The Semantics of Functions with Locally Scoped Names

Steffen Lösch
Trinity College

UNIVERSITY OF CAMBRIDGE

An essay submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

Supervisor: Prof Andrew M. Pitts

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: Steffen.Loesch@cl.cam.ac.uk

June 8, 2010
Declaration

I, Steffen Lösch of Trinity College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 4956

Signed:

Date:

This essay is copyright ©2010 Steffen Lösch.
All trademarks used in this essay are hereby acknowledged.
Abstract

This essay presents the translation from Pitts’ and Stark’s nu-calculus [12] into Odersky’s $\lambda\nu$ [8]. Even though both calculi are extensions of the lambda-calculus with a name generation construct, their semantics are very different. We give a big-step and a frame stack operational semantics for the nu-calculus, as well as a big-step operational and a novel denotational semantics for $\lambda\nu$. The denotational semantics is based on nominal sets, allowing simple definitions and reasoning. We state important properties of it and sketch their proofs. The translation uses the continuation-passing-style and an outline of how to prove its adequacy is given. This means that Odersky’s “pure” notion of local names is more general than the dynamic name allocation in the nu-calculus, given that adequacy holds.
Contents

1 Introduction 1
  1.1 Structure .............................................. 2
  1.2 Contributions ........................................ 2

2 Background 5
  2.1 Permutations .......................................... 5
  2.2 Nominal Sets .......................................... 6
    2.2.1 Nominal Restriction Sets ....................... 8
  2.3 Constructions on Nominal Restriction Sets ............. 9
    2.3.1 Trivial Actions and Restrictions .............. 9
    2.3.2 Atoms ............................................. 9
    2.3.3 Products ......................................... 10
    2.3.4 Functions ....................................... 10
  2.4 Equivariance .......................................... 11

3 Syntax and Typing 13
  3.1 Syntax ................................................ 13
    3.1.1 Types ............................................ 14
    3.1.2 Terms ............................................ 14
    3.1.3 Binding .......................................... 14
  3.2 Typing ................................................ 16

4 Operational Semantics 19
  4.1 Big-Step Semantics of the Nu-Calculus .............. 19
    4.1.1 Values ........................................... 20
    4.1.2 Big-Step Evaluation Relation .................. 20
  4.2 Frame Stack Semantics of the Nu-Calculus .......... 21
    4.2.1 Typed Frame Stacks .............................. 22
    4.2.2 Frame Stack Evaluation Relation ............... 23
    4.2.3 Termination ..................................... 24
Chapter 1

Introduction

Locally scoped names are an important feature of imperative programming languages. For example, local variables in C can be declared at the beginning of each block and are deleted at the end of their block. The concept of local names is used in various other imperative constructs and languages.

When combining functional and imperative features, to obtain a multiple purpose programming language, subtleties can arise. Techniques may interact in unexpected ways and thereby lead to unforeseen errors. The task of programming language semantics is to define languages in a strict mathematical way, such that it is possible to prove their properties.

In this essay, we consider the combination of the lambda-calculus with locally scoped names. We work in a reduced setting, where names can be created and compared, but nothing else. The aim is to comprehend the behaviour of the combination on a small scale, extending the results afterwards.

More specifically, we study two works [12, 8]—developed independently in the 1990s—that combine the simply typed lambda-calculus with a name restriction construct and a name equality test. Pitts’ and Stark’s [12] nu-calculus was designed as a toy-language to study contextual equivalence in ML-like languages. Being a fragment of ML, the nu-calculus has local state and creates fresh names dynamically.
Odersky’s $\lambda\nu$ [8] is intended to provide a basis for modelling functional languages with imperative features, without disturbing familiar properties like function extensionality. Therefore, $\lambda\nu$ is “purer”: it does not contain local state and name restriction is handled by rewriting.

The nu-calculus stimulated various works in semantics, including for example bisimulation models [5], game semantics [1], category theory [15], type theory [3], or reasoning about storages [2]. On the contrary, $\lambda\nu$ remained little explored by the research community.

Recently, Pitts rediscovered $\lambda\nu$ in his work on Nominal System T [11] and sketched a new denotational semantics for it. Building on this discovery, we give a continuation-passing-style translation from the nu-calculus into $\lambda\nu$. Adequacy of the translation is stated, but remains to be proved. It would establish that $\lambda\nu$ is more general than the nu-calculus, a surprising result for many researchers.

1.1 Structure

In Chapter 2 we introduce the framework of nominal sets, that is used for modelling the denotational semantics of $\lambda\nu$. Chapter 3 gives the common syntax and typing relation for the two calculi. Operational semantics of both calculi are discussed in Chapter 4 and the denotational semantics of $\lambda\nu$ is defined in Chapter 5. The CPS translation is given in Chapter 6, Chapter 7 concludes.

1.2 Contributions

Most of the content of this essay has been published in similar form before. However, the material had to be rewritten and adapted to the given context, in this sense it is new. Furthermore, some content was not presented before and had to be developed. Even though frame stacks are known in
the literature, the frame stack semantics of the nu-calculus is a novelty. The big-step semantics of $\lambda\nu$ is also new, because Odersky gave the semantics of $\lambda\nu$ as rewriting system. Pitts [11] only sketches the denotational semantics based on nominal sets very briefly, so the full definition and the proofs about it are original. The CPS translation was mainly given by Pitts in private conversation, but it had to be rewritten to match our setting. Due to time restrictions, the proofs concerning the translation remain as future work.

Remark: For stylistic reasons, the author uses “we” instead of “I”.
Chapter 2

Background

The technicalities of our approach are introduced in this chapter. The content focuses on Pitts’ framework of nominal sets (e.g. [4, 10]) for arguing about names and binding in programming languages.

2.1 Permutations

We start by giving some intuition about the mathematics. Crucial to the nominal treatment of binding is the notion of name permutation. Name permutations are much better behaved than simple (that is, non-capture-avoiding) substitutions. Consider the lambda-abstraction $\lambda x.y$ and the results of permuting and substituting the names $x$ and $y$:

\[
(x \cdot y) \cdot (\lambda x.y) = \lambda y.x \quad \text{(Permutation)}
\]

\[
(\lambda x.y)[x/y] = \lambda x.x \quad \text{(Substitution)}
\]

The result of the substitution is clearly not desirable, because $x$ is captured by the binding construct. In contrast to that, the result of applying a permutation to two $\alpha$-equivalent terms is always $\alpha$-equivalent:

\[
t =_\alpha t' \implies \pi \cdot t =_\alpha \pi \cdot t'
\]
We keep this pleasing behaviour in mind when we define the nominal framework.

Let $\mathbb{A}$ be a countably infinite set with elements $a \in \mathbb{A}$ called atoms. Intuitively, atoms can be considered as names in a program.

**Definition 1.** The group $S_{\mathbb{A}}$ of **finite permutations** of $\mathbb{A}$ consists of all bijections $\pi : \mathbb{A} \to \mathbb{A}$ for which $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is a finite set.

We call the elements of $S_{\mathbb{A}}$ permutations and assume their finiteness implicitly. $S_{\mathbb{A}}$ is a group in the mathematical sense, hence there is a multiplication operation between permutations (given by composition), that is associative, possesses a unit and possesses inverses.

**Definition 2.** The **swapping** of two atoms $(a \ a')$ is the permutation interchanging $a$ and $a'$, while leaving all other atoms fixed:

$$ (a \ a')(a'') := \begin{cases} 
  a & \text{if } a'' = a' \\
  a' & \text{if } a'' = a \\
  a'' & \text{otherwise} 
\end{cases} $$

Every permutation $\pi \in S_{\mathbb{A}}$ can be expressed as the composition of atom swappings. This simplifies many definitions and proofs, because they only have to be given by means of swappings.

**2.2 Nominal Sets**

In order to define nominal sets, we first need to relate permutations to other structures.

**Definition 3.** A $S_{\mathbb{A}}$-**set** is a set $X$ equipped with a permutation action. The permutation action is a function $S_{\mathbb{A}} \times X \to X$, mapping a permutation $\pi$ and an element $x$ to the element $\pi \cdot x$. It is required to be associative and elements must be invariant under the identity permutation:

...
\[
\forall \pi \in S_A. \forall x \in X. \pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x
\]

\[
\forall x \in X. \ id_A \cdot x = x
\]

Every \( S_A \)-set comes with a notion of support. Intuitively, an element of a \( S_A \)-set is supported by \( A \subseteq A \), if it “contains” no element of \( A \).

