A Functional Theory of Local Names

Martin Odersky

Universität Karlsruhe*
76128 Karlsruhe, Germany
odersky@ira.uka.de

Revised from a paper presented at the 21st ACM Symposium on Principles of Programming Languages, January 1994

Abstract

$\lambda\nu$ is an extension of the λ-calculus with a binding construct for local names. The extension has properties analogous to classical λ-calculus and preserves all observational equivalences of λ. It is useful as a basis for modeling wide-spectrum languages that build on a functional core.

1 Introduction

Recent years have given us a good deal of theoretical research on the interaction of imperative programming (exemplified by variable assignment) and functional programming (exemplified by higher order functions) [3, 6, 19, 21, 24]. The common method of all these works is to propose a λ-calculus extended with imperative features and to carry out an exploration of the operational semantics of the new calculus.

Based on our own experience in devising such an extended λ- calculus [13], the present work singles out the *name*, whose only observational property is its identity, as an essential component of any such extension. We present a simple extension of the pure λ-calculus with names; we show by examples how much of the flavor of imperative programming is captured by this simple extension, and we prove compatibility of the extended calculus with the pure calculus in terms of both operational and denotational semantics.

*Most of this work was done while at Yale University

We are in good company; for instance Milner's Turing Award Lecture emphasizes naming as the key idea of the π-calculus [9]. However, Milner relies on names and processes alone, and requires an implementation mapping to recapture functional programming [8]. This implementation is not fully abstract in that it invalidates observational equivalences that hold in a purely functional programming language.

By contrast, this paper presents a syntactic theory for names that builds directly on (call-by-name) λ-calculus. The basic idea is to generalize the notion of constant symbol already present in applied λ-calculus, by introducing an abstraction $\nu n.M$ that binds a name n. Constant symbols in classical applied λ-calculus then become a special case of names that are not bound anywhere. The new calculus, $\lambda\nu$, is pleasingly symmetric: Names can be bound just like placeholder identifiers in λ-abstractions, and both names and identifiers are subject to α-renaming. The difference between the two lies in the operations that can be applied to them. One can substitute a term for an identifier, and one can compare two names for equality, but not vice versa.

In a sense, names are the greatest common denominator of all programming languages that are not purely functional. Hence, one expects a theory that combines names with λ-abstractions to help in understanding design issues of wide-spectrum languages that build on a functional core. So far, the main results of this work are:

- Names can be added to the λ calculus in a referentially transparent way. Full β remains a valid reduction rule.
- The resulting calculus, $\lambda\nu$, is confluent and admits a standard evaluation function.
- The addition of names is fully compatible with functional programming: Every observational equivalence in λ carries over to $\lambda\nu$. This has im-

portant practical consequences. We are guaranteed that every equational technique for verifying, transforming, or compiling functional programs is also applicable to programs with local names.

- The extension property also applies to denotational semantics. There is a model of (simply typed) $\lambda\nu$ that is a conservative extension of the continuous function model of PCF.

Related work. A theory with a scope close to $\lambda\nu$ has also been developed independently by Pitts and Stark [17]. The term languages of both theories are strikingly similar, but their operational semantics are quite different. The nu-calculus of Pitts and Stark is intended to model names as they arise in ML-style references, for instance. It is not intended to be a referentially transparent extension of a functional core (this is discussed further in Section 2).

Recent work on monads [10, 22, 23, 16, 5] shares with $\lambda\nu$ the motivation to extend functional programming languages to new application domains. Monads solve the problem of making sequencing explicit, which is needed if state is to be updated destructively. $\lambda\nu$ solves the orthogonal problem of expressing and encapsulating references. The two techniques complement each other well, as is shown in Example 3.2.

Some of the more syntactic themes of this paper have also been addressed in the context of λ_{var} [13]. The present work extends the scope of [13] with an investigation of models for $\lambda\nu$. It also achieves considerable simplifications by isolating the treatment of names from all other issues of imperative programming. This separation of concerns helped simplify the (rather hard) proofs on the observational equivalence theories of the imperative language. For this reason, we have based an extended version of the λ_{var}-report on $\lambda\nu$ [12].

The rest of this paper is organized as follows. Section 2 describes term syntax and reduction rules of $\lambda\nu$. Section 3 presents two applications of local names, a type reconstruction algorithm and an implementation of state. Section 4 shows properties of $\lambda\nu$, in particular its confluence and its standard evaluation order. Section 5 discusses the observational equivalence theory of $\lambda\nu$ and shows that it is a conservative extension of the corresponding theory of λ. Section 6 gives a denotational semantics for $\lambda\nu$. Section 7 concludes.

2 The $\lambda\nu$ Calculus

Terms. The term-forming productions of $\lambda\nu$ are given in Figure 1. The three productions on the first line are those of classical, pure λ-calculus. The three productions on the next line are particular to $\lambda\nu$. Besides λ-bound identifiers there is a new, countably infinite alphabet of *names*. Names fall into two classes, global and local. A global name n^c is an atomic constant. We assume that there are two such constants denoting the Boolean values *true* and *false*. A local name n^ν is a name that is bound in a name abstraction $\nu n^\nu.M$. In contrast to the case of λ-bound identifiers, nothing is ever substituted for a name. Rather, names can be tested for equality, as in $n_1 == n_2$. Both constants and local names can be operands of $(==)$.

We study here an applied variant of $\lambda\nu$. Accordingly, we have on the last line productions for pairs (M_1, M_2) and applied primitive operators $p\,M$. Primitive operators are always unary, but operators of greater arity can be simulated by currying. We assume that at least the following operators are defined:

$$\begin{array}{lcl} \mathit{pair?}\,(M_1, M_2) & = & \mathit{true} \\ \mathit{pair?}\,\_ & = & \mathit{false} \\[4pt] \mathit{name?}\,n & = & \mathit{true} \\ \mathit{name?}\,\_ & = & \mathit{false} \\[4pt] \mathit{fst}\,(M_1, M_2) & = & M_1 \\ \mathit{snd}\,(M_1, M_2) & = & M_2 \end{array}$$

Notational conventions. We use $BV(M)$ and $FV(M)$ to denote the bound and free identifiers in a term M, respectively. Analogously, $BN(M)$ and $FN(M)$ denote bound and free local names in a term M. A term is *closed* if $FV(M) = FN(M) = \emptyset$. Closed terms are also called *programs*. Note that programs do not contain free local names n^ν, but they may contain constants.

