A Program Logic for Fresh Name Generation

Harold Pancho Eliott[1] and Martin Berger[1,2]

[1] Department of Informatics, University of Sussex, Brighton, UK.
[2] Turing Core, Huawei 2012 Labs, London, UK.

Abstract. We present a program logic for Pitts and Stark's ν-calculus, an extension of the call-by-value simply-typed λ-calculus with a mechanism for the generation of fresh names. Names can be compared for equality and inequality, producing programs with subtle observable properties. Hidden names produced by interactions between generation and abstraction are captured logically with a second-order quantifier over type contexts. We illustrate usage of the logic through reasoning about well-known difficult cases from the literature.

1 Introduction

Naming is a long-standing problem in computer science. Most programming languages can define naming constructs, which, when called, yield a fresh name. The π-calculus [11] made naming and the ν-operator, a constructor for name creation, a first-class construct, leading to a flurry of research, e.g. [6–8, 14, 15, 18]. Initially it was unclear if the π-calculus approach had purchase beyond process calculi. Pitts and Stark [16] as well as Odersky [12] added the ν-operator to the simply-typed λ-calculus (STLC from now on), and showed that the subtleties of naming are already present in the interplay between higher-order functions and fresh name generation. This raises the question of how compositionally to reason about programs that can generate fresh names? There are program logics for ML-like languages that can generate fresh references, such as [4, 19], but, to the best of our knowledge, always in the context of languages with other expressive features such as aliasing, mutable higher-order state or pointer arithmetic, leading to complex logics, where the contribution of fresh name generation to the difficulties of reasoning is not apparent. This is problematic because, while the type Nm carries the same information as $\mathsf{Ref}(\mathsf{Unit})$ in ML, we are often interested in reasoning about languages that combine fresh name generation with other features, such as meta-programming [3]. Can we study reasoning about fresh names in as simple a programming language as possible?

> **Research question**. Is there a Hoare-style program logic for the ν-calculus, conservatively extending program logics for the STLC in a natural manner, that allows for compositional reasoning about fresh name generation?

The present paper gives an affirmative answer to the research question, and presents the first program logic for the ν-calculus.

Informal explanation. By the ν-calculus we mean the STLC with a type Nm of names, a constructor gensym of type $\mathsf{Unit} \to \mathsf{Nm}$ and a destructor, in form of equality and inequality on names (gensym and ν are essentially identical, but the former is more widely used). Immediately we realise that the ν-calculus loses extensionality, as $\mathsf{gensym}() = \mathsf{gensym}()$ evaluates to false. While the loss of extensionality is expected in a stateful language, the ν-calculus does not have state, at least not in a conventional sense.

A first difficulty is expressing freshness in logic. What does it mean for a name x to be fresh? A first idea might be to say that x is guaranteed to be distinct from all existing names. We cannot simply say

$$\{\mathsf{T}\}\ \mathsf{gensym}()\ :_u\ \{\forall x.u \neq x\}$$

since we must prevent $\forall x.u \neq x$ being instantiated to $u \neq u$. We want to say something like:

$$\forall x.\{\mathsf{T}\}\ \mathsf{gensym}()\ :_u\ \{u \neq x\} \tag{1}$$

Unfortunately we cannot quantify over Hoare triples. A second problem is that (1) is not strong enough, in the sense that gensym does not just create names that are fresh w.r.t. existing names, but also w.r.t. all future calls to gensym. We introduce a new quantifier to deal with both problems at the same time. A third difficulty is that fresh names can be exported or remain hidden, with observable consequences. Consider:

$$\mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \lambda y.x = y \tag{2}$$

of type $\mathsf{Nm} \to \mathsf{Bool}$. It receives a name y as argument and compares it with fresh name x. Since x is never exported to the outside, no context can ever supply this fresh x to $\lambda y.x = y$. Hence (2) must be contextually indistinguishable from $\lambda y.\mathsf{false}$. Operationally, this is straightforward. But how can we prove this compositionally? Note that this is not a property of $\lambda y.x = y$, but it is also not a consequence of x's being freshly generated, for x is also fresh in this program:

$$\mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \langle x, \lambda y.x = y\rangle \tag{3}$$

