!] &= [\![\sigma]\!] \to T[\![\sigma']\!]. \end{aligned}$$

Figure 5.7 describes the translation of Reduced ML expressions, with $|-|$ for canonical forms, $[\![-]\!]$ for expressions, and $[\![-]\!]$ for location and variable contexts. This is a straightforward extension of Figure 3.4 on page 44, which does the same for the nu-calculus.

In deference to the use of locations in Reduced ML, variants on l are used in the metalanguage as variables of type Name. In particular, an explicit location is interpreted by itself, regarded as such a variable. We made a similar conflation earlier, with a name n of the nu-calculus being interpreted by the variable $n : \mathit{Name}$ in the metalanguage.

The interpretation respects types and substitution of values:

Lemma 5.7 *For any well typed Reduced ML expression* M, *or expression* C *in canonical form:*

$$\begin{aligned} u, \Gamma \vdash \mathsf{M} : \sigma &\iff [\![u, \Gamma]\!] \vdash [\![\mathsf{M}]\!] : T[\![\sigma]\!] \\ u, \Gamma \vdash \mathsf{C} : \sigma &\iff [\![u, \Gamma]\!] \vdash |\mathsf{C}| : [\![\sigma]\!]. \end{aligned}$$

Proof By induction over the structure of the type judgement in Reduced ML, using uniqueness of types in the metalanguage. □

Canonical forms:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
|\mathsf{x}| &=& x & |()| &=& () \\
|\mathsf{true}| &=& tt & |3| &=& 3 \\
|\mathsf{false}| &=& f\!f & |l| &=& l \\
|\mathsf{fn}\ \mathsf{x}{:}\sigma{\Rightarrow}\mathsf{M}| &=& \lambda x{:}[\![\sigma]\!].[\![\mathsf{M}]\!] & & &
\end{array}$$

Expressions:

$$\begin{array}{rcl}
[\![\mathsf{C}]\!] &=& [|\mathsf{C}|] \\
[\![\mathsf{if}\ \mathsf{B}\ \mathsf{then}\ \mathsf{M}\ \mathsf{else}\ \mathsf{M}']\!] &=& let\, b{\Leftarrow}[\![\mathsf{B}]\!]\ in\ cond(b, [\![\mathsf{M}]\!], [\![\mathsf{M}']\!]) \\
[\![\mathsf{N} + \mathsf{N}']\!] &=& let\, i{\Leftarrow}[\![\mathsf{N}]\!]\ in\ let\, i'{\Leftarrow}[\![\mathsf{N}']\!]\ in\ [plus(i, i')] \\
[\![\mathsf{N} < \mathsf{N}']\!] &=& let\, i{\Leftarrow}[\![\mathsf{N}]\!]\ in\ let\, i'{\Leftarrow}[\![\mathsf{N}']\!]\ in\ [less(i, i')] \\
[\![\mathsf{ref}\ \mathsf{N}]\!] &=& let\, i{\Leftarrow}[\![\mathsf{N}]\!]\ in\ let\, l{\Leftarrow}new\ in\ (set(l, i); [l]) \\
[\![!\mathsf{R}]\!] &=& let\, l{\Leftarrow}[\![\mathsf{R}]\!]\ in\ get(l) \\
[\![\mathsf{R} := \mathsf{N}]\!] &=& let\, l{\Leftarrow}[\![\mathsf{R}]\!]\ in\ let\, i{\Leftarrow}[\![\mathsf{N}]\!]\ in\ set(l, i) \\
[\![\mathsf{R} = \mathsf{R}']\!] &=& let\, l{\Leftarrow}[\![\mathsf{R}]\!]\ in\ let\, l'{\Leftarrow}[\![\mathsf{R}']\!]\ in\ [eq(l, l')] \\
[\![\mathsf{F}\mathsf{M}]\!] &=& let\, f{\Leftarrow}[\![\mathsf{F}]\!]\ in\ let\, m{\Leftarrow}[\![\mathsf{M}]\!]\ in\ fm
\end{array}$$

Contexts:

$$\begin{array}{rcl}
[\![u, \Gamma]\!] &=& l_1, \ldots, l_k : Name,\ x_1 : [\![\sigma_1]\!], \ldots, x_n : [\![\sigma_n]\!] \\
\text{where } u &=& \{l_1, \ldots, l_k\} \\
\Gamma &=& \{\mathsf{x}_1 : \sigma_1, \ldots, \mathsf{x}_\mathsf{n} : \sigma_\mathsf{n}\}
\end{array}$$

Figure 5.7: Interpretation of Reduced ML in the computational metalanguage

Lemma 5.8 *If* $\mathsf{M} \in \mathsf{Exp}_\sigma(u, \Gamma \oplus \{x : \sigma\})$ *and* $\mathsf{C} \in \mathsf{Can}_\sigma(u, \Gamma)$ *then*

$$[\![u, \Gamma]\!] \vdash [\![\mathsf{M}[\mathsf{C}/\mathsf{x}]]\!] = [\![\mathsf{M}]\!][|\mathsf{C}|/\mathsf{x}]$$

in the metalanguage.

Proof By induction on the structure of M, using only the fact that equality in the metalanguage is a congruence. □

Certain definitions are required before we can formulate a notion of correctness for this translation. Recall the abbreviations from page 46:

$$\begin{aligned} (\neq u) &= \{eq(l_i, l_j) = \mathit{ff} \mid 1 \leq i < j \leq k\} \\ let\,(u' \setminus u) \Leftarrow \overrightarrow{new}\; in\, e &= let\, l'_1 \Leftarrow new\; in \ldots let\, l'_{k'} \Leftarrow new\; in\, e \end{aligned}$$

where $u = \{l_1, \ldots, l_k\}$ and $(u' \setminus u) = \{l'_1, \ldots, l'_{k'}\}$. Here $(\neq u)$ asserts that all the locations in u are distinct, and $let\,(u' \setminus u) \Leftarrow \overrightarrow{new}\; in\, e$ extends the available locations from u to u' before computing e. To these we now add:

$$\begin{aligned} assign\,\langle u, s\rangle\; in\, e &= (set(l_1, i_1); \ldots; set(l_k, i_k); e) \\ alter\,\langle u, s\rangle\; to\,\langle u', s'\rangle\; in\, e &= let\,(u' \setminus u) \Leftarrow \overrightarrow{new}\; in\; assign\,\langle u', s'\rangle\; in\, e \end{aligned}$$

where $s = \{l_1 \mapsto i_1, \ldots, l_k \mapsto i_k\} \in \mathsf{State}(u)$ and $s' \in \mathsf{State}(u')$ for some $u' \supseteq u$. Thanks to the various (SWAP) rules in the metalanguage, the ordering of these sets is not important, up to provable equality.

We can now state:

Proposition 5.9 (Correctness of Translation) *If* $\langle u, s\rangle\; \mathsf{M} \Downarrow_\sigma \langle u', s'\rangle\; \mathsf{C}$ *is an evaluation judgement of Reduced ML then*

$$[\![u]\!]; (\neq u) \vdash assign\,\langle u, s\rangle\; in\, [\![\mathsf{M}]\!] = alter\,\langle u, s\rangle\; to\,\langle u', s'\rangle\; in\, [\![\mathsf{C}]\!]$$

can be proved in the metalanguage.

Proof By structural induction on the derivation of the evaluation judgement. Each of the rules in Figure 5.3 translates to a derivation provable in the metalanguage; this is all much the same as Proposition 3.6 for the nu-calculus, and the details are omitted.

