Yale University
Department of Computer Science

A Syntactic Theory of Local Names

Martin Odersky

Yale University, Department of Computer Science,
Box 2158 Yale Station, New Haven, CT 06520

Research Report YALEU/DCS/RR-965
May, 1993

A Syntactic Theory of Local Names

Martin Odersky

Yale University, Department of Computer Science,
Box 2158 Yale Station, New Haven, CT 06520

Abstract

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

1 Introduction

Names are ubiquitous in programming. We find them as the substrate for mutable variables in imperative programming languages, for logic variables in Prolog, or for communication links in process algebras. Milner's Turing Award Lecture [11] emphasizes naming as the key idea of the π-calculus [12, 13]. Even if names are in general introduced as the basis of more complex entieties, they can also be useful on their own. For instance, we often find statements like "let x be some fresh identifier" in definitions of denotational semantics or type checking algorithms.

Names are different from the identifiers serving as placeholders in functional languages: in contrast to a placeholder, nothing is ever substituted for a name. Conversely, equality tests are meaningful for names but not for placeholders. Names are also different from constant strings, at least if the programming language in question admits recursion. In this case, the same text string can denote different names in different recursive instances of a program construct.

This paper presents a syntactic theory that extends the λ-calculus with names. 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: Both placeholder-identifiers and names can be bound, and both 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.

By studying a theory that combines names with λ-abstractions we hope to gain some understanding of the issues involved in the design of wide-spectrum languages that build on a functional core. The main results of this work are:

- Names can be added to the λ calculus in a referentially transparent way. That is, 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. That is, every observational equivalence in λ carries over to $\lambda\nu$. This has important practical consequences. We are guaranteed that every equational technique for verifying, transforming, or compiling functional programs is also applicable to programs with local names.

Many of the themes of this paper have also been addressed in the context of λ_{var} [17]. The main contribution of the present work over [17] is a considerably simpler theory that isolates 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 imperative languages. For this reason, we have based an extended version of the λ_{var}-report on $\lambda\nu$ [16].

Recent work on monads [14, 23, 24, 18, 8] 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 refer-

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

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

Figure 1: Syntax of $\lambda\nu$

β	$(\lambda x.M)\ N$	$\rightarrow$	$[N/x]\,M$	
δ	$p\ V$	$\rightarrow$	$\delta(p, V)$	
eq	$n == n$	$\rightarrow$	*true*	
	$n == m$	$\rightarrow$	*false*	$(n \neq m)$
ν_λ	$\nu n.\lambda x.M$	$\rightarrow$	$\lambda x.\nu n.M$	
ν_p	$\nu n.(M_1, M_2)$	$\rightarrow$	$(\nu n.M_1,\ \nu n.M_2)$	
ν_n	$\nu n.m$	$\rightarrow$	m	$(n \neq m)$

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

ences. The two techniques complement each other well, as is shown in Example 3.2.

Several other calculi use a notion similar to names for expressing mutable variables in imperative languages. The motivation for these calculi comes partly from the desire to explain the semantics of existing languages (e.g. [5, 9] for Scheme or ML, [20, 10, 25] for Algol), and partly from the desire to extend functional programming with full imperative state [22, 17]. The calculi differ in the assumed order of evaluation, in the kinds of side effects that are admitted, in whether state is implicit or explicit, and in many other properties. But they all have in common an alphabet of α-renamable names that identify mutable variables. By factoring out this common concept, and studying it independently and irrespective of the particular additional semantics attached to names, we hope to find a common basis for the various approaches that describe state.

A theory with a motivation close to $\lambda\nu$ has also been developed independently by Pitts and Stark [19]. Even though the term languages of both theories are strikingly similar, their laws are fundamentally different. The nu-calculus of Pitts and Stark is intended to model names as they are used for ML-style references. It is not intended to be a referentially transparent extension of a functional core. Hence, our result of conservative extension does not apply to the nu-calculus.

The rest of this paper is structured as follows. Section 2 describes term syntax and reduction rules of $\lambda\nu$, a calculus of functions and names. 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 function. Sections 5 and 6 discuss the observational equivalence theory of $\lambda\nu$ and prove that it is a conservative extension of the corresponding theory of λ. 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 obtained by currying. We assume that at least the following operators are defined:

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

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 $M \xrightarrow[R]{\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow} N$ 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 of the alphabet of constants.

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

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.

We let letters L, M, N range over terms, and V and W range over values. We let letters m, n range over names. We use a ν-superscript with these to indicate a local name, and a c superscript to indicate a constant. Alternatively, we also use the letter c for a constant.

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

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

Equality $(=)$ is the smallest equivalence relation that contains reduction $(\rightarrow)$.

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^p_c $(c \in \mathit{Names}^c)$. N^p_ν, N^p_λ and $N^p_{(\cdot,\cdot)}$ 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),

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

$$\lambda x.\lambda y.\lambda f.f\ x\ y$$

and associated projections

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

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:

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

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).

3 Applications

To demonstrate how the new constructs in $\lambda\nu$ could be used in a functional programming language, we present two example applications: a type reconstruction algorithm, and an implementation of state-transformers. We use a program much like Haskell, writing `\x -> M` for $\lambda x.M$ and `(.)` for function composition. We extend Haskell by a new term construct `new n ->M`, the ASCII form of $\nu n.M$. Types are presented only informally; they are in general modeled after the type systems of ML and Haskell, with some polymorphic extensions in the second example. A name has type `Name a`, for some type `a`. The typing rule rule for ν is:

$$\frac{\Gamma.n : \mathit{Name}\ \tau' \vdash M : \tau}{\Gamma \vdash \nu n.M : \tau}$$

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. Typically, names are represented as integers, and the name supply indicates the next unused integer. 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`. The function returns 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. Success and failure are represented as the two alternatives of type `E a`. Fresh names are created in the clauses of `tp` that have to do with function abstraction and application. The unification function `unify` takes as arguments two types, a continuation, and a substitution. If the argument types are unifiable, it augments the argument substitution with their most general unifier, passing the result to its continuation.

```
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.

If the arguments are not unifiable, type reconstruction fails.

Example 3.2 (State Transformers)
State-transformers are a way to write imperative programs in a functional programming language, by treating an imperative statement as a function from states to states. They have received much attention recently. State-transformers can be classified according to whether they are global or local, and according to whether state is fixed or dynamic.

[23] and [24] describe *local* state-transformers that can be embedded in other terms and that operate on a *fixed* state data structure. By contrast, [18] describes *global* state-transformers that act as the main program and thus cannot be embedded in another term. State in [18] 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 [6] and [8] contain similar informal proposals.

State is represented as a polymorphic function from 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 function 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, to guarantee that state is single-threaded. Such an implementation could do with just let-polymorphism in the style of ML.

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 continution is an *observer*[1] of type `State -> a`.

Function `begin`, of type `ST a -> a`, allows one to get out of the `ST` monad. `begin` runs its state transformer argument in an empty initial state and with the identity function as continuation.

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 implementation of state in $\lambda\nu$. It is

[1] in the sense of [22].

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

-- Monadic Operators:

return          :: a -> ST a
(>>=)           :: ST a -> (a -> ST b) -> ST b
begin           :: ST a -> a

return a   k s  =  k a s
(p >>= q)  k s  =  p (\x -> q x k) s

begin p         =  p (\x -> x) bottom
bottom          =  bottom

-- State-Based Operators:

newref          :: ST (Name a)
(:=)            :: Name a -> a -> ST ()
deref           :: Name a -> ST a

newref     k s  =  new n -> k n s
(n := a)   k s  =  k () (upd s n a)
deref n    k s  =  k (s n) s

upd s n x m     =  if n == m then x else s m
```

Figure 4: State Transformers

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 [21], 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, which 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 (we take D-patterns to be refutable). 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 **begin**-blocks. This is necessary to guard against access to non-local variables and against export of lo-

cal 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 **begin**-blocks as subterms. It is not clear what is be gained by this method over a program that contains a single, global state, and hence is imperative all the way to the top. Of course, we could have also used $\lambda\nu$-names for the task of marking blocks uniquely. Such a scheme was described by Launchbury [8]. It might lead to more efficient implementations than the simpler method of equating references with names that we have used here.

Interestingly, it is possible in principle to model local state without local names. As part of the proof of observational extension it will be shown in Section 6 that there is another technique, somewhat similar to a de Bruijn numbering [3], that allows names to be distinguished without needing name supplies to be passed. However, due to its complexity the numbering method is hardly practical as a programming technique.

4 Fundamental Theorems

This section details the fundamental laws of $\lambda\nu$: reduction is confluent and there is a standard evaluation function. The treatment largely follows [1], and we assume that the reader is familiar with the some of the more fundamental definitions and theorems given there.

4.1 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

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

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). Let $\lambda'\nu$ be the extension of λ_0 with weighted identifiers x^k and weighted primitive function application $p_0^k\ M$ $(k > 0)$. We define a norm $\|\cdot\|$ on $\lambda'\nu$ terms by

$$\begin{array}{lcl} \|x\| & = & 1 \\ \|x^k\| & = & k \\ \|\lambda x.M\| & = & 1 + \|M\| \\ \|\lambda_0 x.M\| & = & 1 + \|M\| \\ \|M_1\ M_2\| & = & \|M_1\| + \|M_2\| \\ \|n\| & = & 1 \\ \|\nu n.M\| & = & 2 \times \|M\| \\ \|M_1 == M_2\| & = & \|M_1\| + \|M_2\| \\ \|(M_1, M_2)\| & = & 1 + \|M_1\| + \|M_2\| \\ \|p\ M\| & = & 1 + \|M\| \\ \|p_0^k\ M\| & = & k + \|M\| \end{array}$$

A term $M \in \Lambda'\nu$ has a *decreasing weighting* if for every occurrence of a labeled term $(\lambda_0 x.M_1)\ M_2$ in M, every occurrence of x in M_1 has a weight $k \geq \|M_2\|$, and, for every occurrence of a labeled term $p_0^k\ V$ in M, $n + \|V\| > \|\delta(p, V)\|$.

By a proof similar to the one in [1], Lemma 11.2.18(*ii*) one shows that, if M has a decreasing weighting, and $M \xrightarrow[0]{} N$, then N has a decreasing weighting. Furthermore, an easy case analysis on the notion of reduction $\xrightarrow[0]{}$ in $\lambda'\nu$ shows that for all terms $M, N \in \Lambda'\nu$,

$$M \xrightarrow[0]{} N \ \Rightarrow\ \|M\| > \|N\|.$$

Since weightings are well-founded, $\xrightarrow[0]{}$ is strongly normalizing on all terms in $\Lambda'\nu$ that have a decreasing weighting.

Now every term M in Λ_0 can be transformed to a term with a decreasing weighting in $\Lambda'\nu$, by giving appropriate weights to all labeled occurrences of identifiers and primitive functions. As a consequence, $\xrightarrow[0]{}$ is strongly normalizing on $\lambda_0\nu$. ■

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. ■

4.2 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:

$$E \ ::= \ [\,] \mid E\,M \mid p\,E \mid \nu n.E \qquad (2)$$

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 $\xrightarrow[d]{}$ on terms in $\Lambda\nu$ is the smallest relation that satisfies

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

A simple inspection of the productions for E establishes that $\xrightarrow[d]{}$ 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.

Proof: Let Δ be an internal redex in M. By the definition of evaluation contexts (2), there is an evaluation context E, context C, term M' such that

$$M \ \equiv \ E[M'\,(C[\Delta])].$$

(*i*) Let R be the reduct of Δ. Then

$$N \ \equiv \ E[M'\,(C[R])].$$

Assume that N has a head-redex Δ_h. Then because of the form of evaluation contexts (2), either Δ_h is contained in M', or $\Delta_h \equiv M'\,(C[R])$. In both cases, Δ_h is the residual of a head redex in M.

The proofs of parts (*ii*) and (*iii*) are similar. ■

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

$$M \twoheadrightarrow A \ \Leftrightarrow \ M \xrightarrow[d]{}\!\!\!\!\twoheadrightarrow A.$$

Proof: Direction "$\Leftarrow$" follows immediately. To prove "$\Rightarrow$", assume that $M \xrightarrow[d]{}\!\!\!\!\twoheadrightarrow A$. One shows first as an intermediate result that, whenever $M \twoheadrightarrow A$, there is a term N s.t. $M \xrightarrow[d]{}\!\!\!\!\twoheadrightarrow N \xrightarrow[i]{}\!\!\!\!\twoheadrightarrow A$, where the reduction sequence $N \xrightarrow[i]{}\!\!\!\!\twoheadrightarrow A$ 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

In this section we study observational equivalence, 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 $A \in Ans(\mathcal{T})$,

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

Lemma 5.2 For any terms $M, N \in \mathit{Terms}(\mathcal{T})$,

$$\mathcal{T} \models M \cong N \Leftrightarrow \forall C.\mathcal{T} \models C[M] \cong C[N].$$

Proof: "⇒": Assume $M \cong N$, let A be an answer, and let C be a context. Let C' be a context such that $C'[C[M]]$ and $C'[C[N]]$ are closed. Then $M \cong N$ implies

$$C'[C[M]] = A \Leftrightarrow C'[C[N]] = A.$$

Since C' was arbitrary, $C[M] \cong C[N]$.
"⇐": Pick $C = [\,]$. ■

Lemma 5.3 For any two terms $M, N \in \mathit{Terms}(\mathcal{T})$,

$$\mathcal{T} \models \lambda x.M \cong \lambda x.N \Leftrightarrow \mathcal{T} \models M \cong N$$

Proof: "⇒": Assume $\lambda x.M \cong \lambda x.N$, let A be an answer, and let C be a context such that $C[M]$ and $C[N]$ are closed. Then

$$\begin{array}{ll} & C[M] = A \\ \Leftrightarrow & \text{(by } \beta\text{-conversion)} \\ & C[(\lambda y.[y/x]\ M)\ x] = A \\ \Leftrightarrow & \text{(premise)} \\ & C[(\lambda y.[y/x]\ N)\ x] = A \\ \Leftrightarrow & \text{(by } \beta\text{-conversion)} \\ & C[N] = A \end{array}$$

Hence, $M \cong N$.