But in (3), $\lambda y.x = y$ can return true, for example if we use (3) in this context:

$$\mathsf{let}\ p = [\cdot]\ \mathsf{in}\ (\pi_2\ p)(\pi_1\ p)$$

In program logics like [9], the specification of any abstraction $\lambda y.M$ will be a universally quantified formula $\forall y.A$. With fresh names, instantiation of quantification is a core difficulty. Recall that in first-order logic, $\forall y.A$ always implies $A[e/y]$, for all ambient expressions e. Clearly, in the case of (2) we cannot conclude to $A[x/y]$ from $\forall y.A$, because x is, in some sense, not available for instantiation. In contrast, in the case of (3) we can infer $A[x/y]$. Hence we need to answer the question how to express logically the inability to instantiate a universal quantifier with a fresh and hidden name like x in (2). We introduce a novel restricted quantifier, limiting the values based on a type context, and a new quantifier over type contexts to extend the reach of restricted quantifiers.

2 Programming Language

Our programming language is essentially the ν-calculus of [16], with small additions in particular pairs, included for the sake of convenience. We assume a countably infinite set of variables, ranged over by $x, y, ...$ and a countably infinite set, disjoint from variables, of names, ranged over by $r, ...$ Constants ranged over by c are Booleans true, false, and Unit $()$. For simplicity we also call our language ν-calculus. It is given by the following grammar, where α ranges over types, Γ over *standard type contexts* (STC), V over values and M over programs. (Additions over the STLC highlighted.)

$$\alpha \ ::= \ \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nm} \mid \alpha \to \alpha \mid \alpha \times \alpha \qquad \Gamma ::= \emptyset \mid \Gamma, x : \alpha$$

$$\begin{array}{rcl} V & ::= & r \mid \mathsf{gensym} \mid x \mid c \mid \lambda x.M \mid \langle V, V \rangle \\ M & ::= & V \mid MM \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid M = M \\ & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \langle M, M \rangle \mid \pi_i(M) \end{array}$$

Free variables in M, written $\mathsf{fv}(M)$ are defined as usual. M is *closed* if $\mathsf{fv}(M) = \emptyset$. There are no binders for names so the set $\mathring{\mathsf{a}}(M)$ of *all names* in M, is given by the obvious rules, including $\mathring{\mathsf{a}}(r) = \{r\}$, $\mathring{\mathsf{a}}(MN) = \mathring{\mathsf{a}}(M) \cup \mathring{\mathsf{a}}(N)$. If $\mathring{\mathsf{a}}(M) = \emptyset$ then M is *compile-time* syntax. The $\nu n.M$ constructor from the ν-calculus [16] is equivalent to $\mathsf{let}\ n = \mathsf{gensym}()\ \mathsf{in}\ M$ as $\mathsf{gensym}()$ generates fresh names. The typing judgements is $\Gamma \vdash M : \alpha$, with the STC Γ being an unordered mapping from variables to types. Typing rules are standard [13] with the following extensions: $\Gamma \vdash \mathsf{gensym} : \mathsf{Unit} \to \mathsf{Nm}$ and $\Gamma \vdash r : \mathsf{Nm}$.

The operational semantics of our ν-calculus is straightforward and the same as [16]. A *configuration of type* α is a pair (G, M) where M is a closed term of type α, and G a finite set of previously Generated names such that $\mathring{\mathsf{a}}(M) \subseteq G$. The standard call-by-value reduction relation, $\to$, has the following key rules.

$$\begin{array}{rcll} (G,\ (\lambda x.M)V) & \to & (G,\ M[V/x]) & \\ (G,\ \mathsf{gensym}()) & \to & (G \cup \{n\},\ n) & (n \notin G) \\ (G \cup \{n\},\ n = n) & \to & (G \cup \{n\},\ \mathsf{true}) & \\ (G \cup \{n_1, n_2\},\ n_1 = n_2) & \to & (G \cup \{n_1, n_2\},\ \mathsf{false}) & (n_1 \neq n_2) \end{array}$$

$$(G,\ M) \to (G',\ N) \quad \text{implies} \quad (G,\ \mathcal{E}[M]) \to (G',\ \mathcal{E}[N])$$

Here $M[V/x]$ is the usual capture-avoiding substitution, and $\mathcal{E}[\cdot]$ ranges over the usual reduction contexts of the STLC. Finally, $\Downarrow$ is short for $\to^*$.