**Definition 4.** A set of atoms \( A \subseteq A \) supports an element \( x \) of a \( S_A \)-set \( X \), iff \( x \) remains unchanged under a permutation fixing \( A \):

\[
A \text{ supports } x \iff \forall \pi \in S_A. \ (\forall a \in A. \ \pi(a) = a) \Rightarrow \pi \cdot x = x
\]

**Remark 5.** There is an equivalent definition of support based on atom swaps. \( A \subseteq A \) supports \( x \), iff \( x \) remains unchanged under swapping of atoms not in \( A \):

\[
A \text{ supports } x \iff \forall a, a' \in (A - A). \ (a a') \cdot x = x
\]

We are particularly interested in finitely supported elements, i.e. elements being supported by only a finite set of atoms. Every such element has a unique smallest support. This follows from the fact that if \( x \) is supported by two finite sets \( A \) and \( B \), then it is supported by \( A \cap B \).

**Definition 6.** The support of \( x \), denoted by \( \text{supp}(x) \), is the smallest set of atoms supporting it. It is unique for finitely supported elements.

The definition of a nominal set is now easily stated:

**Definition 7.** A \( S_A \)-set \( X \) is a nominal set, iff every element of \( X \) has a finite support:

\[
\forall x \in X. \ \text{supp}(x) \text{ exists and is finite.}
\]

Many sets in programming language semantics are nominal. A common example are the terms of the untyped lambda-calculus.

**Example 8.** The set of terms of the untyped lambda-calculus is given by
the following grammar:

\[ t ::= a | \lambda a.t | t \ t \]

where \( a \in A \). It can be turned into a nominal set by defining the permutation action:

\[
\begin{align*}
\pi \cdot a &:= \pi(a) \\
\pi \cdot (\lambda a.t) &:= \lambda(\pi(a)).(\pi \cdot t) \\
\pi \cdot (t \ t') &:= (\pi \cdot t)(\pi \cdot t')
\end{align*}
\]

The support of a lambda-term is not just the set of free atoms of the term, but the set of all atoms occurring in it.

Nominal sets come with a notion of freshness. It defines precisely what we mean by using a fresh name.

**Definition 9.** An atom \( a \) is **fresh** for a finitely supported element \( x \), written as \( a \# x \), iff \( a \) is not in the support of \( x \).

\[
a \# x \iff a \notin \text{supp}(x)
\]

### 2.2.1 Nominal Restriction Sets

We introduce a restriction operation for nominal sets. This will be useful in the denotational semantics of \( \lambda \nu \) for modelling the name generation construct.

**Definition 10.** A **nominal restriction set** \( X \) is a nominal set with a restriction operation. The restriction operation is a function \( A \times X \rightarrow X \) mapping an atom \( a \) and an element \( x \) to the element \( a \setminus x \), such that the following four properties hold:

\[
\begin{align*}
\pi \cdot (a \setminus x) &= (\pi(a)) \setminus \pi \cdot x \\
\pi \cdot x \setminus a &= x \setminus a
\end{align*}
\]

\( a \# a \setminus x \)
Example 11. The name-restriction operation for the nominal set of untyped lambda-terms of Example 8 is obtained by adjoining a constant \textit{New} to the grammar

\[
t ::= a \mid \lambda a.t \mid t t \mid \text{New}
\]

and defining \(a \setminus t\) to be \(t[\text{New}/a]\).

2.3 Constructions on Nominal Restriction Sets

In this section, we introduce ways of constructing new nominal restriction sets out of existing ones.

2.3.1 Trivial Actions and Restrictions

Any set \(X\) can be turned into a nominal restriction set, adjoining the trivial permutation action and the trivial restriction operation:

\[
\forall \pi \in S_A. \forall x \in X. \pi \cdot x = x
\]

\[
\forall a \in A. \forall x \in X. a \setminus x = x
\]

2.3.2 Atoms

The set of atoms \(A\) itself can be turned into a nominal restriction set, by endowing it with an additional element \textit{New}. Thereby we obtain the nominal restriction set \(A[\text{New}] := A \cup \{\text{New}\}\), with permutation action and restriction
operation defined as follows:

\[
\pi \cdot a := \begin{cases} 
  \text{New} & \text{if } a = \text{New} \\
  \pi(a) & \text{if } a \neq \text{New}
\end{cases}
\]

\[
a \setminus a' := \begin{cases} 
  \text{New} & \text{if } a = a' \\
  a' & \text{if } a \neq a'
\end{cases}
\]

### 2.3.3 Products

Given two nominal restriction sets \(X\) and \(Y\), we can form the nominal restriction product \(X \times Y\) by defining:

\[
\pi \cdot (x, y) := (\pi \cdot x, \pi \cdot y)
\]

\[
a \setminus (x, y) := (a \setminus x, a \setminus y)
\]

### 2.3.4 Functions

Endowing functions with a permutation action and restriction operation is subtle, but very important, because they play a key role in our denotational semantics for \(\lambda\nu\). We start by defining a permutation action on functions.

**Definition 12.** Given a function \(f : X \rightarrow Y\) between two \(S_A\)-sets \(X\) and \(Y\), the **permutation action on the function** \(f\) is defined by:

\[
(\pi \cdot f)(x) := \pi \cdot \left(f(\pi^{-1} \cdot x)\right)
\]

Therefore, the set of function between two \(S_A\)-sets \(X\) and \(Y\) is a \(S_A\)-set \(X \rightarrow Y\). However, \(X \rightarrow Y\) is not a nominal set. To obtain a nominal set, we have to restrict ourselves to finitely supported functions.

**Definition 13.** Given two nominal sets \(X\) and \(Y\), the set of finitely supported
functions $X \rightarrow_{ts} Y$ is defined as its name proposes:

$$
X \rightarrow_{ts} Y := \{ f : X \rightarrow Y \mid f \text{ has finite support} \}
$$

$X \rightarrow_{ts} Y$ is nominal directly by definition. We are now able to explain what we mean by nominal restriction functions.

**Definition 14.** Let a nominal set $X$ and a nominal restriction set $Y$ be given. The set of *nominal restriction functions* is $X \rightarrow_{ts} Y$ equipped with the following restriction operation:

$$(a \setminus f)(x) := a' \setminus \left( \left( (a a') \cdot f \right)(x) \right) \text{ where } a' \# (a, f, x)$$

**Remark 15.** Applying the restriction operation to a finitely supported function returns a unique result. Furthermore, every nominal restriction function $f \in (X \rightarrow_{ts} Y)$ satisfies

$$a \# a \setminus f$$

$$\forall x \in X. a \# x \Rightarrow (a \setminus f)(x) = a \setminus (f(x))$$

and the conditions in Definition 10 hold.

### 2.4 Equivariance

Another important notion of our nominal framework is equivariance. Intuitively, an equivariant function does not “contain” any name.

**Definition 16.** A function $f : X \rightarrow Y$ is *equivariant*, iff it has empty support.
Remark 17. There are other equivalent definitions of equivariance:

\[ \text{supp}(f) = \emptyset \]
\[ \iff \forall \pi \in S_k. \pi \cdot f = f \]
\[ \iff \forall \pi \in S_k. \forall x \in X. f(\pi \cdot x) = \pi \cdot f(x) \]

Equivariant functions play a central role in modelling nominal semantics. Many important operations, such as function application, currying or projections, are equivariant. Nominal sets with equivariant functions form a category and a rich category theory of nominal sets exists. In this essay, however, we will avoid category theoretic terms, for better comprehension.
Chapter 3

Syntax and Typing

The syntax and typing rules we use are identical for the nu-calculus and $\lambda\nu$. This unification is just a slight deviation from the original syntax [12, 8] and the expressiveness remains the same. Details and formal definitions are given in this chapter.