"⇐" is a consequence of Lemma 5.2. ■

Proving Observational Equivalences

The definition of ($\cong$) gives us only a very cumbersome way to reason about observational equivalence, since it relies on a universal quantification over all contexts. We now summarize two theorems that are useful for proving observational equivalences. The proof method is similar to the critical pair technique used in Knuth-Bendix completion [7]. For a more complete treatment, see [15].

The theorems require a more formal treatment of the rules for reduction and observational equivalence than the exposition in previous sections. The following is an excerpt of the relevant definitions of [15].

Both reduction rules and rules for observational equivalences are defined as relations between *meta-terms*. A meta-term is formed from the inductive definition of terms, *meta-variables* a, b, and a substitution construct $[M/x]N$, where M and N are meta-terms. Hence, meta-terms include terms as a subset. A *valuation* is a mapping from meta-variables to meta-terms that is extended homomorphically to a mapping from meta-terms to meta-terms. A notion of reduction is expressed by a set of reduction rules

$$\mathcal{R} = \{L_i \to R_i\}_{i \in I}$$

where I is an arbitrary (possibly infinite) index set, and all L_i and R_i are meta-terms. The reduction relation generated by $\mathcal{R}$ is the smallest relation between meta-terms that contains all valuation instances of rules in $\mathcal{R}$ and that is closed under context formation. Likewise, we can define a notion of *similarity* $\sim$ by a *symmetric* set of rules

$$\mathcal{S} = \{S_j \to T_j\}_{j \in J}$$

where again J is arbitrary and S_j, T_j are meta-terms. $\sim$ is then the smallest relation between meta-terms that contains all valuation instances of rules in $\mathcal{S}$ and that is closed under context formation.

Two meta-terms L_1, L_2 *overlap* at a valuation ρ if there is a non-variable sub(meta-)term M of L_i such that $\rho M \equiv \rho L_j$ $(i, j \in \{0, 1\}\ i \neq j)$.

Two sub(meta-)terms M_1, M_2 of a common meta-term *interfere* wrt a set of meta-terms $\mathcal{L}$ if there is a valuation ρ and there are meta-terms $L_1, L_2 \in \mathcal{L}$ such that $M_1 \equiv \rho L_1$, $M_2 \equiv \rho L_2$ and L_1 and L_2 overlap at ρ. We sometimes do not mention the set $\mathcal{L}$ explicitly. In particular, when we say that a redex Δ and a pattern instance of a similarity P interfere, we mean that they interfere wrt the set of left-hand sides of reduction rules and similarities

$$\{L_i \mid i \in I\} \cup \{S_j \mid j \in J\}.$$

Given similarity $\sim$, we define *parallel similarity* as follows: $M \sim_1 N$ if M and N differ in sub(meta-)terms $M_1, N_1, \ldots, M_n, N_n$, $M_i \sim N_i$ for each $i \in \{1, \ldots n\}$, and the M_i, as well as the N_i, are pairwise non-interfering. The set $\{M_1, \ldots, M_n\}$ sometimes is written as a superscript, as in

$$M \overset{\{M_1,\ldots,M_n\}}{\sim_1} N.$$

Two meta-terms M, N form a *critical pair* if there exists a *root* meta-term R, a redex Δ, and pattern instances of similarities $P_1, \ldots, P_n$ $(n \geq 1)$, such that

$$R \xrightarrow{\Delta} N \qquad \text{and} \qquad R \overset{\{P_1,\ldots,P_n\}}{\sim_1} M$$

and Δ interferes with each P_i $(i = 1, .., n)$. The pair is *deterministically critical* if Δ is head redex in R. We use the notation

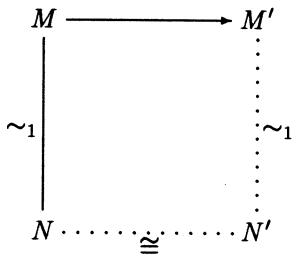
$$M \sim_1 R \to N$$

for a critical pair M, N with root R.

A given set of equations can be shown to be an observational equivalence by formulating it as a similarity relation and showing that it satisfies the following two properties.

$\sim$ *preserves answers* if $M \sim A$ implies $M = A$.

$\sim$ is *locally stable* if, for all critical pairs $N \sim_1 M \to M'$ there is a meta-term N' such that the following diagram commutes.



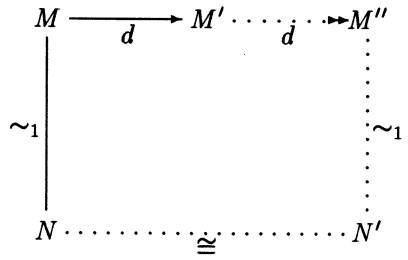
Theorem 5.4 Let $\approx$ be the equivalence closure of $\sim$. If $\sim$ is locally stable and preserves answers then $\approx \subseteq \cong$.

We also need a second, stronger version of Theorem 5.4 that applies if the calculus has a notion of evaluation contexts that satisfies the following two criteria.

Evaluation contexts are *downward closed* if, whenever $C_1[C_2]$ is an evaluation context, then so is C_2. $\sim$ *interferes* with evaluation contexts, if some left-hand-side S_j of $\mathcal{S}$ overlaps with some sub(meta-)term of an inductive production for evaluation contexts.

If these criteria are fulfilled, a weaker notion of *deterministic* local stability is sufficient to guarantee that a similarity is an observational equivalence:

$\sim$ is *deterministically locally stable* if for all deterministically critical pairs $N \sim_1 M \to M'$ there are meta-terms M'', N' such that the following diagram commutes.



Theorem 5.5 Let $\approx$ be the equivalence closure of $\sim$. If

- $\mathcal{T}$ has downward closed evaluation contexts,
- $\sim$ preserves evaluation contexts,
- $\sim$ is deterministically locally stable, and
- $\sim$ is answer-preserving

then $\approx \subseteq \cong$.

A straightforward analysis shows that $\lambda\nu$'s evaluation contexts are downward closed, Furthermore, since the productions for evaluation contexts contain only meta-variables as sub(meta-)terms[2], no similarity relation can interfere with evaluation contexts. Hence, to prove that a similarity is an observational equivalence in $\lambda\nu$ it suffices to show that it satisfies the last two conditions of Theorem 5.5.

Theorem 5.5 will be used extensively in the next section. We use it here in the proof of

Proposition 5.6 The following are observational equivalences in $\lambda\nu$:

$$\nu n.\nu m.M \cong \nu m.\nu n.M \qquad (n \neq m) \qquad (3)$$

$$\nu n.M \cong M \qquad (n \notin FN(M)) \qquad (4)$$

Proof: We prove (3) here; the proof of (4) is given in [15].

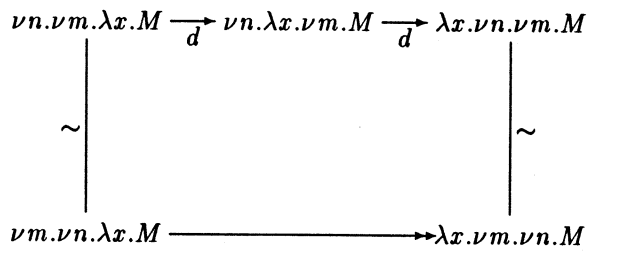
Let a be a meta-variable and let

$$\mathcal{S} = \{\nu n.\nu m.a \cong \nu n.\nu m.a \mid n, m \in Names^{\nu}, n \neq m\}.$$

Let $\sim$ be the compatible valuation closure of $\mathcal{S}$. Since no side of $\mathcal{S}$ matches an answer, $\sim$ vacuously preserves answers. We now show that $\sim$ is also deterministically locally stable.

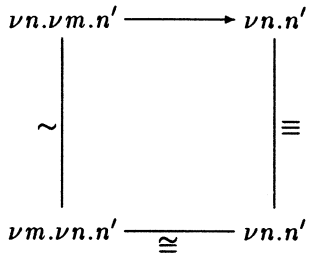
Matching (3) against $\lambda\nu$'s reduction rules establishes that a redex interferes with a pattern instance of $\mathcal{S}$ only if the redex is of the form $\nu m.M$, for some meta-term M, and the pattern instance is $\nu n.\nu m.M$, for $n \neq m$. If $\nu m.M$ is a redex, then M must be a λ-abstraction, a pair, or a name.

If M is a λ-abstraction, say $\lambda x.N$, then the following diagram commutes:



[2]Note that a meta-term may not contain a variable ranging over contexts.

If M is a name, say n', then $m \neq n'$ (otherwise there would be no redex), and the following diagram commutes.



The bottom edge of this diagram is justified by (4), since $m \neq n'$. The remaining case where M is a pair is omitted (it is found in [15]).

This shows that $\sim$ is deterministically locally stable. With Theorem 5.5, (3) follows. ■

6 Observational Extension

Definition 6.1 $\mathcal{T}$ 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 are certain functions (defined below) that map terms of one language to terms of a sublanguage.

6.1 Syntactic Embeddings

Definition 6.2 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 6.3 A term M is λ-closed iff $FV(M) = \emptyset$. M may contain free occurrences of local names.

Definition 6.4 (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,
$$\mathcal{T} \vdash M = A \Leftrightarrow \mathcal{T}_0 \vdash \mathcal{E}[\![M]\!] = A.$$

Syntactic embeddings generalize the syntactic (macro) expansions of [4]. They are useful for two reasons. First, a syntactic embedding $\mathcal{E}$ of $\mathcal{T}$ in $\mathcal{T}_0$ allows us to extend models of $\mathcal{T}_0$ to all of $\mathcal{T}$, simply by composing the denotation function of $\mathcal{T}_0$ with $\mathcal{E}$. As a concrete example, the syntactic embedding from $\lambda\nu$ to λ given in Figure 5 gives us immediately a denotational semantics of $\lambda\nu$. Since a syntactic embedding is the identity on programs in its codomain, the resulting semantics maps λ-terms to the functions they denote in the chosen model of classical λ-calculus. Second, and somewhat related to the first point, the existence of a syntactic embedding between two theories guarantees that all observational equivalences of the base theory are preserved in the extension[3]. Indeed, we have:

Theorem 6.5 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 $\mathcal{T}_0$.

Proof: Assume that $\mathcal{T}_0 \models M \cong N$ Then, forall answers A and $\mathcal{T}_0$-contexts C_0 such that $C_0[M]$ and $C_0[N]$ are closed,

$$\mathcal{T}_0 \vdash C_0[M] = A \Leftrightarrow \mathcal{T}_0 \vdash C_0[N] = A.$$

Assume first that both M and N are λ-closed. Let $\mathcal{E}$ be a syntactic embedding of $\mathcal{T}$ in $\mathcal{T}_0$. Let C be a $\mathcal{T}$-context such that $C[M]$ and $C[N]$ are closed, and let A be an answer such that $C[M] = A$. Then,

$$\begin{aligned}
& \mathcal{T} \vdash C[M] = A \\
\Leftrightarrow\ & (\mathcal{E} \text{ preserves semantics}) \\
& \mathcal{T}_0 \vdash \mathcal{E}[\![C[M]]\!] = A \\
\Leftrightarrow\ & (\mathcal{E} \text{ preserves } \lambda\text{-closed } \mathcal{T}_0 \text{ terms}) \\
& \mathcal{T}_0 \vdash \mathcal{E}[\![C]\!]\,[M] = A \\
\Leftrightarrow\ & (\text{premise: } \mathcal{T}_0 \models M \cong N) \\
& \mathcal{T}_0 \vdash \mathcal{E}[\![C]\!][N] = A \\
\Leftrightarrow\ & (\text{reverse the argument}) \\
& \mathcal{T} \vdash C[N] = A.
\end{aligned}$$

[3] Another approach to proving observational extension is based on constructing a retraction between models rather than a syntactic embedding between term languages [21]. At present, this seems to require a fully abstract denotational semantics of the base language, however.

Since A and C were arbitrary, $\mathcal{T} \models M \cong N$. Now let M and N be arbitrary $\mathcal{T}_0$ terms, with $FV(M) \cup FV(N) = \{x_1, ..., x_n\}$. Then,

$$
\begin{array}{ll}
 & \mathcal{T}_0 \vdash M \cong N \\
\Rightarrow & \text{(Lemma 5.2)} \\
 & \mathcal{T}_0 \vdash \lambda x_1. \ldots \lambda x_n.M \cong \lambda x_1. \ldots \lambda x_n.N \\
\Rightarrow & \text{(first part of proof)} \\
 & \mathcal{T} \vdash \lambda x_1. \ldots \lambda x_n.M \cong \lambda x_1. \ldots \lambda x_n.N \\
\Rightarrow & \text{(Lemma 5.3)} \\
 & \mathcal{T} \vdash M \cong N.
\end{array}
$$

■

6.2 A Syntactic Embedding of $\lambda\nu$ in λ

The main difficulty in constructing a syntactic embedding of $\lambda\nu$ in λ has to do with the representation of names. Since names are subject to α-renaming, which is necessary to avoid name capture problems, we cannot simply assign a fixed constant symbol to a name. Instead, the embedding maps a name occurrence to a *level number* that indicates the number of ν's between the occurrence of the name and its definition. This is somewhat similar to the de Bruijn numbering of λ-bound identifiers [3] (but see below for an important difference).

For instance, in $\nu n.(\nu m.n, n)$, the first occurrence of n in the pair would have level number 1, while the second one would have level number 0. Level numbers have to be adjusted during reduction when ν abstractions are created or destroyed. For instance, $\nu n.(\nu m.n, n)$ reduces (by rule ν_n) to $\nu n.(n, n)$, a term in which both occurrences of n have number 0. The adjustments are performed by functions *inc*, *dec* and *new*. Intuitively, *inc* and *dec* add or subtract one to all level numbers in their argument terms. *new* is like *dec*, except that it is defined only on positive level numbers. For instance, the translation under $\mathcal{N}$ of $\nu n.(\nu m.n, n)$ is *new* (*new* 1), 0)).

Adjustments are also necessary when a term is substituted in another. For instance, $\lambda x.(\nu n.x, x)$ is translated to $\lambda x.(new\ (inc\ x), x)$. This takes account of the increase in the number of surrounding ν's when an argument term is substituted for x.

Note that there is an important difference between de Bruijn numbers and level numbers. Adjustment of de Bruijn numbers is a meta-operation whereas it is crucial that *inc* and *dec* are definable in the λ-calculus itself. Even if we had used a de Bruijn numbering scheme for all identifiers, level numbers and their their shift functions would still be needed.

