Relating Two Semantics of Locally Scoped Names

Steffen Lösch and Andrew M. Pitts

University of Cambridge Computer Laboratory
Cambridge CB3 0FD, UK

Abstract

The operational semantics of programming constructs involving locally scoped names typically makes use of stateful *dynamic allocation*: a set of currently-used names forms part of the state and upon entering a scope the set is augmented by a new name bound to the scoped identifier. More abstractly, one can see this as a transformation of local scopes by expanding them outward to an implicit top-level. By contrast, in a neglected paper from 1994, Odersky gave a stateless lambda calculus with locally scoped names whose dynamics contracts scopes inward. The properties of 'Odersky-style' local names are quite different from dynamically allocated ones and it has not been clear, until now, what is the expressive power of Odersky's notion. We show that in fact it provides a direct semantics of locally scoped names from which the more familiar dynamic allocation semantics can be obtained by continuation-passing style (CPS) translation. More precisely, we show that there is a CPS translation of typed lambda calculus with dynamically allocated names (the Pitts-Stark $\boldsymbol{\nu}$-calculus) into Odersky's $\boldsymbol{\lambda\nu}$-calculus which is computationally adequate with respect to observational equivalence in the two calculi.

1998 ACM Subject Classification F.3.2 Operational semantics; F.3.3 Functional constructs; F.4.1 Lambda calculus and related systems

Keywords and phrases Local names, continuations, typed $\boldsymbol{\lambda}$-calculus, observational equivalence

Digital Object Identifier 10.4230/LIPIcs.CSL.2011.396

1 Introduction

Locally scoped names are a ubiquitous feature of programming languages. Here we will be concerned with properties of this notion that are independent of the nature of the entities being named, be they mutable storage cells, objects, exceptions, communication channels, cryptographic keys, or whatever. The only assumption that we make about names is that the ambient programming language has the ability to test them for equality. The operational semantics of such locally scoped names is commonly specified in terms of *dynamically allocated fresh names*, also known as *generative names*. This is a state-based explanation of the meaning of the scoping construct: to execute a program with a locally scoped name, the current state is augmented with a fresh name and the body of the scope is executed with the scoped name bound to the fresh one. The combination of this simple mechanism with other features, especially higher-order functions as occurs in the ML family of languages, can result in programs with very complicated behaviour. The Pitts-Stark ν-*calculus* [15, 20] was intended to make this point, taking the measure of behaviour to be *observational equivalence* (also known as *contextual equivalence*), the relation between two programs of having the same observable behaviour when placed in any program context. Syntactically, the ν-calculus is simply-typed λ-calculus over ground types Name and Bool (for names and booleans respectively), augmented with a construct $\nu a.\, t$ for restricting the scope of a name a to a term t. The ν-calculus is given an operational semantics that makes it a fragment of Standard ML [10] by interpreting Name as ML's type `unit ref` of references to

Computer Science Logic 2011 (CSL'11).
Editor: Marc Bezem; pp. 396–411
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the unit value and taking $\nu a.\, t$ to be `let a = ref() in t`. The properties of observational equivalence for the ν-calculus turn out to be remarkably complex, despite the simplicity of the language. See [2, Sect. 1] for a survey of the literature on the ν-calculus.

The ν-calculus combines dynamically allocated local names with higher-order functions. But is dynamic allocation the only way to interpret the meaning of locally scoped names? In fact there is another, but much less well known semantics for them. At about the same time that the ν-calculus was introduced, Odersky developed what he called the *$\lambda\nu$-calculus* [12]. Syntactically this is essentially identical to the ν-calculus; it has pairs as well as functions, but the ν-calculus could have had those too (we add them here). However, the local scoping construct $\nu a.\, t$ is given a very different semantics, which we recall in Sect. 4. On the one hand, its most important feature is that it is stateless, or 'referentially transparent'; and Odersky shows that $\lambda\nu$-calculus is a conservative extension of λ-calculus with respect to observational equivalence. On the other hand, it has some properties that seem very strange compared with the more familiar, generative interpretation. For instance, dynamic allocation of locally scoped names generally does not commute with function abstraction; whereas in Odersky's calculus, $\nu a.\, \lambda x \to t$ is observationally equivalent to (indeed, reduces to) $\lambda x \to \nu a.\, t$. (For example, in the ν-calculus $\nu a.\, \lambda x \to a$ and $\lambda x \to \nu a.\, a$ are not observationally equivalent terms of type $\mathsf{Name} \to \mathsf{Name}$—see the discussion after Remark 3.2 below; however, they are observationally equivalent in the $\lambda\nu$-calculus.) Even more radically, in Odersky's calculus there is no sharing of local names between the components of a tuple, since $\nu a.\, (t_1\ ,\ t_2)$ is observationally equivalent to $(\nu a_1.\, t_1[a_1/a]\ ,\ \nu a_2.\, t_2[a_2/a])$.

Contribution of this paper. We shed new light on Odersky's version of locally scoped names by showing that it stands in a surprising relation to the more familiar, dynamic allocation interpretation. We prove that Odersky's version of $\nu a.\, t$ provides a 'direct' meaning for locally scoped names from which the behaviour determined by dynamic allocation can be recovered via continuations. More precisely, we show that a standard continuation passing style (CPS) transformation on typed λ-calculus can be extended to locally scoped names so as to provide a computationally adequate translation of ν-calculus into $\lambda\nu$-calculus. Dynamically allocated names at a particular type are translated to Odersky-style local names at the corresponding function type of continuations. Quite surprisingly, even though Odersky's version of $\nu a.\, (-)$ behaves quite differently with respect to functions compared to the dynamic allocation semantics of $\nu a.\, (-)$, we show that the CPS translation is sound and complete for evaluating boolean terms (Theorem 5.1). Since the translation is compositional, it follows that two ν-calculus terms of any type are observationally equivalent if their CPS-translations are observationally equivalent in the $\lambda\nu$-calculus

Our proof of these results is via a new formulation of $\lambda\nu$-calculus 'big step' operational semantics and via a by-now standard use of Felleisen-style evaluation contexts for ν-calculus. At the heart of the proof we construct (in Sect. 5.2) a logical relation between $\lambda\nu$-calculus and ν-calculus tailored to the CPS transformation. Although we use the methods of operational semantics, as we explain in Sect. 6 our results have their origin in a denotational semantics of dynamic allocation using *nominal sets* [18] and, more recently, a simple nominal sets model for Odersky-style local names [14]. Our results suggest re-evaluating the usefulness of Odersky's semantics of locally scoped names and the concluding section gives some avenues for doing that.

$T \in \mathit{Type} ::=$		$t \in \mathit{Term} ::=$	
Name	names	x	variable, $x \in \mathbb{V}$
Bool	booleans	a	atomic name, $a \in \mathbb{A}$
$T \times T$	pairs	$\nu a.\, t$	locally scoped name
$T \to T$	functions	$t = t$	name equality test
		true	truth
		false	falsity
		$\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$	conditional
		$(t\ ,\ t)$	pair
		$\mathsf{let}\ (x\ ,\ x) = t\ \mathsf{in}\ t$	unpairing
		$\lambda x \to t$	function abstraction
		$t\, t$	application

Figure 1 Syntax

2 Simply Typed λ-Calculus with Local Names

We use the same syntax and typing rules for the Pitts-Stark ν-calculus as for the Odersky $\lambda\nu$-calculus. This unification is just a slight deviation from the original syntax [15, 12], but the expressiveness remains the same. To the usual simply typed λ-calculus with pairs and booleans we add names that can be tested for equality and locally scoped. The types and terms of the resulting language are given in Fig. 1.

It is convenient to use two different sorts of identifier in terms, drawn from disjoint infinite sets $\mathbb{V}$ and $\mathbb{A}$. Elements $x, y, z \ldots$ of $\mathbb{V}$ are called *variables* and elements $a, b, c, \ldots$ of $\mathbb{A}$ are called *atomic names*. We make this syntactic distinction to emphasise the fact that the two different sorts of identifier have different substitution properties. Validity of judgements in the calculi we consider here is preserved under substituting terms for variables; but in general it is only preserved under permutations of atomic names, rather than more general forms of substitution for names.