We use $M \equiv N$ for syntactic equality of terms (modulo α-renaming) and reserve $M = N$ for convertibility. If R is a notion of reduction, we use $\xrightarrow[R]{}$ to express that M reduces in one R reduction step to N, and $M \twoheadrightarrow_R N$ to express that M reduces in zero or more R-steps to N. We also use $M \xrightarrow{\Delta} N$ to express that M reduces to N by contracting redex Δ in M.

The syntactic category of *values* V comprises constants, names, pairs, and λ abstractions. An *observable value* (or *answer*) A is an element of some nonempty subset

x	$\in$	$Idents$	λ-bound identifiers
n	$\in$	$Names = Names^c \cup Names^\nu$	names
n^c	$\in$	$Names^c$	constants
n^ν	$\in$	$Names^\nu$	ν-bound local names
p	$\in$	$Primops$	primitive operators
M	$\in$	$\Lambda\nu$	terms

$$\begin{array}{lcl} M & ::= & x \mid \lambda x.M \mid M_1\ M_2 \\ & \mid & n \mid \nu n.M \mid M_1 == M_2 \\ & \mid & (M_1, M_2) \mid p\ M \end{array}$$

Figure 1: Syntax of $\lambda\nu$

$$\begin{array}{llcll} \beta & (\lambda x.M)\ N & \rightarrow & [N/x]\ M & \\ \delta & p\ V & \rightarrow & \delta(p, V) & \\ & & & & \\ eq & n == n & \rightarrow & true & \\ & n == m & \rightarrow & false & (n \neq m) \\ & & & & \\ \nu_\lambda & \nu n.\lambda x.M & \rightarrow & \lambda x.\nu n.M & \\ \nu_p & \nu n.(M_1, M_2) & \rightarrow & (\nu n.M_1,\ \nu n.M_2) & \\ \nu_n & \nu n.m & \rightarrow & m & (n \neq m) \end{array}$$

Figure 2: Reduction rules for $\lambda\nu$

of the alphabet of constants.

$$\begin{array}{lcl} V & ::= & n \mid (M_1, M_2) \mid \lambda x.M \\ A & \in & Answers \subseteq Names^c \end{array}$$

A context $C[\,]$ is a term with a single hole $[\,]$ in it. $C[M]$ denotes the term that results from replacing the hole in $C[\,]$ with M.

Following Barendregt [1], we take terms that differ only in the representatives of bound identifiers and names to be equal. That is, all terms we write are representatives of equivalence classes of α-convertible terms. To avoid name capture problems in substitutions we restrict ourselves to representatives in which bound and free identifiers are always distinct, and we employ the same conventions for names.

Reduction Rules. Figure 2 gives the reduction rules of $\lambda\nu$. They define a reduction relation between terms in the usual way: we take $(\rightarrow)$ to be the smallest relation on $\Lambda\nu \times \Lambda\nu$ that contains the rules in Figure 2 and that, for any context C, is closed under the implication

$$M \rightarrow N \ \Rightarrow\ C[M] \rightarrow C[N].$$

Rule β is the usual reduction rule of pure λ-calculus. Rule δ expresses rewriting of applied primitive operators. To abstract from particular primitive operators and their rewrite rules, we only require the existence of a partial function δ from primitive operators p and values V to terms. δ can be arbitrary, as long as its result does not depend on the body of a an argument function, or the value of a local argument name. That is, we postulate that for every primitive operator p there exist closed terms N_c^p $(c \in Names^c)$. N_ν^p, N_λ^p and $N_{(\cdot,\cdot)}^p$ such that for all values V for which $\delta(p, V)$ is defined:

$$\delta(p,V) \;=\; \begin{cases} N^p_c & \textbf{if } V \equiv c \\ N^p_\nu & \textbf{if } V \text{ is a local name} \\ N^p_\lambda & \textbf{if } V \text{ is a } \lambda\text{-abstraction} \\ N^p_{(\cdot,\cdot)}\; M_1\; M_2 & \textbf{if } V \equiv (M_1, M_2) \end{cases}$$

Note that all primitive operators are strict, since δ requires its arguments to be values.

The remaining rules of Figure 2 are particular to $\lambda\nu$. Rule *eq* defines (==) to be syntactic identity. Rule ν_λ says that ν- and λ-prefixes commute. Rule ν_p says that ν-prefixes distribute through pairs. Finally, rule ν_n says that a ν-prefixes is absorbed by any name that differs from the name bound in the prefix. Taken together, these rules have the effect of pushing names into a term, thus exposing the term's outer structure and allowing it to interact with its environment.

An important consequence of these rules is that the term $\nu n.n$ cannot be reduced further, but is not a value either, and hence cannot be decomposed or compared. In other words, the identity of a name is known only within its (dynamic) scope. This does not restrict expressiveness since it is always possible to extend the scope of a variable by passing the "rest" of the computation as a continuation (see the examples in the next section).

An Alternative. Instead of "pushing" ν-prefixes into a term, one might also consider to "pull" them out of a function application. I.e. rather than with the ν rules of Figure 2 one might want to work with the rules

$$\begin{array}{lll} \nu_L & (\nu n.M_1)\; M_2 & \rightarrow\; \nu n.(M_1\; M_2) \\ \nu_R & M_1\; (\nu n.M_2) & \rightarrow\; \nu n.(M_1\; M_2). \end{array}$$

These rules can be regarded as an axiomatization of *gensym* in Scheme. They closely correspond to the operational semantics of the nu-calculus [17].