We have to represent level numbers in such a way that they can be distinguished from other symbols in the translated terms. There are a number of ways of doing so. For instance, level numbers can be represented as pairs of a marker name, that is chosen to be distinct from the names of a mapped term M, and a Church-Numeral. (Note that this means that identity of the marker names depends on the mapped term). The precise representation of level numbers is irrelevant here; we abstract from it by denoting level numbers with numerals, and assume that we have a test function *lnum*?, successor and predecessor functions $(+1)$ and (-1), and equality *eq_lnum* on level numbers. In the following, we let the letters k and l range over level numbers.

Figure 5 gives an embedding $\mathcal{E}$ of $\lambda\nu$ in λ. The embedding is defined in terms of another syntactic mapping $\mathcal{N}$, that takes as additional arguments a level number l that indicates the current ν-nesting level, and an environment ρ that maps identifiers and names to the nesting level of their definition.

Definition 6.6 (Admissible Environments) Given a term M and a level number l, the set $Env(M, l)$ of environments that are *admissible* for M at level l is the set of all finite mappings ρ from identifiers and names to level numbers in $0..l$ that satisfy the following two requirements:

1. ρ is defined for each identifier and name free in M.
2. ρ maps different names to different level numbers.

Furthermore, we assume that each $\rho \in Env(M, l)$ is undefined on all bound identifiers and names in M. This can always be achieved by choosing an appropriate α-representative for M.

Notation. $\rho[x \mapsto a]$ (or $\rho[n \mapsto a]$) denotes the extension of environment ρ by the binding that maps x (or n) to a. To save parentheses, we assume that extension binds more strongly than function application, i.e $f\ \rho[x \mapsto a]$ parses $f\ (\rho[x \mapsto a])$.

The embedding is defined in terms of a number of auxiliary functions, which for the moment are assumed to be primitive. They are:

- A function *eq*, such that

$$
\begin{array}{lcl}
\delta(eq, c) & = & \lambda x.if\ (lnum?\ x)\ false\ (x == c) \\
\delta(eq, l) & = & \lambda x.if\ (name?\ x)\ false\ (eq\_lnum\ x\ l)
\end{array}
$$

  and such that $\delta(eq, V)$ is undefined for values that are not constants or level numbers.

$$
\begin{array}{lcl}
\mathcal{E}[\![M]\!] & = & \mathcal{N}[\![M]\!]\ 0\ \bot \\
\\
\mathcal{N}[\![x]\!]\ l\ \rho & = & inc^{(l-\rho x)}\ x \\
\mathcal{N}[\![\lambda x.M]\!]\ l\ \rho & = & \lambda x.\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l] \\
\mathcal{N}[\![M_1\ M_2]\!]\ l\ \rho & = & (\mathcal{N}[\![M_1]\!]\ l\ \rho)\ (\mathcal{N}[\![M_2]\!]\ l\ \rho) \\
\mathcal{N}[\![n^c]\!]\ l\ \rho & = & n^c \\
\mathcal{N}[\![n^\nu]\!]\ l\ \rho & = & l\text{-}\rho n^\nu \\
\mathcal{N}[\![\nu n.M]\!]\ l\ \rho & = & new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]) \\
\mathcal{N}[\![M_1 == M_2]\!]\ l\ \rho & = & eq\ (\mathcal{N}[\![M_1]\!]\ l\ \rho)\ (\mathcal{N}[\![M_2]\!]\ l\ \rho) \\
\mathcal{N}[\![(M_1, M_2)]\!]\ l\ \rho & = & (\mathcal{N}[\![M_1]\!]\ l\ \rho,\ \mathcal{N}[\![M_2]\!]\ l\ \rho) \\
\mathcal{N}[\![p\ M]\!]\ l\ \rho & = & p'\ (\mathcal{N}[\![M]\!]\ l\ \rho) \\
\mathcal{N}[\![[\,]]\!]\ l\ \rho & = & [\,]
\end{array}
$$

Figure 5: Syntactic embedding of $\lambda\nu$ in λ.

- For each primitive function in $\lambda\nu$, a primitive function p' such that $\delta(p', \mathcal{N}[\![V]\!]\ l\ \rho)$ is defined for all $l \geq 0$, $\rho \in Env(V, l)$ iff $\delta(p, V)$ is defined, and

$$
\delta(p', V) = \begin{cases}
\mathcal{N}[\![N^p_c]\!]\ 0\ \bot & \textbf{if } V \equiv c \\
\mathcal{N}[\![N^p_\nu]\!]\ 0\ \bot & \textbf{if } V \text{ is a level number} \\
\mathcal{N}[\![N^p_\lambda]\!]\ 0\ \bot & \textbf{if } V \text{ is a } \lambda\text{-abstraction} \\
(\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ 0\ \bot)\ M_1\ M_2 & \textbf{if } V \equiv (M_1, M_2)
\end{cases}
$$

- Primitive functions *inc*, *dec*, *new*, such that

$$
\begin{array}{lcll}
\delta(inc, l) & = & l{+}1 & \\
\delta(inc, c) & = & c & \\
\delta(inc, \lambda x.M) & = & \lambda x.[dec\ x/x]\ (inc\ M) & \\
\delta(inc, (M, N)) & = & (inc\ M, inc\ N) & \\
\\
\delta(dec, l) & = & l{-}1 & \\
\delta(dec, c) & = & c & \\
\delta(dec, \lambda x.M) & = & \lambda x.[inc\ x/x]\ (dec\ M) & \\
\delta(dec, (M, N)) & = & (dec\ M, dec\ N) & \\
\\
\delta(new, l) & = & l{-}1 & \textbf{if } l \neq 0 \\
\delta(new, c) & = & c & \\
\delta(new, \lambda x.M) & = & \lambda x.[inc\ x/x]\ (new\ M) & \\
\delta(new, (M, N)) & = & (new\ M, new\ N) &
\end{array}
$$

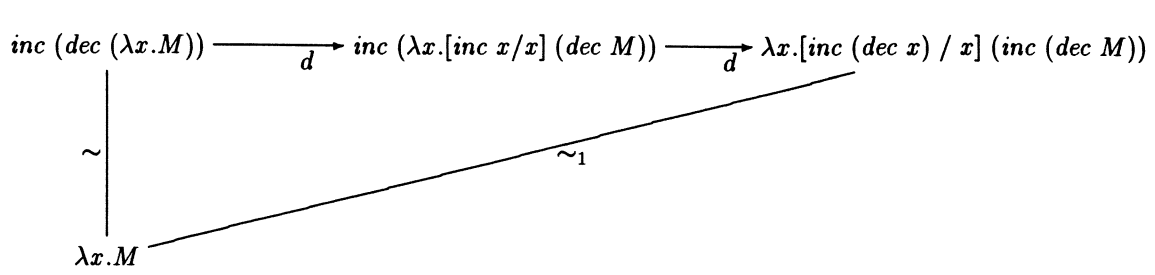
Ultimately, all these functions will be defined as λ-terms, but for now we assume that they are primitive functions. To keep the different calculi apart, we call the variant of λ with the new primitive functions λ'.

One checks easily that $\mathcal{N}$ preserves closed Λ-terms. Hence, the hardest part of showing that $\mathcal{N}$ is a syntactic embedding of $\lambda\nu$ in λ concerns the proof that $\mathcal{N}$ is semantics preserving. This proof is rather convoluted; it occupies all of the remainder of this section.

Outline of the proof of semantics preservation. The proof is in four parts. In the first part, we show that equal terms in $\lambda\nu$ are mapped (via $\mathcal{N}$) to equal terms in λ' (Lemma 6.32). In the second part, we show that $\mathcal{N}$ has a left inverse, $\mathcal{N}^{-1}$ (Lemma 6.34). We use the left inverse in the third part to show that different $\lambda\nu$ terms are mapped to different λ' terms (Lemma 6.38). In the fourth part, we use the previous results to show that $\mathcal{N}$ preserves semantics (Lemma 6.39). This is also the part where we drop the assumption that *eq*, *inc*, *dec*, *new* and the p' are primitive (Corollary 6.40).

The first part is the hardest. Its cornerstone is a syntactic characterization of the result of applying *inc* to an $\mathcal{N}$-mapped term (Lemma 6.29). It relates function *inc* with $\Uparrow_k$, a syntactic shift operator on level numbers (Lemma 6.29). Informally, $\Uparrow_k$ increments all nonnegative level numbers greater than or equal to k in a term. The detour via $\Uparrow_k$ and $\Downarrow_k$ is necessary since *new* and *dec* do not distribute through function application, but $\Downarrow_k$ does, and we need this property to carry out the structural induction in later proofs.

An important property of $\Uparrow_k$ and its inverse $\Downarrow_k$ is that both mappings are preserved under reduction. (Lemma 6.26, Lemma 6.27). The following lemmas prepare the way for these results.



(5)

First, some facts about *inc*, *dec* and *new*:

Lemma 6.7 For all $M \in \Lambda'$,

$$(i) \qquad inc\ (dec\ M) \cong M$$
$$(ii) \qquad dec\ (inc\ M) \cong M$$

Proof: (i) We use the critical pairs technique described in Section 5. Let

$$\mathcal{S} = \{inc\ (dec\ a) \sim a\}$$

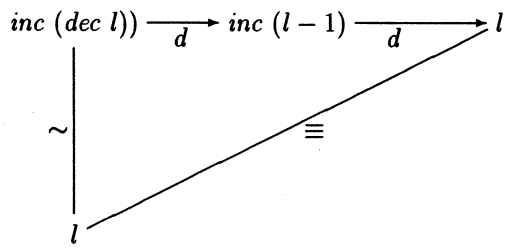
for some meta-variable a and let $\sim$ be the symmetric compatible valuation closure of $\mathcal{S}$. We use Theorem 5.5 to show that $\sim \subseteq \cong$.

It was shown already in [15] that $\lambda\nu$ has downwards closed evaluation contexts that are preserved by every similarity relation. Hence, it remains to be shown that $\sim$ preserves answers and that it is deterministically locally stable.

If $M \sim A$, then, by the definition of $\mathcal{S}$, M must be of the form $inc\ (dec\ A)$. This term reduces to A, hence $M = A$. This shows that $\sim$ is answer preserving. We now show that $\sim$ is deterministically locally stable.

Since the RHS of $\mathcal{S}$ consists a single meta-variable, it cannot interfere with a redex. The left hand side can interfere in exactly one way, namely $\Delta \equiv dec\ M$, for some term M. Furthermore, since Δ is a redex, M must be a λ-abstraction, a pair, a name, or a level number. We distinguish the four cases. In the case of a $\lambda-$abstraction, diagram (5) commutes.

In the case of a level number, the following diagram commutes.



(6)

The other two cases are similar. This shows that $\sim$ is deterministically locally stable. Thus, by Theorem 5.5, $\sim \subseteq \cong$.

The proof of (ii) is analogous. ■

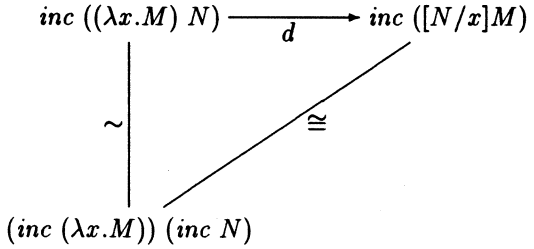
Lemma 6.8 For all M, N in Λ',

$$inc\ (M\ N) \cong (inc\ M)\ (inc\ N)$$

Proof: Let

$$\mathcal{S} = \{inc(a\ b) \sim (inc\ a)\ (inc\ b)\}$$

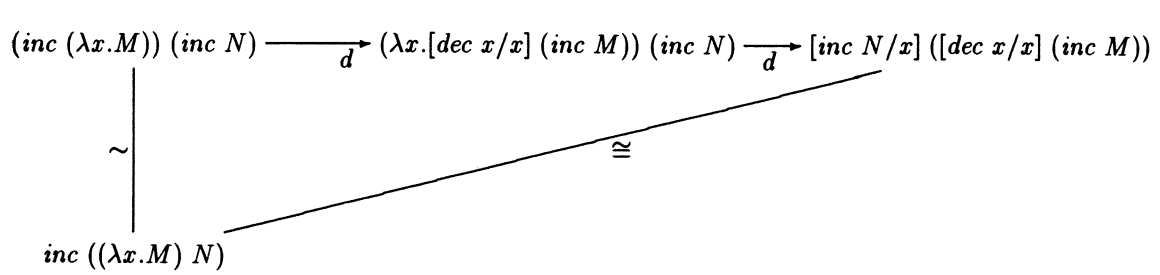
and let $\sim$ be the symmetric compatible valuation closure of $\mathcal{S}$. We use Theorem 5.5 to show that $\sim \subseteq \cong$. Clearly, no side of $\mathcal{S}$ matches an answer, hence $\sim$ vacuously preserves answers. We now show that $\sim$ is deterministically locally stable. Assume first that a Δ interferes with the LHS of $\mathcal{S}$. Then Δ must be of the form $(\lambda x.M)\ N$, for some terms M, N. In this case the following diagram commutes.



(7)

The $\cong$ diagonal follows from the following argument.

$$\begin{array}{ll} & (inc\ (\lambda x.M))\ (inc\ N) \\ = & \text{(by definition of } inc\text{)} \\ & (\lambda x.[dec\ x/x]\ (inc\ M))\ (inc\ N) \\ = & \text{(by } \beta\text{-reduction)} \\ & [inc\ N/x]\ ([dec\ x/x]\ (inc\ M)) \\ \equiv & \text{(by combining the substitutions)} \\ & [dec\ (inc\ N)\ /\ x]\ (inc\ M) \\ \cong & \text{(by Lemma 6.7)} \\ & [N/x]\ (inc\ M) \end{array}$$



(8)

Assume now that a redex Δ interferes deterministically with the right hand side of $\mathcal{S}$. Then Δ must be of the form *inc* L, for some term L that is a λ-abstraction, a pair, a name, or a level number. Furthermore, since Δ is by assumption head-redex, it must match the function part (*inc* a) of the RHS of $\mathcal{S}$. We distinguish according to the form of L.