As a matter of notation we write $t[t_1/x_1, \ldots, t_n/x_n]$ for the (capture-avoiding, simultaneous) substitution of terms $t_1, \ldots, t_n$ for free occurrences of the distinct variables $x_1, \ldots, x_n$ in the term t. We identify terms up to α-equivalence of bound variables and bound atomic names. The binding forms are as follows: free occurrences of a in t become bound in $\nu a.\, t$; free occurrences of x_1 and x_2 in t' become bound in $\mathsf{let}\ (x_1\ ,\ x_2) = t\, \mathsf{in}\, t'$; and free occurrences of x in t become bound in $\lambda x \to t$. We write $\mathit{fv}(t)$ and $\mathit{fn}(t)$ respectively for the finite sets of free variables and free atomic names of t. We say that a term t is *variable-closed* if $\mathit{fv}(t) = \emptyset$ (even if $\mathit{fn}(t)$ is non-empty).

The grammar in Fig. 1 specifies 'raw' terms, but we are only interested in well-typed terms. We specify those via an inductively defined typing relation $\Gamma \vdash t : T$, where the *typing context* $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$ is a finite map from variables x_i to types T_i whose domain $\mathit{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ contains the set $\mathit{fv}(t)$ of free variables of t. Rather than also recording the free atomic names of t in the typing context, we have chosen to leave them implicit, because it simplifies notation later. Thus the typing rules involving names are as follows.

$$\frac{}{\Gamma \vdash a : \mathsf{Name}} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \nu a.\, t : T} \qquad \frac{\Gamma \vdash t : \mathsf{Name} \qquad \Gamma \vdash t' : \mathsf{Name}}{\Gamma \vdash t = t' : \mathsf{Bool}}$$

The typing rules for the other syntactic constructs are entirely conventional, so we omit them here.

$$\frac{\overline{a} \cup \{a\} \,,\, t \Downarrow_\nu \overline{a}' \,,\, v}{\overline{a} \,,\, \nu a.\, t \Downarrow_\nu \overline{a}' \,,\, v}\ (a \notin \overline{a}) \qquad \frac{}{\overline{a} \,,\, v \Downarrow_\nu \overline{a} \,,\, v}\ (v = a, \mathsf{true}, \mathsf{false}, \lambda x \to t)$$

$$\frac{\overline{a} \,,\, t_1 \Downarrow_\nu \overline{a}' \,,\, a_1 \qquad \overline{a}' \,,\, t_2 \Downarrow_\nu \overline{a}'' \,,\, a_2}{\overline{a} \,,\, t_1 = t_2 \Downarrow_\nu \overline{a}'' \,,\, \delta_{a_1\, a_2}}\ \text{where } \delta_{a_1\, a_2} \triangleq \begin{cases} \mathsf{true} & \text{if } a_1 = a_2 \\ \mathsf{false} & \text{if } a_1 \neq a_2 \end{cases}$$

$$\frac{\overline{a} \,,\, t_1 \Downarrow_\nu \overline{a}' \,,\, \mathsf{true} \qquad \overline{a}' \,,\, t_2 \Downarrow_\nu \overline{a}'' \,,\, v}{\overline{a} \,,\, \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow_\nu \overline{a}'' \,,\, v} \qquad \frac{\overline{a} \,,\, t_1 \Downarrow_\nu \overline{a}' \,,\, \mathsf{false} \qquad \overline{a}' \,,\, t_3 \Downarrow_\nu \overline{a}'' \,,\, v}{\overline{a} \,,\, \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow_\nu \overline{a}'' \,,\, v}$$

$$\frac{\overline{a} \,,\, t_1 \Downarrow_\nu \overline{a}' \,,\, v_1 \qquad \overline{a}' \,,\, t_2 \Downarrow_\nu \overline{a}'' \,,\, v_2}{\overline{a} \,,\, (t_1 \,,\, t_2) \Downarrow_\nu \overline{a}'' \,,\, (v_1 \,,\, v_2)} \qquad \frac{\overline{a} \,,\, t \Downarrow_\nu \overline{a}' \,,\, (v_1 \,,\, v_2) \qquad \overline{a}' \,,\, t'[v_1/x_1, v_2/x_2] \Downarrow_\nu \overline{a}'' \,,\, v}{\overline{a} \,,\, \mathsf{let}\ (x_1 \,,\, x_2) = t\ \mathsf{in}\ t' \Downarrow_\nu \overline{a}'' \,,\, v}$$

$$\frac{\overline{a} \,,\, t_1 \Downarrow_\nu \overline{a}' \,,\, \lambda x \to t \qquad \overline{a}' \,,\, t_2 \Downarrow_\nu \overline{a}'' \,,\, v \qquad \overline{a}'' \,,\, t[v/x] \Downarrow_\nu \overline{a}''' \,,\, v'}{\overline{a} \,,\, t_1\, t_2 \Downarrow_\nu \overline{a}''' \,,\, v'}$$

$$\text{where } v \in \mathit{Val} ::= x \mid a \mid \mathsf{true} \mid \mathsf{false} \mid (v \,,\, v) \mid \lambda x \to t$$

Figure 2 ν-Calculus evaluation relation

We will be concerned with various congruence relations between well-typed terms. Here is the general definition of such a relation (cf. [13, Definition 7.5.1]).

▶ **Definition 2.1.** A *type-respecting binary relation* is specified by a set $\mathcal{R}$ of quadruples (Γ, t_1, t_2, T), where $\Gamma \vdash t_1 : T$ and $\Gamma \vdash t_2 : T$. We write $\Gamma \vdash t_1 \ \mathcal{R}\ t_2 : T$ instead of $(\Gamma, t_1, t_2, T) \in \mathcal{R}$. Such a relation is a *congruence* if it is reflexive, symmetric, transitive and *compatible* with the term-forming operations. The latter means

$$\begin{aligned}
&\Gamma \vdash a\ \mathcal{R}\ a : \mathsf{Name} \\
&\Gamma \vdash t_1\ \mathcal{R}\ t_2 : T \ \Rightarrow\ \Gamma \vdash \nu a.\, t_1\ \mathcal{R}\ \nu a.\, t_2 : T \\
&\Gamma \vdash t_1\ \mathcal{R}\ t_2 : \mathsf{Name} \ \wedge\ \Gamma \vdash t : \mathsf{Name} \ \Rightarrow\ \Gamma \vdash (t_1 = t)\ \mathcal{R}\ (t_2 = t) : \mathsf{Bool} \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Gamma \vdash (t = t_1)\ \mathcal{R}\ (t = t_2) : \mathsf{Bool}
\end{aligned}$$

and similar conditions for the other term-forming operations.

3 ν-Calculus

The language of the previous section becomes the ν-calculus [15, 20] if we evaluate locally scoped names $\nu a.\, t$ using the mechanism of dynamic allocation (and use call-by-value evaluation for pairs and functions). Figure 2 gives rules in the style of the Definition of Standard ML [10] for inductively defining a relation $\overline{a} \,,\, t \Downarrow_\nu \overline{a}' \,,\, v$, where

- $\overline{a}$ and $\overline{a}'$ are finite subsets of $\mathbb{A}$ with $\overline{a} \subseteq \overline{a}'$;
- t and v are variable-closed terms with $\mathit{fn}(t) \subseteq \overline{a}$ and $\mathit{fn}(v) \subseteq \overline{a}'$;
- $v \in \mathit{Val} \subseteq \mathit{Term}$ is a *value*, as specified by the grammar in Fig. 2.