3.1 Syntax

$\lambda\nu$ and the nu-calculus are both extensions of the simply typed lambda-calculus with locally scoped names. One can create a new local name with $\nu$ and compare names with $==$. Hence the only thing known about a name is its identity, but nothing else.
3.1.1 Types

There are four different kinds of types: booleans, names, pairs and functions. The set of types $Typ$ is defined by the following grammar:

$$T \in Typ ::= \begin{align*}
\text{Bool} & \quad \text{(booleans)} \\
\mid & \quad \text{Name} \quad \text{(names)} \\
\mid & \quad T \times T \quad \text{(pairs)} \\
\mid & \quad T \rightarrow T \quad \text{(functions)}
\end{align*}$$

3.1.2 Terms

The set of terms $Term$ is given as:

$$t \in Term ::= \begin{align*}
x & \quad \text{(variables)} \\
\mid & \quad a \quad \text{(atomic names)} \\
\mid & \quad \nu a.t \quad \text{(name restriction)} \\
\mid & \quad t == t \quad \text{(name equality test)} \\
\mid & \quad \text{true} \quad \text{(boolean value true)} \\
\mid & \quad \text{false} \quad \text{(boolean value false)} \\
\mid & \quad \text{if } t \text{ then } t \text{ else } t \quad \text{(conditional)} \\
\mid & \quad (t, t) \quad \text{(pairing)} \\
\mid & \quad \text{case } t \text{ of } (x, x) \rightarrow t \quad \text{(pair destruction)} \\
\mid & \quad \lambda x.t \quad \text{(function abstraction)} \\
\mid & \quad t \ t \quad \text{(function application)}
\end{align*}$$

Here, $x$ is a member of an infinite set of variables $\mathbb{V}$, and $a$ is member of an infinite set of atoms (or names) $\mathbb{A}$. The two sets are disjoint.

3.1.3 Binding

Binding constructs are name restriction, pair destruction and function abstraction. The functions $fv : Term \rightarrow \mathbb{V}$, returning the free variables of a
term, and \( fn : \text{Term} \rightarrow \mathcal{A} \), returning the free names of a term, are defined as:

\[
\begin{align*}
fv(x) & := \{x\} \\
fv(a) & := \{} \\
fv(\nu a.t) & := fv(t) \\
fv(t_1 == t_2) & := fv(t_1) \cup fv(t_2) \\
fv(b) & := \{} \quad \text{for } b = \text{true} \text{ or } b = \text{false} \\
fv(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) & := fv(t_1) \cup fv(t_2) \cup fv(t_3) \\
fv((t_1, t_2)) & := fv(t_1) \cup fv(t_2) \\
fv(\text{case } t \text{ of } (x_1, x_2) \rightarrow t') & := fv(t) \cup (fv(t') - \{x_1, x_2\}) \\
fv(\lambda x.t) & := fv(t) - \{x\} \\
fv(t_1 t_2) & := fv(t_1) \cup fv(t_2)
\end{align*}
\]

\[
\begin{align*}
fn(x) & := \{} \\
fn(a) & := \{a\} \\
fn(\nu a.t) & := fn(t) - \{a\} \\
fn(t_1 == t_2) & := fn(t_1) \cup fn(t_2) \\
fn(b) & := \{} \quad \text{for } b = \text{true} \text{ or } b = \text{false} \\
fn(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) & := fn(t_1) \cup fn(t_2) \cup fn(t_3) \\
fn((t_1, t_2)) & := fn(t_1) \cup fn(t_2) \\
fn(\text{case } t \text{ of } (x_1, x_2) \rightarrow t') & := fn(t) \cup fn(t') \\
fn(\lambda x.t) & := fn(t) \\
fn(t_1 t_2) & := fn(t_1) \cup fn(t_2)
\end{align*}
\]

Implicitly, we identify terms up to \( \alpha \)-equivalence of variables and names.

Please note that the language is very simple. There are no recursive functions or while-loops, so every term evaluates to a normal form.
3.2 Typing

A typing environment $\Gamma : \mathcal{V} \to \mathcal{T}$ is a function mapping a finite number of variables to types. We write $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$ for the function mapping $x_i$ to $T_i$ for $1 \leq i \leq n$, and is undefined otherwise. The notation $\Gamma, x : T$ stands for the environment mapping $x$ to $T$ and behaving like $\Gamma$ otherwise. The typing judgement takes the following form:

$$\Gamma \vdash t : T$$

$\Gamma$ is a typing environment, $t \in \text{Term}$ is a term and $T \in \text{Typ}$ is a type. It is defined by recursion:

- **var**: $\frac{1 \leq i \leq n}{x_1 : T_1, \ldots, x_n : T_n \vdash x_i : T_i}$
- **name**: $\frac{a \in \mathbb{A}}{\Gamma \vdash a : \text{Name}}$
- **loc**: $\frac{\Gamma \vdash t : T}{\Gamma \vdash \lambda a.t : T}$
- **equ**: $\frac{\Gamma \vdash t_1 : \text{Name}}{\Gamma \vdash t_1 ::= t_2 : \text{Bool}}$
- **tbool**: $\frac{\Gamma \vdash t_1 : \text{Bool}}{\Gamma \vdash \text{true} : \text{Bool}}$
- **fbool**: $\frac{\Gamma \vdash t_1 : \text{Bool}}{\Gamma \vdash \text{false} : \text{Bool}}$
- **if**: $\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$
- **pair**: $\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2}$
- **case**: $\frac{\Gamma \vdash t : T \times T \quad x_1, x_2 \notin \text{dom}(\Gamma)}{\Gamma \vdash \text{case } t \text{ of } (x_1, x_2) \rightarrow t' : T'}$
- **abs**: $\frac{\Gamma, x : T \vdash t : T' \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \lambda x.t : T \rightarrow T'}$
- **app**: $\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T'}{\Gamma \vdash t_1 t_2 : T'}$

The type of a name restriction is the one of its underlying term and only names can be checked for equality.

Variable-closed expressions of type $T$ can have free names and are defined
as:

$$\text{Term}_T := \{ t \in \text{Term} \mid \emptyset \vdash t : T \}$$

To give some intuition about typeability, we conclude this chapter with a few interesting typing judgements.

$$\emptyset \vdash \nu a. a \ : \ \text{Name}$$

$$\{b_1 : \text{Bool}\} \vdash \nu a. \nu a'. \lambda b_2. \text{if } a == a' \text{ then } b_1 \text{ else } b_2 : \text{Bool} \rightarrow \text{Bool}$$

$$\emptyset \vdash \text{case } (\nu a. a, \nu a. a) \text{ of } (x_1, x_2) \rightarrow x_1 == x_2 : \text{Bool}$$
Chapter 4

Operational Semantics

Even though the nu-calculus and $\lambda\nu$ have identical syntax, their semantics are very different. The nu-calculus uses a call-by-value evaluation strategy with ML-like local state. In contrast to that, $\lambda\nu$ is call-by-name and instead of having local state it treats name restrictions by “pushing the restriction inside terms”. In this chapter, we introduce both operational semantics and briefly discuss their differences.

4.1 Big-Step Semantics of the Nu-Calculus

The operational semantics of the nu-calculus is presented in two different styles: big-step and frame stack. The big-step semantics is used in the original paper [12], and is easier to comprehend, but the frame stack semantics simplifies proving semantic properties.

We use the notation $\delta(a, a')$ for the equality check of two atomic names:

$$
\delta(a, a') := \begin{cases} 
\text{true} & \text{if } a = a' \\
\text{false} & \text{if } a \neq a'
\end{cases}
$$
4.1.1 Values

The set of values $Val$ of the nu-calculus can be built from the following grammar:

$$ v \in Val ::= x \quad \text{(variables)} $$

$$ \text{\mid} a \quad \text{(atomic names)} $$

$$ \text{\mid} \text{true} \quad \text{(boolean value true)} $$

$$ \text{\mid} \text{false} \quad \text{(boolean value false)} $$

$$ \text{\mid} (v, v) \quad \text{(pairing)} $$

$$ \text{\mid} \lambda x.t \quad \text{(function abstraction)} $$

We define the set of variable-closed values of type $T$, possibly containing free names:

$$ Val_T := \{ v \in Val \mid \emptyset \vdash v : T \} $$

4.1.2 Big-Step Evaluation Relation

The big-step evaluation relation takes the following form:

$$ s \vdash t \downarrow (s')v $$

where $s, s' \subseteq_{\text{fin}} \mathcal{A}$ are disjoint finite sets of names, $t \in \text{Term}_T$ is a variable-closed term, $v \in Val_T$ is a value of the same type, and $fn(t) \subseteq s$ as well as $fn(v) \subseteq (s \cup s')$ always hold.
The rules of the big-step evaluation relation are given as follows:

\[\text{CAN} \quad s \vdash v \Downarrow \{\}v\]
\[\text{LOC} \quad s \cup a \vdash t \Downarrow (s_1)v \quad a \notin s\]
\[\text{EQU} \quad s \vdash t_1 \Downarrow (s_1)a_1 \quad s \cup s_1 \vdash t_2 \Downarrow (s_2)a_2 \quad s \vdash t_1 == t_2 \Downarrow (s_1 \cup s_2)\delta(a_1, a_2)\]
\[\text{IF1} \quad s \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow (s_1 \cup s_2)v\]
\[\text{IF2} \quad s \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow (s_1 \cup s_2)v\]
\[\text{PAIR} \quad s \vdash t_1 \Downarrow (s_1)v_1 \quad s \cup s_1 \vdash t_2 \Downarrow (s_2)v_2 \quad s \vdash (t_1, t_2) \Downarrow (s_1 \cup s_2)(v_1, v_2)\]
\[\text{CASE} \quad s \vdash t \Downarrow (s_1)(v_1, v_2) \quad s \cup s_1 \vdash t'[v_1/x_1, v_2/x_2] \Downarrow (s_2)v \quad s \vdash \text{case } t \text{ of } (x_1, x_2) \rightarrow t' \Downarrow (s_1 \cup s_2)v\]
\[\text{APP} \quad s \vdash t_1 \Downarrow (s_1)\lambda x. t' \quad s \cup s_1 \vdash t_2 \Downarrow (s_2)v' \quad s \cup s_1 \cup s_2 \vdash t'[v'/x] \Downarrow (s_3)v \quad s \vdash t_1 \ t_2 \Downarrow (s_1 \cup s_2 \cup s_3)v\]