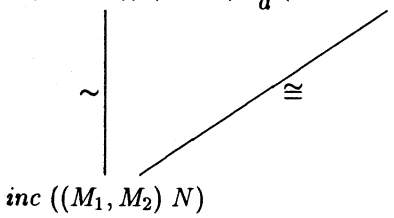
If L is a λ–abstraction $\lambda x.M$, diagram (8) commutes.

The $\cong$ diagonal in (8) holds, since

$$\begin{array}{ll} & inc\ ((\lambda x.M)\ N) \\ = & \text{(by } \beta\text{-reduction)} \\ & inc\ ([N/x]\ M) \\ \cong & \text{(by Lemma 6.7)} \\ & [inc\ N/x]\ ([dec\ x/x]\ (inc\ M)) \end{array}$$

If L is a pair (M_1, M_2), the following diagram commutes.

(9)



The $\cong$ diagonal holds by a further application of Theorem 5.4 since both sides of $\cong$ are "stuck", i.e. they do not interfere with any redex.

The cases where L is a name or a level number are analogous to the last case. The preceding discussion showed that $\sim$ is weakly locally stable. By Theorem 5.5, $\sim \subseteq \cong$. ■

The proofs of the following three lemmas are omitted. They are like the last two proofs routine applications of Theorem 5.5.

Lemma 6.9 For all primitive operators p', terms $M \in \Lambda'$,

$$inc\ (p'\ M) \ \cong\ p'\ (inc\ M)$$

Lemma 6.10 For all $M, N \in \Lambda'$,

$$inc\ (eq\ M\ N) \ \cong\ eq\ (inc\ M)\ (inc\ N)$$

Lemma 6.11 For all terms $M \in \Lambda'$,

$$new\ M\ =\ dec\ (neq\ 0\ M),$$

where

$$\begin{array}{lcll} neq\ l\ k & = & k & \text{if } l \neq k \\ neq\ l\ c & = & c & \\ neq\ l\ (\lambda x.M) & = & \lambda x.neq\ l\ M & \\ neq\ l\ (M,N) & = & (neq\ l\ M, neq\ l\ N) & \end{array}$$

Next, some facts about $\mathcal{N}$ that are easily shown by an induction on the structure of the the term argument of $\mathcal{N}$.

Lemma 6.12 For all terms $M \in \Lambda\nu$, level numbers $0 \leq k \leq l$, environments $\rho \in Env(M, l)$,

(i) If x is not free in M,

$$\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto k] \equiv \mathcal{N}[\![M]\!]\ l\ \rho.$$

(ii) If n is not free in M,

$$\mathcal{N}[\![M]\!]\ l\ \rho[n \mapsto k] \equiv \mathcal{N}[\![M]\!]\ l\ \rho.$$

Lemma 6.13 For all closed terms $M \in \Lambda\nu$, level numbers l, environments $\rho \in Env(M, l)$,

$$\mathcal{N}[\![M]\!]\ l\ \rho \ \equiv\ \mathcal{N}[\![M]\!]\ 0\ \bot.$$

Lemma 6.14 For all $M \in \Lambda\nu$, level numbers $0 \leq k < l$, environments $\rho \in Env(M, l)$,

$$[inc\ x/x]\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto k{+}1]) \ \equiv\ \mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto k]$$

Proof: By an induction on the structure of M. The interesting base case for (i) is:

$$
\begin{array}{ll}
& [inc\ x/x]\ (\mathcal{N}[\![x]\!]\ l\ \rho[x \mapsto k{+}1]) \\
\equiv & \text{(by definition of } \mathcal{N}\text{, substitution)} \\
& inc^{l-(k+1)}\ (inc\ x) \\
\equiv & \text{(by definition of } \mathcal{N}\text{)} \\
& \mathcal{N}[\![x]\!]\ l\ \rho[x \mapsto k]
\end{array}
$$

The corresponding case for (ii) is analogous. The other base cases and induction steps are all routine. ■

We now work towards giving a syntactic characterization of *inc* ($\mathcal{N}[\![M]\!]\ l\ \rho$) that relates *inc* with the syntactic operator $\Uparrow_k$. $\Uparrow_k$ and its inverse $\Downarrow_k$ are defined in Figures 6 and 7 (Note that $\Downarrow_k$ is a partial function). Before we come to the correspondence result of (Lemma 6.29), we need to prove some facts about these shift operators.

Lemma 6.15 $\Downarrow_k$ and $\Uparrow_k$ are inverses. For all $M \in \Lambda'$, level numbers k,

$$\Downarrow_k\ (\Uparrow_k\ M)\ \equiv\ M,$$

and for all $M \in \Lambda'$, level numbers k, such that $\Downarrow_k\ M$ is defined,

$$\Uparrow_k\ (\Downarrow_k\ M)\ \equiv\ M$$

Proof: An easy induction on the structure of M. ■

Lemma 6.16 $\Uparrow\_$ and $\Downarrow\_$ are compositional: For all contexts C, level numbers k, there exists an level number k' such that, for arbitrary $x \notin FV\ (C[M])$,

$$
\begin{array}{rcl}
\Uparrow_k\ (C[M]) & \equiv & [\Uparrow_{k'} M/x]\ (\Uparrow_k\ (C[x])) \\
\Downarrow_k\ (C[M]) & \equiv & [\Downarrow_{k'} M/x]\ (\Downarrow_k\ (C[x]))
\end{array}
$$

Proof: An easy structural induction on C. ■

Lemma 6.17 For all level numbers k, l_1, l_2,

$$l_1 < l_2\ \Rightarrow\ \Uparrow_k\ l_1 < \Uparrow_k\ l_2.$$

If $l_1 \neq k$ and $l_2 \neq k$ then also

$$l_1 < l_2\ \Rightarrow\ \Downarrow_k\ l_1 < \Downarrow_k\ l_2.$$

Proof: Easy. ■

Lemma 6.18 For all $M \in \Lambda'$, level numbers k, identifiers x,

$$
\begin{array}{rcl}
\Uparrow_k\ ([inc\ x/x]\ M) & \equiv & [inc\ x/x]\ (\Uparrow_k\ M) \\
\Uparrow_k\ ([dec\ x/x]\ M) & \equiv & [dec\ x/x]\ (\Uparrow_k\ M).
\end{array}
$$

Furthermore, provided either side is defined,

$$
\begin{array}{rcl}
\Downarrow_k\ ([inc\ x/x]\ M) & \equiv & [inc\ x/x]\ (\Downarrow_k\ M) \\
\Downarrow_k\ ([dec\ x/x]\ M) & \equiv & [dec\ x/x]\ (\Downarrow_k\ M),
\end{array}
$$

Proof: An easy induction on the structure of M. ■

Definition 6.19 The *dec-excess* of a subterm N of a Λ'-term M is the number of *dec* and *new* terms occurring on the path between N and the root of M minus the number of *inc* terms occurring on that path.

Definition 6.20 A term M is *balanced* if every occurrence of a bound identifier in M has the same *dec*-excess as its binding λ-term.

Lemma 6.21 For all terms $M \in \Lambda\nu$, level numbers l, environments $\rho \in Env(M, l)$: $\mathcal{N}[\![M]\!]\ l\ \rho$ is balanced.

Proof: Follows directly from the definition of $\mathcal{N}$. ■

Lemma 6.22 For all terms $M, N \in \Lambda'$, if M is balanced and $M \rightarrow N$ then N is balanced.

Proof: A straightforward analysis of the different reductions in λ'. ■

Lemma 6.23 For all terms $M \in \Lambda\nu$, level numbers k: If M is closed then

$$\Uparrow_k\ (\mathcal{N}[\![M]\!]\ 0\ \bot)\ \equiv\ \mathcal{N}[\![M]\!]\ 0\ \bot\ \equiv \Downarrow_k\ (\mathcal{N}[\![M]\!]\ 0\ \bot)$$

Proof: An easy induction on the structure of M. ■

Lemma 6.24 ($\Uparrow_k$-Substitution) For all terms $M, N \in \Lambda'$, level numbers k, if $\lambda x.M$ is balanced then

$$[\Uparrow_k\ N/x]\ (\Uparrow_k\ M)\ \equiv \Uparrow_k\ ([N/x]M).$$

Proof: Let l be a level number and let C be an n-hole context such that

$$M \equiv C[x, ..., x], \tag{10}$$

and x is not free in C. Since $\lambda x.M$ is balanced, the *dec*-excess of every occurrence of x in M is 0. This means that there is an equal number of *inc* and *dec*/*new* nodes on every path from the root of C to a hole. Therefore, by the definition of $\Uparrow_k$, there is a n-hole context C' such that for all terms $N_1, ..., N_n$

$$\Uparrow_k\ (C[N_1, ..., N_n])\ \equiv\ C'[\Uparrow_k\ N_1, ..., \Uparrow_k\ N_n]. \tag{11}$$

$$
\begin{array}{lcll}
\Uparrow_k l & \equiv & l{+}1 & \textbf{if } k \leq l \\
 & \equiv & l & \textbf{if } k > l \\
\Uparrow_k x & \equiv & x & \\
\Uparrow_k c & \equiv & c & \\
\Uparrow_k (\lambda x.M) & \equiv & \lambda x.\, \Uparrow_k M & \\
\Uparrow_k (MN) & \equiv & (\Uparrow_k M)\, (\Uparrow_k N) & \\
\Uparrow_k (p'\, M) & \equiv & p'\, (\Uparrow_k M) & \\
\Uparrow_k (eq\ M\ N) & \equiv & eq\ (\Uparrow_k M)\ (\Uparrow_k N) & \\
\Uparrow_k (dec\ M) & \equiv & dec\ (\Uparrow_{k+1} M) & \\
\Uparrow_k (inc\ M) & \equiv & inc\ (\Uparrow_{k-1} M) & \\
\Uparrow_k (new\ M) & \equiv & \Uparrow_k (dec\ (neq\ 0\ M)) &
\end{array}
$$

Figure 6: Up-shift operator $\Uparrow_k$.

Hence,

$$
\begin{array}{ll}
 & [\Uparrow_k N/x]\ (\Uparrow_k M) \\
\equiv & \text{(by (10))} \\
 & [\Uparrow_k N/x]\ (\Uparrow_k C[x,...,x]) \\
\equiv & \text{(by (11))} \\
 & [\Uparrow_k N/x]\ (C'[\Uparrow_k x,...,\Uparrow_k x]) \\
\equiv & \text{(by the definition of } \Uparrow_k) \\
 & [\Uparrow_k N/x]\ (C'[x,...,x]) \\
\equiv & \text{(by substitution)} \\
 & C'[\Uparrow_k N,...,\Uparrow_k N] \\
\equiv & \text{(by (11))} \\
 & \Uparrow_k (C[N,...,N]) \\
\equiv & \text{(by (10))} \\
 & \Uparrow_k ([N/x]M).
\end{array}
$$

■

Lemma 6.25 ($\Downarrow_k$-Substitution) For all terms $M, N \in \Lambda'$, level numbers k, if $\lambda x.M$ is balanced and $\Downarrow_k N$ and $\Downarrow_k M$ are both defined then

$$[\Downarrow_k N/x]\ (\Downarrow_k M) \equiv \Downarrow_k ([N/x]M).$$

Proof: Analogous to the proof of Lemma 6.24 ■

Lemma 6.26 For all terms $M, N \in \Lambda'$, level numbers k,

$$M \to N \ \Rightarrow\ \Uparrow_k M \to \Uparrow_k N$$

Proof: We distinguish according to the form of the redex Δ in $M \to N$. Assume first that Δ coincides with M.

Case $(\lambda x.M)\ N$ In this case:

$$
\begin{array}{ll}
 & \Uparrow_k ((\lambda x.M)\ N) \\
\equiv & (\Uparrow_k \lambda x.M)\ (\Uparrow_k N) \\
\equiv & (\lambda x.\, \Uparrow_k M)\ (\Uparrow_k N) \\
 & [\Uparrow_k N/x]\ (\Uparrow_k M) \\
\equiv & \text{(by Lemma 6.24)} \\
 & \Uparrow_k ([N/x]M)
\end{array}
$$

Case $inc\ l$ If $l < k{-}1$, we have:

$$
\begin{array}{ll}
 & \Uparrow_k (inc\ l) \\
\equiv & inc\ (\Uparrow_{k-1} l) \\
\equiv & inc\ l \\
\to & l{+}1 \\
\equiv & \Uparrow_k (l{+}1)
\end{array}
$$

On the other hand, if $l \geq k{-}1$, we have:

$$
\begin{array}{ll}
 & \Uparrow_k (inc\ l) \\
\equiv & inc\ (\Uparrow_{k-1} l) \\
\equiv & inc\ (l{+}1) \\
\to & l{+}2 \\
\equiv & \Uparrow_k (l{+}1)
\end{array}
$$

$$
\begin{array}{lcll}
\Downarrow_k l & \equiv & l\text{–}1 & \textbf{if } k < l \\
 & \equiv & l & \textbf{if } k > l \\
\Downarrow_k x & \equiv & x & \\
\Downarrow_k c & \equiv & c & \\
\Downarrow_k (\lambda x.M) & \equiv & \lambda x.\, \Downarrow_k M & \\
\Downarrow_k (MN) & \equiv & (\Downarrow_k M)\ (\Downarrow_k N) & \\
\Downarrow_k (p'\ M) & \equiv & p'\ (\Downarrow_k M) & \\
\Downarrow_k (eq\ M\ N) & \equiv & eq\ (\Downarrow_k M)\ (\Downarrow_k N) & \\
\Downarrow_k (dec\ M) & \equiv & dec\ (\Downarrow_{k+1} M) & \\
\Downarrow_k (inc\ M) & \equiv & inc\ (\Downarrow_{k-1} M) & \\
\Downarrow_k (new\ M) & \equiv & \Downarrow_k (dec\ (neq\ 0\ M)) &
\end{array}
$$

Figure 7: Down-shift operator $\Downarrow_k$.