We use this relation to define observational equivalence for the ν-calculus, $\Gamma \vdash t_1 \approx_\nu t_2 : T$. To do so, we believe it is helpful to take the abstract, *relational* point of view first advocated by Gordon and Lassen [8]. We wish $\approx_\nu$ to be a congruence in the sense of Definition 2.1

and to be ν-*adequate* for observing evaluation of boolean terms in the sense that

$$\emptyset \vdash t_1 \approx_\nu t_2 : \mathsf{Bool} \ \wedge \ fn(t_1, t_2) \subseteq \overline{a} \ \Rightarrow \ (\overline{a}\,, t_1 \Downarrow_\nu \_\,, \mathsf{true} \Leftrightarrow \overline{a}\,, t_2 \Downarrow_\nu \_\,, \mathsf{true}) \tag{1}$$

$$\text{where} \quad \overline{a}\,, t \Downarrow_\nu \_\,, \mathsf{true} \ \triangleq \ (\exists \overline{a}')\, \overline{a}\,, t \Downarrow_\nu \overline{a}'\,, \mathsf{true}. \tag{2}$$

▶ **Definition 3.1** (ν-Calculus observational equivalence)**.** Arguing as in the proof of [13, Theorem 7.5.3], we have that the union of all type-respecting binary relations that are both compatible (Definition 2.1) and have the ν-adequacy property (1) is an equivalence relation; and hence it is the largest ν-adequate congruence relation. We denote it by $\approx_\nu$ and call it ν-*calculus observational equivalence*.

The fact that in (1) we observe convergence just to true is not significant; also observing convergence to false, or to a particular atomic name, does not change $\approx_\nu$. On the other hand, just observing convergence *per se* would result in a trivial equivalence, since ν-calculus lacks any non-terminating features such as fixpoint recursion (as a matter of choice rather than necessity).

▶ **Remark 3.2** (contextual equivalence)**.** The terms 'observational equivalence' and 'contextual equivalence' are used more or less interchangeably in the literature on ν-calculus. (One might say that they are contextually equivalent terms.) We have chosen the first, because we favour the more abstract, 'context-free' characterization that we have used as the definition. However, it is possible to give a more concrete characterization of $\approx_\nu$ in terms of substitution of terms into term contexts, that is, syntax-trees with a hole; see [15, Definition 4]. Both free variables and free atomic names in terms may get captured by this form of substitution. So term contexts are not identified up to α-equivalence and one has to give separate and more elaborate typing rules for them. These complications are avoided by using the relational definition we have given.

However one defines it, the properties of $\approx_\nu$ are known to be very complicated; see [2] for a recent discussion of this fact. In particular, terms of function or product type do not behave extensionally up to observational equivalence. For example

$$\emptyset \vdash \nu a.\, \lambda x \to a \not\approx_\nu \lambda x \to \nu a.\, a : \mathsf{Name} \to \mathsf{Name} \tag{3}$$

(since applying $\lambda f \to \nu a.\, (f\ a = f\ a)$ to each term gives terms that evaluate to true and false respectively); and yet applying these two terms to any name yields observationally equivalent results. Similarly

$$\emptyset \vdash \nu a.\, \nu b.\, (a\ ,\ b) \not\approx_\nu \nu a.\, (a\ ,\ a) : \mathsf{Name} \times \mathsf{Name} \tag{4}$$

(since applying $\lambda x \to \mathsf{let}\ (x_1\ ,\ x_2) = x\ \mathsf{in}\ (x_1 = x_2)$ to each term gives terms that evaluate to false and true respectively); and yet applying first and second projection functions to them yields observationally equivalent results in each case.

4 $\lambda\nu$-Calculus

Figure 3 inductively defines a state-free evaluation relation $t \Downarrow_{\lambda\nu} c$, where t and c are variable-closed terms (possibly with free atomic names) and c is a *canonical form*, that is, in the subset $Cf \subseteq Term$ of terms specified by the grammar at the bottom of the figure. The rules for evaluating booleans, pairs and functions are just those of the pure call-by-name typed λ-calculus. It is the first rule in the figure, for evaluating $\nu a.\, t$, that embodies

$$\frac{t \Downarrow_{\lambda\nu} c}{\nu a.\, t \Downarrow_{\lambda\nu} a \setminus c} \text{ where } a \setminus c \triangleq \begin{cases} \nu a.\, a & \text{if } c = a \\ c & \text{if } c \in (\mathbb{A} - \{a\}) \cup \{\nu a.\, a, \mathsf{true}, \mathsf{false}\} \\ (\nu a.\, t_1 \ , \ \nu a.\, t_2) & \text{if } c = (t_1 \ , \ t_2) \\ \lambda x \to \nu a.\, t & \text{if } c = \lambda x \to t \end{cases}$$

$$\frac{}{c \Downarrow_{\lambda\nu} c} \qquad \frac{t_1 \Downarrow_{\lambda\nu} c_1 \qquad t_2 \Downarrow_{\lambda\nu} c_2}{t_1 = t_2 \Downarrow_{\lambda\nu} \delta_{c_1\, c_2}} \text{ where } \delta_{c_1\, c_2} \triangleq \begin{cases} \mathsf{true} & \text{if } c_1 = c_2 \\ \mathsf{false} & \text{if } c_1 \neq c_2 \end{cases}$$

$$\frac{t_1 \Downarrow_{\lambda\nu} \mathsf{true} \qquad t_2 \Downarrow_{\lambda\nu} c}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow_{\lambda\nu} c} \qquad \frac{t_1 \Downarrow_{\lambda\nu} \mathsf{false} \qquad t_3 \Downarrow_{\lambda\nu} c}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow_{\lambda\nu} c}$$

$$\frac{t \Downarrow_{\lambda\nu} (t_1 \ , \ t_2) \qquad t'[t_1/x_1, t_2/x_2] \Downarrow_{\lambda\nu} c}{\mathsf{let}\ (x_1 \ , \ x_2) = t\ \mathsf{in}\ t' \Downarrow_{\lambda\nu} c} \qquad \frac{t_1 \Downarrow_{\lambda\nu} \lambda x \to t \qquad t[t_2/x] \Downarrow_{\lambda\nu} c}{t_1\, t_2 \Downarrow_{\lambda\nu} c}$$

$$\text{where } c \in Cf ::= a \mid \nu a.\, a \mid \mathsf{true} \mid \mathsf{false} \mid (t \ , \ t) \mid \lambda x \to t$$

Figure 3 $\lambda\nu$-Calculus evaluation relation

Odersky's semantics of locally scoped names from [12]. Compared with the corresponding rule in Fig. 2, whose effect is to extrude local scopes outward to the top level, here scoping intrudes through pairing and function abstraction until it reaches canonical booleans and names.

▶ **Remark 4.1** (the 'anonymous name' anon $\triangleq \nu a.\, a$)**.** What we here call the $\lambda\nu$-calculus is essentially the typed calculus described in [12, Sect. 6] equipped with the deterministic evaluation relation sketched in Sect. 4 of that paper (although our description of evaluation in Fig. 3 is more direct). However, there are two related respects in which our calculus differs. Firstly, we choose to regard the term anon $\triangleq \nu a.\, a$ as a canonical form of type Name and secondly, we take the boolean term anon = anon to evaluate to true. Whereas Odersky takes both terms to be stuck with respect to evaluation (and to be bottom, denotationally). Other choices are possible; for example one might take anon to be canonical, but have anon = anon evaluate to false, or be stuck. Such choices clearly affect the properties of $\lambda\nu$-calculus contextual equivalence and hence potentially affect the adequacy of translations of ν-calculus into $\lambda\nu$-calculus that we develop in the next section.

Our motivation for taking anon to be a canonical form comes from the nominal sets model of Odersky-style local names described in [14], where anon is a non-bottom value. Having stuck terms, Odersky's original typed system fails to satisfy the usual 'progress' part of type soundness, whereas here we have the following result.

▶ **Theorem 4.2** ($\lambda\nu$-calculus type soundness and totality)**.** *In the $\lambda\nu$-calculus, well-typed variable-closed terms possess unique canonical forms: for all $\emptyset \vdash t : T$, there is a unique c satisfying $\emptyset \vdash c : T$ and $t \Downarrow_{\lambda\nu} c$.*

Proof. The proof that evaluation preserves typing is routine. That evaluation is single-valued follows from the fact that it does not create free atomic names ($t \Downarrow_{\lambda\nu} c \Rightarrow fn(c) \subseteq fn(t)$); this follows in turn from the fact that in the derived operation $a \setminus c$ on canonical forms used to evaluate $\nu a.\, t$, free occurrences of a in c become bound in $a \setminus c$. Finally one has to prove that evaluation of well-typed terms is total. This can be done by adapting the

usual argument for simply typed λ-calculus using Tait-style computability predicates; we omit the details here. ◀

As for the ν-calculus, we can give a simple, 'relational' definition of observational equivalence.