### 4.2 Frame Stack Semantics of the Nu-Calculus

In this section, we present the frame stack semantics, which describes the operational semantics in terms of steps of an abstract machine. It takes many steps to evaluate a term and in this sense the frame stack semantics is finer grained than the big-step semantics. This notion of evaluation, more elementary in a certain sense, is useful for many proofs. Frame stacks were
introduced by Wright and Felleisen [16].

4.2.1 Typed Frame Stacks

The set of frame stacks \( \text{Stack} \) is given by:

\[
\text{Fs} \in \text{Stack} := \ Id \quad \text{(empty)}
\]

\[
\mid \text{Fs} \circ \mathcal{F} \quad \text{(non-empty)}
\]

where \( \mathcal{F} \) is an evaluation frame:

\[
\mathcal{F} := \text{if } [-] \text{ then } t \text{ else } t
\]

\[
\mid [-] == t
\]

\[
\mid a == [-]
\]

\[
\mid ([-], t)
\]

\[
\mid (v, [-])
\]

\[
\mid [-] t
\]

\[
\mid v [-]
\]

\[
\mid \text{case } [-] \text{ of } (x, x) \to t
\]

Basically, an evaluation frame is a flat term with a hole. A frame stack is a composition of evaluation frames and can be considered as a complex term with a hole. \( \mathcal{F}[t] \) is the term obtained by filling the hole of the evaluation frame \( \mathcal{F} \) with the term \( t \). The term \( \text{Fs}[t] \) is defined as follows:

\[
\text{Id}[t] := t
\]

\[
(\text{Fs} \circ \mathcal{F})[t] := \text{Fs}[\mathcal{F}[t]]
\]

The set \( fv(\text{Fs}) \) is the set containing all free variables of the constituent frames of a frame stack \( \text{Fs} \). Likewise, the set \( fn(\text{Fs}) \) is the set containing all free names of the constituent frames of a frame stack \( \text{Fs} \). \( \text{Fs} \) is variable-closed iff \( fv(\text{Fs}) \) is empty.

Not all frame stacks are well-typed. Therefore, we have to define a typing
relation for frame stacks. It takes the form:

\[ \vdash F_s : T \rightarrow T' \]

where \( F_s \) is a well-typed, variable-closed frame stack taking an argument of type \( T \) and returning a result of type \( T' \). This relation is defined recursively:

\[ \begin{align*}
\text{id} & \vdash \text{id} : T \rightarrow T \\
\vdash F_s : T' \rightarrow T'' & \text{fs} \{ x : T \} \vdash F[x] : T' \quad x \notin v(F) \\
\vdash F_s \circ F : T \rightarrow T'' & \text{fs}
\end{align*} \]

The set of well-typed variable-closed frame stacks taking an argument of type \( T \) is defined as:

\[ \text{Stack}_T := \{ F_s \in \text{Stack} \mid \exists T' \in \text{Typ.} \vdash F_s : T \rightarrow T' \} \]

### 4.2.2 Frame Stack Evaluation Relation

The frame-stack evaluation relation is defined by transitions:

\[ (F_s, t) \rightarrow (F_{s'}, t') \]

where \( t \) and \( t' \) are well-typed variable-closed expressions and \( F_s \) and \( F_{s'} \) are well-typed variable-closed frame stacks. The transitions are defined by:

*Case \( t \) is not a value:*

\[ \begin{align*}
(F_s, \text{if } t_1 \text{ then } t_2 \text{ else } t_3) & \rightarrow (F_s \circ (\text{if } [-] \text{ then } t_2 \text{ else } t_3), t_1) \\
(F_s, t_1 \text{ == } t_2) & \rightarrow (F_s \circ ([ - ] \text{ == } t_2), t_1) \\
(F_s, (t_1, t_2)) & \rightarrow (F_s \circ ([ - ], t_2), t_1) \\
(F_s, t_1 t_2) & \rightarrow (F_s \circ ([ - ] t_2), t_1) \\
(F_s, \text{case } t \text{ of } (x_1, x_2) \rightarrow t') & \rightarrow (F_s \circ \text{ case } [-] \text{ of } (x_1, x_2) \rightarrow t'), t) \\
(F_s, \nu a.t) & \rightarrow (F_s, t) \text{ if } a \notin fn(F_s)
\]
Case \( t \) is a value:

\[
\langle \mathcal{F} \circ (\text{if } [-] \text{ then } t_1 \text{ else } t_2), v \rangle \rightarrow \langle \mathcal{F}, t_1 \rangle \quad \text{if } v = \text{true} \\
\langle \mathcal{F} \circ (\text{if } [-] \text{ then } t_1 \text{ else } t_2), v \rangle \rightarrow \langle \mathcal{F}, t_2 \rangle \quad \text{if } v = \text{false} \\
\langle \mathcal{F} \circ ([-] == t), a \rangle \rightarrow \langle \mathcal{F} \circ (a == [-]), t \rangle \\
\langle \mathcal{F} \circ (a == [-]), a' \rangle \rightarrow \langle \mathcal{F} \circ \delta(a,a') \rangle \\
\langle \mathcal{F} \circ ([], t), v \rangle \rightarrow \langle \mathcal{F} \circ (v, [-]), t \rangle \\
\langle \mathcal{F} \circ (v, [-]), v' \rangle \rightarrow \langle \mathcal{F}, (v, v') \rangle \\
\langle \mathcal{F} \circ ([], t), v \rangle \rightarrow \langle \mathcal{F} \circ (v [-]), t \rangle \\
\langle \mathcal{F} \circ (v [-]), v' \rangle \rightarrow \langle \mathcal{F}, t[v'/x] \rangle \quad \text{if } v = \lambda x.t \\
\langle \mathcal{F} \circ \text{(case } [-] \text{ of } (x_1, x_2) \rightarrow t), (v_1, v_2) \rangle \rightarrow \langle \mathcal{F}, t[v_1/x_1, v_2/x_2] \rangle
\]

In contrast to the big-step semantics, no additional set of names is needed. We can now clarify the relationship between the frame stack and the big-step semantics.

**Remark 18.** The evaluation relations coincide. For all frame stacks \( \mathcal{F}s \), terms \( t \) and values \( v \), there exists a set of names \( s \), such that the following equivalence holds:

\[
\langle \mathcal{F}, t \rangle \rightarrow^* \langle \text{Id}, v \rangle \iff \left( \text{fn}(\mathcal{F}s) \cup \text{fn}(t) \right) \vdash \mathcal{F}s[t] \Downarrow (s)v
\]

### 4.2.3 Termination

We define a notion of termination for the frame stack semantics. It might seem strange to define a notion of termination in a setting of the nu-calculus, where no non-terminating terms exist. However, this simplifies forthcoming proofs about the CPS translation. Define:

\[
\langle \mathcal{F}, t \rangle \Downarrow \text{true} \iff \langle \mathcal{F}, t \rangle \rightarrow^* \langle \text{Id}, \text{true} \rangle
\]

where \( \rightarrow^* \) is the reflexive-transitive closure of \( \rightarrow \).
4.3 Big-Step Semantics of $\lambda\nu$

In this section we introduce the other calculus presented in this essay: $\lambda\nu$. There are considerable differences on the level of semantics between $\lambda\nu$ and the $\mu$-calculus. In his original paper [8], Odersky presented the operational semantics of $\lambda\nu$ as a term rewriting system, whereas we give a call-by-name big-step semantics. Further deviations from the original $\lambda\nu$ are discussed in Section 4.3.3.