Most of the work is done by the rules

$$\frac{[\![u']\!]; (\neq u') \vdash assign\,\langle u', s'\rangle\; in\, e = alter\,\langle u', s'\rangle\; to\,\langle u'', s''\rangle\; in\, e'}{[\![u]\!]; (\neq u) \vdash alter\,\langle u, s\rangle\; to\,\langle u', s'\rangle\; in\, e = alter\,\langle u, s\rangle\; to\,\langle u'', s''\rangle\; in\, e'}\;(u \subseteq u')$$

and

$$\frac{[\![u]\!] \vdash assign\,\langle u, s\rangle\; in\, e = alter\,\langle u, s\rangle\; to\,\langle u', s'\rangle\; in\, e'}{[\![u]\!] \vdash assign\,\langle u, s\rangle\; in\; let\, x \Leftarrow e\; in\, e'' = alter\,\langle u, s\rangle\; to\,\langle u', s'\rangle\; in\; let\, x \Leftarrow e'\; in\, e''}$$

both of which follow easily from the definitions of $assign$ and $alter$. The (FETCH) and (ALTER) rules accessing the store also use the derived equalities:

$$\begin{aligned} [\![u]\!] \vdash\; assign\,\langle u, s\rangle\; in\, e &= alter\,\langle u, s\rangle\; to\,\langle u, s\rangle\; in\, e \\ [\![u]\!]; (\neq u) \vdash\; assign\,\langle u, s\rangle\; in\; get(l) &= assign\,\langle u, s\rangle\; in\, [i] \qquad s(l) = i \\ [\![u]\!]; (\neq u) \vdash\; assign\,\langle u, s\rangle\; in\; set(l, i) &= assign\,\langle u, s\{l \mapsto i\}\rangle\; in\, [()]. \end{aligned}$$

The first of these is trivial, and the other two follow from (READ), (WRITE) and the various (SWAP) rules of the metalanguage. □

If we assume that the metalanguage is consistent, then the interpretation is adequate with respect to contextual equivalence in Reduced ML. As with the nu-calculus, this assumption is justified by the existence of non-trivial categorical models, to be presented later.

Proposition 5.10 (Adequacy of Translation) *If* $\mathsf{M}_1, \mathsf{M}_2 \in \mathsf{Exp}_\sigma(u, \Gamma)$ *are two expressions of Reduced ML, with*

$$[\![u, \Gamma]\!]; (\neq u) \vdash [\![\mathsf{M}_1]\!] = [\![\mathsf{M}_2]\!]$$

provable in the metalanguage, then they are contextually equivalent $u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2$.

Proof Suppose that $\mathsf{P}\langle\!\langle - \rangle\!\rangle$ is some program context defined over u. By the compositionality of the translation $[\![-]\!]$, and Lemma 5.8 on the substitution of values,

$$[\![u]\!]; (\neq u) \vdash [\![\mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M}_1 \rangle\!\rangle]\!] = [\![\mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M}_2 \rangle\!\rangle]\!].$$

If now $s \in \mathsf{State}(u)$, then there are evaluation judgements

$$\langle u, s \rangle\, \mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M_i} \rangle\!\rangle \Downarrow_{bool} \langle u_i, s_i \rangle\, b_i$$

for $i = 1, 2$ and some $b_1, b_2 \in \{\mathsf{true}, \mathsf{false}\}$. The correctness result above gives

$$[\![u]\!]; (\neq u) \vdash \mathit{assign}\ \langle u, s \rangle\ \mathit{in}\ [\![\mathsf{P}\langle\!\langle (\vec{\mathsf{x}})\mathsf{M_i} \rangle\!\rangle]\!] = \mathit{alter}\ \langle u, s \rangle\ \mathit{to}\ \langle u_i, s_i \rangle\ \mathit{in}\ [|b_i|]$$

for $i = 1, 2$, and so

$$[\![u]\!]; (\neq u) \vdash \mathit{alter}\ \langle u, s \rangle\ \mathit{to}\ \langle u_1, s_1 \rangle\ \mathit{in}\ [|b_1|] = \mathit{alter}\ \langle u, s \rangle\ \mathit{to}\ \langle u_2, s_2 \rangle\ \mathit{in}\ [|b_2|].$$

We can apply the (MONO$^+$) rule to obtain

$$[\![u]\!]; (\neq u) \vdash |b_1| = |b_2|$$

from which $b_1 = b_2$, and so $u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2$ as required.

The only significant difference between this and the proof of Proposition 3.7 for the nu-calculus is that instead of (DROP) and (MONO) we needed the stronger (MONO$^+$) rule for the final step. ◻

Thus we can use the metalanguage to reason about the contextual equivalence of Reduced ML expressions. This has power similar to the strong applicative equivalence of Section 5: it has some uses, but is intolerant of garbage. This is improved by the addition of the rule

(DROP′) $$\frac{\Gamma \vdash i : \mathit{Int} \qquad \Gamma \vdash e : TA}{\Gamma \vdash \mathit{let}\, n \Leftarrow \mathit{new}\ \mathit{in}\ (\mathit{set}(n, i); e) = e}$$

as a replacement for the ordinary (DROP) rule. With (DROP′), reasoning in the metalanguage is like applicative equivalence: it is successful in most cases where dynamically generated store is used only for temporary variables. Unfortunately, the simpler categorical models do not satisfy this extra rule.

As with the nu-calculus, directly reasoning in the metalanguage cannot cope with more subtle interactions of privacy and higher-order functions, represented by example (7) onwards. These require particular concrete models, for example using relational techniques, or perhaps some relational enhancement to the metalanguage. The first of these we discuss below, the second is work for the future.

8 Categorical Models

In Chapter 3 we saw that if a category satisfies certain requirements then it provides a model for the nu-calculus. Specifically, its internal language will include the computational metalanguage with names; correctness and adequacy results then carry over from metalanguage to category. Careful choice of category can prove a range of contextual equivalences: as with the parametric functor category $\mathcal{P}$ from Chapter 4, which uses categories with relations and is fully abstract at ground and first-order types.

The same method applies in the presence of store: given certain additional conditions, which correspond to the extra features of the metalanguage for store, a category can be used to model Reduced ML. Further, it happens that all the categories given earlier, as models for names, are also suitable to model store, by a general construction.

Recall from Section 4 of Chapter 3 that a category $\mathcal{C}$ is suitable to model the metalanguage with names if the following hold:

- It is cartesian closed.
- It has a strong monad T with units $\eta_A : A \to TA$ all monomorphisms.
- It has a disjoint coproduct $1 + 1$.
- There is a distinguished decidable object N.
- There is a distinguished morphism $new : 1 \to TN$ satisfying certain equations.

To model store too, the following requirements are sufficient:

- The monad T satisfies the *strong mono requirement*, that all the strength maps $t_{A,B} : A \times TB \to T(A \times B)$ are monomorphisms. In fact, it is enough to show that all the $t_{A,1} : A \times T1 \to T(A \times 1)$ are monic. This is equivalent to the (MONO$^+$) rule for computations, from Figure 5.6.

  This property of the monad T does not occur elsewhere in the literature; it would certainly be interesting to know whether it arises in any other context. Roughly speaking, it asserts that elements of T-types may have side-effects, but cannot entirely divert the course of computations. Monads for exceptions, or non-termination, would not satisfy the strong mono requirement.

  This condition implies the ordinary mono requirement, that all the unit maps $\eta_A : A \to TA$ are monic. Conversely, if $\mathcal{C}$ is affine and satisfies the mono requirement, then it satisfies the strong mono requirement. This corresponds to the metalanguage derivation of (MONO$^+$) from the (DROP$^+$) and (MONO) rules.

- There is a distinguished object I, used to interpret the type of integers. This requires an accompanying collection of morphisms, including for example

  $$less : I \times I \to 1 + 1, \qquad plus : I \times I \to I, \qquad \text{and} \qquad 3 : 1 \to I,$$

  which must satisfy various arithmetic equalities. Equivalently, I must be a 'natural numbers object', a categorical generalisation of $\mathbb{N}$; see [52, p. 269].