▶ **Definition 4.3** ($\lambda\nu$-Calculus observational equivalence)**.** We define $\approx_{\lambda\nu}$, to be the largest congruence relation satisfying the following *$\lambda\nu$-adequacy property*:

$$\emptyset \vdash t_1 \approx_{\lambda\nu} t_2 : \mathsf{Bool} \;\Rightarrow\; (t_1 \Downarrow_{\lambda\nu} \mathsf{true} \;\Leftrightarrow\; t_2 \Downarrow_{\lambda\nu} \mathsf{true}). \tag{5}$$

It can be constructed by observing that the union of all $\lambda\nu$-adequate and compatible type-respecting relations is an equivalence relation (as well as being $\lambda\nu$-adequate and compatible).

Modulo the changes mentioned in Remark 4.1, $\approx_{\lambda\nu}$ is essentially the same notion that Odersky defines more concretely with term contexts [12, Sect. 5]. He shows that it has many pleasant properties in common with the pure typed λ-calculus, such as extensionality for functions and products. One can show that

$$\emptyset \vdash t : T \;\wedge\; t \Downarrow_{\lambda\nu} c \;\Rightarrow\; \emptyset \vdash t \approx_{\lambda\nu} c : T \tag{6}$$

$$\Gamma \vdash t : T \;\wedge\; a \notin fn(t) \;\Rightarrow\; \Gamma \vdash \nu a.\, t \approx_{\lambda\nu} t : T \tag{7}$$

$$\Gamma \vdash t : T \;\Rightarrow\; \Gamma \vdash \nu a.\, \nu a'.\, t \approx_{\lambda\nu} \nu a'.\, \nu a.\, t : T. \tag{8}$$

Hence in particular the pairs of terms in (3) and (4) are observationally equivalent in the $\lambda\nu$-calculus. So $\approx_\nu$ and $\approx_{\lambda\nu}$ are not at all the same. Indeed in view of property (6), the evaluation rules in Fig. 3 imply the characteristic 'scope intrusion' laws

$$\Gamma \vdash \nu a.\, \lambda x \to t \approx_{\lambda\nu} \lambda x \to \nu a.\, t : T_1 \to T_2 \tag{9}$$

$$\Gamma \vdash \nu a.\, (t_1\ ,\ t_2) \approx_{\lambda\nu} (\nu a.\, t_1\ ,\ \nu a.\, t_2) : T_1 \times T_2 \tag{10}$$

that distinguish Odersky-style local names from dynamically allocated ones.

5 Translating ν to $\lambda\nu$

Figure 4 gives a *continuation-passing style* (CPS) transformation of the types and terms of the typed λ-calculus from Sect. 2. The transformations for values ($v \mapsto v^\bullet$) and for terms ($t \mapsto t^\circ$) are defined by mutual recursion on the structure of these expressions.

The part of the transformation that does not concern local names is very standard: we have combined Moggi's call-by-value translation of λ-calculus into his computational metalanguage [11] with an interpretation of that metalanguage that uses the continuation monad $\mathcal{C}(-) \triangleq (- \to \mathsf{Bool}) \to \mathsf{Bool}$. The part of the transformation that *does* concern local names is pleasingly simple; *dynamically allocated local names at a type T are transformed into Odersky-style local names at type $\mathcal{C}\overline{T}$*: $(\nu a.\, t)^\circ = \nu a.\, t^\circ$.

Recalling the definitions of $\Downarrow_\nu$ and $\approx_\nu$ for the ν-calculus from Sect. 3 and $\Downarrow_{\lambda\nu}$ and $\approx_{\lambda\nu}$ for the $\lambda\nu$-calculus from Sect. 4, we can now state the main result of the paper.

▶ **Theorem 5.1** (computational adequacy)**.** *(i) For all $\emptyset \vdash t : \mathsf{Bool}$, with $fn(t) \subseteq \overline{a}$ say,*

$$\overline{a}\,, t \Downarrow_\nu \_\,, \mathsf{true} \;\Leftrightarrow\; t^\circ(\lambda x \to x) \Downarrow_{\lambda\nu} \mathsf{true}. \tag{11}$$

(ii) For all $\Gamma \vdash t_i : T$ $(i = 1, 2)$, if $\overline{\Gamma} \vdash t_1^\circ \approx_{\lambda\nu} t_2^\circ : \mathcal{C}\overline{T}$, then $\Gamma \vdash t_1 \approx_\nu t_2 : T$.

- Types $\mapsto$ types $\overline{T}$ $\left\{\begin{array}{rcl} \overline{\mathsf{Name}} & = & \mathsf{Name} \\ \overline{\mathsf{Bool}} & = & \mathsf{Bool} \\ \overline{T_1 \times T_2} & = & \overline{T_1} \times \overline{T_2} \\ \overline{T_1 \to T_2} & = & \overline{T_1} \to \mathcal{C}\overline{T_2} \\ \text{where} \quad \mathcal{C}T & \triangleq & (T \to \mathsf{Bool}) \to \mathsf{Bool}. \end{array}\right.$
- Typing contexts $\Gamma \mapsto$ typing contexts $\overline{\Gamma}$ $\left\{\begin{array}{rcl} \overline{\emptyset} & = & \emptyset \\ \overline{\Gamma, x : T} & = & \overline{\Gamma}, x : \overline{T}. \end{array}\right.$
- Values $\Gamma \vdash v : T \mapsto$ canonical forms $\overline{\Gamma} \vdash v^\bullet : \overline{T}$

$$\begin{array}{rl} v^\bullet = v & \text{for } v = x, a, \mathsf{true}, \mathsf{false} \\ (v_1 \ , \ v_2)^\bullet = & (v_1^\bullet \ , \ v_2^\bullet) \\ (\lambda x_1 \to t_1)^\bullet = & \lambda x_1 \to t_1^\circ. \end{array}$$

- Terms $\Gamma \vdash t : T \mapsto$ terms $\overline{\Gamma} \vdash t^\circ : \mathcal{C}\overline{T}$

$$\begin{array}{rl} v^\circ = & \lambda k \to k\, v^\bullet \\ (\nu a.\, t)^\circ = & \nu a.\, t^\circ \\ (t_1 = t_2)^\circ = & \lambda k \to t_1^\circ(\lambda x \to t_2^\circ(\lambda x' \to \mathsf{if}\ x = x'\ \mathsf{then}\ k\, \mathsf{true}\ \mathsf{else}\ k\, \mathsf{false})) \\ (\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3)^\circ = & \lambda k \to t_1^\circ(\lambda x \to \mathsf{if}\ x\ \mathsf{then}\ t_2^\circ\, k\ \mathsf{else}\ t_3^\circ\, k) \\ (t_1 \ , \ t_2)^\circ = & \lambda k \to t_1^\circ(\lambda x \to t_2^\circ(\lambda x' \to k\,(x \ , \ x'))) \quad \text{when } (t_1 \ , \ t_2) \notin \mathit{Val} \\ (\mathsf{let}\ (x_1 \ , \ x_2) = t_1\ \mathsf{in}\ t_2)^\circ = & \lambda k \to t_1^\circ(\lambda x \to \mathsf{let}\ (x_1 \ , \ x_2) = x\ \mathsf{in}\ t_2^\circ\, k) \\ (t_1\, t_2)^\circ = & \lambda k \to t_1^\circ(\lambda x \to t_2^\circ(\lambda x' \to x\, x'\, k)) \\ (\text{where } k, x, x' \notin \mathit{fv}(v, t_1, t_2, t_3)). & \end{array}$$

Figure 4 CPS transformation

Part (ii) of the theorem follows from part (i), because the CPS transformation is compositional. More precisely, referring to Definition 2.1, the type-respecting binary relation

$$\mathcal{R} \ \triangleq \ \{(\Gamma, t_1, t_2, T) \mid \Gamma \vdash t_1 : T \ \wedge \ \Gamma \vdash t_2 : T \ \wedge \ \overline{\Gamma} \vdash t_1^\circ \approx_{\lambda\nu} t_2^\circ : \mathcal{C}\overline{T}\}$$

is easily seen to be a congruence; additionally it has the ν-adequacy property (1) by virtue of (i) and because $\approx_{\lambda\nu}$ is a $\lambda\nu$-adequate congruence. So since $\approx_\nu$ is by definition the largest ν-adequate congruence, it contains $\mathcal{R}$—as required for property (ii).