Rule ν_L is η-equivalent to rule ν_λ. But adding rule ν_R to the λ calculus breaks the Church-Rosser property. For instance,

$$(\lambda x.(x,x))\;(\nu n.n)$$

reduces (with β) to $(\nu n.n, \nu n.n)$ but also reduces (with ν_R and then β) to $\nu n.(n,n)$, and the two reducts do not reduce by $\beta\nu_L\nu_R$ to a common term. Hence, β needs to be abandoned if we want to have a confluent calculus with ν_R.

The difference between the the nu-calculus and $\lambda\nu$ can also be illustrated by looking at their reductions on the term

$$(\lambda x.x == x)\;(\nu n.n).$$

With ν_R and a suitably restricted β-rule this reduces to *true*, while in $\lambda\nu$ this reduces to

$$\nu n.n == \nu n.n,$$

a term in normal form that is not a value (such terms are often called "stuck"). Intuitively, reduction gets stuck since the value of a symbol is undefined outside its scope. This restriction is required to ensure that all equalities of the underlying λ-calculus are preserved. Indeed, the preservation law even extends to all observational equivalences (Theorem 5.9).

A Note on Church-Encoding Pairs. We have chosen to make the pairing function $(\cdot,\cdot)$ a primitive term constructor with associated primitive projections *fst* and *snd*. What would have happened if we had encoded pairs as functions instead? The Church-encoding of pairs defines a pairing function

$$P \;=\; \lambda x.\lambda y.\lambda f.f\; x\; y$$

and associated projections

$$\begin{array}{lll} \pi_1 & = & \lambda p.p\;(\lambda x.\lambda y.x) \\ \pi_2 & = & \lambda p.p\;(\lambda x.\lambda y.y). \end{array}$$

The crucial question is what happens to ν_p, or, rather its Church-encoded form

$$\nu n.P\; M\; N \;=\; P\;(\nu n.M)\;(\nu n.N). \qquad (1)$$

It is easily verified that this not an equality derivable from the other reductions. On the other hand, if we apply a projection π_i to each side of (1) then we *do* get an equality that is derivable from β and ν_λ. This is shown by some straightforward computation:

$$\begin{array}{ll} & \pi_1\;(\nu n.P\; M\; N) \\ = & (\text{by definition of } \pi_1,\; P) \\ & (\lambda p.p\;(\lambda x.\lambda y.x))\;(\nu n.\lambda f.f\; M\; N) \\ = & (\text{by } \beta) \\ & (\nu n.\lambda f.f\; M\; N)\;(\lambda x.\lambda y.x) \\ = & (\text{by } \nu_\lambda) \\ & (\lambda f.\nu n.f\; M\; N)\;(\lambda x.\lambda y.x) \\ = & (\text{by } \beta) \\ & \nu n.M \\ = & (\text{by definition of } \pi_1,\; P \text{ and } \beta) \\ & \pi_1\;(P\;(\nu n.M)\;(\nu n.N)) \end{array}$$

The case where the projection is π_2 is completely analogous. In summary, the ν_p rule for Church-encoded pairs is subsumed by ν_λ and β, as long as pairs are used as intended (i.e. only projections are applied to them).

```
data Id          =  String
data Term        =  ID Id | AP Term Term |
                    LAM Id Term
data TID         =  Name ()
data Type        =  TV TID | Type :-> Type
data E a         =  Suc a | Err

type TypeEnv     =  Id -> Type
type Subst       =  Type -> Type
type SubstTran a =  (Subst -> E a) -> Subst -> E a

upd              :: (a -> b) -> a -> b -> (a -> b)
upd f x a y      =  if y == x then a else f y

mgu              :: Type -> Type -> E Subst
-- most general unifier; definition is left out
```

```
unify            :: Type -> Type -> SubstTran a

unify t1 t2 k s  =  case mgu (s t1) (s t2) of
                      Suc s' -> k (s . s')
                      Err    -> Err

tp               :: TypeEnv -> Term -> Type
                    -> SubstTran a

tp e (ID n) t    =  unify (e n) t
tp e (AP a b) t  =  new n ->
                      tp e a (TV n :-> t) .
                      tp e b (TV n)
tp e (LAM x a) t =  new n -> new m ->
                      unify (TV n :-> TV m) t .
                      tp (upd e x (TV n)) a (TV m)
```

Figure 3: Type reconstruction algorithm for the simply typed λ-calculus.

3 Applications

To demonstrate how the $\lambda\nu$-extensions can be used in a functional programming language, we study two example applications: a type reconstruction algorithm and an implementation of state transformers. We use a programming notation that extends Haskell with a new construct `new n -> M`, the ASCII form of $\nu n.M$. A name has type `Name a`, for some type `a`. The typing rule rule for `new` is:

$$\frac{\Gamma.\mathtt{n} : \mathtt{Name}\ \tau' \ \vdash\ \mathtt{M} : \tau}{\Gamma\ \vdash\ \mathtt{new\ n\ \text{->}\ M} : \tau}$$

Example 3.1 (Type Reconstruction) Type reconstruction algorithms for polymorphically typed languages need to define fresh identifiers for type variables "on the fly". To this purpose, a name supply is usually passed along as an additional argument to the type reconstruction function. As an alternative, we present here a type reconstruction algorithm for the simply typed λ-calculus that replaces the name supply by bound $\lambda\nu$ names.

The code for the type checker is given in Figure 3. Types are either variables `TV n` or function types `t1 :-> t2`. The identifying part `n` of a type variable `TV n` is a name (of type `TID`, which is a synonym for `Name ()`). The main function `tp` constructs a proof for a goal $e\ \vdash\ a : t$, where e is a typing environment, a is a term, and t is a type. `e`, `a` and `t` are the first three arguments of `tp`.

Fresh names are created in the clauses of `tp` that have to do with function abstraction and application. `tp` is written in continuation passing style in order to extend the scope of names as far as needed. Its result is a substitution transformer (of type `SubstTran`), which is a mapping that takes a continuation and a substitution and yields either failure or succeeds with some result type that is determined by the continuation.