- There are distinguished morphisms $get : N \to TI$ and $set : N \times I \to T1$ such that the following assertions in the internal language of C are satisfied:

$$\begin{array}{rcl}
n : N,\, i : I & \vdash & let\, x \Leftarrow set(n,i)\; in\; get(n) \;=\; let\, x \Leftarrow set(n,i)\; in\; [i] \\
n : N,\, i, i' : I & \vdash & let\, x \Leftarrow set(n,i)\; in\; set(n,i') \;=\; set(n,i') \\
n, n' : N,\, i, i' : I & \vdash & (eq(n,n') = \mathit{ff}) \implies \\
 & & \quad let\, x \Leftarrow set(n,i)\; in\; set(n',i') \\
 & & \qquad = let\, x \Leftarrow set(n',i')\; in\; set(n,i) \\
a : A,\, n : N,\, i : I & \vdash & let\, x \Leftarrow set(n,i)\; in\; (let\, n' \Leftarrow new\; in\; f(a,n')) \\
 & & \quad = let\, n' \Leftarrow new\; in\; (let\, x \Leftarrow set(n,i)\; in\; f(a,n')).
\end{array}$$

  In this last equation, f is any morphism $A \times N \to TB$. These are clearly just the storage rules of Figure 5.6; we could express them by commutative diagrams in $\mathcal{C}$, but their meaning then disappears in an excess of variable manipulation.

Given such a category $\mathcal{C}$, the embedding of the computational metalanguage as its internal language proceeds exactly as before. The additional types *Unit* and *Int* are interpreted by the objects 1 and I respectively, and the extra term-forming operations give morphisms as in Figure 5.8. All the other details are as described earlier, on page 54 and in Figure 3.5. Any equation provable in the metalanguage for store will hold in the category $\mathcal{C}$; consequently, any non-degenerate $\mathcal{C}$ demonstrates that the metalanguage is consistent.

The translation of the previous section now gives an interpretation of Reduced ML in the category $\mathcal{C}$. For each expression $\mathsf{M} \in \mathsf{Exp}_\sigma(u,\Gamma)$ there is a morphism

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

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

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

Here $N^{|u|}$ is the object of $|u|$-tuples of names. As with the nu-calculus, we take the subobject $(\neq u) \rightarrowtail N^{|u|}$ of distinct $|u|$-tuples and define the composite morphisms:

$$\begin{array}{rcll}
[\![\mathsf{M}]\!]_{\Gamma,\neq u} & = & \left((\neq u) \times [\![\Gamma]\!] \rightarrowtail N^{|u|} \times [\![\Gamma]\!] \xrightarrow{[\![\mathsf{M}]\!]} T[\![\sigma]\!]\right) & \mathsf{M} \in \mathsf{Exp}_\sigma(u,\Gamma) \\
|\mathsf{C}|_{\Gamma,\neq u} & = & \left((\neq u) \times [\![\Gamma]\!] \rightarrowtail N^{|u|} \times [\![\Gamma]\!] \xrightarrow{|\mathsf{C}|} [\![\sigma]\!]\right) & \mathsf{C} \in \mathsf{Can}_\sigma(u,\Gamma) \\
[\![\mathsf{M}]\!]_{\neq u} & = & \left((\neq u) \rightarrowtail N^{|u|} \xrightarrow{[\![\mathsf{M}]\!]} T[\![\sigma]\!]\right) & \mathsf{M} \in \mathsf{Exp}_\sigma(u) \\
|\mathsf{C}|_{\neq u} & = & \left((\neq u) \rightarrowtail N^{|u|} \xrightarrow{|\mathsf{C}|} [\![\sigma]\!]\right) & \mathsf{C} \in \mathsf{Can}_\sigma(u)
\end{array}$$

Correctness and adequacy results carry over to the categorical model. To express them properly, we use the derived morphism constructors

$$\frac{e : (\neq u) \times [\![\Gamma]\!] \to TA}{assign\, \langle u, s\rangle\; in\; e : (\neq u) \times [\![\Gamma]\!] \to TA}\; (s \in \mathsf{State}(u))$$

$$\frac{}{\Gamma \vdash () : \mathit{Unit}} \quad \longmapsto \quad \frac{}{! : \Gamma \to 1}$$

$$\frac{}{\Gamma \vdash 3 : \mathit{Int}} \quad \longmapsto \quad \frac{}{\Gamma \xrightarrow{!} 1 \xrightarrow{3} I}$$

$$\frac{\Gamma \vdash i, i' : \mathit{Int}}{\Gamma \vdash \mathit{plus}(i, i') : \mathit{Int}} \quad \longmapsto \quad \frac{i : \Gamma \to I \qquad i' : \Gamma \to I}{\Gamma \xrightarrow{\langle i, i' \rangle} I \times I \xrightarrow{\mathit{plus}} I}$$

$$\frac{\Gamma \vdash i, i' : \mathit{Int}}{\Gamma \vdash \mathit{less}(i, i') : \mathit{Bool}} \quad \longmapsto \quad \frac{i : \Gamma \to I \qquad i' : \Gamma \to I}{\Gamma \xrightarrow{\langle i, i' \rangle} I \times I \xrightarrow{\mathit{less}} 1 + 1}$$

$$\frac{\Gamma \vdash n : \mathit{Name}}{\Gamma \vdash \mathit{get}(n) : T\mathit{Int}} \quad \longmapsto \quad \frac{n : \Gamma \to N}{\Gamma \xrightarrow{n} N \xrightarrow{\mathit{get}} TI}$$

$$\frac{\Gamma \vdash n : \mathit{Name} \qquad \Gamma \vdash i : \mathit{Int}}{\Gamma \vdash \mathit{set}(n, i) : T\mathit{Unit}} \quad \longmapsto \quad \frac{n : \Gamma \to N \qquad i : \Gamma \to I}{\Gamma \xrightarrow{\langle n, i \rangle} N \times I \xrightarrow{\mathit{set}} T1}$$

Figure 5.8: Additional morphisms to interpret terms of the metalanguage

and

$$\frac{e' : (\neq u') \times [\![\Gamma]\!] \to TA}{\mathit{alter}\ \langle u, s\rangle\ \mathit{to}\ \langle u', s'\rangle\ \mathit{in}\ e' : (\neq u) \times [\![\Gamma]\!] \to TA}\ (u \subseteq u',\ s' \in \mathsf{State}(u')).$$

These represent exactly the same abbreviations as in the metalanguage: they give computations that initialise the store before doing their work.

Proposition 5.11 (Correctness) *If* $\langle u, s\rangle\ \mathsf{M} \Downarrow_\sigma \langle u', s'\rangle\ \mathsf{C}$ *is an evaluation judgement of Reduced ML then*

$$\mathit{assign}\ \langle u, s\rangle\ \mathit{in}\ [\![\mathsf{M}]\!]_{\neq u} = \mathit{alter}\ \langle u, s\rangle\ \mathit{to}\ \langle u', s'\rangle\ \mathit{in}\ [\![\mathsf{C}]\!]_{\neq u}$$

as morphisms in category $\mathcal{C}$.