4.3.1 Canonical Forms

We need to change our notion of value, to account for the call-by-name evaluation. The set $Can$ of canonical forms of $\lambda\nu$ is defined as follows:

$$c \in Can \ ::= \ a \quad \text{(atomic names)}$$
$$| \ \nu a.a \quad \text{(New)}$$
$$| \ \text{true} \quad \text{(boolean value true)}$$
$$| \ \text{false} \quad \text{(boolean value false)}$$
$$| \ (t,t) \quad \text{(pairing)}$$
$$| \ \lambda x.t \quad \text{(function abstraction)}$$

We deliberately add the term $\nu a.a$ to the canonical forms and call it “New”. Since terms are implicitly identified up to alpha-equivalence, $\nu a.a$ is equal to $\nu a'.a'$ for any $a' \in A$.

Canonical forms can be regarded as the call-by-name analogue of values. We define well-typed variable-closed canonical forms as in Section 4.1.1:

$$Can_T := \{ c \in Can \mid \emptyset \vdash c : T \}$$

It is desirable to include $\nu a.a$ in the equality check. Therefore we change the
definition of $\delta$ to $\delta_c$:

$$
\delta_c(c, c') := \begin{cases} 
\text{true} & \text{if } c = c' \\
\text{false} & \text{if } c \neq c'
\end{cases}
$$

where $c, c' \in (A \cup \{\nu a.a\})$.

### 4.3.2 Big-Step Evaluation Relation

The big-step evaluation relation for $\lambda \nu$ takes the following form:

$$
t \Downarrow c
$$

where $t$ is a well-typed variable-closed term and $c$ is a well-typed variable-closed canonical form of the same type. We proceed with the definition of the evaluation relation:

- **CAN**
  
  $$
c \Downarrow c
  $$

- **CASE**
  
  $$
t \Downarrow (t_1, t_2) \quad t'[t_1/x_1, t_2/x_2] \Downarrow c \quad \text{case } t \text{ of } (x_1, x_2) \rightarrow t' \Downarrow c
  $$

- **IF1**
  
  $$
  \begin{align*}
  t_1 & \Downarrow \text{true} \\
  t_2 & \Downarrow c \\
  \text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \Downarrow c
  \end{align*}
  $$

- **IF2**
  
  $$
  \begin{align*}
  t_1 & \Downarrow \text{false} \\
  t_3 & \Downarrow c \\
  \text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \Downarrow c
  \end{align*}
  $$

- **EQU**
  
  $$
  t_1 \Downarrow c_1 \quad t_2 \Downarrow c_2 \quad c_1, c_2 \in (A \cup \{\nu a.a\}) \\
  t_1 == t_2 \Downarrow \delta_c(c_1, c_2)
  $$

- **APP**
  
  $$
  \begin{align*}
  t_1 & \Downarrow \lambda x. t' \\
  t_2 & \Downarrow c \\
  t'[t_2/x] & \Downarrow c
  \end{align*}
  $$

- **NUA**
  
  $$
  t \Downarrow a \quad \nu a.t \Downarrow \nu a.a
  $$

- **NUNAME**
  
  $$
  \begin{align*}
  t & \Downarrow c \\
  c & \in \{a', \text{true}, \text{false}, \nu a.a\} \\
  \nu a.t & \Downarrow c
  \end{align*}
  $$

- **NULAM**
  
  $$
  \begin{align*}
  t & \Downarrow \lambda x. t' \\
  \nu a.t & \Downarrow \lambda x. \nu a.t
  \end{align*}
  $$

- **NUPAIR**
  
  $$
  \begin{align*}
  t & \Downarrow (t_1, t_2) \\
  \nu a.t & \Downarrow (\nu a.t_1, \nu a.t_2)
  \end{align*}
  $$
4.3.3 Deviations from the Original $\lambda \nu$

Instead of giving the operational semantics in terms of a rewriting system—as it is done in Odersky’s original paper [8]—we present the operational semantics as a big-step evaluation relation. The big-step semantics is more restrictive: Only closed expressions are evaluated and terms do not evaluate under lambda abstractions or pairs. However, there is no conceptual difference between the presentations, because the big-step evaluation relation is included in the reflexive-transitive closure of the rewriting system. Our presentation is therefore a reformulation rather than a deviation.

The major difference between Odersky’s system and ours lies in the treatment of the term $\nu a.a$. Motivated by the forthcoming denotational semantics based on nominal sets, we include $\nu a.a$ in the set of canonical forms. Furthermore, we allow $\nu a.a$ to appear in equality tests, making the following evaluations valid:

\[
\begin{align*}
a &= \nu a.a \Downarrow \text{false} \\
\nu a.a &= \nu a.a \Downarrow \text{true}
\end{align*}
\]

Odersky, on the other hand, considers $\nu a.a$ to be stuck. That means $\nu a.a$ is not a value, but cannot be further rewritten either. Therefore, the term $\nu a.a \equiv \nu a.a$ is stuck as well, even though it is typeable. Odersky’s typing system coincides with ours and is given in Figure 5 of his original paper [8].

Apart from the treatment of $\nu a.a$, the two operational semantics are similar in spirit, if not in form.

4.4 Comparison

We demonstrated the operational semantics of the nu-calculus and $\lambda \nu$. Even though the two languages have the same syntax, there are significant differences between them. We discuss these differences and illustrate them by
examples.

First of all, the nu-calculus is call-by-value while \( \lambda \nu \) is call-by-name. There is a rich literature concerning the relationship between these two evaluation strategies (see for example [13]), so we omit a discussion here.

A more interesting difference is the treatment of the name restriction construct \( \nu \). The nu-calculus uses ML-like local state and names are created dynamically. In contrast to that, \( \lambda \nu \) does not have local state and name restrictions are “pushed into terms” by rewriting. In this sense, \( \lambda \nu \) can be considered “purer” than the nu-calculus; features characteristic to pure languages (e.g. Milner’s “Context Lemma” [6]) hold for \( \lambda \nu \), but not for the nu-calculus. We highlight this by exhibiting terms that evaluate differently in the two languages.

**Example 19.** We already know the following term from the last section:

\[
\nu a. a = \nu a.a
\]

- Nu-calculus: false
- \( \lambda \nu \): true

**Example 20.** Of course, there are also similarities in the treatment of names. In both calculi, the \( \nu \)-construct corresponds to new name generation:

\[
\nu a. \nu a'.(a == a')
\]

- Nu-calculus: false
- \( \lambda \nu \): false

**Example 21.** This example illustrates a different behaviour of name restriction in combination with lambda-abstractions. Even though \( a' \) is free, the given term is valid because we evaluate variable-closed but possibly name-open terms.

\[
\left((\nu a.\lambda x.a')a\right) = \nu a.a
\]
• Nu-calculus: false
• \( \lambda\nu \): true

**Example 22.** \( \lambda\nu \) and the nu-calculus treat the interaction between name restriction and pairs differently:

\[
\text{case } (\nu a, \nu a' (a, a')) \text{ of } (x_1, x_2) \rightarrow (x_1 == x_2)
\]

• Nu-calculus: false
• \( \lambda\nu \): true
Chapter 5

Denotational Semantics of $\lambda\nu$

This chapter defines a denotational semantics based on nominal sets for $\lambda\nu$. Through the use of nominal sets, our denotational semantics is simpler than the original one proposed by Odersky [8]. It is computationally adequate, but probably not fully abstract. Our approach is based on the denotational semantics of Nominal System T, which is briefly sketched in Pitts’ paper [11].

The semantics of $\lambda\nu$ is interpreted in a framework based on nominal sets. In particular, types denote nominal restriction sets, as defined in Chapter 2. A $\lambda\nu$ term denotes a finitely supported function from the nominal restriction set denoting its typing environment to the nominal restriction set denoting its type.

5.1 Denotations of Types

We define the denotation of each possible kind of type: booleans, names, products and function types.

- **Booleans**: The ground type Bool is interpreted as the trivial nominal restriction set having two elements $true$ and $false$:

  $$[[\text{Bool}]] := \{true, false\}$$
Permutation action and restriction operation are given as in Section 2.3.1.

- **Names**: Names are interpreted as the nominal restriction set of atoms from Section 2.3.2 with permutation action and restriction operation defined there.

  \[
  [\text{Name}] := A[\text{New}]
  \]

  The additional element \text{New} can be considered as the interpretation of \(\nu a.a\). We will justify this later.