The rest of this section sketches the proof of part (i) of the theorem. We first re-formulate the operational semantics of the ν-calculus in terms of an abstract machine with frame stacks. As a result, property (11) becomes a statement about machine configurations with an empty stack that can be deduced from a more general bi-implication involving arbitrary frame stacks (Corollary 5.8). The left-to-right part of this bi-implication is straightforward; the right-to-left part is harder and we prove it by constructing a suitable logical relation between the $\lambda\nu$-calculus and the ν-calculus.

5.1 Abstract machine

Although the 'big step' operational semantics of Sect. 3 gives a clear specification of the ν-calculus, experience has shown that a small-step semantics formulated in the style of Felleisen with evaluation contexts [5] is better suited for developing the properties of the associated observational equivalence, $\approx_\nu$; and to make proofs about evaluation contexts easier to formalize, it pays to write them 'inside out' as a list of basic contexts (evaluation

$$
\begin{aligned}
\langle F\,,\nu a.\,t\rangle &\to_\nu \langle F\,,t\rangle \quad \text{if } a \notin \mathit{fn}(F)\\
\langle F\,,t_1 = t_2\rangle &\to_\nu \langle F \circ (\cdot = t_2)\,,t_1\rangle\\
\langle F\,,\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3\rangle &\to_\nu \langle F \circ (\mathsf{if}\cdot\mathsf{then}\ t_2\ \mathsf{else}\ t_3)\,,t_1\rangle\\
\langle F\,,(t_1\ ,\ t_2)\rangle &\to_\nu \langle F \circ (\cdot\ ,\ t_2)\,,t_1\rangle \quad \text{when } (t_1\ ,\ t_2) \notin \mathit{Val}\\
\langle F\,,\mathsf{let}\ (x_1\ ,\ x_2) = t\ \mathsf{in}\ t'\rangle &\to_\nu \langle F \circ (\mathsf{let}\ (x_1\ ,\ x_2) = \cdot\ \mathsf{in}\ t')\,,t\rangle\\
\langle F\,,t_1\,t_2\rangle &\to_\nu \langle F \circ (\cdot\,t_2)\,,t_1\rangle\\
\langle F \circ (\cdot = t)\,,a\rangle &\to_\nu \langle F \circ (a = \cdot)\,,t\rangle\\
\langle F \circ (a_1 = \cdot)\,,a_2\rangle &\to_\nu \langle F\,,\delta_{a_1\,a_2}\rangle\\
\langle F \circ (\mathsf{if}\cdot\mathsf{then}\ t\ \mathsf{else}\ t')\,,\mathsf{true}\rangle &\to_\nu \langle F\,,t\rangle\\
\langle F \circ (\mathsf{if}\cdot\mathsf{then}\ t\ \mathsf{else}\ t')\,,\mathsf{false}\rangle &\to_\nu \langle F\,,t'\rangle\\
\langle F \circ (\cdot\ ,\ t)\,,v\rangle &\to_\nu \langle F \circ (v\ ,\ \cdot)\,,t\rangle\\
\langle F \circ (v_1\ ,\ \cdot)\,,v_2\rangle &\to_\nu \langle F\,,(v_1\ ,\ v_2)\rangle\\
\langle F \circ (\mathsf{let}\ (x_1\ ,\ x_2) = \cdot\ \mathsf{in}\ t)\,,(v_1\ ,\ v_2)\rangle &\to_\nu \langle F\,,t[v_1/x_1.v_2/x_2]\rangle\\
\langle F \circ (\cdot\,t)\,,v\rangle &\to_\nu \langle F \circ (v\,\cdot)\,,t\rangle\\
\langle F \circ (\lambda x \to t)\,\cdot\,,v\rangle &\to_\nu \langle F\,,t[v/x]\rangle
\end{aligned}
$$

where $F \in \mathit{Stack} ::= \mathsf{Id} \mid F \circ E$ and
$E \in \mathit{Frame} ::= \cdot = t \mid v = \cdot \mid \mathsf{if}\cdot\mathsf{then}\ t\ \mathsf{else}\ t \mid (\cdot\ ,\ t) \mid (v\ ,\ \cdot) \mid \mathsf{let}\ (x\ ,\ x) = \cdot\ \mathsf{in}\ t \mid \cdot\,t \mid v\,\cdot$

Figure 5 ν-Calculus abstract machine

frames). Figure 5 formulates the operational semantics of the ν-calculus in this style. It defines a binary relation $\to_\nu$ between configurations of the form $\langle F\,,t\rangle$, where

- F is a *frame stack* (a list of *evaluation frames* E, defined by the grammar in the figure);
- t is a term;
- both F and t are variable-closed.

Note the first transition in Fig. 5, for dynamically allocated local names. The use of sets of atomic names $\overline{a}$ as states in the definition of $\Downarrow_\nu$ is not necessary for $\to_\nu$; the implicit state of a configuration $\langle F\,,t\rangle$ is its finite set $\mathit{fn}(F) \cup \mathit{fn}(t)$ of free atomic names. The termination relation (2) used in the definition of $\approx_\nu$ can be characterized in terms of termination of the abstract machine, as follows.

▶ **Lemma 5.2.** *Let t be a variable-closed term, with $\mathit{fn}(t) \subseteq \overline{a}$ say. Then $\overline{a}\,,t \Downarrow_\nu \_\,,\mathsf{true}$ holds iff $\langle \mathsf{Id}\,,t\rangle \to_\nu^* \langle \mathsf{Id}\,,\mathsf{true}\rangle$, where $\to_\nu^*$ denotes the reflexive-transitive closure of $\to_\nu$.*

Proof. For the left-to-right implication, one can show (by induction on the derivation from the rules in Fig. 2) that $\overline{a}\,,t \Downarrow_\nu \overline{a}'\,,v$ implies $(\forall F)\ \mathit{fn}(F) \cap \overline{a}' = \emptyset \ \Rightarrow\ \langle F\,,t\rangle \to_\nu^* \langle F\,,v\rangle$. The right-to-left implication can be deduced from

$$\langle F\,,t\rangle \to_\nu \langle F'\,,t'\rangle \ \wedge\ \mathit{fn}(F,t,F',t') \subseteq \overline{a} \ \wedge\ \overline{a}\,,F'[t'] \Downarrow_\nu \_\,,v \ \Rightarrow\ \overline{a}\,,F[t] \Downarrow_\nu \_\,,v \tag{12}$$

where the term $F[t]$ is defined by recursion on the length of the frame stack F

$$\mathsf{Id}[t] = t \quad \text{and} \quad (F \circ E)[t] = F[E[t/\cdot]] \tag{13}$$

and where $E[t/\cdot]$ is the term obtained from an evaluation frame E by replacing its hole $\cdot$ by the term t. Property (12) is proved by case analysis on the definition of $\to_\nu$ in Fig. 5, using

$$\overline{a}\,,F[t] \Downarrow_\nu \overline{a}'\,,v \ \Leftrightarrow\ (\exists \overline{a}'', v')\ \overline{a}\,,t \Downarrow_\nu \overline{a}''\,,v' \ \wedge\ \overline{a}''\,,F[v'] \Downarrow_\nu \overline{a}'\,,v$$