Proof Follows from Proposition 5.9. □

Proposition 5.12 (Adequacy) *Suppose that* $\mathcal{C}$ *is non-degenerate, in that* $0 \not\cong 1$. *Then for all* $\mathsf{M}_1, \mathsf{M}_2 \in \mathsf{Exp}_\sigma(u, \Gamma)$:

$$[\![\mathsf{M}_1]\!]_{\Gamma,\neq u} = [\![\mathsf{M}_2]\!]_{\Gamma,\neq u} \quad \Longrightarrow \quad u, \Gamma \vdash \mathsf{M}_1 \approx_\sigma \mathsf{M}_2.$$

Proof Exactly as for Proposition 5.10. □

So a non-degenerate category that satisfies certain conditions can be used to model Reduced ML. It validates reasoning in the metalanguage, and may be used itself to prove contextual equivalences. Any categorical model will confirm basic examples such as (3), (4) and (6); more abstract models prove additional equivalences.

9 Example Categories

In the same way that the metalanguage for store is an extension of that for names, all the categories given earlier as models for the nu-calculus are also suitable to model Reduced ML. We need simply apply a general side-effects construction outlined by Moggi in [67, §4.1.2]. Essentially, consideration of the nu-calculus has isolated the problematic parts of ML references; actual storage of values is straightforward, if a little tedious in detail.

Suppose that we have a category $\mathcal{C}$ with strong monad (T, η, μ, t), object of names N and morphism new suitable to model the nu-calculus. We first require an object I to interpret the integers. In the category $\mathit{Set}^{\mathcal{I}}$ for example, the constant functor to the set $\mathbb{Z}$ will do; or in $\mathbf{B}G$, the set $\mathbb{Z}$ with trivial G-action. Based on this, define the *state object* to be:

$$S = I^N.$$

For example in $\mathit{Set}^{\mathcal{I}}$, at stage $u \in \mathcal{I}$ we have that $Su \cong \{s \mid s : (u+1) \to \mathbb{Z}\}$. The extra 1 in the domain of s describes its behaviour at all later stages; this gives a default value for new cells.

We can now construct another strong monad (T', η', μ', t') which allows for change of state. For any object A of $\mathcal{C}$,

$$T'A \;=\; (T(A \times S))^S$$

and if $f : A \to B$ then

$$T'f \;=\; (T(f \times S))^S \;:\; T'A \to T'B.$$

This satisfies the strong mono requirement if T does; which is automatic, as all the monads for names given earlier are both affine and satisfy the ordinary mono requirement.

For example, in $Set^{\mathcal{I}}$ an element of $T'Au$ specifies, for every later stage u' and state $s \in Su'$, a result a and final state s', perhaps at some further stage u''. Functorality adds a degree of uniformity to this behaviour, as does the quotient from the definition of T.

Using the internal language of the original monad, the morphisms to accompany T are defined by:

$$\begin{aligned} a : A \;&\vdash\; \eta'_A a \;=\; \lambda s{:}S.[\langle a, s\rangle] \;:\; T'A \\ e : T'(T'A) \;&\vdash\; \mu'_A e \;=\; \lambda s{:}S.(let\,\langle e', s'\rangle{\Leftarrow}es\; in\; e's') \;:\; T'A \\ a : A,\, e : T'B \;&\vdash\; t'_{A,B}\langle a, b\rangle \;=\; \lambda s{:}S.let\,\langle b, s'\rangle{\Leftarrow}es\; in\,[\langle\langle a, b\rangle, s'\rangle] \;:\; T'(A \times B). \end{aligned}$$

The same object of names N will do, with a different map

$$new' \;=\; \lambda s{:}S.let\, n{\Leftarrow}new\; in\,[\langle n, s\rangle] \;:\; T'N.$$

Notice that this makes no mention of initialisation. In fact in a category such as $Set^{\mathcal{I}}$, every element of the state object S will have some default value for new cells. However, this plays no part in the interpretation of Reduced ML, because the expression $(\mathsf{ref}\, i)$ always provides an initial value.

Manipulation of store is effected by the following maps:

$$\begin{aligned} n : N \;&\vdash\; get(n) \;=\; \lambda s{:}S.[\langle sn, s\rangle] \;:\; TI \\ n : N,\, i : I \;&\vdash\; set(n, i) \;=\; \lambda s{:}S.[\langle (), \lambda n'{:}N.cond(eq(n', n), i, sn')\rangle] \;:\; T1. \end{aligned}$$

Here get simply looks up the value associated with the name n in state s, while set creates a modified state, which differs from s only at the name n. The various equalities that new', get and set must satisfy follow directly from their definitions and the given properties of new. Thus in $\mathcal{C}$ we have a model of Reduced ML, with store managed in a simple way and the original monad T dealing with dynamic creation of cells.

This applies to all the the categories that Chapter 3 introduces: $Set^{\mathcal{I}}$, the atomic topos $\mathcal{A}$, and continuous G-sets $\mathbf{B}G$. As it happens, these do not give particularly abstract models; they validate all reasoning in the metalanguage but go no further. They even fail to confirm the equivalence

2. $\quad u, \Gamma \vdash \quad \mathsf{let\,val\,r = ref}\, i\, \mathsf{in\,M\,end} \;\approx_\sigma\; \mathsf{M} \qquad i \in \mathbb{Z}, \quad \mathsf{r} \notin fv(\mathsf{M}).$

Although the name creation monad T in these models will factor out all unused names, the cell r is touched once, when it is initialised with the value i.

Working over categories with relations, the parametric functor category $\mathcal{P}$, from Section 4 of Chapter 4, is rather better. The model of Reduced ML in $\mathcal{P}$ confirms equivalences (2)–(7), (11) and any others where dynamically generated store is used only for temporary variables. The relations that do this are exactly as for the nu-calculus: spans that identify common visible locations, and ignore private locations.

To make this model still more abstract, we have to adjust the detailed construction of T' from T. The simplest approach here would be to replace $S = I^N$ with some more sophisticated state object, perhaps with extra relational structure. This mirrors the development of operational logical relations based on state, as described in Section 5. It seems that this should be enough to construct a model that validates all the equivalences of Section 4, with no need to go outside the 2-categorical structure of categories with relations.

Sieber, in [117], has an impressive full abstraction result for certain models of store in Algol-like languages, up to second-order types. This uses a sophisticated relational structure over stores, and has similarities to the functor categories described above. It is not yet known whether this result can be carried over to models for references in Reduced ML, nor how powerful it might be there.

Chapter 6

Conclusion

In this final chapter we outline possibilities for further research, describe other published work in the same area, and summarise the results of the dissertation.

1 Directions for Future Research

There are several lines of inquiry that follow on from the work in this dissertation, some that strengthen existing results and others that suggest new ones. We first look at ideas for the nu-calculus and names in general, and then discuss extensions to the work of Chapter 5 on ML-style references.

1.1 The Nu-Calculus

The methods we have developed here are sufficient to prove all the contextual equivalences presented in Section 5 of Chapter 2; yet none are complete, or can express the equivalence $\approx_{\sigma \to \sigma'}$ simply in terms of $\approx_\sigma$ and $\approx_{\sigma'}$. We do not know whether the metalanguage can be made complete for reasoning about contextual equivalence, and we have no fully abstract categorical model. A solution to any of these problems would neatly round off the theory of the nu-calculus.

One possibility is to adapt O'Hearn and Riecke's recent fully abstract model of PCF, which is based on Jung and Tiuryn's logical relations of varying arity [80, 36]. Another approach is to construct a category from the types and expressions of the nu-calculus itself, in the style of Milner's term model for the simply-typed lambda-calculus [59]. Here full abstraction is immediate as contextual equivalence is built in from the start; however there are complications, in particular the need for equalizers in the category.

The requirements given in Chapter 3 for a categorical model of the nu-calculus are sufficient but perhaps a little arbitrary, and this might be improved. For example, it may be that if a category $\mathcal{C}$ has a strong monad T, then the object of names N and morphism new are characterised by some universal property, as with a natural numbers object [42]. The first example of a model was in $Set^{\mathcal{I}}$, and it could be significant that this category is symmetric monoidal closed, with a multiplication '$\otimes$' and exponential '$\multimap$' derived from the operation '$+$' on $\mathcal{I}$. Such structure captures notions of support and non-interference between terms, but it remains to be seen whether this can be generalised.