- **Products**: The interpretation of a product is the product of two nominal restriction sets, as defined in Section 2.3.3:

  \[
  [T_1 \times T_2] := [T_1] \times [T_2]
  \]

- **Functions**: A function type denotes the set of finitely supported functions between two nominal restriction sets. Again, permutation and restriction are defined in the background chapter, specifically in Section 2.3.4.

  \[
  [T_1 \rightarrow T_2] := [T_1] \rightarrow_{\text{fs}} [T_2]
  \]

- **Typing environments**: To give a denotation to a typing environment like \(\Gamma = \{t_1 : T_1,.., t_n : T_n\}\), we take the product of its constituent types:

  \[
  [\Gamma] := [T_1] \times .. \times [T_n]
  \]

### 5.2 Denotations of Expressions

An expression denotes a finitely supported function from its typing environment to its type:

\[
[\Gamma \vdash t : T] \in ([\Gamma] \rightarrow_{\text{fs}} [T])
\]
Both the typing environment and the resulting type denote nominal restriction sets. Sometimes we omit the typing information and write \([t]\) instead of \([\Gamma \vdash t : T]\).

It is convenient to introduce an additional semantic function \(if_X : [\text{Bool}] \times X \times X \to X\) for conditionals:

\[
if_X(b, x_1, x_2) := \begin{cases} 
x_1 & \text{if } b = \text{true} \\
x_2 & \text{if } b = \text{false}
\end{cases}
\]

Denotations of expressions are defined by recursion on the typing rules:

<table>
<thead>
<tr>
<th>Typing Rule</th>
<th>Denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{var} \quad 1 \leq i \leq n \quad x_1 : T_1, \ldots, x_n : T_n \vdash x_i : T_i)</td>
<td>(\pi_i)</td>
</tr>
<tr>
<td>(\text{name} \quad a \in A \quad \Gamma \vdash a : \text{Name})</td>
<td>(\lambda g.a)</td>
</tr>
<tr>
<td>(\text{loc} \quad \Gamma \vdash t : T \quad a \in A \quad \Gamma \vdash \nu a.t : T)</td>
<td>(f)</td>
</tr>
<tr>
<td>(\text{equ} \quad \Gamma \vdash t_1 : \text{Name} \quad \Gamma \vdash t_2 : \text{Name} \quad \Gamma \vdash t_1 == t_2 : \text{Bool})</td>
<td>(f_1, f_2 \quad \lambda g.\delta_c(f_1(g), f_2(g)))</td>
</tr>
<tr>
<td>(\text{tbool} \quad \Gamma \vdash \text{true} : \text{Bool})</td>
<td>(\lambda g.\text{true})</td>
</tr>
<tr>
<td>(\text{fbool} \quad \Gamma \vdash \text{false} : \text{Bool})</td>
<td>(\lambda g.\text{false})</td>
</tr>
</tbody>
</table>
We use the function $\delta_c$ from Section 4.3.1 in the if-rule, allowing for a slight confusion between operational and denotational values.

The conclusion of the loc-rule $a \triangleright f$ is the finitely supported function $f$ applied to the restriction operation defined in Section 2.3.4. We can now show how
the term νa.a relates to the element New ∈ [[Name]]:

\[
\begin{align*}
[[\nu a.a]] & \quad \text{loc-rule} \quad a \backsim [a] \\
& \quad \text{name-rule} \quad a \backsim (\lambda g.a) \\
& \quad \text{restriction on functions} \quad a' \backsim ((a \ a') \cdot (\lambda g.a)) \\
& \quad \text{swapping on terms} \quad a' \backsim (\lambda g.a') \\
& \quad \text{η-reduction} \quad \lambda g'.(a' \backsim (\lambda g.a')) \ g' \\
& \quad \text{application} \quad \lambda g'.a' \backsim ((\lambda g.a') \ g') \\
& \quad \text{restriction on atoms} \quad \lambda g'.New
\end{align*}
\]

5.3 Properties

The recursive definition above only defines a relation \((t, f)\), where \(t\) is a typeable term and \(f\) is a function. We have to show that this relation fulfils the properties we expect from a denotational semantics. The central property is the computational adequacy: if two terms have the same denotation then they are contextually equivalent. In general, two terms are contextually equivalent, if they have the same operational behaviour in every context. For brevity, the proofs are only sketched. Statements that have not been proved by the author are marked as conjectures.

**Conjecture 23 (Well-definedness).** For every typing judgement \(\Gamma \vdash t : T\), its unique denotation is a well-defined finitely supported function from \([\Gamma]\) to \([T]\):

\([\Gamma \vdash t : T] \in [\Gamma] \rightarrow_{fs} [T]\)

**Proof sketch.** This proof requires several lemmas:

- The denotation relation is functional.

- The denotation relation always returns a finitely supported function with the right domain and codomain.

- The denotation relation respects alpha-equivalence of names and vari-
We have proved the first two bullet points by induction on the typing rules, while the third bullet point can be proved by Pitts’ alpha-structural induction and recursion principles [10].

**Proposition 24** (Compositionality). *If two terms have the same denotation, then their denotation is equal in every context. Let any well-typed context $C[-]$ be given, it follows that:*

$$[[t]] = [[t']] \implies [C[t]] = [C[t']]$$

*Proof sketch.* By induction on the structure of $C[-]$. For the proof of soundness, we need a lemma about substitutions.

**Lemma 25** (Substitution lemma). *Suppose* $$\Gamma \vdash t : T$$ $$\Gamma, x : T \vdash t' : T'$$

*then for all* $g \in [[\Gamma]]$ *we have*

$$[[t'[t/x]]](g) = [[t']](g, [[t]](g))$$

*Proof sketch.* By induction on the structure of $t'$.

It is possible to prove soundness with the help of the substitution lemma.

**Proposition 26** (Soundness). *If a given term evaluates to a canonical form, then they have the same denotation:*

$$t \Downarrow c \implies [[t]] = [[c]]$$

*Proof sketch.* By induction on the evaluation relation $\Downarrow$.

**Proposition 27** (Adequacy). *If the denotation of a closed term of type $\text{Bool}$
is true, then it evaluates to true. Suppose:

$$\emptyset \vdash t : \text{Bool}$$

then

$$[t] = [\text{true}] \implies t \Downarrow \text{true}$$

Proof sketch. By a logical relation argument. Details can be found in the appendix.

We are now able to prove computational adequacy.

**Theorem 28** (Computational adequacy). If two typeable terms $t$ and $t'$ have the same denotation, then they are contextually equivalent.

$$[t] = [t'] \implies t \equiv_{\text{ctx}} t'$$

Proof sketch. By combining compositionality, soundness and adequacy.

Computational adequacy enables us to prove the contextual equivalence of many terms by checking if their denotations are equal. For example, we obtain the following contextual equivalences:

$$\nu a. t \equiv_{\text{ctx}} t \quad \text{if } a \notin \text{fn}(t)$$

$$\nu a.\nu a'. t \equiv_{\text{ctx}} \nu a'.\nu a. t$$
Chapter 6

CPS Translation

Translating the nu-calculus into $\lambda\nu$ is the main goal of this essay. The details are given in this chapter. The translation is done in continuation-passing-style (CPS), which allows us, among other things, to switch from call-by-value to call-by-name evaluation strategies. Continuations were developed by the compilers community in the early 1970s and CPS-translations in the context of the lambda-calculus were introduced by Plotkin [13]. Our CPS translation is inspired by Shinwell’s and Pitts’ denotational semantics for FreshML [14] and is closely related to Moggi’s monad-based approach for modelling semantics [7].