Case $inc\ (\lambda x.M)$ In this case we have:

$$
\begin{array}{ll}
 & \Uparrow_k (inc\ (\lambda x.M)) \\
\equiv & inc\ (\lambda x.\ \Uparrow_{k-1} M) \\
\rightarrow & \text{(by definition of } inc\text{)} \\
 & \lambda x.inc\ ([dec\ x/x]\ (\Uparrow_{k-1} M)) \\
\equiv & \text{(by Lemma 6.18)} \\
 & \lambda x.inc\ (\Uparrow_{k-1} ([dec\ x/x]\ M)) \\
\equiv & \Uparrow_k (\lambda x.inc\ ([dec\ x/x]\ M))
\end{array}
$$

Case $inc\ c$ Case $inc\ (M,N)$ are both easy.

Case $dec\ l$ Case $dec\ (\lambda x.M)$ Case $dec\ c$

Case $dec\ (M,N)$ are analogous to the last four cases.

Case $eq\ l_1\ l_2$ If $l_1 = l_2$ then

$$
\begin{array}{ll}
 & \Uparrow_k (eq\ l_1\ l_2) \\
\equiv & eq\ (\Uparrow_k l_1)\ (\Uparrow_k l_2) \\
\rightarrow & \text{(by definition of } eq\text{)} \\
 & true \\
\equiv & \Uparrow_k true
\end{array}
$$

If $l_1 \neq l_2$ then, with Lemma 6.17, $\Uparrow_k l_1 \neq \Uparrow_k l_2$, and we have:

$$
\begin{array}{ll}
 & \Uparrow_k (eq\ l_1\ l_2) \\
\equiv & eq\ (\Uparrow_k l_1)\ (\Uparrow_k l_2) \\
\rightarrow & \text{(by definition of } eq\text{)} \\
 & false \\
\equiv & \Uparrow_k false
\end{array}
$$

Case $p'\ M$ In this case we further distinguish according to the form of M. If M is a level number l, then $\Uparrow_k l$ is some level number l' and we have:

$$
\begin{array}{ll}
 & \Uparrow_k (p'\ l) \\
\equiv & p'(\Uparrow_k l) \\
\equiv & p'\ l' \\
\rightarrow & \text{(by definition of } p'\text{)} \\
 & \mathcal{N}[\![N^p_\nu]\!]\ 0\ \bot \\
\equiv & \text{(by Lemma 6.23)} \\
 & \Uparrow_k (\mathcal{N}[\![N^p_\nu]\!]\ 0\ \bot)
\end{array}
$$

if M is a pair (M_1, M_2), we have

$$
\begin{array}{ll}
 & \Uparrow_k (p'\ (M_1, M_2)) \\
\equiv & p'(\Uparrow_k M_1, \Uparrow_k M_2) \\
\rightarrow & \text{(by definition of } p'\text{)} \\
 & (\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ 0\ \bot)\ (\Uparrow_k M_1)\ (\Uparrow_k M_2) \\
\equiv & \text{(by Lemma 6.23)} \\
 & \Uparrow_k (\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ 0\ \bot)\ (\Uparrow_k M_1)\ (\Uparrow_k M_2) \\
\equiv & \Uparrow_k ((\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ 0\ \bot)\ M_1\ M_2)
\end{array}
$$

The remaining two cases, where M is a λ-abstraction or a constant, are similar.

This concludes the analysis for the case where $\Delta \equiv M$. We now show the lemma for the general case that Δ is contained in M. Let C and R be such that

$$M \equiv C[\Delta] \xrightarrow{\Delta} C[R] \equiv N$$

By Lemma 6.16, there is an level number k' such that for all terms L, $\Uparrow_k (C[L]) \equiv [\Uparrow_{k'} L/x] (C[x])$ By the first part of the proof, $\Uparrow_{k'} \Delta \to \Uparrow_{k'} R$. Hence,

$$\begin{array}{rl} & \Uparrow_k M \equiv \Uparrow_k (C[\Delta]) \equiv [\Uparrow_{k'} \Delta/x] (C[x]) \\ \to & [\Uparrow_{k'} R/x] (C[x]) \equiv \Uparrow_k (C[R]) \equiv \Uparrow_k N. \end{array}$$

■

The reverse of Lemma 6.26 also holds:

Lemma 6.27 For all terms $M, N \in \Lambda'$, level numbers k, if $M \to N$ and $\Downarrow_k M$ is defined, then so is $\Downarrow_k N$ and

$$\Downarrow_k M \to \Downarrow_k N.$$

Proof: Largely analogous to the proof of Lemma 6.26. We distinguish according to the form of the redex in $M \to N$. The only three cases that are not completely symmetric to cases in the proof of Lemma 6.26 are.

Case $inc\ l$ We distinguish two subcases: $l < k{-}1$ and $l > k{-}1$. ($l = k{-}1$ is impossible, since in this case neither $\Downarrow_k (inc\ l)$ nor $\Downarrow_k (l{+}1)$ are defined). If $l < k{-}1$, we have:

$$\begin{array}{rl} & \Downarrow_k (inc\ l) \\ \equiv & inc\ (\Downarrow_{k-1} l) \\ \equiv & inc\ l \\ \to & l{+}1 \\ \equiv & \Downarrow_k (l{+}1). \end{array}$$

On the other hand, if $l > k{-}1$, we have:

$$\begin{array}{rl} & \Downarrow_k (inc\ l) \\ \equiv & inc\ (\Downarrow_{k-1} l) \\ \equiv & inc\ (l{-}1) \\ \to & l \\ \equiv & \Downarrow_k (l{+}1). \end{array}$$

Case $dec\ l$ We distinguish again two subcases: $l < k{+}1$, and $l > k{+}1$. (Case $l = k{+}1$ is impossible). If $l < k{+}1$, we have:

$$\begin{array}{rl} & \Downarrow_k (dec\ l) \\ \equiv & dec\ (\Downarrow_{k+1} l) \\ \equiv & dec\ l \\ \to & l{-}1 \\ \equiv & \Downarrow_k (l{-}1). \end{array}$$

On the other hand, if $l > k{+}1$, we have:

$$\begin{array}{rl} & \Downarrow_k (dec\ l) \\ \equiv & dec\ (\Downarrow_{k+1} l) \\ \equiv & dec\ (l{-}1) \\ \to & l{-}2 \\ \equiv & \Downarrow_k (l{-}1). \end{array}$$

Case $eq\ l_1\ l_2$ Since $\Downarrow_k (eq\ l_1\ l_2)$ is defined, and

$$\Downarrow_k (eq\ l_1\ l_2) \equiv eq\ (\Downarrow_k l_1)\ (\Downarrow_k l_2),$$

$\Downarrow_k l_1$ and $\Downarrow_k l_2$ are both defined, which implies $l_1 \neq k$, $l_2 \neq k$. With Lemma 6.17, it follows that

$$l_1 = l_2 \Leftrightarrow \Downarrow_k l_1 = \Downarrow_k l_2.$$

The case then follows as in Lemma 6.26. ■

We now use the fact that $\Uparrow_k$ and $\Downarrow_k$ are stable under reduction for a stepping stone towards Lemma 6.29.

Lemma 6.28 For all terms M,

$$neq\ 0\ (\Uparrow_1 M) \cong neq\ 1\ (\Uparrow_0 M). \qquad (12)$$

Proof: We use Theorem 5.5. (12) corresponds to the symmetric system of similarities

$$\begin{array}{rcl} \mathcal{S} & \stackrel{\text{def}}{=} & \{\ neq\ i\ (\Uparrow_j M) \sim neq\ j\ (\Uparrow_i M) \\ & & \mid M \in \Lambda', i, j \in \{0,1\}, i \neq j \\ & & \} \end{array}$$

Let $\sim$ be the compatible valuation closure of $\mathcal{S}$. We show that $\sim$ is deterministically locally stable. Let (P, Q) be a deterministically critical pair. Then (P, Q) falls into one of the categories (1.)–(5.) below. We show in each case that the critical pair can be completed.

1. The redex is contained in $\Uparrow_i M$, i.e.

$$\begin{array}{rcl} P & \equiv & neq\ i\ (\Uparrow_j M) \sim neq\ j\ (\Uparrow_i M) \xrightarrow[d]{} neq\ j\ N \\ & \equiv & Q \end{array}$$

where $M, N \in \Lambda'$, $\Uparrow_i M \xrightarrow[d]{} N$, $i, j \in \{0,1\}, i \neq j$.

In this case,

$$\begin{array}{ll} & P \\ \equiv & neq\ i\ (\Uparrow_j\ M) \\ \equiv & \text{(by Lemma 6.15)} \\ & neq\ i\ (\Uparrow_j\ (\Downarrow_i\ (\Uparrow_i\ M))) \\ \rightarrow & \text{(by Lemma 6.27, Lemma 6.26)} \\ & neq\ i\ (\Uparrow_j\ (\Downarrow_i\ N)) \\ \sim & \text{(by definition of } \sim) \\ & neq\ j\ (\Uparrow_i\ (\Downarrow_i\ N)) \\ \equiv & \text{(by Lemma 6.15)} \\ & neq\ j\ N \\ \equiv & Q \end{array}$$

Therefore, the following diagram commutes.

$$\begin{array}{ccc} neq\ j\ (\Uparrow_i\ M) & \xrightarrow[d]{} & neq\ j\ N \equiv Q \\ \sim \big| & & \big| \sim \\ P \equiv neq\ i\ (\Uparrow_j\ M) & \longrightarrow & neq\ i\ (\Uparrow_j\ (\Downarrow_i\ N)) \end{array}$$

2. The redex involves *neq* and a λ-abstraction, i.e.

$$\begin{array}{lll} P & \equiv & neq\ i\ (\Uparrow_j\ \lambda x.M)\ \sim\ neq\ j\ (\Uparrow_i\ \lambda x.M) \\ & & \xrightarrow[d]{}\ \lambda x.neq\ j\ (\Uparrow_i\ M) \\ & \equiv & Q \end{array}$$

where $M \in \Lambda'$, $i,j \in \{0,1\}, i \neq j$. In this case,

$$\begin{array}{ll} & P \\ \equiv & neq\ i\ (\Uparrow_j\ \lambda x.M) \\ \equiv & neq\ i\ (\lambda x.\ \Uparrow_j\ M) \\ \rightarrow & \lambda x.neq\ i\ (\Uparrow_j\ M) \\ \sim & \lambda x.neq\ j\ (\Uparrow_i\ M) \\ \equiv & Q \end{array}$$

Therefore, the following diagram commutes.

$$\begin{array}{ccc} neq\ j\ (\Uparrow_i\ \lambda x.M) & \xrightarrow[d]{} & \lambda x.neq\ j\ (\Uparrow_i\ M) \equiv Q \\ \sim \big| & & \big| \sim \\ P \equiv neq\ i\ (\Uparrow_j\ \lambda x.M) & \longrightarrow & \lambda x.neq\ i\ (\Uparrow_j\ M) \end{array}$$

3. The redex involves *neq* and a pair, i.e.

$$\begin{array}{lll} P & \equiv & neq\ i\ (\Uparrow_j\ (M,N))\ \sim\ neq\ j\ (\Uparrow_i\ (M,N)) \\ & & \xrightarrow[d]{}\ (neq\ j\ (\Uparrow_i\ M), neq\ j\ (\Uparrow_i\ N)) \\ & \equiv & Q \end{array}$$

where $M,N \in \Lambda'$, $i,j \in \{0,1\}, i \neq j$. This is essentially analogous to the last case.

4. The redex involves *neq* and a constant, i.e.

$$P \equiv neq\ i\ (\Uparrow_j\ c)\ \sim\ neq\ j\ (\Uparrow_i\ c)\ \xrightarrow[d]{}\ c \equiv Q$$

where c is a constant, $i,j \in \{0,1\}, i \neq j$. Then $P \equiv neq\ i\ (\Uparrow_j\ c) \rightarrow c$ which completes the critical pair.

5. The redex involves *neq* and a level number, i.e.

$$P \equiv neq\ i\ (\Uparrow_j\ l)\ \sim\ neq\ j\ (\Uparrow_i\ l)\ \xrightarrow[d]{}\ \Uparrow_i\ l \equiv Q$$

where l is a level number, $i,j \in \{0,1\}, i \neq j$. We distinguish the two subcases $i = 0, j = 1$ and $i = 1, j = 0$. If $i = 0, j = 1$ then the root of the critical pair is $neq\ 1\ (\Uparrow_0\ l)$. Since this is also a redex we must have $\Uparrow_0\ l \neq 1$, and therefore $l \neq 0$. But this implies $\Uparrow_1\ l \equiv \Uparrow_0\ l \neq 0$. Therefore, $neq\ 0\ (\Uparrow_1\ l) \rightarrow \Uparrow_1\ l$. In summary, the following diagram commutes:

$$\begin{array}{ccc} neq\ 1\ (\Uparrow_0\ l) & \xrightarrow[d]{} & \Uparrow_0\ l \equiv Q \\ \sim \big| & & \big| \equiv \\ P \equiv neq\ 0\ (\Uparrow_1\ l) & \longrightarrow & \Uparrow_1\ l \end{array}$$

The subcase where $i = 1$ and $j = 0$ is completely symmetric.

(1.)–(5.) together show that $\mathcal{S}$ is locally stable. Furthermore, since neither side of this system matches an answer, $\sim$ vacuously preserves answers. By Theorem 5.5, $\sim \subseteq \cong$. ■

Lemma 6.29 Let σ be the syntactic mapping that substitutes $inc\ x$ for x in M, for every free identifier x in M. That is, for all terms M,

$$\sigma\ M\ =\ [inc\ x/x]_{x \in FV(M)}\ M.$$

Then for all $M \in \Lambda\nu$, level numbers l, $\rho \in Env(M,l)$,

$$inc\ (\mathcal{N}[\![M]\!]\ l\ \rho)\ \cong\ \sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho))$$

Proof: By induction on the structure of M.