Example 3.2 (State Transformers)
Using state-transformers, one can write imperative programs in a functional programming language, by treating an imperative statement as a function from states to states (and, possibly, intermediate results). State transformers can be classified according to whether they are global or local, and according to whether state is fixed or dynamic.

[22] and [23] describe *local* state-transformers that can be embedded in other terms and that operate on a *fixed* state data structure. By contrast, [16] describes *global* state-transformers that act as the main program and thus cannot be embedded in another term. State in [16] is *dynamic*, i.e. it consists of a heap with dynamically created references.

Figure 4 shows an implementation of local state-transformers with dynamic state. This is to my knowledge the first fully formal treatment of this class of state-transformers, even if [4] and [5] contain similar informal proposals.

State is represented as a polymorphic function from

```
  type State  =  all a. Name a -> a
  type ST a   =  all b. (a -> State -> b) -> State -> b

-- Monadic Operators:                                      -- State-Based Operators:

  return         :: a -> ST a                                newref         :: ST (Name a)
  (>>=)          :: ST a -> (a -> ST b) -> ST b              (:=)           :: Name a -> a -> ST ()
  pure           :: ST a -> a                                deref          :: Name a -> ST a

  return a   k s  =  k a s                                   newref     k s  =  new n -> k n s
  (p >>= q)  k s  =  p (\x -> q x k) s                       (n := a)   k s  =  k () (upd s n a)
                                                             deref n    k s  =  k (s n) s
  pure p          =  p (\x -> \s -> x) bot
  bot             =  bot                                     upd s n x m     =  if n == m then x else s m
```

Figure 4: State Transformers

names of type `Name a` to terms of type `a`. Its type is:

```
    all a.Name a -> a
```

A state-transformer of type `ST a` is a function that takes a continuation and a state as arguments, and returns the result of the continuation. Its type is:

```
    all b.(a -> State -> b) -> State -> b
```

Note that the polymorphic types of state and state transformers exceed the capabilities of first-order type systems such as Haskell's or ML's. However, an efficient implementation of state transformers would treat type `ST a` as an abstract data type and would hide type `State` altogether in order to guarantee that state is single-threaded. Such an implementation could do with just ML-style let-polymorphism.

State transformers form a Kleisli monad, with `return` as the monad unit, and with infix `(>>=)` as the "bind" operator. If we leave out the redundant state parameter `s` this is just the standard continuation monad. The result type of a continuation is an observer of type `State -> a` (as in [21]).

Function `pure`, of type `ST a -> a`, allows one to get out of the `ST` monad. `pure` runs its state transformer argument in an empty initial state with a continuation that yields its first argument as answer.

The remaining operations access state. `newref` returns a freshly allocated reference as result. Its implementation is based on ν-abstraction. `(n := a)` updates the state, returning the unit value as result, while `deref n` returns the current value of the state at reference `n`.

This concludes our first implementation of state in $\lambda\nu$. It is perhaps surprising how simple such an implementation can be, once the problem of expressing local names is taken care of. However, one could argue that we have oversimplified, in that the implementation of Figure 4 does not really describe state! Indeed, there are two trouble-spots.

The first problem is caused by the fact that the state argument `s` is not linear in the definition of `deref`. Therefore, access to state is only single-threaded if the application `s n` in the body of `deref` gets resolved before control is passed to the continuation. But nothing in the implementation forces this evaluation order! One could solve the problem by making continuations strict in their first argument. However, this forces `s n` to be reduced to a *value*, which is needlessly drastic. To ensure single-threadedness, it is enough to just perform the function application without further evaluation.

Another problem concerns the meaning of readers and assignments that involve names from some outer block. In the implementation of Figure 4, such accesses are not errors. Instead, the read or write is performed on a locally allocated cell that is named by the non-local name. Therefore, the same name might identify several locations in different states. This approach, which is similar to the semantics of state in [20], is perfectly acceptable from a theoretical standpoint. But it raises some implementation problems, since it prevents the identification of names with machine addresses.

Both problems are solved by a slightly more refined implementation that marks stored terms with a data constructor. We modify the type of state as follows:

```
type State  =  all a. Name a -> D a
data D a    =  D a
```

The implementation of the state-based operators then becomes:

```
newref   k s  =  new n ->
                   k n (upd s n (D bottom))

(n := a) k s  =  case s n of
                   D b -> k () (upd s n (D a))
deref n  k s  =  case s n of
                   D a -> k a s
```

In the new implementation, the `case` construct in the body of `deref` forces `s n` to be evaluated before control is passed to the case-branch. This takes care of the first problem. Moreover, both readers and writers require that an entry for the accessed reference exists in the local state, and `newref` allocates such an entry for a freshly created reference. This takes care of the second problem.

The contribution of $\lambda\nu$ to this implementation is rather subtle. It consists of the ν-abstraction in the code of `newref` and the equality test in function `upd`. Nevertheless, the presence of local names is important for modeling dynamic local state in a simple way. To see this, let's try to model local state without local names, by representing heaps as arrays with references as indices, say. Now, any implementation of local state has to distinguish between variables that are defined in different `pure`-blocks. This is necessary to guard against access to non-local variables and against export of local variables out of their block, both referentially opaque operations. A straightforward scheme to distinguish between variables defined in different blocks would pass a name supply to each block, such that the block, and all the variables defined in it, can be tagged with a unique identifier. The problem with this scheme is that it has a "poisoning" effect on the environment that surrounds a block. Each function now has to pass along name-supply arguments even if the function itself does not contain `pure`-blocks as subterms. It is not clear what is gained by this method over a program that contains a single, global state, and hence is imperative all the way to the top.

4 Reduction

This section details the fundamental laws of $\lambda\nu$-reduction: reduction is confluent and there is a standard evaluation order. The treatment largely follows [1], and we assume that the reader is familiar with some of the more fundamental definitions and theorems given there. Most of the proofs in this and the following chapters are sketched or left out; for a more detailed treatment, see [11].