6.1 Definition

We define the CPS translation on types, values, terms, frame stacks and typing environments of the nu-calculus. It is helpful for the translation to consider terms and types of the nu-calculus and $\lambda\nu$ to be different. However, this is not the case in reality. Terms and types of both calculi are defined in Chapter 3.
6.1.1 Types

A translated type is written as $\overline{T}$, where $T$ is a nu-calculus type and $\overline{T}$ is a $\lambda\nu$ type. For notational convenience we define for each type $T$:

$$T^\perp := T \to \text{Bool}$$

The translation is given by:

$$\overline{\text{Name}} := \text{Name}$$
$$\overline{\text{Bool}} := \text{Bool}$$
$$\overline{T_1 \times T_2} := \overline{T_1} \times \overline{T_2}$$
$$\overline{T_1 \to T_2} := \overline{T_1} \to \overline{T_2}^\perp = \overline{T_1} \to (\overline{T_2} \to \text{Bool}) \to \text{Bool}$$

6.1.2 Values

We write $v^\bullet$ for a nu-calculus value $v$ translated into a $\lambda\nu$ canonical form. Values and canonical forms are defined in Section 4.1.1 and Section 4.3.1. We define:

$$x^\bullet := x$$
$$a^\bullet := a$$
$$\text{true}^\bullet := \text{true}$$
$$\text{false}^\bullet := \text{false}$$
$$(v_1, v_2)^\bullet := (v_1^\bullet, v_2^\bullet)$$
$$(\lambda x.t)^\bullet := \lambda x.t^\bullet$$
6.1.3 Terms

The last line of the translation on values already uses the translation on terms. It is denoted by $t^\circ$, for a nu-calculus term $t$, and we define it by:

$$
v^\circ := \lambda k. k \ v^\bullet \quad \text{for } k \notin fv(v^\bullet)
$$

$$(\nu a. t)^\circ := \nu a. t^\circ
$$

$$(t_1 == t_2)^\circ := \lambda k. \ t_1^\circ \left( \lambda a_1. t_2^\circ ( \lambda a_2. \text{if } a_1 == a_2 \text{ then } (k \ true) \text{ else } (k \ false) \right)
$$

for $k \notin fv(t_1^\circ, t_2^\circ, a_1, a_2)$ and $a_1 \notin fv(t_2^\circ, a_2)$

$$\text{(if } t_1 \text{ then } t_2 \text{ else } t_3)^\circ := \lambda k. \ t_1^\circ \left( \lambda b_1. \text{if } b_1 \text{ then } (t_2^\circ k) \text{ else } (t_3^\circ k) \right)
$$

for $k \notin fv(t_1^\circ, t_2^\circ, t_3^\circ, b_1)$ and $b_1 \notin fv(t_2^\circ, t_3^\circ)$

$$\text{(for } k \notin fv(t_1^\circ, t_2^\circ, v_1, v_2) \text{ and } v_1 \notin fv(t_2^\circ, v_2)
$$

$$(\text{case } t \text{ of } (x_1, x_2) \rightarrow t')^\circ := \lambda k. t^\circ \left( \lambda p. \text{case } p \text{ of } (x_1, x_2) \rightarrow t'^\circ k \right)
$$

for $k \notin fv(t^\circ, t'^\circ, p, x_1, x_2)$ and $p \notin fv(t'^\circ, x_1, x_2)$

$$\text{(for } k \notin (t_1^\circ, t_2^\circ, f, v) \text{ and } f \notin fv(t_2^\circ, v)
$$

6.1.4 Frame Stacks

For the translation of frame stacks (see Section 4.2.1) into $\lambda \nu$ terms, we use the same notation as for values.

$$Id^\bullet := \lambda x. x
$$

$$(\mathcal{F}s \circ \mathcal{F})^\bullet := \lambda x. (\mathcal{F}[x])^\circ \ (\mathcal{F}s^\bullet)
$$
6.1.5 Typing Environments

A typing environment $\Gamma = \{ x_1 : T_1, \ldots, x_n : T_n \}$ is translated by:

$$\overline{\Gamma} := \{ x_1 : \overline{T_1}, \ldots, x_n : \overline{T_n} \}$$

6.2 Properties of the Translation

The translation from the nu-calculus into $\lambda \nu$ is defined syntactically. In this section, we give properties of the translation, to show that it is semantically sensible. We conclude with the main theorem of this essay: the adequacy of the translation for contextual equivalence. It states that if two translated terms are contextually equivalent in $\lambda \nu$, then the original terms are contextually equivalent in the nu-calculus.

It is sufficient to exclusively consider frame stacks mapping to the type Bool, because we are interested in contextual equivalence only. Hence we redefine:

$$Stack_T := \{ F_s \in Stack \mid \vdash F_s : T \rightarrow Bool \}$$

**Conjecture 29.** For every typing environment $\Gamma$, type $T$, value $v$, term $t$ and frame stack $F_s$, the following implications hold:

$$\begin{align*}
\Gamma \vdash v : T & \Rightarrow \overline{\Gamma} \vdash v^\ast : \overline{T} \\
\Gamma \vdash t : T & \Rightarrow \overline{\Gamma} \vdash t^o : \overline{T}^\perp \\
\vdash F_s : T \rightarrow Bool & \Rightarrow \emptyset \vdash F_s^\ast : \overline{T}^\perp
\end{align*}$$

**Proof sketch.** By induction on the corresponding typing rules.

With the above implications it should be clearer why the definition of the translation is sensible.

In the monadic interpretation, $v^\ast$ can be considered as arrow into a value, $t^o$ can be considered as arrow into a continuation monad and $F_s^\ast$ can be
Conjecture 30. We list some more important technical properties:

\[
\begin{align*}
fv(v^*) &= fv(v) \\
fn(v^*) &= fn(v) \\
fv(t^0) &= fv(t) \\
fn(t^0) &= fn(t)
\end{align*}
\]

\[
\begin{align*}
(v'[v/x])[v^*/x] &= v'[v^*/x] \\
(t[v/x])[v^*/x] &= t[v^*/x]
\end{align*}
\]

Proof sketch. We prove each column by simultaneous induction on the

It is now possible to prove that the translation is sound.

Conjecture 31 (Soundness of the translation for termination). For all \( t \in \text{Term}_T \) and \( \mathcal{F}s \in \text{Stack}_T \) it holds that:

\[
(\mathcal{F}s, t) \downarrow \text{true} \implies t^0 \mathcal{F}s^* \downarrow \text{true}
\]

Proof sketch. By case analysis on the frame stack evaluation relation.

With the help of a logical relations argument, very similar to the one given
in Pitts’ paper concerning contextual equivalence in a subset of ML [9], we
are able to prove the backward direction of Conjecture 31.

Conjecture 32 (Adequacy of the translation for termination). For all \( t \in \text{Term}_T \) and \( \mathcal{F}s \in \text{Stack}_T \), it holds that:

\[
\begin{align*}
t^0 \mathcal{F}s^* \downarrow \text{true} \implies (\mathcal{F}s, t) \downarrow \text{true}
\end{align*}
\]

Proof sketch. The proof probably works with a logical relation for expressions
\( t \triangleleft t' : T \), where \( t \in \text{Term}_{\mathcal{T} \downarrow} \) and \( t' \in \text{Term}_T \). This relation is defined in
terms of a relation for frame stacks \( t \blacktriangleright \mathcal{F}s : T \rightarrow \text{Bool} \), where \( t \in \text{Term}_{\mathcal{T} \downarrow} \) and \( \mathcal{F}s \in \text{Stack}_T \). Similarly, the relation on frame stacks is defined in terms
of a relation for values \( t \blacktriangleright v : T \) with \( t \in \text{Term}_T \) and \( v \in Val_T \).

The adequacy of the translation can be proved by using Conjecture 32.

Conjecture 33 (Adequacy of the translation for contextual equivalence).
For all terms \( t, t' \in \text{Term}_T \), if their translation is contextually equivalent,
then the terms themselves are contextually equivalent:

\[ \Gamma \vdash t \cong t' : \overline{T} \perp \perp \implies \Gamma \vdash t \cong t' : T \]

Proof sketch. Probably follows from Lemma 32.

A consequence of Conjecture 33 is that \(\lambda\nu\) is more general than the nu-calculus, because contextual equivalence in \(\lambda\nu\) determines contextual equivalence in the nu-calculus.
Chapter 7

Conclusions

We presented the nu-calculus and $\lambda\nu$ together with a translation between them. The two calculi were given a joint syntax and typing relation and additionally operational semantics of different types were defined. A core part of the essay was the denotational semantics for $\lambda\nu$, formulated in terms of nominal restriction sets. It turned out to be pleasingly simple. We translated the nu-calculus into $\lambda\nu$ in a semantically sensible way. Thereby we demonstrated that $\lambda\nu$ can be considered to be more general than the nu-calculus.

7.1 Potential Future Work

Conjecture 33, i.e. the full proof of the adequacy of the CPS translation for contextual equivalence, remains to be proved. We hope that the proof will be straightforward, because we have quite a clear idea about how to proceed.

The problem of full abstraction of the translation, that is the implication from right to left in Conjecture 33, is also worth investigating. We are not sure whether it holds and it is desirable to have a proof or a counter-example.