3 Logical Language

This section defines the syntax of the logic. As is customary for program logics, ours is an extension of first order logic with equality (alongside axioms for arithmetic). *Expressions*, ranged over by e,e',..., *formulae*, ranged over by A, B, C, ...

and *Logical Type Contexts* (LTCs), ranged over by $\mathit{\Gamma}$, $\mathit{\Gamma}'$, $\mathit{\Gamma}_i$, ... , are given by the grammar below. (Extensions over [9] highlighted.)

$$\begin{array}{lcl} e & ::= & x^\alpha \mid c \mid \langle e, e\rangle \mid \pi_i(e) \\ \mathit{\Gamma} & ::= & \emptyset \mid \mathit{\Gamma}{+}x:\alpha \mid \mathit{\Gamma}{+}\delta:\mathbb{TC} \\ A & ::= & e = e \mid \neg A \mid A \wedge A \mid e \bullet e = x^\alpha\{A\} \mid \forall x^\alpha \in (\mathit{\Gamma}).A \mid \forall \delta.A \end{array}$$

Expressions, e, are standard, where constants, c, range over Booleans and (), but do *not* include names or gensym as constants. Equality, negation and conjunction are standard. Evaluation formulae $e \bullet e' = m\{A\}$ internalise triples [9] and express that if the program denoted by e is executed with argument denoted by e', then the result, denoted by m, satisfies A. Since the ν-calculus has no recursion, all applications terminate and we do not distinguish partial from total correctness. We write $e_1 \bullet e_2 = e_3$ as shorthand for $e_1 \bullet e_2 = m\{m = e_3\}$.

Given variables represent values, ensuring hidden names cannot be revealed in an unsafe manner requires the idea that a value is *derived* from an LTC if a name free term uses the variables in the LTC to evaluate to said value. Specifically define a name as reachable from said LTC if it can be derived from it, and hidden otherwise.

Freshness is not an absolute notion. Instead, a name is fresh with respect to something, in this case names generated in the past, and future of the computation. Formulae refer to names by variables, and variables are tracked in the STC. Freshness is now defined in two steps: (1) First we characterise freshness of a name w.r.t. the current STC, meaning the name cannot be derived from the variables in the STC. Then, (2) we define freshness w.r.t. all future extension of the current STC, details in Sec. 4. The modal operator is used in [19] in order to express "for all future extensions", but we found modalities inconvenient, since they don't allow us to name extensions. We introduce a new quantifier $\forall x^\alpha \in (\mathit{\Gamma}).A$ instead, where $\mathit{\Gamma}$ ranges over LTCs from which x can be derived. To make this precise, we need LTCs (explained next), a generalisation of STCs.

LTCs. Like STCs, LTCs map variables to types, and are needed for typing expressions, formulae and triples (introduced in Sec. 6), LTCs generalise STCs in two ways: they are *ordered*, and they don't just contain program variables, but also *type context variables* (TCVs), ranged over by δ. TCVs are always mapped to the new type $\mathbb{TC}$, short for *type context*. The ordering in LTCs is essential because $\mathit{\Gamma}{+}\delta:\mathbb{TC}$ implies δ represents an *extension* of the LTC $\mathit{\Gamma}$.

Restricted universal quantification. The meaning of $\forall x^\alpha \in (\mathit{\Gamma}).A$ is intuitively simple: A must be true for all x that range only over values of type α, derived from $\mathit{\Gamma}$ that do *not* reveal hidden names. For example if the model contained the name r but only as $\lambda y.y = r$, then r was hidden and whatever x in $\forall x^\alpha \in (\mathit{\Gamma}).A$ ranged over, it must not reveal r. Formalising this requirement is subtle.

Quantification over LTCs. Below we formalise the axiomatic semantics of gensym by saying that the result of each call to this function is fresh w.r.t. all

future extensions of the present state (with the present state being included). The purpose of $\forall\delta.A$ is to allow us to do so: $\forall\delta.A$ implies for all future states derived from the current state (included), when the LTC for that state is assigned to the TCV δ, then A holds.