Confluence

We show in this section analogues for $\lambda\nu$ of the Finite Developments and Church-Rosser theorems for the λ-calculus.

Definition 4.1 Let $\lambda_0\nu$ be the extension of $\lambda\nu$ with labeled redexes $(\lambda_0 x.M)\,N$ and $p_0\ V$ and with labeled reduction rules

$$\begin{array}{lll} \beta_0 : & (\lambda_0 x.M)\,N & \to\ [N/x]\,M \\ \delta_0 : & p_0\ V & \to\ \delta(p,V). \end{array}$$

Let $\xrightarrow[0]{}$ be the reduction relation generated by β_0, δ_0, eq, ν_λ, ν_p, ν_n.

Theorem 4.2 (Finite Developments) $\xrightarrow[0]{}$ is strongly normalizing.

Proof: The proof is similar to the proof of finite developments in the pure λ calculus ([1],CH.11,§2). We construct a family of non-negative decreasing weightings and show that each reduction step maps a term with a decreasing weighting to a term with a smaller decreasing weighting.

Theorem 4.3 The notion of reduction in $\lambda\nu$ is Church-Rosser: if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ then there is a term M_3 s.t. $M_1 \twoheadrightarrow M_3$ and $M_2 \twoheadrightarrow M_3$.

Proof: Using a case analysis on reduction rules, coupled with a case analysis on the relative position of redexes, one shows that the notion of reduction $\delta\nu$ is weakly Church-Rosser and commutes with β. Then by Theorem 4.2 and Newman's lemma ([1],CH.3,§1) $\delta\nu$ is Church-Rosser, and together with the lemma of Hindley/Rosen ([1],CH.3,§3) this implies the proposition.

Evaluation

As programmers, we are interested not only in proving equality of terms, but also in evaluating them, i.e. reducing them to an answer. We now define a computable evaluation function that maps a term to an answer A iff $\lambda\nu \vdash M = A$. Following Felleisen [2], the evaluation function is defined by means of a *context machine.* At every step, the machine separates its argument term deterministically into an evaluation context and a redex and then performs a reduction on the redex. Evaluation stops once the argument is an answer. Evaluation contexts for $\lambda\nu$ are defined as follows:

$$E \ ::= \ [\,] \mid E\,M \mid p\,E \mid \nu n.E \tag{2}$$

The first three clauses generate evaluation contexts for the applied call-by-name λ-calculus, whereas the last clause is particular to $\lambda\nu$.

Definition. The *deterministic reduction* relation $\underset{d}{\rightarrow}$ on terms in $\Lambda\nu$ is the smallest relation that satisfies

$$M \xrightarrow{M} N \ \Rightarrow \ E[M] \underset{d}{\rightarrow} E[N].$$

A simple inspection of the productions for E establishes that $\underset{d}{\rightarrow}$ is indeed deterministic:

Proposition 4.4 For any redexes Δ_1, Δ_2 and evaluation contexts E_1, E_2,

$$E_1[\Delta_1] \equiv E_2[\Delta_2] \ \Rightarrow \ E_1 \equiv E_2 \wedge \Delta_1 \equiv \Delta_2.$$

A redex Δ is a *head redex* of a term M if $M \equiv E[\Delta]$, for some evaluation context E. A redex that is not head redex is called an *internal redex.* Reduction of internal redexes keeps head and internal redexes separate, in the sense of

Lemma 4.5 Let M be a program s.t. $M \xrightarrow{\Delta} N$ where Δ is an internal redex of M. Then,
(i) If N has a head redex then so has M,
(ii) the residual of M's head redex is head redex in N,
(iii) the residuals of every internal redex in M are internal redexes in N.

Theorem 4.6 (Correspondence) For every program $M \in \Lambda\nu$ and every answer A,

$$M \twoheadrightarrow A \ \Leftrightarrow \ M \underset{d}{\twoheadrightarrow} A.$$

Proof: Direction "$\Leftarrow$" follows immediately. To prove "$\Rightarrow$", assume that $M \underset{d}{\twoheadrightarrow} A$. One shows first as an intermediate result that, whenever $M \twoheadrightarrow A$, there is a term N s.t. $M \underset{d}{\twoheadrightarrow} N \underset{i}{\twoheadrightarrow} A$, where the reduction sequence $N \underset{i}{\twoheadrightarrow} A$ from N to A consists of only internal reductions. This result corresponds to the main lemma for the Curry/Feys standardization theorem ([1],CH.11,§4) and has exactly the same proof. That proof uses only the theorem of finite developments (Theorem 4.2 for $\lambda\nu$) and a lemma equivalent to Lemma 4.5. The proposition then follows from the observation that no internal $\lambda\nu$ reduction ends in an answer, hence we must have $N \equiv A$.

5 Observational Equivalence

Observational equivalence is the most comprehensive notion of equivalence between program fragments. Intuitively, two terms are observationally equivalent if they cannot be distinguished by some experiment. Experiments wrap a term in some arbitrary context that binds all free identifiers and local names in a term. The only observation allowed in an experiment is whether the resulting program reduces to an answer, and, if so, to which one. We define observational equivalence for arbitrary extensions of applied λ calculus. In the following, let $\mathcal{T}$ be an equational theory that extends λ and has term language $Terms(\mathcal{T})$ and a set of answers $Ans(\mathcal{T}) \subset Names^c(\mathcal{T})$. We assume that $Names^c(\mathcal{T}) \backslash Ans(\mathcal{T})$ is infinite.