Both the nu-calculus and $\lambda\nu$ are very minimalistic languages. It would be interesting to see whether our statements still hold when we add more features. Obvious extensions are recursive functions and recursively defined datatypes.
Appendix A

Proof Details

A.1 Proof of Proposition 27

Definition 34. For every type $T$, define the logical relation 

$$
\triangleright_T \subseteq [T] \times Term_T
$$

by 

\begin{align*}
    d \triangleright_{\text{Bool}} t & \iff t \Downarrow d \\
    d \triangleright_{\text{Name}} t & \iff t \Downarrow d \\
    d \triangleright_{T_1 \times T_2} t & \iff \pi_1 t \triangleright_{T_1} \pi_1(d) \land \pi_2 t \triangleright_{T_2} \pi_2(d) \\
    d \triangleright_{T_1 \rightarrow T_2} t & \iff \forall d' \in [T_1], t' \in Term_{T_1}. d' \triangleright_{T_1} t' \Rightarrow d(d') \triangleright_{T_2} t t'
\end{align*}

In the definition, we allow a slight confusion between syntactic and semantic constructs. For example we identify the syntactic value true $\in$ Term_{Bool} with the semantic element true $\in [\text{Bool}]$. The same is done for names, where we identify for example New with $\nu a.a$.

Additionally, we introduced new syntactic constructs $\pi_1$ and $\pi_2$. They can
be considered as abbreviations:

$$\pi_1 t := \text{case } t \text{ of } (x_1, x_2) \rightarrow x_1$$

$$\pi_2 t := \text{case } t \text{ of } (x_1, x_2) \rightarrow x_2$$

With these definitions, we get the following derivable evaluation rules:

$$\begin{array}{c}
\text{PI1} & t \Downarrow (t_1, t_2) & t_1 \Downarrow c & \text{PI2} & t \Downarrow (t_1, t_2) & t_2 \Downarrow c \\
\hline
\pi_1 t \Downarrow c & & & \pi_2 t \Downarrow c & &
\end{array}$$

Again, we allow notational overlapping, because the semantic functions of the first and second projections are denoted by $\pi_1$ and $\pi_2$ as well.

**Lemma 35.** For every type $T$, element $d \in \llbracket T \rrbracket$, terms $t, t' \in \text{Term}_T$ and canonical form $c \in \text{Can}_T$, it holds that:

$$d \triangleleft_T t \ & \forall c.(t \Downarrow c \Rightarrow t' \Downarrow c) \implies d \triangleleft_T t'$$

*Proof sketch.* By induction on the structure of $T$.

**Definition 36.** For every typing environment $\Gamma = \{t_1 : T_1, \ldots, t_n : T_n\}$, define the set of $\Gamma$-pairs as:

$$\{(g, p) \mid g \in \llbracket T_1 \rrbracket \times \ldots \times \llbracket T_n \rrbracket \ & \ p \in \text{Term}_{T_1} \times \ldots \times \text{Term}_{T_n}\}$$

Define also for a $\Gamma$-pair $(g, p)$, where $g = (g_1, \ldots, g_n)$ and $p = (p_1, \ldots, p_n)$:

$$g \triangleleft_\Gamma p \iff \forall i \in \{1, \ldots, n\}. g_i \triangleleft_{T_i} p_i$$

**Lemma 37** (Fundamental property of the logical relation). Let a term $t$ such that $\Gamma \vdash t : T$ holds with $\Gamma = \{t_1 : T_1, \ldots, t_n : T_n\}$ be given. Then for all $\Gamma$-pairs $(g, p)$ it holds that:

$$g \triangleleft_\Gamma p \implies \llbracket \Gamma \vdash t : T \rrbracket (g) \triangleleft_T t[p/x]$$

where $t[p/x]$ is an abbreviation for $t[p_1/x_1, \ldots, p_n/x_n]$. For $\Gamma = \emptyset$ we obtain
the special case:

\[ [t] \trianglelefteq_T t \]

**Proof.** By induction on the typing rules. We only prove the cases for conditionals and for name restrictions in detail, the other cases work similarly.

- **if-rule:** Let a typeable term \( \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \) and a \( \Gamma \)-pair \( (g, p) \) be given. By the induction hypothesis, we know:

\[ [t_1](g) \trianglelefteq _{\text{Bool}} t_1[p/x] \quad \text{(A.1)} \]
\[ [t_2](g) \trianglelefteq_T t_2[p/x] \quad \text{(A.2)} \]
\[ [t_3](g) \trianglelefteq_T t_3[p/x] \quad \text{(A.3)} \]

If \( [t_1](g) = \text{true} \):

\[
\begin{align*}
\text{Definition 34} & \quad \text{true} \trianglelefteq _{\text{Bool}} t_1[p/x] \\
\text{IF1-rule of } \Downarrow & \quad t_1[p/x] \Downarrow \text{true} \\
\forall c. t_2[p/x] \Downarrow c & \Rightarrow \text{if } t_1[p/x] \text{ then } t_2[p/x] \text{ else } t_3[p/x] \Downarrow c \\
[t_2](g) \trianglelefteq_T \text{if } t_1[p/x] \text{ then } t_2[p/x] \text{ else } t_3[p/x]
\end{align*}
\]

and because of

\[
[t_2](g) = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \text{ }(g)
\]

\[
\text{if } t_1[p/x] \text{ then } t_2[p/x] \text{ else } t_3[p/x] = \text{if } t_1 \text{ then } t_2 \text{ else } t_3[p/x]
\]

we know that

\[
[t_1 \text{ then } t_2 \text{ else } t_3](g) \trianglelefteq_T \text{if } t_1 \text{ then } t_2 \text{ else } t_3[p/x]
\]

which was to show.
If $\llbracket t_1 \rrbracket(g) = false$:

- \[ (A.1) \Rightarrow false \triangleleft_{\text{Bool}} t_1[p/x] \]
- \[ t_1[p/x] \downarrow false \]
- \[ \text{IF2-rule of } \downarrow \Rightarrow \forall c. t_3[p/x] \downarrow c \Rightarrow if \ t_1[p/x] \text{ then } t_2[p/x] \text{ else } t_3[p/x] \downarrow c \]

\[ \llbracket t_3 \rrbracket(g) \triangleleft_T \text{ if } t_1[p/x] \text{ then } t_2[p/x] \text{ else } t_3[p/x] \]

and because of

\[ \llbracket t_3 \rrbracket(g) = if_T (\llbracket t_1 \rrbracket(g), \llbracket t_2 \rrbracket(g), \llbracket t_3 \rrbracket(g)) = \llbracket \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rrbracket[g] \]

we know that

\[ \llbracket \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rrbracket(g) \triangleleft_T (\text{if } t_1 \text{ then } t_2 \text{ else } t_3)(p/x) \]

which was to show.

- loc-rule: Let a typeable term $\Gamma \vdash \nu a.t : T$ and a $\Gamma$-pair $(g,p)$ be given.

By the induction hypothesis, we know:

\[ \llbracket t \rrbracket(g) \triangleleft_T t[p/x] \]

And we also know:

\[ \llbracket \nu a.t \rrbracket(g) = (a \smallsetminus \llbracket t \rrbracket(g)) \triangleleft_{\text{Type}} a \smallsetminus t \llbracket g \rrbracket \]

\[ (\nu a.t)[p/x] = \nu a.t[p/x] \]

So it remains to show that:

\[ \llbracket t \rrbracket(g) \triangleleft_T t[p/x] \Rightarrow a \smallsetminus \llbracket t \rrbracket(g) \triangleleft_T \nu a.t[p/x] \]

This is proved in Lemma 40, which requires two other lemmas in its proof.
Lemma 38. For all $a, t, c$ it holds that:

$$\nu a. \pi_i t \Downarrow c \implies \pi_i \nu a. t \Downarrow c \quad \text{for } i \in \{1, 2\}$$

Proof sketch. By case analysis on the structure of $c$.

Lemma 39. For all $a, t_1, t_2, c$ with $a \neq t_2$, it holds that:

$$\nu a. (t_1 t_2) \Downarrow c \implies (\nu a. t_1) t_2 \Downarrow c$$

Proof sketch. By case analysis on the structure of $c$.

Lemma 40. For all $\Gamma \vdash t : T$ and $\Gamma$-pairs $(g, p)$:

$$[ t ](g) \triangleleft_T t[p/x] \implies a \setminus [ t ](g) \triangleleft_T \nu a. t[p/x]$$

Proof sketch. By induction on the structure of $T$, using Lemma 38 for product types and Lemma 39 for function types.

• The other cases are proved like the if-rule.

With the fundamental property of the logical relation, we are able to prove adequacy:

Let

$$[ t ] = true$$

be given. By Lemma 37 we know that:

$$[ t ] \triangleleft_{Bool} t$$

holds and by the definition of $\triangleleft_{Bool}$ it follows that:

$$t \Downarrow true$$
Bibliography