A convenient shorthand, the freshness predicate. We express freshness of the name x relative to the LTC $\mathit{\Gamma}$ as $\forall z \in (\mathit{\Gamma}).x \neq z.$ and, as this predicate is used pervasively, abbreviate it to $x\#\mathit{\Gamma}$. Intuitively, $x\#\mathit{\Gamma}$, a variant of a similar predicate in [19], states that the name denoted by x is not derivable, directly or indirectly, from the LTC $\mathit{\Gamma}$.

Typing of expressions, formulae and triples. We continue with setting up definitions that allow us to type expressions, formulae and triples. The ordered union of $\mathit{\Gamma}$ and $\mathit{\Gamma}'$ with $\mathsf{dom}(\mathit{\Gamma}) \cap \mathsf{dom}(\mathit{\Gamma}') = \emptyset$ is written $\mathit{\Gamma}+\mathit{\Gamma}'$, and should be understood as: every variable from $\mathsf{dom}(\mathit{\Gamma})$ comes before every variable from $\mathsf{dom}(\mathit{\Gamma}')$. Other abbreviations include $\exists x^{\alpha} \in (\mathit{\Gamma}).A \stackrel{def}{=} \neg\forall x^{\alpha} \in (\mathit{\Gamma}).\neg A$, and where α is obvious $(\mathit{\Gamma}+y) \stackrel{def}{=} (\mathit{\Gamma}+y : \alpha)$ (respectively $(\mathit{\Gamma}+\delta) \stackrel{def}{=} (\mathit{\Gamma}+\delta : \mathbb{TC})$). For simplicity, where not explicitly required, $\mathit{\Gamma}+\delta$ is written δ. Functions on LTCs are defined as expected including mapping variables, $\mathit{\Gamma}(x)$, and TCVs, $\mathit{\Gamma}(\delta)$; obtaining the domain, $\mathsf{dom}(\mathit{\Gamma})$; ordered removal of a variable, $\mathit{\Gamma}\backslash x$; ordered removal of all TCV, $\mathit{\Gamma}\backslash_{-TCV}$; and removal of TCV to produce a STC, $\mathit{\Gamma}\downarrow_{-TC}$. We define free variables of LTC, $\mathsf{fv}(\mathit{\Gamma}) \stackrel{def}{=} \mathsf{dom}(\mathit{\Gamma}\downarrow_{-TC}) \stackrel{def}{=} \mathsf{dom}(\mathit{\Gamma}\backslash_{-TCV})$, then free variables of formulae defined as expected, with the addition of $\mathsf{fv}(x\#\mathit{\Gamma}) \stackrel{def}{=} \mathsf{fv}(\mathit{\Gamma}) \cup \{x\}$, $\mathsf{fv}(\forall x \in (\mathit{\Gamma}).A) \stackrel{def}{=} (\mathsf{fv}(A)\backslash\{x\}) \cup \mathsf{fv}(\mathit{\Gamma})$, and $\mathsf{fv}(\forall\delta.A) \stackrel{def}{=} \mathsf{fv}(A)$. Similarly $\mathsf{ftcv}(\mathit{\Gamma})$ and $\mathsf{ftcv}(A)$ define all TCV occurring in $\mathit{\Gamma}$ and unbound by $\forall\delta.$ in A respectively, calling $\mathit{\Gamma}$ *TCV-free* if $\mathsf{ftcv}(\mathit{\Gamma}) \stackrel{def}{=} \emptyset$. The typing judgement for LTCs, written $\mathit{\Gamma} \Vdash \mathit{\Gamma}'$, checks that $\mathit{\Gamma}'$ is an 'ordered subset' of $\mathit{\Gamma}$. Type checks on expressions, formulae and triples use LTC as the base, written $\mathit{\Gamma} \Vdash e : \alpha$, $\mathit{\Gamma} \Vdash A$ and $\mathit{\Gamma} \Vdash \{A\}\ M :_u \{B\}$ respectively. Fig. 1 gives the rules defining the typing judgements. From now on we adhere to the following convention: *All expressions, formulae and triples are typed*, and we will mostly omit being explicit about typing.