The success of logical relations suggests that it might be good to add them to the metalanguage, following Plotkin and Abadi's logic for relational parametricity in System F [97].

The idea is that this would match the ease of equational reasoning with the power of logical relations.

A *logic of properties* for the nu-calculus would characterise expressions by the properties that they satisfy, so for example

$$(\{n\} \to \{true\}) \ \wedge \ (\{n'\} \to \{false\})$$

might specify how a function of type $(\nu \to o)$ treats the names n and n'. This is based on the process logics of concurrency theory, with a duality between axiomatic and denotational views of properties [27, 3]. The treatment of new names is problematic; one possibility is to define quantifiers

$$\begin{array}{ll} \forall n.\phi & \text{`for all (fresh) names } n\text{, } \phi \text{ holds'} \\ \exists n.\phi & \text{`for some (private) name } n\text{, } \phi \text{ holds'} \end{array}$$

but it is not clear how best to formalise these.

The definition of the monad T for the parametric functor category $\mathcal{P}$ of Chapter 4 is strikingly similar to Plotkin and Abadi's proof rule for existential types in System F [97]. There is in fact a common origin in Reynolds' notion of 'relational parametricity', but there may also be a closer connection in the form of a direct translation of the nu-calculus into System F. This would use an existential type, or possibly a bounded existential, to conceal the generation of new names [65, 14]. O'Hearn and Riecke have used polymorphism, in a different way, to interpret Algol-like state [79]; Launchbury and Peyton-Jones have also suggested existentials to manage state in Haskell [45].

Quite separate from the refinements of nu-calculus reasoning are the applications to other areas. We have already looked at references in ML, but exceptions, datatypes and structures are dynamically generated too. The same methods may help with models of names in the pi-calculus, or the 'of course' modality from linear logic [61, 6].

A final possibility is that the metalanguage for names of Chapter 3 could be reused to interpret a lazy nu-calculus, with $[\![\sigma \to \sigma']\!] = T[\![\sigma]\!] \to T[\![\sigma']\!]$ for call-by-name function types. The (DROP) and (SWAP) rules then express precisely how abstract names transcend the vagaries of evaluation order in a lazy language.

1.2 Reduced ML

Chapter 5 outlines a number of methods for reasoning about references in Reduced ML; the most promising are operational logical relations based on state and models that use categories with relations. An immediate task is to investigate these further, to establish their strengths and identify their limitations.

In traditional imperative languages almost all uses of assignment are mundane and predictable, even trivial. One of the triumphs of functional programming is that these are eliminated; as a consequence, only the interesting uses of store remain. Reference types in Standard ML can for example be used:

- as local 'own' variables, persisting over several invocations of a function;
- as variables shared between functions in a package;
- to encapsulate state within an object of user defined type;

- as pointers within a graph or similar data structure.

The examples of Chapter 5 only begin to illustrate these possibilities, and the development of reasoning systems must go hand in hand with work on using the full power of references. The interaction with higher-order functions is particularly important: the `memo` and `profile` examples are just a start in the right direction.

The step from Reduced ML to the full language has several effects on references, including:

- storage of values of function type;
- recursively defined functions that use store;
- recursively defined (user declared) datatypes that incorporate store.

In combination these create considerable difficulties for reasoning and the construction of models. Nevertheless the work on Reduced ML — in particular the identification of some categorical requirements for a model of store — should provide a good foundation for future efforts.

2 Related Work

There is a considerable body of work on the subject of functional programming and state; much of it concerned with store, and a little with the issue of names. This section surveys just a sample of the approaches taken, and most of the items cited contain further useful references.

Odersky has developed a theory $\lambda\nu$ that adds local names to the lambda-calculus, and preserves all existing contextual equivalences [74, 75]. Syntactically this language is very like the nu-calculus; differences are that $\lambda\nu$ is untyped and has a call-by-name reduction strategy, with the possibility of 'stuck' terms. So, taking example (10) of Chapter 2, in $\lambda\nu$ the expression $(\lambda x.x == x)(\nu n.n)$ reduces first to $\nu n.n == \nu n.n$ and is then stuck; the equivalent nu-calculus expression evaluates to $true$. Odersky works around the limited scope of names by using a continuation-passing style of programming; he also shows how monadic 'state transformers' can be combined with names to code extensible store.

Odersky, Rabin and Hudak earlier proposed a language λ_{var} which adds variables and assignment to the lambda-calculus [76]. As with $\lambda\nu$, this has the strong property that every contextual equivalence of the lambda-calculus also holds in the extended system.

Augustsson, Rittri and Synek describe a method for distributing a supply of names around a functional program using a binary tree [9]. This is a convenient structure, but still requires an imperative $gensym$ for a practical implementation.

Demers and Donahue have given an equational theory for the Russell language [16], and Boehm has described an axiomatic method for reasoning about side-effects there [12, 13]. However, although Russell has both higher-order functions and storage cells, their interaction is sharply constrained by the 'import rule'.

Mason and Talcott have developed operational methods for reasoning about LISP programs in [54, 55] and, with Honsell and Smith, in [30]. They consider an untyped language with call-by-value semantics and dynamically generated mutable cells. There are substantial example proofs of program equivalences in Mason's thesis [53], though the techniques described are restricted to the language without lambda abstraction or higher-order functions. Mason's notion of 'strong isomorphism' compares with our applicative equivalence for Reduced ML, as applied to ground types.

Felleisen and others have added variable assignment and control operators to the call-by-value lambda-calculus [21, 20]. They present a syntactic, equational theory for the lambda-calculus, and show that it can be extended with certain axioms for reasoning about state.

The Imperative Lambda Calculus of Swarup, Reddy and Ireland [123, 124] is a typed lambda-calculus with references and state. The type system has three distinct layers: applicative, mutable and observer types. This makes a clear distinction between imperative and applicative programming styles.

Riecke has proposed the use of an 'effects delimiter' to manage imperative additions to a functional language [111]; this ensures that particular pieces of code can be treated as if they were purely functional. Gifford and Lucassen's 'effect system' is a more detailed technique that annotates types to record and control possible side-effects [49].

The introduction to Chapter 5 has already described Reynolds' 'Algol-like' languages, and how they compare with Reduced ML. Work on semantics for local variables in this setting has included functor categories [87, 88, 81, 127], categories with relations [82, 83], logical relations [116, 117] and parametric polymorphism [79]. The Algol type system makes a clear distinction between commands, which alter the state, and expressions, which return values. This separation has motivated attempts to formally constrain the use of state, as in Reynolds' 'specification logic' [106, 108, 126, 84] and 'syntactic control of interference' [105, 110, 125, 78].

Various methods have been proposed for carrying out stateful programming in the purely functional language Haskell. Wadler suggests the use of monads to structure functional programs in ways that simulate features such as global state or exception handlers [133, 134, 135]. Hudak's 'mutable abstract datatypes' [32] encode state-like objects using various techniques, including monadic and continuation-passing styles. Given a set of axioms with a certain linearity property, they can then be implemented efficiently using in-place update.

Most recently Launchbury and Peyton Jones have proposed an ingenious use of polymorphic types that wraps up 'state transformers' and presents them as pure functions [89, 43, 44, 45]. This allows efficient implementation without compromising full use of higher-order functions and lazy evaluation. It is even possible to wrap up calls to C code so that they appear functional and are executed lazily.