Case c In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![c]\!]\ l\ \rho) \\ \equiv & inc\ c \\ = & c \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![c]\!]\ l\ \rho) \end{array}$$

Case n^ν In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![n^\nu]\!]\ l\ \rho) \\ \equiv & inc\ (l{-}\rho n^\nu) \\ = & \text{(by definition of } inc\text{)} \\ & l{-}\rho n^\nu{+}1 \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![n^\nu]\!]\ l\ \rho) \end{array}$$

Case x In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![x]\!]\ l\ \rho) \\ \equiv & inc\ (inc^{l-\rho x}\ x) \\ \equiv & inc^{l-\rho x}(inc\ x) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![x]\!]\ l\ \rho)) \end{array}$$

Case $\lambda x.M$ In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![\lambda x.M]\!]\ l\ \rho) \\ \equiv & inc\ (\lambda x.\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l]) \\ = & \text{(by definition of } inc\text{)} \\ & \lambda x.[dec\ x/x]\ (inc\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l])) \\ = & \text{(by the induction hypothesis)} \\ & \lambda x.[dec\ x/x]\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l]))) \\ \equiv & \sigma\ (\lambda x.[dec\ x/x]\ ([inc\ x/x] \\ & \qquad (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l])))) \\ \cong & \text{(by Lemma 6.7)} \\ & \sigma\ (\lambda x.\ \Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l])) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![\lambda x.M]\!]\ l\ \rho)) \end{array}$$

Case $\nu n.M$ In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![\nu n.M]\!]\ l\ \rho) \\ \equiv & inc\ (new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])) \\ = & \text{(by Lemma 6.11)} \\ & inc\ (dec\ (neq\ 0\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]))) \\ \cong & \text{(by Lemma 6.7)} \\ & dec\ (inc\ (neq\ 0\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]))) \\ \cong & \text{(by Lemma 6.8)} \\ & dec\ (neq\ 1\ (inc\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]))) \\ \cong & \text{(by the induction hypothesis)} \\ & dec\ (neq\ 1\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])))) \\ \equiv & \text{(by rearrangement)} \\ & \sigma\ (dec\ (neq\ 1\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])))) \\ \cong & \text{(by Lemma 6.28)} \\ & \sigma\ (dec\ (neq\ 0\ (\Uparrow_1\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])))) \\ \equiv & \text{(by definition of } \Uparrow_k\text{)} \\ & \sigma\ (\Uparrow_0\ (dec\ (neq\ 0\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])))) \\ \cong & \text{(by Lemma 6.11, Lemma 6.26)} \\ & \sigma\ (\Uparrow_0\ (new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]))) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![\nu n.M]\!]\ l\ \rho)) \end{array}$$

Case MN In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![MN]\!]\ l\ \rho) \\ \cong & \text{(by Lemma 6.8)} \\ & (inc\ (\mathcal{N}[\![M]\!]\ l\ \rho))\ (inc\ (\mathcal{N}[\![N]\!]\ l\ \rho)) \\ \cong & \text{(by the induction hypothesis)} \\ & (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho)))\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![N]\!]\ l\ \rho))) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![MN]\!]\ l\ \rho)) \end{array}$$

Case $M == N$ In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![M == N]\!]\ l\ \rho) \\ \equiv & inc\ (eq\ (\mathcal{N}[\![M]\!]\ l\ \rho)\ (\mathcal{N}[\![N]\!]\ l\ \rho)) \\ \cong & \text{(by Lemma 6.10)} \\ & eq\ (inc\ (\mathcal{N}[\![M]\!]\ l\ \rho))\ (inc\ (\mathcal{N}[\![N]\!]\ l\ \rho)) \\ \cong & \text{(by the induction hypothesis)} \\ & eq\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho)))\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![N]\!]\ l\ \rho))) \\ \equiv & \sigma\ (\Uparrow_0\ (eq\ (\mathcal{N}[\![M]\!]\ l\ \rho)\ (\mathcal{N}[\![N]\!]\ l\ \rho))) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![M == N]\!]\ l\ \rho)) \end{array}$$

Case $p\ M$ In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![p\ M]\!]\ l\ \rho) \\ \equiv & inc\ (p'\ (\mathcal{N}[\![M]\!]\ l\ \rho)) \\ \cong & \text{(by Lemma 6.9)} \\ & p'\ (inc\ (\mathcal{N}[\![M]\!]\ l\ \rho)) \\ \cong & \text{(by the induction hypothesis)} \\ & p'\ (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho))) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![p'\ M]\!]\ l\ \rho)) \end{array}$$

Case (M, N) In this case:

$$\begin{array}{ll} & inc\ (\mathcal{N}[\![(M,N)]\!]\ l\ \rho) \\ = & \text{(by definition of } inc\text{)} \\ & (inc\ (\mathcal{N}[\![M]\!]\ l\ \rho), inc\ (\mathcal{N}[\![N]\!]\ l\ \rho)) \\ \cong & \text{(by the induction hypothesis)} \\ & (\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho)), \sigma\ (\Uparrow_0\ (\mathcal{N}[\![N]\!]\ l\ \rho))) \\ \equiv & \sigma\ (\Uparrow_0\ (\mathcal{N}[\![(M,N)]\!]\ l\ \rho)) \end{array}$$

■

Using this characterization of inc in terms of $\Uparrow_k$ gives us

Lemma 6.30 For all $M \in \Lambda\nu$, level numbers l, $\rho \in Env(M, l)$,

$$inc\ (\mathcal{N}[\![M]\!]\ l\ \rho) \cong \mathcal{N}[\![M]\!]\ (l{+}1)\ \rho.$$

Proof: By Lemma 6.29,

$$inc\ (\mathcal{N}[\![M]\!]\ l\ \rho) \cong \sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho). \qquad (13)$$

An easy induction on the structure of M shows that

$$\sigma\ (\Uparrow_0\ (\mathcal{N}[\![M]\!]\ l\ \rho) \cong \mathcal{N}[\![M]\!]\ (l{+}1)\ \rho \qquad (14)$$

(13) and (14) together imply the lemma. ■

The next result states that $\mathcal{N}$ commutes (wrt $\cong$) with substitution.

Lemma 6.31 (Substitution) For all $M, N \in \Lambda\nu$, level numbers l, $\rho \in Env((\lambda x.M)\ N, l)$,

$$[\mathcal{N}[\![N]\!]\ l\ \rho\ /\ x]\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l]) \cong \mathcal{N}[\![[N/x]\ M]\!]\ l\ \rho.$$

Proof: We use a structural induction on M. A direct proof of the equivalence fails for the case $M \equiv \nu n.M'$. We prove instead a stronger proposition:

For all $M, N \in \Lambda\nu$, level numbers l, k, environments $\rho' \in Env(N, l)$ and $\rho \in Env(\lambda x.M, l{+}k)$, such that ρ extends ρ',

$$\begin{array}{ll} & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}[\![M]\!]\ (l{+}k)\ \rho[x \mapsto l]) \quad (15) \\ \cong & \mathcal{N}[\![[N/x]\ M]\!]\ (l{+}k)\ \rho. \end{array}$$

Case x In this case:

$$\begin{array}{ll} & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}[\![x]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\ \equiv & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (inc^k\ x) \\ \equiv & inc^k\ (\mathcal{N}[\![N]\!]\ l\ \rho') \\ \cong & \text{(by Lemma 6.30)} \\ & \mathcal{N}[\![N]\!]\ (l{+}k)\ \rho' \\ \equiv & \text{(by Lemma 6.12)} \\ & \mathcal{N}[\![[N/x]\ x]\!]\ (l{+}k)\ \rho \end{array}$$

Case y **where** $y \neq x$ In this case:

$$\begin{array}{ll} & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}[\![y]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\ \equiv & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (inc^{l+k-\rho y}\ y) \\ \equiv & inc^{l+k-\rho y}\ y \\ \equiv & \mathcal{N}[\![[N/x]y]\!]\ (l{+}k)\ \rho \end{array}$$

Case c and Case n^ν are similar to the last case.

Case $\lambda y.M$ In this case:

$$\begin{array}{ll} & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}[\![\lambda y.M]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\ \equiv & \lambda y.[\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x] \\ & \qquad (\mathcal{N}[\![M]\!]\ (l{+}k)\ \rho[x \mapsto l][y \mapsto l{+}k]) \\ \cong & \text{(by the induction hypothesis)} \\ & \lambda y.\mathcal{N}[\![[N/x]M]\!]\ (l{+}k)\ \rho[y \mapsto l{+}k] \\ \equiv & \mathcal{N}[\![[N/x]\ \lambda y.M]\!]\ (l{+}k)\ \rho \end{array}$$

Case $\nu n.M$ In this case:

$$\begin{array}{ll} & [\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}[\![\nu n.M]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\ \equiv & new\ ([\mathcal{N}[\![N]\!]\ l\ \rho'\ /\ x] \\ & \qquad (\mathcal{N}[\![M]\!]\ (l{+}k{+}1)\ \rho[x \mapsto l][n \mapsto l{+}k{+}1])) \\ \equiv & \text{(by the induction hypothesis)} \\ & new\ (\mathcal{N}[\![[N/x]M]\!]\ (l{+}k{+}1) \\ & \qquad \rho[x \mapsto l][n \mapsto l{+}k{+}1]) \\ \equiv & \mathcal{N}[\![[N/x]\ \nu n.M]\!]\ (l{+}k)\ \rho \end{array}$$

Case $M_1\ M_2$ Case $M == N$ Case (M_1, M_2)

Case $p\ M$ are all easy induction steps. ■

We now show that the translation $\mathcal{N}$ is stable under reduction in $\lambda\nu$:

Lemma 6.32 (Simulation) For all terms $M, N \in \Lambda$, level numbers l, environments $\rho \in Env(M, l)$: If $M \to N$ then $\rho \in Env(N, l)$ and

$$\mathcal{N}[\![M]\!]\ l\ \rho \cong \mathcal{N}[\![N]\!]\ l\ \rho. \tag{16}$$

Proof: Assume $M \to N$ and let $\rho \in Env(M, l)$. Then all free identifiers and names in N are also free in M, therefore $\rho \in Env(N, l)$. To show (16), we distinguish according to the notion of reduction in $M \to N$. By an argument analogous to the one at the end of the proof of Lemma 6.26, we can assume w.l.o.g. that the redex in the reduction from M to N coincides with M.

Case β In this case the redex is of the form $(\lambda x.M)\ N$, and we have:

$$\begin{array}{ll} & \mathcal{N}[\![(\lambda x.M)\ N)]\!]\ l\ \rho \\ \equiv & (\lambda x.\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l])\ (\mathcal{N}[\![N]\!]\ l\ \rho) \\ = & \text{(by reduction in } \lambda') \\ & [\mathcal{N}[\![N]\!]\ l\ \rho\ /\ x]\ (\mathcal{N}[\![M]\!]\ l\ \rho[x \mapsto l]) \\ = & \text{(by Lemma 6.31)} \\ & \mathcal{N}[\![[N/x]\ M]\!]\ l\ \rho \end{array}$$

Case δ In this case the redex is of the form $p\ V$, where p is a primitive operator and V is a value. We further distinguish according to the form of V. If V is a local name n^ν, then $\mathcal{N}[\![n^\nu]\!]\ l\ \rho$ is a level number and we have:

$$\begin{array}{ll} & \mathcal{N}[\![p\ n^\nu]\!]\ l\ \rho \\ \equiv & p'\ (\rho n^\nu) \\ = & \text{(by reduction in } \lambda') \\ & \mathcal{N}[\![N^p_\nu]\!]\ 0\ \bot \\ \equiv & \text{(by Lemma 6.13)} \\ & \mathcal{N}[\![N^p_\nu]\!]\ l\ \rho \\ \equiv & \mathcal{N}[\![\delta(p, n^\nu)]\!]\ l\ \rho \end{array}$$

If V is a pair (M_1, M_2), then

$$\begin{array}{ll} & \mathcal{N}[\![p\ (M_1, M_2)]\!]\ l\ \rho \\ \equiv & p'\ (\mathcal{N}[\![M_1]\!]\ l\ \rho, \mathcal{N}[\![M_2]\!]\ l\ \rho) \\ = & \text{(by reduction in } \lambda') \\ & (\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ 0\ \bot)\ (\mathcal{N}[\![M_1]\!]\ l\ \rho)\ (\mathcal{N}[\![M_2]\!]\ l\ \rho) \\ \equiv & \text{(by Lemma 6.13)} \\ & (\mathcal{N}[\![N^p_{(\cdot,\cdot)}]\!]\ l\ \rho)\ (\mathcal{N}[\![M_1]\!]\ l\ \rho)\ (\mathcal{N}[\![M_2]\!]\ l\ \rho) \\ \equiv & \mathcal{N}[\![\delta(p, (M_1, M_2))]\!]\ l\ \rho \end{array}$$

The remaining cases, where V is a constant or a λ-abstraction, are similar.

Case eq In this case the redex is of the form $n == m$ where n and m are names. If n and m are the same local name or constant then:

$$\begin{array}{ll} & \mathcal{N}[\![n == n]\!]\ l\ \rho \\ \equiv & eq\ (\mathcal{N}[\![n]\!]\ l\ \rho)\ (\mathcal{N}[\![n]\!]\ l\ \rho) \\ = & \text{(by reduction in } \lambda') \\ & true \\ \equiv & \mathcal{N}[\![true]\!]\ l\ \rho \end{array}$$

If n and m are different local names then ρn and ρm are different level numbers, because ρ is injective. Therefore we have:

$$\begin{array}{ll} & \mathcal{N}[\![n == m]\!]\ l\ \rho \\ \equiv & eq\ (\rho n)\ (\rho m) \\ = & \text{(by reduction in } \lambda') \\ & false \\ \equiv & \mathcal{N}[\![false]\!]\ l\ \rho \end{array}$$

A similar argument can be made in the case of different constants n, m.

If n is a local name and m is a constant, then $\mathcal{N}[\![n]\!]\ l\ \rho$ is a level number and $\mathcal{N}[\![m]\!]\ l\ \rho$ is a constant and we have:

$$\begin{array}{ll} & \mathcal{N}[\![n == m]\!]\ l\ \rho \\ \equiv & eq\ (\rho n)\ m \\ = & \text{(by reduction in } \lambda') \\ & false \\ \equiv & \mathcal{N}[\![false]\!]\ l\ \rho \end{array}$$

The remaining case, where n is a constant and m is a local name, is symmetric.