Definition 5.1 Two terms $M, N \in Terms(\mathcal{T})$ are *observationally equivalent* in $\mathcal{T}$, written $\mathcal{T} \models M \cong N$, iff for all contexts C in $Terms(\mathcal{T})$ such that $C[M]$ and $C[N]$ are closed, and for all answers $A \in Ans(\mathcal{T})$,

$$\mathcal{T} \vdash C[M] = A \ \Leftrightarrow \ \mathcal{T} \vdash C[N] = A.$$

Proposition 5.2 The following are observational equivalences in $\lambda\nu$:

$$\begin{array}{lcll} \nu n.\nu m.M & \cong & \nu m.\nu n.M & (n \neq m) \\ \nu n.M & \cong & M & (n \notin FN(M)) \end{array}$$

Definition 5.3 $\mathcal{T}$ is an *observational extension* of $\mathcal{T}_0$ if $Terms(\mathcal{T}) \supseteq Terms(\mathcal{T}_0)$ and, for all $M \in Terms(\mathcal{T}_0)$,

$$\mathcal{T}_0 \models M \cong N \ \Rightarrow \mathcal{T} \models M \cong N.$$

The extension is *conservative* if the implication can be strengthened to an equivalence.

The main result of this section states that $\lambda\nu$ is an observational extension of λ. The proof relies on the construction of a syntactic embedding from $\lambda\nu$ to λ. Syntactic embeddings were first defined in [13]; we use here the following, simplified definitions.

Definition 5.4 Given an inductively defined term language $Terms$, an *extended term* is formed from the inductive definitions of $Terms$ and $[\,]$. (Hence, both terms and contexts are extended terms).

Definition 5.5 A term M is λ-closed iff $FV(M) = \emptyset$. M may contain free occurrences of local names.

Definition 5.6 (Syntactic Embedding) Let $\mathcal{T}$ and $\mathcal{T}_0$ be extensions of λ such that $Terms(\mathcal{T}) \supseteq Terms(\mathcal{T}_0)$

and $Ans(\mathcal{T}) = Ans(\mathcal{T}_0)$. Let $\mathcal{E}$ be a syntactic mapping from extended $\mathcal{T}$-terms to extended $\mathcal{T}_0$-terms. Then $\mathcal{E}$ is a *syntactic embedding* of $\mathcal{T}$ in $\mathcal{T}_0$ if it satisfies the following two requirements.

1. $\mathcal{E}$ preserves λ-closed $\mathcal{T}_0$-subterms. For all $\mathcal{T}$-contexts C, λ-closed $\mathcal{T}_0$-terms M,

$$\mathcal{T}_0 \vdash \mathcal{E}[\![C[M]]\!] = \mathcal{E}[\![C]\!]\,[M].$$

2. $\mathcal{E}$ preserves semantics. For all closed $\mathcal{T}$-terms M, answers A,

$$\mathcal{T} \vdash M = A \Leftrightarrow \mathcal{T}_0 \vdash \mathcal{E}[\![M]\!] = A.$$

Theorem 5.7 Let $\mathcal{T}$ and $\mathcal{T}_0$ be extensions of λ such that $Terms(\mathcal{T}) \supseteq Terms(\mathcal{T}_0)$ and $Ans(\mathcal{T}) = Ans(\mathcal{T}_0)$. If there is a syntactic embedding of $\mathcal{T}$ in $\mathcal{T}_0$ then $\mathcal{T}$ is an observational extension of $\mathcal{T}_0$.

The next lemma was shown in [11].

Lemma 5.8 There exists a syntactic embedding of $\lambda\nu$ in λ.

Together with Theorem 5.7, this implies:

Theorem 5.9 $\lambda\nu$ is a conservative observational extension of λ.

Proof: By Lemma 5.8, $\mathcal{E}$ is a syntactic embedding of $\lambda\nu$ in λ. By Theorem 5.7 this implies that $\lambda\nu$ is an observational extension of λ. That the extension is conservative follows directly from the observation that $\lambda\nu$-convertibility is a conservative extension of λ-convertibility.

6 Denotational Semantics

We develop a denotational semantics for a typed version of $\lambda\nu$ that results from adding ν-abstractions to PCF terms. The semantics is an extension of the continuous function model for PCF [18]. In that sense, it follows the spirit of previous sections, where $\lambda\nu$ was studied as an extension of λ-calculus, rather than as a theory of its own.

We use a "possible worlds" semantics [15], where a world is characterized by a finite set of names. Intuitively, these are the names available for program evaluation. As a new twist, the meaning of the term $\nu n.M$ in a world W is the *intersection* of the meaning of M in all possible worlds that extend W with a new suitable location. A location is suitable if it does not clash with locations used in other parts of the program. Instead of trying to trace these locations explicitly, we simply choose the "best" co-finite set L of possible candidate locations in the information ordering. I.e.

$$[\![\nu n.M]\!]\,\rho \;=\; \bigcup_{L \in \wp^{cofin}(Name)} \;\bigcap_{l \in L} [\![M]\!]\,\rho[n \mapsto l]$$

It is a consequence of Theorem 6.9 that the least upper bound always exists. The meaning of all other constructs is the same as in PCF.

Example 6.1 The meaning of $\nu n.n$ is bottom:

$$\begin{aligned} [\![\nu n.n]\!]\,\rho &= \textstyle\bigcup_{L \in \wp^{cofin}(Name)} \bigcap_{l \in L} l \\ &= \bot \end{aligned}$$

This corresponds to the term $\nu n.n$ being "stuck" in the reduction semantics. It reflects on the fact that the identity of a name is known only within its scope.

Example 6.2 The meaning of $\nu n.\nu m.n == m$ is $false$. Indeed,

$$[\![\nu n.\nu m.n == m]\!]\,\rho \;=\; \textstyle\bigcup_K \bigcap_{k \in K} \bigcup_L \bigcap_{l \in L} k = l$$

where K and L range over $\wp^{cofin}(Name)$. If K, k, and L are chosen, then $\bigcap_{l \in L} k = l$ is either $\bot$ (if $k \in L$) or $false$ (if $k \notin L$). Hence, for any given K and k, the value of $\bigcup_{L \in \wp^{cofin}(Name)} \bigcap_{l \in L} k = l$ is $false$. But this implies $[\![\nu n.\nu m.n == m]\!]\,\rho = false$.