This work on state in Haskell is impressive, but somewhat orthogonal to the results of this dissertation; it seeks to hide state rather than understand it. As Launchbury and Peyton Jones point out [45, §10], some of the most interesting uses of state, including a distributed name supply, cannot be entirely concealed. Their approach is to use the unsafe $interleaveST$ combinator; our results on names are the kind of reasoning that shows when this is acceptable.

Many people have paid particular attention to the issue of update-in-place and associated efficiences of implementation. Examples include the 'unique' types of Clean [11], Guzman and Hudak's single-threaded lambda-calculus [26], and numerous applications of linear logic [5, 29, 41, 136, 137]. This work gives useful insights into the way functional languages make use of the store available to them; it is however quite distinct from the aim of this dissertation, which is to look at names and references as convenient and powerful programming constructs in their own right, independent of implementation details.

3 Summary of Results

In this dissertation we have used the nu-calculus, a small experimental language, to explore the interaction between higher-order functions and dynamically generated names. Contextual equivalence between expressions of the nu-calculus has provided our benchmark measure of their behaviour; we have described a collection of examples, and have put forward a number of techniques for proving them.

Interesting properties of names in the nu-calculus include their generativity and scoping, together with the issues of privacy and visibility. Contextual equivalence can capture all of these, but it is a difficult relation to demonstrate directly. To remedy this, the major part of this dissertation has comprised the development of some practical proof methods.

- A context lemma that reduces the range of settings in which expressions must be tested for equivalence.
- Applicative equivalence, an operational relation that is easy to demonstrate and proves a range of basic contextual equivalences.
- A translation from the nu-calculus to a computational metalanguage for names. This allows equational reasoning with similar power to applicative equivalence.
- A general scheme for categorical models using a strong monad, with two particular examples: the functor category $Set^{\mathcal{I}}$ and the category $\mathbf{B}G$ of continuous G-sets.
- Operational logical relations, a more sophisticated technique that distinguishes between private and public uses of names. This is complete for proving contextual equivalence up to first-order types.
- A model $\mathcal{P}$ based on categories with relations. This has close links to operational logical relations, and is fully abstract at ground and first-order types.
- Predicated logical relations, an operational method that uses a finer analysis of how names are used, to prove even more contextual equivalences than logical relations alone.

For each of these methods we have described a proof of correctness, and have investigated the range of contextual equivalences validated.

We have also demonstrated how the same techniques can be applied to generative features within a larger programming language, specifically integer references in a subset of Standard ML. Both operational and denotational methods can be smoothly extended to handle store; we have given various uses of logical relations and have described in outline some examples of categorical models.

In summary, this work has shown that dynamically generated names can be added to a functional language in a safe and well-behaved way. Their combination with higher-order functions exhibits subtle and interesting behaviour, and we have developed powerful methods for reasoning about this. We have also outlined how the same approach may work for the extensible, mutable store of Standard ML.

Names and ML-style references have generally been regarded as ‘impure’ language features. In the light of this dissertation, and other similar work, there is every prospect that they can be brought within the fold as powerful and intuitive programming constructs, suitable for clear and safe use in any functional language.

Appendix A

The Meyer-Sieber Examples

Meyer and Sieber, in [57], give a series of examples that illustrate the behaviour of local store in an Algol-like language. For convenience these are reproduced here as MS1–MS7. Further details can be found in the original paper, which also presents an 'invariant-preserving' model that validates all but the final example.

Corresponding to contextual equivalence, some of the examples use a notion of 'observational congruence' between expressions. Others assert that a particular block of code always diverges; equivalently, that it is observationally congruent to a *diverge* command.

MS1 The block below can be replaced by the call to P.

$$\begin{array}{l} \mathbf{begin} \\ \quad \mathbf{new}\, x; \\ \quad P \qquad\qquad \%\ P\ \mathit{is\ declared\ elsewhere} \\ \mathbf{end} \end{array}$$

MS2 The block below always diverges.

$$\begin{array}{l} \mathbf{begin} \\ \quad \mathbf{new}\, x; \\ \quad x := 0; \\ \quad P; \\ \quad \mathbf{if}\ \mathit{contents}(x) = 0\ \mathbf{then}\ \mathit{diverge}\ \mathbf{fi} \\ \mathbf{end} \end{array}$$

In both these examples, the local variable x is invisible to the procedure P.

MS3 The blocks

$$\mathbf{begin}\ \mathbf{new}\, x;\ \mathbf{new}\, y;\ x := 0;\ Q(x, y)\ \mathbf{end}$$

and

$$\mathbf{begin}\ \mathbf{new}\, x;\ \mathbf{new}\, y;\ x := 0;\ Q(y, x)\ \mathbf{end}$$

are observationally congruent. Here Q cannot distinguish the local variables x and y, and must treat them uniformly.

MS4 The block below always diverges.

$$
\begin{array}{l}
\textbf{begin} \\
\quad \textbf{new}\, x;\ \textbf{new}\, y; \\
\quad \textbf{procedure}\ \mathit{Twice};\ \textbf{begin}\ y := 2 * \mathit{contents}(y)\ \textbf{end}; \\
\quad x := 0;\ y := 0; \\
\quad Q(\mathit{Twice}); \qquad\qquad \%\ \mathit{Q\ is\ declared\ elsewhere} \\
\quad \textbf{if}\ \mathit{contents}(x) = 0\ \textbf{then}\ \mathit{diverge}\ \textbf{fi} \\
\textbf{end}
\end{array}
$$

In this example Q is only able to access the variable y, and not x. Rather confusingly, it happens that the procedure Q cannot in fact change the value of y from zero anyhow, due to the unfortunate choice of initial value; starting with $y := 1$ would perhaps make the example clearer.

MS5 The block below always diverges.

$$
\begin{array}{l}
\textbf{begin} \\
\quad \textbf{new}\, x; \\
\quad \textbf{procedure}\ \mathit{Add\_2}; \qquad\qquad \%\ \mathit{Add\_2\ is\ the\ ability\ to\ add\ 2\ to\ x} \\
\qquad \textbf{begin}\ x := \mathit{contents}(x) + 2\ \textbf{end}; \\
\quad x := 0; \\
\quad Q(\mathit{Add\_2}); \qquad\qquad\qquad \%\ \mathit{Q\ is\ declared\ elsewhere} \\
\quad \textbf{if}\ \mathit{contents}(x) \bmod 2 = 0\ \textbf{then}\ \mathit{diverge}\ \textbf{fi} \\
\textbf{end}
\end{array}
$$

Here the external procedure Q is given the power to change x, but only in increments of two. Thus the contents of x remains even.

MS6 The block below always diverges.

$$
\begin{array}{l}
\textbf{begin} \\
\quad \textbf{new}\, x; \\
\quad \textbf{procedure}\ \mathit{AlmostAdd\_2}(z); \\
\qquad \textbf{begin if}\ z = x\ \textbf{then}\ x := 1\ \textbf{else}\ x := \mathit{contents}(x) + 2\ \textbf{fi end}; \\
\quad x := 0; \\
\quad P(\mathit{AlmostAdd\_2}); \\
\quad \textbf{if}\ \mathit{contents}(x) \bmod 2 = 0\ \textbf{then}\ \mathit{diverge}\ \textbf{fi} \\
\textbf{end}
\end{array}
$$

The test $z = x$ here compares locations, rather than their contents. As P does not know of x, the conditional in $\mathit{AlmostAdd\_2}$ always takes the **else** branch.