- Frame stacks $\Gamma \vdash F : T \to \mathsf{Bool} \mapsto$ canonical forms $\overline{\Gamma} \vdash F^* : \overline{T} \to \mathsf{Bool}$

$$
\begin{aligned}
\mathsf{Id}^* &= \lambda x \to x \\
(F \circ (\cdot = t_2))^* &= \lambda x \to t_2^\circ(\lambda x' \to \mathsf{if}\ x = x'\ \mathsf{then}\ F^*\mathsf{true}\ \mathsf{else}\ F^*\mathsf{false}) \\
(F \circ (v_1 = \cdot))^* &= \lambda x \to \mathsf{if}\ v_1^\bullet = x\ \mathsf{then}\ F^*\mathsf{true}\ \mathsf{else}\ F^*\mathsf{false} \\
(F \circ (\mathsf{if}\ \cdot\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2))^* &= \lambda x \to \mathsf{if}\ x\ \mathsf{then}\ t_1^\circ F^*\ \mathsf{else}\ t_2^\circ F^* \\
(F \circ (\cdot\ ,\ t_2))^* &= \lambda x \to t_2^\circ(\lambda x' \to F^*(x\ ,\ x')) \\
(F \circ (v_1\ ,\ \cdot))^* &= \lambda x \to F^*(v_1^\bullet\ ,\ x) \\
(F \circ (\mathsf{let}\ (x_1\ ,\ x_2) = \cdot\ \mathsf{in}\ t))^* &= \lambda x \to \mathsf{let}\ (x_1\ ,\ x_2) = x\ \mathsf{in}\ t^\circ F^* \\
(F \circ (\cdot\, t_2))^* &= \lambda x \to t_2^\circ(\lambda x' \to x\, x'\, F^*) \\
(F \circ (v_1\, \cdot))^* &= \lambda x \to v_1^\bullet\, x\, F^* \\
\text{(where } x, x' \notin fv(v_1, t_1, t_2, t, F) &\text{ and } x_1, x_2 \notin fv(F))
\end{aligned}
$$

Figure 6 CPS transformation for frame stacks

which in turn is proved by induction on the length of F. ◀

▶ **Definition 5.3** (typed frame stacks)**.** We use the typing relation for terms from Sect. 2 to type ν-calculus frame stacks by substituting a fresh variable for the hole. Thus we write $\Gamma \vdash F : T' \to T$ to mean that $\Gamma, x : T' \vdash F[x] : T$ holds for some/any $x \notin dom(\Gamma)$. (An equivalent, syntax-directed inductive definition of $\Gamma \vdash F : T' \to T$ is of course possible.)

▶ **Notation 5.4.** For each type $T \in Type$, we write $Term(T)$ for the variable-closed terms of type t, that is, those $t \in Term$ satisfying $\emptyset \vdash t : T$. (Note that such a t may have free atomic names.) Similarly $Val(T)$ and $Stack(T' \to T)$ denote the sets of variable-closed ν-calculus values and frame stacks of types T and $T' \to T$ respectively. We define $Config(T) \triangleq \{\langle F\,, t\rangle \mid (\exists\, T' \in Type)\ F \in Stack(T' \to T)\ \wedge\ t \in Term(T')\}$.

The CPS transformation for terms can be extended to frame stacks. This is done in Fig. 6 and the next lemma proves the soundness of the transformation.

▶ **Lemma 5.5** (soundness of the CPS transformation)**.** *For each* $\langle F\,, t\rangle \in Config(\mathsf{Bool})$*, if* $\langle F\,, t\rangle \to_\nu^* \langle \mathsf{Id}\,, \mathsf{true}\rangle$ *then* $t^\circ\, F^* \Downarrow_{\lambda\nu} \mathsf{true}$.

Proof. Note that $\mathsf{true}^\circ\, \mathsf{Id}^* = (\lambda k \to k\, \mathsf{true})\, (\lambda x \to x) \Downarrow_{\lambda\nu} \mathsf{true}$. So it suffices to show that if $\langle F\,, t\rangle \to_\nu \langle F'\,, t'\rangle$ and $t'^\circ\, F'^* \Downarrow_{\lambda\nu} \mathsf{true}$, then $t^\circ\, F^* \Downarrow_{\lambda\nu} \mathsf{true}$. This can be proved by case analysis on the definition of $\to_\nu$ in Fig. 5. For the two cases in that figure involving substitution of values for variables one first needs to show $(t[v/x])^\circ = t^\circ[v^\bullet/x]$, which can be done by induction on the structure of t. ◀

5.2 Logical relation

To prove the converse of Lemma 5.5 we use the following logical relation between the $\lambda\nu$-calculus and the ν-calculus.

▶ **Definition 5.6.** The relation $t' \blacktriangleleft v : T$, where $T \in Type$, $v \in Val(T)$ and $t' \in Term(\overline{T})$, is defined by recursion on the structure of types T, making use of auxiliary relations $\triangleleft$ and

$\triangleleft^*$ for terms and frame stacks that are defined in terms of $\blacktriangleleft$:

$$\begin{aligned}
t' \blacktriangleleft a : \mathsf{Name} &\Leftrightarrow t' \Downarrow_{\lambda\nu} a \\
t' \blacktriangleleft b : \mathsf{Bool} &\Leftrightarrow t' \Downarrow_{\lambda\nu} b \quad (b \in \{\mathsf{true}, \mathsf{false}\}) \\
t' \blacktriangleleft (v_1 \,,\, v_2) : T_1 \times T_2 &\Leftrightarrow (\forall t_1, t_2)\, t' \Downarrow_{\lambda\nu} (t_1 \,,\, t_2) \;\Rightarrow\; t_1 \blacktriangleleft v_1 : T_1 \;\wedge\; t_2 \blacktriangleleft v_2 : T_2 \\
t' \blacktriangleleft v : T_1 \to T_2 &\Leftrightarrow (\forall t_1, v_1)\, t_1 \blacktriangleleft v_1 : T_1 \;\Rightarrow\; t'\, t_1 \triangleleft v\, v_1 : T_2
\end{aligned}$$

where for $T \in \mathit{Type}$, $t \in \mathit{Term}(T)$, $t' \in \mathit{Term}(\mathcal{C}\overline{T})$, $F \in \mathit{Stack}(T \to \mathsf{Bool})$ and $t'' \in \mathit{Term}(\overline{T} \to \mathsf{Bool})$ we define

$$\begin{aligned}
t' \triangleleft t : T &\triangleq (\forall t_1, F)\, t_1 \triangleleft^* F : T \to \mathsf{Bool} \;\Rightarrow\; t'\, t_1 \Downarrow_{\lambda\nu} \mathsf{true} \;\Rightarrow\; \langle F \,,\, t\rangle \to_\nu^* \langle \mathsf{Id} \,,\, \mathsf{true}\rangle \\
t'' \triangleleft^* F : T \to \mathsf{Bool} &\triangleq (\forall t_1, v)\, t_1 \blacktriangleleft v : T \;\Rightarrow\; t''\, t_1 \Downarrow_{\lambda\nu} \mathsf{true} \;\Rightarrow\; \langle F \,,\, v\rangle \to_\nu^* \langle \mathsf{Id} \,,\, \mathsf{true}\rangle.
\end{aligned}$$