In the rest of this section, we make these notions precise. In particular, we need to give a semantic characterization of the functions that belong to a world W – informally, these are the functions that access only locations in W. We also have deal with the fact that the *lub* of a chain of functions that access strictly increasing sets of locations accesses an infinite number of locations, and hence is not a member of any world. As a consequence, our domains form a locally complete partial order (*lcpo*) [7] rather than a *cpo*.

We base our discussion on a typed version of $\lambda\nu$, given by the typing rules in Figure 5. We also assume the usual constants and operations of PCF, without listing their typing rules explicitly.

Definition 6.3 Let $Name$ be a countably infinite set of names, and let $m, n \in Name$. The *exchange* $X_{m,n}$ is the unique logical relation such that for names x, y,

$$\begin{aligned} x\, X_{m,n}\, y \;\Leftrightarrow\; & m = x \wedge y = n \;\vee \\ & m = y \wedge x = n \;\vee \\ & m \neq x = y \neq n, \end{aligned}$$

$$
\begin{array}{llll}
(ID) & \Gamma, x{:}\tau \vdash x : \tau & (NAME) & \Gamma, n{:}Name \vdash n : Name \\[2ex]
(ABS) & \dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \tau} & (NU) & \dfrac{\Gamma, n{:}Name \vdash M : \tau}{\Gamma \vdash \nu n.M : \tau} \\[2ex]
(APPL) & \dfrac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau} & (EQ) & \dfrac{\Gamma \vdash M : Name \qquad \Gamma \vdash N : Name}{\Gamma \vdash M == N : Bool}
\end{array}
$$

Figure 5: Typing Rules for $\lambda\nu$

$$
\begin{array}{lll}
(ID) & [\![\Gamma, x{:}\tau \rhd x : \tau]\!]\ \rho & = \ \rho\ x \\
(ABS) & [\![\Gamma \rhd \lambda x.M : \sigma \to \tau]\!]\ \rho & = \ \lambda y.[\![\Gamma, x{:}\sigma \rhd M : \tau]\!]\ \rho[x \mapsto y] \\
(APPL) & [\![\Gamma \rhd M\ N : \tau]\!]\ \rho & = \ ([\![\Gamma \rhd M : \sigma \mapsto \tau]\!]\ \rho)\ ([\![\Gamma \rhd N : \sigma]\!]\ \rho) \\[1ex]
(NAME) & [\![\Gamma, n{:}Name \rhd n : Name]\!]\ \rho & = \ \rho\ n \\
(NU) & [\![\Gamma \rhd \nu n.M : \tau]\!]\ \rho & = \ \bigcup_{L \in \wp^{cofin}(Name)} \bigcap_{l \in L} [\![\Gamma, n{:}Name \rhd M : \tau]\!]\ \rho[n \mapsto l] \\
(EQ) & [\![\Gamma \rhd M == N : Bool]\!]\ \rho & = \ [\![\Gamma \rhd M : Name]\!]\ \rho = [\![\Gamma \rhd N : Name]\!]\ \rho
\end{array}
$$

Figure 6: Semantic Function $[\![\cdot]\!]$

for elements of other ground types,

$$x\ X_{m,n}\ y \quad \Leftrightarrow \quad x = y,$$

and such that $\bot\ X_{m,n}\ \bot$.

Exchanges have the property that they are closed under intersections and unions:

Lemma 6.4 (i) If, for all $i \in I$, $A_i\ X_{m,n}\ B_i$, then

$$\bigcap_{i \in I} A_i\ \ X_{m,n}\ \ \bigcap_{i \in I} B_i.$$

(ii) If $\{A_i \mid i \in I\}$ and $\{B_i \mid i \in I\}$ are directed sets and for all $i \in I$, $A_i\ \ X_{m,n}\ \ B_i$, then

$$\bigcup_{i \in I} A_i\ X_{m,n}\ \bigcup_{i \in I} B_i.$$

Definition 6.5 The *smooth set* of a value $x \in D$,

$$smooth(x) \quad = \quad \{m : Name \mid \exists L \in \wp^{cofin}(Name).\ \forall n \in L.\ x\ X_{m,n}\ x\}.$$

The *support* of x is the complement of its smooth set,

$$support(x) = Name \setminus smooth(x).$$

Informally, $support(x)$ is x if x is a name, and is the set of names accessed by x if x is a function. A characterization of *support* and *smooth* that is easier to use in proofs is given by:

Lemma 6.6

$$m \in smooth(x) \quad \Leftrightarrow \quad \forall n \in smooth(x).\ x\ X_{m,n}\ x.$$

This equivalence cannot be used to define *smooth*, however, since its right hand side is not monotonic in $smooth(x)$.

Example 6.7 The support of the name n is $\{n\}$. The support of the function $f \overset{def}{=} \lambda x.x == m$ is $\{m\}$. This can be derived as follows: Let n be any name different from m. Then $m\ X_{m,n}\ n$. But $f\ m \neq f\ n$, which proves $\neg(f\ X_{m,n} f)$ and hence shows that m is not in $smooth(f)$. On the other hand, let k, l be arbitrary names different from m. It is easy to check that $f\ X_{k,l}\ f$. Hence, by Lemma 6.6, $smooth(f) \supseteq Name \backslash \{m\}$. In summary, $smooth(f) = Name \backslash \{m\}$, and hence $support(f) = \{m\}$.

Definition 6.8 For type τ and finite name set W, the domains $[\![\tau]\!]_W$ and $[\![\tau]\!]$ are defined as follows:

$[\![Name]\!]_W \;=\; W_\perp.$

For all other ground types o, $[\![o]\!]_W$ is the usual interpretation of o in PCF.

$[\![\sigma \to \tau]\!]_W = \{f : [\![\sigma]\!] \xrightarrow{lc} [\![\tau]\!] \mid \mathit{support}(f) \subseteq W\}$, where $D \xrightarrow{lc} E$ denotes the locally continuous functions from D to E.