MS7 The block

$$
\begin{array}{l}
\textbf{begin} \\
\quad \textbf{new}\, x; \\
\quad \textbf{procedure}\, \mathit{Add\_1};\ \textbf{begin}\ x := \mathit{contents}(x) + 1\ \textbf{end}; \\
\quad P(\mathit{Add\_1}) \\
\textbf{end}
\end{array}
$$

and the block

$$
\begin{array}{l}
\textbf{begin} \\
\quad \textbf{new}\, x; \\
\quad \textbf{procedure}\, \mathit{Add\_2};\ \textbf{begin}\ x := \mathit{contents}(x) + 2\ \textbf{end}; \\
\quad P(\mathit{Add\_2}) \\
\textbf{end}
\end{array}
$$

are observationally congruent. The procedure calls $P(\mathit{Add\_1})$ and $P(\mathit{Add\_2})$ differ only in their effect on x; for while they can alter the variable, they cannot read it. As x is then immediately deallocated, the two blocks are equivalent.

Bibliography

[1] *Applications of Categories in Computer Science: Proceedings of the LMS Symposium, Durham 1991*. London Mathematical Society Lecture Note Series 177. Cambridge University Press, 1992. (pp. 136, 138)

[2] M. Abadi and G. Plotkin. Arities in relational parametricity. Unpublished preprint, November 1993. (p. 70)

[3] S. Abramsky. *Domain Theory and the Logic of Observable Properties*. PhD thesis, Queen Mary College, University of London, October 1987. (pp. 33, 123)

[4] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990. (pp. 2, 8, 32)

[5] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. (p. 125)

[6] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Technical Report 92/24, Department of Computing, Imperial College, September 1992. (p. 123)

[7] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. (p. 5)

[8] L. Augustsson. A compiler for Lazy ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227. ACM Press, 1984. (p. 3)

[9] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4:117–123, January 1994. (p. 124)

[10] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North Holland, 1984. (p. 2)

[11] E. Barendsen and J. E. W. Smetsers. Conventional and uniqueness typing in graph rewriting. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 761. Springer-Verlag, 1993. (p. 125)

[12] H.-J. Boehm. *A Logic for the Russell Programming Language*. PhD thesis, Cornell University, Ithaca, New York, February 1984. Also published as Technical Report 84-593, Department of Computer Science, Cornell University. (p. 124)

[13] H.-J. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985. (p. 124)

[14] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985. (p. 123)

[15] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice Hall, 1980. (p. 6)

[16] A. J. Demers and J. E. Donahue. Making variables abstract: An equational theory for Russell. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–72. ACM Press, January 1983. Also published as Technical Report 82-534, Department of Computer Science, Cornell University. (p. 124)

[17] J. Fairbairn. Ponder and its type system. Technical Report 31, University of Cambridge Computer Laboratory, November 1982. (p. 5)

[18] J. Fairbairn. *Design and Implementation of a Simple Typed Language based on the Lambda-Calculus*. PhD thesis, University of Cambridge, May 1984. Also published as Technical Report 75, University of Cambridge Computer Laboratory. (p. 5)

[19] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North Holland, 1986. (p. 19)

[20] M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989. (p. 125)

[21] 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. (p. 125)

[22] *Functional Programming Languages and Computer Architecture: Proceedings of the 5th ACM Conference, Cambridge, MA, USA, August 26–30, 1991*, Lecture Notes in Computer Science 523. Springer-Verlag, 1991. (pp. 139, 140)

[23] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thése de doctorat d'état, Université Paris VII, 1972. (p. 5)

[24] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989. (p. 5)

[25] A. D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, September 1994. (pp. 1, 27, 32, 33, 106)

[26] J. C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–345. IEEE Computer Society Press, 1990. (p. 125)

[27] M. Hennessey. *Algebraic Theory of Processes*. MIT Press, 1989. (p. 123)

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. (p. 6)

[29] S. Holmström. A linear functional language. In *Proceedings of the Aspenæs Workshop on Implementation of Lazy Functional Languages*, Report 53, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, pages 13–52, 1988. (p. 125)

[30] F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects, 1993. To appear in *Information and Computation*. (pp. 27, 101, 124)

[31] D. Howe. Equality in lazy computation systems. In *LICS '89* [47], pages 198–201. (pp. 32, 33, 106)

[32] P. Hudak. Mutable abstract datatypes. Research Report YALEU/DCS/RR-914, Yale University Department of Computer Science, 1992. (p. 125)

[33] P. Hudak, S. Peyton Jones, P. Wadler, *et al.* Report on the programming language Haskell – a non-strict, purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992. (p. 2)

[34] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992. (p. 5)

[35] M. P. Jones. An introduction to Gofer. Included as part of the distribution for Gofer version 2.30a, June 1994. (p. 2)

[36] A. Jung and J. Tiuryn. A new characterization of lambda definability. In *TLCA '93* [129], pages 245–257. (pp. 91, 122)

[37] G. M. Kelly and R. H. Street. Review of the elements of 2-categories. In *Proceedings of the Sydney Category Theory Seminar*, Lecture Notes in Mathematics 420, pages 75–103. Springer-Verlag, 1974. (p. 78)

[38] D. J. King and J. Launchbury. Functional graph algorithms with depth-first search. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, pages 145–155. Springer-Verlag, July 1993. (p. 2)

[39] A. Kock. Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, XXI:1–10, 1970. (p. 53)

[40] A. Kock. Bilinearity and cartesian closed monads. *Mathematica Scandinavica*, 29:161–174, 1971. (p. 53)

[41] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. (p. 125)

[42] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics 7. Cambridge University Press, 1986. (pp. 7, 51, 122)

[43] J. Launchbury. Lazy imperative programming. In *SIPL '93* [118], pages 46–56. (p. 125)

[44] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1994. (p. 125)

[45] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 1995. To appear. (pp. 123, 125)

[46] A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. In *SIPL '93* [118], pages 101–119. (p. 93)

[47] *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1989. (pp. 133, 135)

[48] C. H. Lindsay and S. G. van der Meulen. *Informal Introduction to Algol 68*. North Holland, Amsterdam, 1980. (p. 5)

[49] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88* [99], pages 47–57. (p. 125)

[50] Q. Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *Mathematical Foundations of Programming Semantics: Proceedings of the 7th International Conference*, Lecture Notes in Computer Science 598, pages 1–40. Springer-Verlag, 1992. (p. 8)

[51] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer-Verlag, 1971. (p. 7)

[52] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992. (pp. 7, 59, 63, 116)

[53] 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. (p. 124)

[54] I. A. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):297–327, July 1991. (pp. 107, 124)

[55] I. A. Mason and C. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105:167–215, 1992. (pp. 107, 124)

[56] D. C. J. Matthews. *Programming Language Design with Polymorphism*. PhD thesis, University of Cambridge, 1983. Also published as Technical Report 49, University of Cambridge Computer Laboratory. (p. 5)

[57] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *POPL '88* [99], pages 191–203. (pp. 93, 102, 128)

[58] *Mathematical Foundations of Computer Science: Proceedings of the 18th International Symposium MFCS '93, Gdańsk, Poland, August 30–September 3, 1993*, Lecture Notes in Computer Science 711. Springer-Verlag, 1993. (pp. 137, 140)

[59] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977. (pp. 6, 7, 26, 122)

[60] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, January 1993. Turing Award lecture. (p. 4)

[61] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992. (pp. 4, 123)

[62] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990. (pp. 3, 11, 99)

[63] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 303–314. IEEE Computer Society Press, 1987. (p. 57)

[64] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991. (p. 57)

[65] J. C. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. (p. 123)

[66] E. Moggi. Computational lambda-calculus and monads. In *LICS '89* [47], pages 14–23. (pp. 8, 37)

[67] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1990. (pp. 8, 37, 57, 119)

[68] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. (pp. 8, 37, 43, 51, 52, 79)

[69] E. Moggi. A semantics for evaluation logic, 1993. To appear in *Fundamenta Informaticae*. (p. 59)

[70] J. H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968. (p. 6)

[71] P. Naur *et al.* Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, January 1963. (p. 4)