The relation $\blacktriangleleft$ is extended to substitutions:

$$\Gamma \vdash \rho \blacktriangleleft \sigma \;\triangleq\; (\forall x \in \mathit{dom}(\Gamma))\; \rho(x) \blacktriangleleft \sigma(x) : \Gamma(x)$$

where ρ (respectively σ) ranges over finite functions from variables to variable-closed terms (respectively variable-closed values). Finally we extend the relations to open values, terms and frame stacks:

$$\begin{aligned}
\Gamma \vdash t' \blacktriangleleft v : T &\triangleq \overline{\Gamma} \vdash t' : \overline{T} \;\wedge\; \Gamma \vdash v : T \;\wedge\; (\forall \rho, \sigma)\, \Gamma \vdash \rho \blacktriangleleft \sigma \;\Rightarrow\; t'[\rho] \blacktriangleleft v[\sigma] : T \\
\Gamma \vdash t' \triangleleft t : T &\triangleq \overline{\Gamma} \vdash t' : \overline{T} \;\wedge\; \Gamma \vdash t : T \;\wedge\; (\forall \rho, \sigma)\, \Gamma \vdash \rho \blacktriangleleft \sigma \;\Rightarrow\; t'[\rho] \triangleleft t[\sigma] : T \\
\Gamma \vdash t' \triangleleft^* F : T \to \mathsf{Bool} &\triangleq \overline{\Gamma} \vdash t' : \overline{T} \to \mathsf{Bool} \;\wedge\; \Gamma \vdash F : T \to \mathsf{Bool} \;\wedge \\
&\qquad (\forall \rho, \sigma)\, \Gamma \vdash \rho \blacktriangleleft \sigma \;\Rightarrow\; t'[\rho] \triangleleft^* F[\sigma] : T \to \mathsf{Bool}
\end{aligned}$$

▶ **Theorem 5.7** (fundamental property of the logical relation).

$$\Gamma \vdash v : T \;\Rightarrow\; \Gamma \vdash v^\bullet \blacktriangleleft v : T \tag{14}$$

$$\Gamma \vdash t : T \;\Rightarrow\; \Gamma \vdash t^\circ \triangleleft t : T \tag{15}$$

$$\Gamma \vdash F : T \to \mathsf{Bool} \;\Rightarrow\; \Gamma \vdash F^* \triangleleft^* F : T \to \mathsf{Bool}. \tag{16}$$

Proof (sketch). Properties (14) and (15) are proved simultaneously by induction on the structure of v and t; and then (16) follows by induction on the structure of F. Here we give just the induction step for the case of locally scoped names; and for this it suffices to show that $t' \triangleleft t : T$ implies $\nu a.\, t' \triangleleft \nu a.\, t : T$ So suppose

$$t' \triangleleft t : T. \tag{17}$$

Referring to the definition of $\triangleleft$ in terms of $\triangleleft^*$ in Definition 5.6, we have to show that if

$$t_1 \triangleleft^* F : T \to \mathsf{Bool} \tag{18}$$

$$(\nu a.\, t')t_1 \Downarrow_{\lambda\nu} \mathsf{true} \tag{19}$$

then $\langle F \,,\, \nu a.\, t\rangle \to_\nu^* \langle \mathsf{Id} \,,\, \mathsf{true}\rangle$. Since we identify terms up to α-equivalence of bound atomic names, we may assume $a \notin \mathit{fn}(t_1, F)$. It follows from the definition of $\Downarrow_{\lambda\nu}$ in Fig. 3 that (19) implies $t' t_1 \Downarrow_{\lambda\nu} \mathsf{true}$, since $a \notin \mathit{fn}(t_1)$. From this, (17) and (18), the definition of $\triangleleft$ gives us $\langle F \,,\, t\rangle \to_\nu^* \langle \mathsf{Id} \,,\, \mathsf{true}\rangle$. Then since $a \notin \mathit{fn}(F)$, from the definition of $\to_\nu$ in Fig. 5 we get $\langle F \,,\, \nu a.\, t\rangle \to_\nu^* \langle \mathsf{Id} \,,\, \mathsf{true}\rangle$, as required. ◀

▶ **Corollary 5.8.** *If* $\langle F\,,t\rangle \in \mathit{Config}(\mathsf{Bool})$*, then*

$$\langle F\,,t\rangle \to_\nu^* \langle \mathsf{Id}\,,\mathsf{true}\rangle \;\Leftrightarrow\; t^\circ F^* \Downarrow_{\lambda\nu} \mathsf{true}. \tag{20}$$

Proof. The left-to-right implication in (20) is the soundness Lemma 5.5. For the converse, since $\langle F\,,t\rangle \in \mathit{Config}(\mathsf{Bool})$, we have $t \in \mathit{Term}(T)$ and $F \in \mathit{Stack}(T \to \mathsf{Bool})$ for some $T \in \mathit{Type}$. Note that by the fundamental property of the logical relation (Theorem 5.7) we have $t^\circ \lhd t : T$ and $F^* \lhd^* F : T \to \mathsf{Bool}$. Then the right-to-left implication follows immediately from the definition of $\lhd$ in terms of $\lhd^*$ in Definition 5.6. ◀

We can now complete the proof of the main theorem.

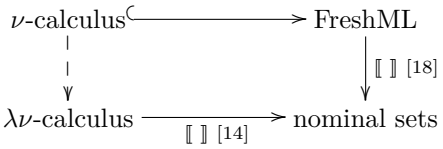
Proof of Theorem 5.1. We have already noted how part (ii) of the theorem follows from part (i). For the latter, combine Lemma 5.2 with the special case of Corollary 5.8 when $F = \mathsf{Id}$, for which $F^* = \mathsf{Id}^* = \lambda x \to x$. ◀

6 A Denotational Perspective

The results in this paper have two sources of inspiration.

- The FreshML language [19], which adds to an ML-like language facilities for declaring and computing with data involving name-binding operations. The 'fresh' in FreshML refers to the fact that the language's mechanism for computing with bound names involves dynamic allocation of fresh names. FreshML's type system ensures that even though programmers have access to the names of bound entities, α-renamed variants of data are indistinguishable up to observational equivalence in the language. This is proved in [18] via a denotational semantics of FreshML (and hence of dynamically allocated local names) using nominal sets [7].
- A nominal sets semantics for Odersky-style locally scoped names given by Pitts in connection with his work on structural recursion modulo α-equivalence [14].

In retrospect, one can see that the denotational semantics in [18] uses a continuation monad in order for the denotation of types to be valued in the 'nominal restriction sets' of [14, Sect. 2.3], rather than just in nominal sets; the restriction operation is then used to interpret locally scoped names. Thus the following picture emerges.



The dotted arrow is the syntactic translation of ν-calculus into $\lambda\nu$-calculus that we have developed in this paper. It composes with the denotational semantics in [14] to recover that in [18] when restricted to the sub-language of FreshML consisting of the ν-calculus.

This suggests an alternative approach to the main result, Theorem 5.1. Instead of the direct, operationally-based proof we have given, one could define a denotational semantics of $\lambda\nu$-calculus using nominal sets, as in [14]. Composing with the CPS transformation gives a denotational semantics for ν-calculus which can be proved adequate for $\Downarrow_\nu$ by constructing a logical relation between semantics and syntax along the lines of that in [18, Sect. 3]. The right-to-left implication in (20) follows from this adequacy result and hence we get an alternative, albeit less direct, proof of the main theorem.

7 Failure of Full Abstraction

Theorem 5.1(ii) says that the CPS translation of ν-calculus into $\lambda\nu$-calculus reflects observational equivalence. If a language translation not only reflects observational equivalence but also preserves it, then one says that the translation is *fully abstract* (by analogy with the use of that terminology for a denotational semantics). For this property to hold, roughly speaking the target language must not be able to observe more about a translated term than is possible in the source language. This is certainly not the case for the CPS translation we have used in this paper. For example, it follows from the theory in [15] that in the ν-calculus the values $v_1 \triangleq \lambda f \to (\lambda x \to \mathsf{true})(f\,\mathsf{true})$ and $v_2 \triangleq \lambda f \to \mathsf{true}$ satisfy $\emptyset \vdash v_1 \approx_\nu v_2 : (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}$. However, in the $\lambda\nu$-calculus one has

$$\emptyset \vdash v_1^\circ \not\approx_{\lambda\nu} v_2^\circ : \mathcal{C}\overline{(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}}$$

because one can calculate from the definition in Fig. 3 that $v_1^\circ(\lambda f \to f\,F\,T) \Downarrow_{\lambda\nu} \mathsf{false}$ and $v_2^\circ(\lambda f \to f\,F\,T) \Downarrow_{\lambda\nu} \mathsf{true}$, where $F \triangleq \lambda x \to \lambda k \to \mathsf{false} \in \mathit{Term}(\overline{\mathsf{Bool} \to \mathsf{Bool}})$ and $T \triangleq \lambda x \to \mathsf{true} \in \mathit{Term}(\mathsf{Bool} \to \mathsf{Bool})$. (For simplicity we have used a pair of values whose equivalence depends upon the absence of non-terminating features in the ν-calculus; more complicated counter-examples exist if one adds recursion to the calculi.)