$[\![\tau]\!] \;=\; \bigcup_{W \in \wp^{fin}(Name)} [\![\tau]\!]_W.$

The interpretation of $\lambda\nu$ terms is defined in Figure 6. Let Γ be a set of type hypotheses and let W be a finite set of names. A (Γ, W)-environment is a function ρ on identifiers and names that maps each identifier $x \in \mathit{dom}(\Gamma)$ to a value in $[\![\Gamma(x)]\!]$, and that maps each name $n \in \mathit{dom}(\Gamma)$ to a unique name in W. The semantic function $[\![\cdot]\!]$ takes as arguments a type judgement $\Gamma \rhd M : \tau$ and a (Γ, W)-environment ρ. It yields a value in $[\![\tau]\!]_W$.

Theorem 6.9 For all valid type judgements $\Gamma \vdash M : \tau$, finite name sets W and (Γ, W)-environments ρ,

$$[\![\Gamma \vdash M : \tau]\!]\;\rho \;\in\; [\![\tau]\!]_W.$$

Proof: A standard induction on type derivations. The following lemma is needed for the abstraction case.

Lemma 6.10 Let $m, n \in Name$. Let $\Gamma \vdash M : \tau$ be a valid type judgement. Let ρ, ρ' be (Γ, W) environments such that, for all $x \in \mathit{dom}(\Gamma)$, $\rho\ x \;\; X_{m,n} \;\; \rho'\ x$. Then

$$[\![\Gamma \rhd M : \tau]\!]\;\rho \;\; X_{m,n} \;\; [\![\Gamma \rhd M : \tau]\!]\;\rho'.$$

Theorem 6.11 $[\![\cdot]\!]$ defines a computationally adequate model of $\lambda\nu$.

Proof: One verifies easily that all reductions in $\lambda\nu$ are equalities in the model. To show adequacy, we adapt Plotkin's adequacy proof for PCF [18]. Say M is *computable* if one of conditions (1)-(4) holds.
(1) M is closed of ground type, and $[\![M]\!] = [\![A]\!]$ implies $M \twoheadrightarrow A$.
(2) M is closed, of type $\sigma \to \tau$, and $M\ N$ is computable for all closed, computable terms N of type σ.
(3) $x : \tau$ is free in M, and $[N/x]M$ is computable for all closed, computable terms N of type τ.
(4) $n : Name$ is free in M, and $\nu n.M$ is computable.
Using structural induction on M, one shows that every term in $\lambda\nu$ is computable, which implies the proposition.

The model fails to be fully abstract. A counter-example to full abstraction is as follows. Consider the program fragment

$$\begin{array}{l} \nu m. \\ \quad f(\lambda x.\textbf{if } x == m \textbf{ then } x \textbf{ else } \perp) \textbf{ and} \\ \quad f(\lambda x.\textbf{if } x == m \textbf{ then } \perp \textbf{ else } x) \end{array}$$

for an arbitrary Boolean ranged function f, defined elsewhere. An easy case analysis shows that this fragment is observationally equivalent to

$$f(\perp).$$

However, the two fragments are distinguished in our model. This can be seen by substituting for f the function F defined below.

$$F(x) = \begin{cases} \textbf{true} & \textbf{if}\, \mathit{support}(x) \neq \emptyset \\ \textbf{false} & \textbf{otherwise} \end{cases}$$

A similar example was suggested to us by Peter O'Hearn. It remains to be seen whether recent advances in models for Algol-like languages [14] are applicable in the setting of $\lambda\nu$.

7 Conclusions

We have studied reduction semantics, observational equivalence theory and denotational semantics of $\lambda\nu$, a theory for functions that create local names. Each of these three equational theories for $\lambda\nu$ is a conservative extension of the corresponding standard theory for λ (respectively PCF). $\lambda\nu$ is in that sense fully compatible with functional programming. There is also good evidence that it is a useful foundation for modelling many constructs that so far were outside the domain of functional programming. For instance, Example 3.2 shows how imperative programming with mutable local variables can be expressed in $\lambda\nu$. It would be interesting to see other applications of the calculus, such as in logic or concurrent programming.

Acknowledgements This work was supported in part by grant N00014-91-J-4043 from DARPA. I thank Vincent Dornic, Paul Hudak, Peter O'Hearn and Dan Rabin for their comments on earlier versions of the paper. Dan Rabin in particular helped to improve its presentation considerably. John Launchbury, Jayadev Misra, David Turner and Phil Wadler also commented on this work in helpful discussions.

References

[1] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[2] E. Crank and M. Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 233–244, January 1991.

[3] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[4] P. Hudak and D. Rabin. Mutable abstract datatypes – or – how to have your state and munge it too. Research Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, July 1992.

[5] J. Launchbury. Lazy imperative programming. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 46–56, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[6] I. Mason and C. Talcott. Axiomatising operational equivalence in the presence of side effects. In *IEEE Symposium on Logic in Computer Science*, pages 284–303, Asilomar, California, June 1989.

[7] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 191–203. ACM, ACM Press, Januar y 1988.

[8] R. Milner. Functions as processes. Rapport de Recherche 1154, INRIA Sophia-Antipolis, February 1990.

[9] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, January 1993. Turing Award lecture.

[10] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 1989 IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[11] M. Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Department of Computer Science, Yale University, May 1993.

[12] M. Odersky and D. Rabin. The unexpurgated call-by-name, assignment, and the lambda-calculus. Research Report YALEU/DCS/RR-930, Department of Computer Science, Yale University, May 1993.

[13] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, call-by-value, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[14] P. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 171–184. ACM Press, January 1993.

[15] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, August 1982.

[16] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, January 1993.

[17] A. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 31–45, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[18] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[19] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[20] J. G. Riecke. Delimiting the scope of effects. In *Proc. Conf. on Functional Programming and Computer Architecture*, pages 146–155, June 1993.

[21] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[22] P. Wadler. Comprehending monads. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 61–78, June 1990.

[23] P. Wadler. The essence of functional programming. In *Proc.19th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[24] S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 57–70. ACM Press, January 1993.