[72] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991. (p. 5)

[73] F. Nielson and H. R. Nielson. Layered predicates. In *Proceedings of the 1992 Workshop on Semantics: Foundations and Applications*, Lecture Notes in Computer Science 666, pages 425–456. Springer-Verlag, 1992. Also published as Technical Report DAIMI PB-423, Computer Science Department, Aarhus University. (p. 91)

[74] M. Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Yale University Department of Computer Science, May 1993. (p. 124)

[75] M. Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM Symposium on Principles of Programming Languages*, pages 48–59. ACM Press, 1994. (p. 124)

[76] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In *POPL '93* [100], pages 43–56. (p. 124)

[77] P. W. O'Hearn. Linear logic and interference control (preliminary report). In *Category Theory and Computer Science 1991*, Lecture Notes in Computer Science 530, pages 74–93. Springer-Verlag, 1991. (p. 93)

[78] P. W. O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3:435–465, 1993. (pp. 93, 125)

[79] P. W. O'Hearn and J. G. Riecke. Fully abstract translations and parametric polymorphism. In *Programming Languages and Systems — ESOP '94*, Lecture Notes in Computer Science 788, pages 454–468. Springer-Verlag, 1994. (pp. 123, 125)

[80] P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF, June 1994. Submitted to *Information and Computation*. (pp. 91, 122)

[81] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science 1991* [1], pages 217–238. (pp. 57, 93, 125)

[82] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. Technical Report SU-CIS-93-30, School of Computer and Information Science, Syracuse University, October 1993. (pp. 8, 78, 93, 103, 108, 125)

[83] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables (preliminary report). In *POPL '93* [100], pages 171–184. (pp. 8, 93, 125)

[84] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, part 2. *Information and Computation*, 107(1):25–57, November 1993. (pp. 93, 125)

[85] A. Ohori. Representing object identity in a pure functional language. In *ICDT '90: Proceedings of the Third International Conference on Database Theory*, Lecture Notes in Computer Science 470. Springer-Verlag, 1990. (p. 4)

[86] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 1994. To appear. (p. 2)

[87] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982. (pp. 93, 125)

[88] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, 1985. (pp. 57, 93, 125)

[89] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL '93* [100], pages 71–84. (p. 125)

[90] 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. (pp. 8, 37, 39)

[91] A. M. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *MFCS '93* [58], pages 122–141. (pp. -5, 33)

[92] A. M. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *SIPL '93* [118], pages 31–45. (pp. -5, 33)

[93] A. M. Pitts, I. Stark, and V. de Paiva. State and monadic ML. In preparation, 1995. (p. 107)

[94] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. (pp. 6, 19, 20, 24)

[95] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977. (pp. 6, 7)

[96] G. Plotkin. Lambda-definability in the full type heirarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980. (pp. 8, 33)

[97] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA '93* [129], pages 361–375. (pp. 5, 70, 122, 123)

[98] C. G. Ponder, P. McGeer, and A. P. Ng. Are applicative languages inefficient? *ACM SIGPLAN Notices*, 23(6):135–139, June 1988. (p. 2)

[99] *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, January 1988*. ACM Press, 1988. (p. 134)

[100] *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*. ACM Press, 1993. (p. 136)

[101] U. S. Reddy. Global state considered unnecessary. In *SIPL '93* [118], pages 120–135. (p. 93)

[102] U. S. Reddy. Passivity and independence. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 342–352. IEEE Computer Society Press, 1994. (p. 93)

[103] J. C. Reynolds. GEDANKEN — A simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970. (p. 5)

[104] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, Lecture Notes in Computer Science 19, pages 408–425. Springer-Verlag, April 1974. (p. 5)

[105] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, 1978. (p. 125)

[106] J. C. Reynolds. *The Craft of Programming*. Prentice Hall, 1981. (p. 125)

[107] J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North Holland, 1981. (pp. 4, 5, 57, 93)

[108] J. C. Reynolds. Idealized Algol and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982. (p. 125)

[109] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North Holland, 1983. (p. 8)

[110] J. C. Reynolds. Syntactic control of interference, part II. In *Proceedings of ICALP '89*, Lecture Notes in Computer Science 372, pages 704–722. Springer-Verlag, 1989. (p. 125)

[111] J. G. Riecke. Delimiting the scope of effects. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 146–155. ACM Press, 1993. (p. 125)

[112] J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 1995. To appear. (p. 2)

[113] E. Ritter and A. M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *Typed Lambda Calculi and Applications: TLCA '95*. Springer-Verlag Lecture Notes in Computer Science, 1995. To appear. (pp. 99, 106)

[114] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358, 1993. (p. 24)

[115] K. Sieber. Reasoning about sequential functions via logical relations. In *Applications of Categories in Computer Science 1991* [1], pages 258–269. (p. 8)

[116] K. Sieber. New steps towards full abstraction for local variables. In *SIPL '93* [118], pages 88–100. (pp. 93, 125)

[117] K. Sieber. Full abstraction for the second order subset of an Algol-like language (preliminary report). Technical Report A 01/94, Universität des Saarlandes, Saarbrücken, January 1994. (pp. 93, 108, 121, 125)

[118] *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark, June 12, 1993*, Yale University Department of Computer Science Research Report YALEU/DCS/RR-968, 1993. (pp. 134, 137, 138)

[119] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, November 1982. (p. 7)

[120] I. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 1995. To appear. (p. -5)

[121] R. Statman. Completeness, invariance and λ-definability. *Journal of Symbolic Logic*, 47:17–26, 1982. (p. 8)

[122] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977. (p. 7)

[123] V. Swarup and U. S. Reddy. A logical view of assignments. In *Constructivity in Computer Science 1991*, Lecture Notes in Computer Science 613. Springer-Verlag, 1991. (p. 125)

[124] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *FPCA '91* [22], pages 192–214. (p. 125)

[125] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983. (pp. 93, 125)

[126] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990. (pp. 93, 125)

[127] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991. (pp. 93, 125)

[128] M. Tillotson. Introduction to the functional programming language "Ponder". Technical Report 65, University of Cambridge Computer Laboratory, May 1985. (p. 2)

[129] *Typed Lambda Calculi and Applications: Proceedings of TLCA '93, March 16–18, 1993, Utrecht, the Netherlands*, Lecture Notes in Computer Science 664. Springer-Verlag, 1993. (pp. 133, 137)

[130] D. A. Turner. Miranda: a non-strict functional language with polymorphic types*. In *Functional Programming Languages and Computer Architecture 1985*, Lecture Notes in Computer Science 201, pages 1–16. Springer-Verlag, 1985. (p. 2)

[131] R. van der Linden. The ANSA naming model. Architecture Report APM.1003.01, Architecture Projects Management Ltd., Cambridge, UK, July 1993. (p. 4)

[132] A. van Wijngaarden *et al. Revised Report On the Algorithmic Language Algol 68*. Springer, 1976. (p. 5)

[133] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (pp. 3, 125)

[134] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992. (pp. 3, 125)

*Miranda is a trademark of Research Software Limited

[135] P. Wadler. Monads and functional programming. In *Proceedings of the 1992 Marktoberdorf Summer School on Program Design Calculi*, NATO ASI Series F: Computer and System Sciences 118. Springer-Verlag, 1993. (p. 125)

[136] P. Wadler. A taste of linear logic. In *MFCS '93* [58]. (p. 125)

[137] D. Wakeling and C. Runciman. Linearity and laziness. In *FPCA '91* [22], pages 215–240. (p. 125)

[138] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, 1979. (p. 4)

[139] G. C. Wraith. A note on categorical datatypes. In *Category Theory and Computer Science 1989*, Lecture Notes in Computer Science 389. Springer-Verlag, 1989. (p. 5)