Note that these evaluations do not involve the novel parts of Fig. 3 to do with locally scoped names. Thus this failure of full abstraction has more to do with the nature of continuation-passing transformations than with locally scoped names. Can the CPS transformation we have studied here be modified to give a translation of dynamic allocation into a calculus with Odersky-style local names that is fully abstract? One possibility is to change to a version of $\lambda\nu$ with linear function types ($\multimap$) and make use of *linearly used continuations*, $((-) \to R) \multimap R$. In particular, it would be interesting to consider the relationship between dynamic allocation and Odersky-style locally scoped names within the *enriched effect calculus* of Egger *et al*, for which the linearly-used CPS translation has a very strong self-duality property [4]. Another possibility is to add locally scoped names to the polymorphic λ-calculus and use *continuations with polymorphic result type*, $\forall R.\,((-) \to R) \to R$, for which the work of Ahmed and Blume [1] suggests there may be a full abstraction result.

Should one care about the full abstraction property? The ν-calculus and the $\lambda\nu$-calculus are not ends in themselves; they are merely vehicles for studying the semantics of higher-order functions with locally scoped names in as simple a setting as possible. One should certainly consider extending the results of this paper to richer languages, beginning by making them Turing-powerful. This could be done by adding fixpoint recursion for functions. It is reasonable to expect the CPS transformation to extend to a computationally adequate translation of such extended languages; whereas any full abstraction result for a modification of the translation probably would not survive such additions.

8 Translating $\lambda\nu$ to ν

Having given a computationally adequate translation of ν-calculus into $\lambda\nu$-calculus, it is natural to consider such a translation in the reverse direction as well. We sketch one in this section, leaving the details for future work.

The main idea is to translate an Odersky-style locally scoped name (at type T say) into the ν-calculus by dynamically generating a fresh name a and then applying to the translated body a function $a\setminus_T \in \mathit{Val}(T \to T)$ that implements the operation $c \mapsto a\setminus c$ in Fig. 3. This

is defined by recursion on the structure of the type T:

$$
\begin{aligned}
a\setminus_{\mathsf{Name}} &= \lambda x \to \mathsf{if}\ x = a\ \mathsf{then}\ \nu a.\, a\ \mathsf{else}\ x \\
a\setminus_{\mathsf{Bool}} &= \lambda x \to x \\
a\setminus_{T_1 \times T_2} &= \lambda x \to \mathsf{let}\ (x_1\ ,\ x_2) = x\ \mathsf{in}\ (a\setminus_{T_1} x_1\ ,\ a\setminus_{T_2} x_2) \\
a\setminus_{T_1 \to T_2} &= \lambda f \to \lambda x \to a\setminus_{T_2} (f\ x).
\end{aligned}
$$

We would like any computationally adequate translation of $\lambda\nu$-calculus into ν-calculus to be robust with respect to adding extra features such as fixpoint recursion, where the difference between call-by-name and call-by-value becomes visible up to observational equivalence. For a translation to adequately reflect the call-by-name evaluation relation in Fig. 3, one could combine the above idea for implementing Odersky-style $\nu a.\,(-)$ with a standard translation of call-by-name into call-by-value based on using a lifting monad $L(-) = \mathsf{Unit} \to (-)$ to delay evaluation at appropriate points. We did not include a one-element type Unit in the λ-calculus of Sect. 2, but could easily have done so; one could instead use $\mathsf{Bool} \to (-)$ for L. A simpler alternative would be to switch to a call-by-value version of the $\lambda\nu$-calculus.

9 Conclusion

We have shown that Odersky's semantics for $\nu a.\, t$ provides a direct meaning for locally scoped names in higher-order functions (and pairs) from which the more common semantics in terms of dynamic allocation can be recovered via a continuation-passing transformation. This does not help much with understanding the subtle properties of observational equivalence in the Pitts-Stark ν-calculus, because of the complicated nature of the CPS translation. However, the result does shed new light upon the expressive power of the relatively unfamiliar semantics of local names given by Odersky. We have seen that dynamically allocated local names can be encoded with Odersky-style local names. That suggests re-evaluating the usefulness of Odersky's notion. We conclude by mentioning some avenues for doing that.

- Figures 2 and 3 can easily be augmented with evaluation rules for expressions of recursively defined and polymorphic types. We believe our main result (Theorem 5.1) will scale to this extension, using the technique of *step-indexing* to overcome the difficulty of defining a suitable logical relation in the presence of recursive types (see [3], for example). So extended, the $\lambda\nu$-calculus gives a core non-strict functional programming language with Odersky-style local names. For reasons of efficiency one would prefer call-by-need rather than the call-by-name operational semantics in Figure 3. Can one of the standard operational descriptions of call-by-need [9, 17] be combined with this form of locally scoped name? The difficulty is to reconcile its characteristic property of 'scope intrusion', that is, moving $\nu a.\,(-)$ inward to evaluation sites, with local (recursively defined) heaps, $\mathsf{let}\{x_1 = e_1, \ldots, x_n = e_n\}\ \mathsf{in}\ (-)$.
- Odersky's $\nu a.\,(-)$ gives a version of locally scoped names whose evaluation is free of side-effects. Therefore it makes sense to add it to meta-languages for describing the denotational semantics of effects, such as Moggi's computational λ-calculus [11] or the enriched effect calculus of Egger *et al* [4]. Is this a useful extension of such languages?
- Following [14], it should be possible to produce a version of FreshML [19] in which the use of dynamically allocated names is replaced by Odersky-style local names and yet the language still respects α-renaming of bound names in data, up to observational equivalence. The convenient expressive power of FreshML would not be affected and one would regain programming laws for observational equivalence (such as extensionality of

function expressions) that are disrupted by dynamic allocation. (This purer version of FreshML would not pass all of Pottier's criteria for purity [16], since it would admit the anonymous name $\nu a.\, a$ as a value.)

- Fernández and Gabbay [6] consider rewriting for nominal terms extended with 'name generation', a non-binding scoping construct. The formal relationship between this notion and dynamically allocated, or Odersky-style, local names needs clarifying. In any case, the combination of Odersky-style local names with term-rewriting seems worth investigating.

Acknowledgement

Some of the results in this paper are drawn from Lösch's MPhil dissertation; he gratefully acknowledges the support of a Gates Cambridge Scholarship.

References

1 A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proc. ICFP 2008*, pages 157–168. ACM Press.

2 N. Benton and V. Koutavas. A mechanized bisimulation for the nu-calculus. Technical Report MSR-TR-2008-129, Microsoft Research Cambridge, October 2007.

3 D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proc. ICFP 2010*, pages 143–156. ACM Press.

4 J. Egger, R. Møgelberg, and A. Simpson. Linearly-used continuations in the enriched effect calculus. In *Proc. FOSSACS 2010*, volume 6014 of Springer *Lecture Notes in Computer Science*, pages 18–32.

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

6 M. Fernández and M. J. Gabbay. Nominal rewriting with name generation: Abstraction vs. locality. In *Proc. PPDP 2005*, pages 47–58. ACM Press.

7 M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

8 S. B. Lassen. Relational reasoning about contexts. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 91–135. Cambridge University Press, 1998.

9 J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL 1993*, pages 144–154. ACM.

10 R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

12 M. Odersky. A functional theory of local names. In *Proc. POPL 1994*, pages 48–59. ACM Press.

13 A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

14 A. M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 21(3):235–286, 2011.

15 A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS 1993*, volume 711 of Springer *Lecture Notes in Computer Science*, pages 122–141.

16 F. Pottier. Static name control for FreshML. In *Proc. LICS 2007*, pages 356–365. IEEE Computer Society Press.

17 P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7:231–264, 1997.

18 M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

19 M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP 2003*, pages 263–274. ACM Press.

20 I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.