Case ν_λ In this case the redex is of the form $\nu n.\lambda x.M$, and we have:

$$\begin{array}{ll} & \mathcal{N}[\![\nu n.\lambda x.M]\!]\ l\ \rho \\ \equiv & new\ (\lambda x.\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1][x \mapsto l{+}1]) \\ \equiv & \lambda x.new\ ([inc\ x/x] \\ & \qquad (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1][x \mapsto l{+}1])) \\ = & \text{(by Lemma 6.14)} \\ & \lambda x.new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[x \mapsto l][n \mapsto l{+}1]) \\ \equiv & \mathcal{N}[\![\lambda x.\nu n.M]\!]\ l\ \rho \end{array}$$

Case ν_p In this case the redex is of the form $\nu n.(M, N)$, and we have:

$$
\begin{array}{ll}
& \mathcal{N}[\![\nu n.(M,N)]\!]\, l\, \rho \\
\equiv & new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1], \\
& \qquad \mathcal{N}[\![N]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]) \\
= & \text{(by definition of } new\text{)} \\
& (new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]), \\
& \qquad new\ (\mathcal{N}[\![N]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])) \\
\equiv & (\mathcal{N}[\![\nu n.M]\!]\, l\, \rho, \mathcal{N}[\![\nu n.N]\!]\, l\, \rho)
\end{array}
$$

Case ν_n In this case the redex is of the form $\nu n.m$, where m is a name different from n. If m is a constant, we have:

$$
\begin{array}{ll}
& \mathcal{N}[\![\nu n.m]\!]\, l\, \rho \\
\equiv & new\ m \\
= & \text{(by reduction in } \lambda'\text{)} \\
& m \\
\equiv & \mathcal{N}[\![\nu n.m]\!]\, l\, \rho
\end{array}
$$

On the other hand, if m is a local name, we have:

$$
\begin{array}{ll}
& \mathcal{N}[\![\nu n.m]\!]\, l\, \rho \\
\equiv & new\ (l{+}1{-}\rho m) \\
= & \text{(by reduction in } \lambda'\text{, since } \rho\ m \leq l\text{)} \\
& l{-}\rho m \\
\equiv & \mathcal{N}[\![\nu n.m]\!]\, l\, \rho
\end{array}
$$

■

The preceding lemma showed that $\mathcal{N}$ maps equal terms in $\Lambda\nu$ to equal terms in Λ'. We also need to show the converse: different terms in $\Lambda\nu$ are mapped to different terms in Λ'. To show this, we define in Figure 8 a mapping $\mathcal{N}^{-1}$ which is a left inverse of $\mathcal{N}$. $\mathcal{N}^{-1}$ is a partial mapping from terms in Λ', level numbers l and environments $\rho \in \bigcup\{Env(M,l) \mid M \in \Lambda\nu\}$ to terms in $\Lambda\nu$.

Definition 6.33 For a term $M \in \Lambda'$, let

$$
\begin{array}{rcl}
Env^{-1}(M,l) & = & \{\rho \mid \exists M' \in \Lambda\nu.\ \rho \in Env(M',l) \\
& & \qquad \wedge\ \mathcal{N}^{-1}[\![M]\!]\, l\, \rho \text{ is defined}\}.
\end{array}
$$

We observe that $\mathcal{N}^{-1}$ is well-defined as a function. Since ρ maps different names to different level numbers, there is for any given pair (k,l) at most one name n such that $\rho n = l{-}k$. Therefore, $\mathcal{N}^{-1}$ is a function.

Lemma 6.34 For all $M \in \Lambda\nu$, level numbers l, $\rho \in Env(M,l)$,

$$\mathcal{N}^{-1}[\![\mathcal{N}[\![M]\!]\, l\, \rho]\!]\, l\, \rho \equiv M$$

Proof: An easy induction on the structure of M. We present only two sample cases.

Case x In this case:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![\mathcal{N}[\![x]\!]\, l\, \rho]\!]\, l\, \rho \\
\equiv & \text{(by definition of } \mathcal{N}\text{)} \\
& \mathcal{N}^{-1}[\![inc^{l-\rho x}\ x]\!]\, l\, \rho \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}\text{)} \\
& \mathcal{N}^{-1}[\![x]\!]\ (\rho x)\ \rho \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}\text{)} \\
& x
\end{array}
$$

Case $\nu n.M$ In this case:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![\mathcal{N}[\![\nu n.M]\!]\, l\, \rho]\!]\, l\, \rho \\
\equiv & \text{(by definition of } \mathcal{N}\text{)} \\
& \mathcal{N}^{-1}[\![new\ (\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1])]\!]\, l\, \rho \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}\text{)} \\
& \nu n.\mathcal{N}^{-1}[\![\mathcal{N}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1]]\!] \\
& \qquad (l{+}1)\ \rho[n \mapsto l{+}1] \\
\equiv & \text{(by the induction hypothesis)} \\
& \nu n.M
\end{array}
$$

■

Lemma 6.35 For all terms $M \in \Lambda'$, level numbers $0 \leq k \leq l$, environments $\rho \in Env^{-1}(M,l)$,

(i) If x is not free in M,

$$\mathcal{N}^{-1}[\![M]\!]\, l\, \rho[x \mapsto k] \equiv \mathcal{N}^{-1}[\![M]\!]\, l\, \rho.$$

(ii) If n is not free in M,

$$\mathcal{N}^{-1}[\![M]\!]\, l\, \rho[n \mapsto k] \equiv \mathcal{N}^{-1}[\![M]\!]\, l\, \rho.$$

Proof: Easy. ■

Lemma 6.36 For all $M \in \Lambda'$, level numbers $0 \leq k \leq l$, environments $\rho \in Env^{-1}(M,l)$,

$$
\begin{array}{ll}
(i) & \mathcal{N}^{-1}[\![M]\!]\, l\, \rho[x \mapsto k] \\
& \equiv\ \mathcal{N}^{-1}[\![[dec\ x/x]\ M]\!]\, l\, \rho[x \mapsto k{+}1] \\
(ii) & \mathcal{N}^{-1}[\![M]\!]\, l\, \rho[x \mapsto k{+}1] \\
& \equiv\ \mathcal{N}^{-1}[\![[inc\ x/x]\ M]\!]\, l\, \rho[x \mapsto k]
\end{array}
$$

Proof: (i) By an induction on the structure of M. The only interesting case is Case x. In this case,

$$
\begin{array}{lcll}
\mathcal{N}^{-1}[\![k]\!]\ l\ \rho & \equiv & n & \textbf{if } \rho n = l\text{-}k \\
\mathcal{N}^{-1}[\![x]\!]\ l\ \rho & \equiv & x & \textbf{if } \rho x = l \\
\mathcal{N}^{-1}[\![c]\!]\ l\ \rho & \equiv & c & \\
\mathcal{N}^{-1}[\![\lambda x.M]\!]\ l\ \rho & \equiv & \lambda x.\mathcal{N}^{-1}[\![M]\!]\ l\ \rho[x \mapsto l] & \\
\mathcal{N}^{-1}[\![M_1\ M_2]\!]\ l\ \rho & \equiv & (\mathcal{N}^{-1}[\![M_1]\!]\ l\ \rho)\ (\mathcal{N}^{-1}[\![M_2]\!]\ l\ \rho) & \\
\mathcal{N}^{-1}[\![(M_1, M_2)]\!]\ l\ \rho & \equiv & (\mathcal{N}^{-1}[\![M_1]\!]\ l\ \rho,\ \mathcal{N}^{-1}[\![M_2]\!]\ l\ \rho) & \\
\mathcal{N}^{-1}[\![p'\ M]\!]\ l\ \rho & \equiv & p\ (\mathcal{N}^{-1}[\![M]\!]\ l\ \rho) & \\
\mathcal{N}^{-1}[\![eq\ M_1\ M_2]\!]\ l\ \rho & \equiv & \mathcal{N}^{-1}[\![M_1]\!]\ l\ \rho == \mathcal{N}^{-1}[\![M_2]\!]\ l\ \rho & \\
\mathcal{N}^{-1}[\![inc\ M]\!]\ (l{+}1)\ \rho & \equiv & \mathcal{N}^{-1}[\![M]\!]\ l\ \rho & \\
\mathcal{N}^{-1}[\![dec\ M]\!]\ l\ \rho & \equiv & \mathcal{N}^{-1}[\![M]\!]\ (l{+}1)\ \rho & \\
\mathcal{N}^{-1}[\![new\ M]\!]\ l\ \rho & \equiv & \nu n.\mathcal{N}^{-1}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1] & \textbf{where } n \notin DOM(\rho) \cup FN(M)
\end{array}
$$

Figure 8: Syntactic embedding $\mathcal{N}^{-1}$.

$$
\begin{array}{ll}
 & \mathcal{N}^{-1}[\![x]\!]\ l\ \rho[x \mapsto k] \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}) \\
 & inc^{l-k}\ x \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}) \\
 & \mathcal{N}^{-1}[\![x]\!]\ (l{+}1)\ \rho[x \mapsto k{+}1] \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}) \\
 & \mathcal{N}^{-1}[\![[dec\ x/x]\ x]\!]\ l\ \rho[x \mapsto k{+}1]
\end{array}
$$

The proof of (ii) is analogous. ■

We now show the dual of the substitution lemma (Lemma 6.31).

Lemma 6.37 (Substitution-Dual) For all $M, N \in \Lambda\nu$, level numbers l, $\rho \in Env^{-1}((\lambda x.M)\ N, l)$,

$$
\begin{array}{l}
[\mathcal{N}^{-1}[\![N]\!]\ l\ \rho\ /\ x]\ (\mathcal{N}^{-1}[\![M]\!]\ l\ \rho[x \mapsto l]) \\
\quad \equiv\ \mathcal{N}^{-1}[\![[N/x]\ M]\!]\ l\ \rho.
\end{array}
$$

Note that this is stronger than the corresponding statement of Lemma 6.31, in that we have $\equiv$ instead of $\cong$.

Proof: We use a structural induction on M. A direct proof of the proposition fails for the case $M \equiv \nu n.M'$. We prove instead a stronger proposition:

For all $M, N \in \Lambda\nu$, level numbers l, k, environments $\rho' \in Env^{-1}(N, l)$ and $\rho \in Env^{-1}(\lambda x.M, l{+}k)$, such that ρ extends ρ',

$$
\begin{array}{l}
[\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![M]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\
\quad \equiv\ \mathcal{N}^{-1}[\![[N/x]\ M]\!]\ (l{+}k)\ \rho.
\end{array}
$$

Case x In this case $k = 0$ and

$$
\begin{array}{ll}
 & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![x]\!]\ l\ \rho[x \mapsto l]) \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}) \\
 & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ x \\
\equiv & \text{(by substitution)} \\
 & \mathcal{N}^{-1}[\![[N/x]\ x]\!]\ l\ \rho' \\
\equiv & \text{(by Lemma 6.35)} \\
 & \mathcal{N}^{-1}[\![[N/x]\ x]\!]\ l\ \rho
\end{array}
$$

Case y **where** $y \neq x$ In this case $\rho y = l{+}k$, and

$$
\begin{array}{ll}
 & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![y]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\
\equiv & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ y \\
\equiv & \mathcal{N}^{-1}[\![[N/x]\ y]\!]\ (l{+}k)\ \rho
\end{array}
$$

Case c and Case l are similar to the last case.

Case $\lambda y.M$ In this case:

$$
\begin{array}{ll}
 & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![\lambda y.M]\!]\ (l{+}k)\ \rho[x \mapsto l]) \\
\equiv & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x] \\
 & \quad (\lambda y.\mathcal{N}^{-1}[\![M]\!]\ (l{+}k)\ \rho[x \mapsto l][y \mapsto l{+}k]) \\
\equiv & \text{(by the induction hypothesis)} \\
 & \lambda y.\mathcal{N}^{-1}[\![[N/x]M]\!]\ (l{+}k)\ \rho[x \mapsto l][y \mapsto l{+}k] \\
\equiv & \mathcal{N}^{-1}[\![[N/x]\ \lambda y.M]\!]\ (l{+}k)\ \rho[x \mapsto l]
\end{array}
$$

Case *inc M* In this case $l+k > 0$, and

$$
\begin{array}{ll}
& [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![inc\ M]\!]\ (l+k)\ \rho[x \mapsto l]) \\
\equiv & [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![M]\!]\ (l+k-1)\ \rho[x \mapsto l]) \\
\equiv & \text{(by the induction hypothesis)} \\
& \mathcal{N}^{-1}[\![[N/x]\ M]\!]\ (l+k-1)\ \rho \\
\equiv & \mathcal{N}^{-1}[\![[[N/x]\ (inc\ M)]\!]\ (l+k)\ \rho
\end{array}
$$

Case *dec M* is similar to the last case.

Case *new M* In this case:

$$
\begin{array}{ll}
& [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x]\ (\mathcal{N}^{-1}[\![new\ M]\!]\ (l+k)\ \rho[x \mapsto l]) \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}, \\
& \qquad \text{for some } n \notin DOM(\rho) \cup FN(M)) \\
& [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho'\ /\ x] \\
& \quad (\nu n.\mathcal{N}^{-1}[\![M]\!]\ (l+k+1)\ \rho[x \mapsto l][n \mapsto l+k+1]) \\
\equiv & \text{(by the induction hypothesis)} \\
& \nu n.\mathcal{N}^{-1}[\![[N/x]\ M]\!]\ (l+k+1)\ \rho[n \mapsto (l+k+1] \\
\equiv & \mathcal{N}^{-1}[\![[N/x]\ (new\ M)]\!]\ (l+k)\ \rho
\end{array}
$$

Case $M_1\ M_2$ Case *p M* Case *eq M N* and

Case (M_1, M_2) are all simple induction steps. ■

We now show in the dual of Lemma 6.32 that the reverse translation $\mathcal{N}^{-1}$ is stable under reduction in λ':

Lemma 6.38 (Simulation-Dual) For all $M \in \Lambda'$, level numbers l, $\rho \in Env^{-1}(M, l)$, if $M \to N$ then

$$\mathcal{N}^{-1}[\![M]\!]\ l\ \rho = \mathcal{N}^{-1}[\![N]\!]\ l\ \rho$$

Note that this is stronger than the corresponding statement of Lemma 6.32, in that we have $=$ instead of $\cong$.

Proof: We distinguish according to the form of the redex in $M \to N$. Since $\mathcal{N}^{-1}$ is compositional, we may assume w.l.o.g. that M itself is the redex in the reduction step (this is justified as in the proof of Lemma 6.26).

Case $(\lambda x.M)\ N$ In this case:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![(\lambda x.M)\ N]\!]\ l\ \rho \\
\equiv & \text{(by definition of } \mathcal{N}^{-1}) \\
& (\lambda x.\mathcal{N}^{-1}[\![M]\!]\ l\ \rho[x \mapsto l])\ (\mathcal{N}^{-1}[\![N]\!]\ l\ \rho) \\
= & \text{(by reduction in } \lambda') \\
& [\mathcal{N}^{-1}[\![N]\!]\ l\ \rho\ /\ x]\ (\mathcal{N}^{-1}[\![M]\!]\ l\ \rho[x \mapsto l]) \\
\equiv & \text{(Lemma 6.37)} \\
& \mathcal{N}^{-1}[\![[N/x]\ M]\!]\ l\ \rho
\end{array}
$$

Case $p'\ V$ If V is a level number k then $\mathcal{N}^{-1}[\![k]\!]\ l\ \rho$ is a local name n^ν, and we have:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![p'\ k]\!]\ l\ \rho \\
\equiv & p\ n^\nu \\
= & \text{(by reduction in } \lambda\nu) \\
& N^p_\nu \\
\equiv & \text{(by Lemma 6.34)} \\
& \mathcal{N}^{-1}[\![\mathcal{N}[\![N^p_\nu]\!]\ l\ \rho]\!]\ l\ \rho \\
\equiv & \text{(by Lemma 6.13)} \\
& \mathcal{N}^{-1}[\![\mathcal{N}[\![N^p_\nu]\!]\ 0\ \bot]\!]\ l\ \rho \\
\equiv & \mathcal{N}^{-1}[\![\delta(p', k)]\!]\ l\ \rho
\end{array}
$$

If V is a λ-abstraction $\lambda x.M$, we have:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![p'\ (\lambda x.M)]\!]\ l\ \rho \\
\equiv & p\ (\lambda x.\mathcal{N}^{-1}[\![M]\!]\ l\ \rho[x \mapsto l]) \\
= & \text{(by reduction in } \lambda\nu) \\
& N^p_\lambda \\
\equiv & \text{(by Lemma 6.34)} \\
& \mathcal{N}^{-1}[\![\mathcal{N}[\![N^p_\lambda]\!]\ l\ \rho]\!]\ l\ \rho \\
\equiv & \text{(by Lemma 6.13)} \\
& \mathcal{N}^{-1}[\![\mathcal{N}[\![N^p_\lambda]\!]\ 0\ \bot]\!]\ l\ \rho \\
\equiv & \mathcal{N}^{-1}[\![\delta(p', \lambda x.M)]\!]\ l\ \rho
\end{array}
$$

The remaining cases, where V is a constant or a pair, are similar.

Case *eq V W* In this case V and W are level-numbers or constants, otherwise we would not have a redex. In each case $\mathcal{N}^{-1}[\![V]\!]\ l\ \rho$ and $\mathcal{N}^{-1}[\![W]\!]\ l\ \rho$ are names. Moreover, $\mathcal{N}^{-1}[\![V]\!]\ l\ \rho == \mathcal{N}^{-1}[\![W]\!]\ l\ \rho$ iff V and W are the same level number or constant. Hence, if $eq\ V\ W = true$ in λ', then:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![eq\ V\ W]\!]\ l\ \rho \\
\equiv & V == W \\
= & true \\
\equiv & \mathcal{N}^{-1}[\![true]\!]\ l\ \rho
\end{array}
$$

On the other hand, $eq\ V\ W = false$ in λ', then:

$$
\begin{array}{ll}
& \mathcal{N}^{-1}[\![eq\ V\ W]\!]\ l\ \rho \\
\equiv & V == W \\
= & false \\
\equiv & \mathcal{N}^{-1}[\![false]\!]\ l\ \rho
\end{array}
$$

Case *inc* $(\lambda x.M)$ In this case $l > 0$, since otherwise

$\rho \notin Env^{-1}(inc\ (\lambda x.M), l)$, and we have

$$\begin{array}{ll} & \mathcal{N}^{-1}[\![inc\ (\lambda x.M)]\!]\ l\ \rho \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}) \\ & \lambda x.\mathcal{N}^{-1}[\![M]\!]\ (l{-}1)\ \rho[x \mapsto l{-}1] \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}) \\ & \lambda x.\mathcal{N}^{-1}[\![inc\ M]\!]\ l\ \rho[x \mapsto l{-}1] \\ \equiv & \text{(by Lemma 6.36)} \\ & \lambda x.\mathcal{N}^{-1}[\![[dec\ x/x]\ (inc\ M)]\!]\ l\ \rho[x \mapsto l] \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![\lambda x.[dec\ x/x]\ (inc\ M)]\!]\ l\ \rho \end{array}$$

Case $inc\ k$ In this case $l > 0$, and we have:

$$\begin{array}{ll} & \mathcal{N}^{-1}[\![inc\ k]\!]\ l\ \rho \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![k]\!]\ (l{-}1)\ \rho \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}, \\ & \quad \text{for some } n \text{ s.t. } \rho n = l{-}1{-}k) \\ & n \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![k{+}1]\!]\ l\ \rho \end{array}$$

Case $inc\ c$ In this case $l > 0$, and we have:

$$\begin{array}{ll} & \mathcal{N}^{-1}[\![inc\ c]\!]\ l\ \rho \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![c]\!]\ (l{-}1)\ \rho \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & c \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![c]\!]\ l\ \rho \end{array}$$

Case $inc\ (M_1, M_2)$ is an easy induction step.

Case $dec\ \lambda x.M$ Case $dec\ k$ Case $dec\ c$ and Case $dec\ (M_1, M_2)$ are symmetric to the last four cases.

Case $new\ (\lambda x.M)$ In this case:

$$\begin{array}{ll} & \mathcal{N}^{-1}[\![new\ (\lambda x.M)]\!]\ l\ \rho \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}, \\ & \quad \text{for some name } n \notin DOM(\rho) \cup FN(T)) \\ & \nu n.\lambda x.\mathcal{N}^{-1}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1][x \mapsto l{+}1] \\ = & \text{(by reduction in } \lambda') \\ & \lambda x.\nu n.\mathcal{N}^{-1}[\![M]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1][x \mapsto l{+}1] \\ \equiv & \text{(by Lemma 6.36)} \\ & \lambda x.\nu n.\mathcal{N}^{-1}[\![[inc\ x/x]M]\!]\ (l{+}1)\ \rho[x \mapsto l][n \mapsto l{+}1] \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![\lambda x.[inc\ x/x]\ (new\ M)]\!]\ l\ \rho \end{array}$$

Case $new\ k$ In this case $k > 0$, since otherwise there would be no redex, and we have:

$$\begin{array}{ll} & \mathcal{N}^{-1}[\![new\ k]\!]\ l\ \rho \\ \equiv & \text{(by definition of } \mathcal{N}^{-1}, \\ & \quad \text{for some name } n \notin DOM(\rho) \cup FN(T)) \\ & \nu n.\mathcal{N}^{-1}[\![k]\!]\ (l{+}1)\ \rho[n \mapsto l{+}1] \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}, \\ & \quad \text{for some name } m \text{ s.t. } \rho m = l{+}1{-}k) \\ & \nu n.m \\ = & \text{(by reduction in } \lambda', \text{ since } m \neq n) \\ & m \\ \equiv & \text{(by the definition of } \mathcal{N}^{-1}) \\ & \mathcal{N}^{-1}[\![k{-}1]\!]\ l\ \rho \end{array}$$

Case $new\ c$ and Case $new\ (M_1, M_2)$ are both easy. ■

Now everything is in place for the main lemma of this section:

Lemma 6.39 For all closed $M \in \Lambda\nu$, answers A,

$$\lambda\nu \vdash M = A \ \Leftrightarrow\ \lambda' \vdash \mathcal{N}[\![M]\!]\ 0\ \bot = A.$$

Proof: "$\Rightarrow$": Assume $M = A$. Then $M \twoheadrightarrow A$ since $\lambda\nu$ is Church-Rosser. By an induction on the length of reduction from M to A, using Lemma 6.32 at each step,

$$\mathcal{N}[\![M]\!]\ 0\ \bot \cong \mathcal{N}[\![A]\!]\ 0\ \bot.$$

But $\mathcal{N}[\![A]\!]\ 0\ \bot \equiv A$, hence $\mathcal{N}[\![M]\!]\ 0\ \bot \cong A$. By the definition of $\cong$, this implies $\mathcal{N}[\![M]\!]\ 0\ \bot = A$.

"$\Leftarrow$": Assume $\mathcal{N}[\![M]\!]\ 0\ \bot = A$. Then $\mathcal{N}[\![M]\!]\ 0\ \bot \twoheadrightarrow A$ since λ' is Church-Rosser. By an induction on the length

of reduction from M to A, using Lemma 6.38 at each step,

$$\mathcal{N}^{-1}[\![\mathcal{N}[\![M]\!]\,0\,\bot]\!]\,0\,\bot = \mathcal{N}^{-1}[\![A]\!]\,0\,\bot$$

The left-hand side of this equality is M by Lemma 6.34, whereas the right-hand side is A by the definition of $\mathcal{N}^{-1}$. Hence, $M = A$. ■

We now drop the assumption that $\mathcal{N}$ maps to Λ', where all p', *eq*, *inc*, *dec* and *new* are primitive functions, and state the equivalent of Lemma 6.39 for λ. This rests on the fact that all p', *eq*, *inc*, *dec*, and *new* are definable in λ. The definitions for p', *eq*, *inc* are:

$$\begin{aligned}
p' &= \lambda x.\mathit{if}\ (\mathit{lnum}?\ x)\ (\mathcal{N}[\![N_\nu^p]\!]\,0\,\bot)\ (p\ x)\\
eq' &= \lambda x.\lambda y.\mathit{if}(\mathit{lnum}?\ x)\\
&\qquad \mathit{if}(\mathit{lnum}?\ y)\\
&\qquad\quad (\mathit{eq_lnum}\ x\ y)\\
&\qquad\quad (x == y)\\
&\qquad (x == y)\\
\mathit{inc} &= \lambda x.\mathit{if}\ (\mathit{lnum}?\ x)\\
&\qquad (x+1)\\
&\qquad (\mathit{if}\ (\mathit{name}?\ x)\\
&\qquad\quad x\\
&\qquad\quad (\mathit{if}\ (\mathit{pair}?\ x)\\
&\qquad\qquad (\mathit{inc}\ (\mathit{fst}\ x), \mathit{inc}\ (\mathit{snd}\ x))\\
&\qquad\qquad (\lambda y.\mathit{inc}\ (x\ (\mathit{dec}\ y)))))
\end{aligned}$$

The definitions of *dec* and *new* are analogous to the one for *inc*.

Let $\tilde{\mathcal{N}}$ be the syntactic mapping defined as in Figure 5 except that instead of the primitive functions p', *eq*, *inc*, *dec*, *new* their defining λ-terms are used. Then clearly,

$$\lambda' \vdash \mathcal{N}[\![M]\!]\,l\,\rho \Leftrightarrow \lambda \vdash \tilde{\mathcal{N}}[\![M]\!]\,l\,\rho.$$

With Lemma 6.39, this implies:

Corollary 6.40 For all closed $M \in \Lambda\nu$, answers A,

$$\lambda\nu \vdash M = A \Leftrightarrow \lambda \vdash \mathcal{N}[\![M]\!]\,0\,\bot = A.$$

Corollary 6.41 $\mathcal{E}$ is a syntactic embedding of $\lambda\nu$ in λ

Proof: It is straightforward to verify that $\tilde{\mathcal{E}}$ preserves closed Λ subterms. With Corollary 6.40, $\mathcal{E}$ preserves semantics. ■

6.3 Putting It All Together

Theorem 6.42 $\lambda\nu$ is a conservative observational extension of λ.

Proof: By Corollary 6.41, $\mathcal{E}$ is a syntactic embedding of $\lambda\nu$ in λ. By Theorem 6.5 this implies that $\lambda\nu$ is an observational extension of λ. To show that the extension is conservative, assume $\lambda\nu \models M \cong N$, for terms $M, N \in \Lambda$. Then we have

$$\lambda\nu \vdash C[M] = A \Leftrightarrow \lambda\nu \vdash C[N] = A$$

for all $\Lambda\nu$-contexts C such that $C[M]$ and $C[N]$ are closed and therefore also for all such Λ-contexts. Since terms $M \in \Lambda$ have only β and δ redexes, and since λ is closed under $\beta\delta$ reduction, this implies $\lambda \models M \cong N$. ■

7 Conclusions

$\lambda\nu$ is a syntactic theory for functions that create local names. The calculus is fully compatible with functional programming, in the sense that all observational equivalences of functional programming are preserved. There is 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 showed how imperative programming with mutable 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.

Some other areas are left to future research. In particular, one would like to study models for local names. Another question concerns implementation: How expensive are local names in practice? There exists a prototype implementation of $\lambda\nu$ in Yale Haskell, but it is not sufficiently integrated with the compiler to answer questions of efficiency.

Acknowledgements This work was supported in part by grant N00014-91-J-4043 from DARPA. I thank Kung Chen, Vincent Dornic and Dan Rabin for their comments on earlier versions of the paper. I had many important discussions with them and also with Paul Hudak. John Launchbury, Jayadev Misra, David Turner and Phil Wadler also commented on this work in useful 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 Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 233–244. ACM Press, January 1991.

[3] N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[4] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[5] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Department of Computer Science, Rice University, June 1989.

[6] 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.

[7] D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Compuatational Problems in Abstract Algebra*, pages 263–297. Pergamon, Oxford, 1970.

[8] J. Launchbury. Lazy imperative programming. In *Proc. SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*. Yale University Technical Report, June 1993.

[9] 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.

[10] I. A. Mason and C. L. Talcott. References, local variables, and operational reasoning. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 186–197, Los Alamitos, California, June 1992. IEEE Computer Society Press.

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

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100:1–40, 1992.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100:41–77, 1992.

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

[15] M. Odersky. A syntactic method for proving observational equivalences. Research Report YALEU/DCS/RR-964, Department of Compter Science, Yale University, May 1993.

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

[17] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 43–57, January 1993.

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

[19] A. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names. In *Proc. SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*. Yale University Technical Report, June 1993.

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

[21] J. G. Riecke. Delimiting the scope of effects. In *Functional Programming and Computer Architecture*, June 1993.

[22] 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.

[23] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, June 1990.

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

[25] 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 57–71, January 1993.