Advanced substitutions. Reasoning with quantifiers requires quantifier instantiation. This is subtle with $\forall\delta.A$, and we need to define two substitutions, $A[e/x]_{\mathit{\Gamma}}$ (substitutes expressions for variables) and $A[\mathit{\Gamma}_0/\delta]_{\mathit{\Gamma}}$ (LTCs substituted for TCVs). First extend the definition *e is free for* x^{α} *in* A in [10], to ensure if e contains destructors i.e. $\pi_i(\ )$ or $=$, then all free occurrences of x in any LTC $\mathit{\Gamma}_0$ in A must imply $\mathit{\Gamma}_0 \Vdash e : \alpha$. Below, we assume the standard substitution $e[e'/x]$ of expressions for variables in expressions, simple details omitted.

We define $A[e/x]_{\mathit{\Gamma}}$, *logical substitution of e for x in A in the context of* $\mathit{\Gamma}$, if e is free for x in A and e is typed by $\mathit{\Gamma}$, by the following clauses (simple cases omitted) and the auxiliary operation on LTCs below. We often write $A[e/x]$ for $A[e/x]_{\mathit{\Gamma}}$.

$$\frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \Vdash b : \mathsf{Bool}} \quad \frac{-}{\Gamma \Vdash () : \mathsf{Unit}} \quad \frac{\Gamma(x) = \alpha}{\Gamma \Vdash x : \alpha} \quad \frac{\Gamma \Vdash e : \alpha \quad \Gamma \Vdash e' : \beta}{\Gamma \Vdash \langle e, e' \rangle : \alpha \times \beta} \quad \frac{\Gamma \Vdash e : \alpha_1 \times \alpha_2}{\Gamma \Vdash \pi_i(e) : \alpha_i}$$

$$\frac{-}{\Gamma \Vdash \emptyset} \quad \frac{\Gamma \Vdash \Gamma_0}{\Gamma + x : \alpha \Vdash \Gamma_0 + x : \alpha} \quad \frac{\Gamma \Vdash \Gamma_0}{\Gamma + \delta \Vdash \Gamma_0 + \delta} \quad \frac{\Gamma \Vdash \Gamma_0}{\Gamma + \Gamma' \Vdash \Gamma_0}$$

$$\frac{\Gamma \Vdash e_1 : \alpha \quad \Gamma \Vdash e_2 : \alpha}{\Gamma \Vdash e_1 = e_2} \quad \frac{\Gamma \Vdash A_1 \quad \Gamma \Vdash A_2}{\Gamma \Vdash A_1 \wedge A_2} \quad \frac{\Gamma \Vdash A}{\Gamma \Vdash \neg A} \quad \frac{\Gamma + \delta : \mathbb{TC} \Vdash A}{\Gamma \Vdash \forall \delta . A}$$

$$\frac{\Gamma \Vdash e : \alpha \to \beta \quad \Gamma \Vdash e' : \alpha \quad \Gamma + x : \beta \Vdash A}{\Gamma \Vdash e \bullet e' = x^{\beta}\{A\}} \quad \frac{\Gamma \Vdash x : \mathsf{Nm} \quad \Gamma \Vdash \Gamma'}{\Gamma \Vdash x \# \Gamma'}$$

$$\frac{\Gamma \Vdash \Gamma' \quad \Gamma + x : \alpha \Vdash A}{\Gamma \Vdash \forall x^{\alpha} \in (\Gamma') . A} \quad \frac{\Gamma \Vdash A \quad \Gamma \downarrow_{-TC} \vdash M : \alpha \quad \Gamma + m : \alpha \Vdash B}{\Gamma \Vdash \{A\}\ M :_m \{B\}}$$

Fig. 1. Typing rules for LTCs, expressions, formulae and triples (see Sec. 6). Simple cases omitted. M in the last rule is compile-time syntax.

$$\begin{array}{rcll}
(e_1 \bullet e_2 = m\{A\})[e/x]_{\Gamma} & \stackrel{def}{=} & e_1[e/x] \bullet e_2[e/x] = m\{A[e/x]_{\Gamma + m}\} & (x \neq m,\ m \notin \mathsf{fv}(\Gamma)) \\
(y \# \Gamma')[e/x]_{\Gamma} & \stackrel{def}{=} & y[e/x] \# (\Gamma'[e/x]_{\Gamma}) & \\
(\forall m \in (\Gamma') . A)[e/x]_{\Gamma} & \stackrel{def}{=} & \forall m \in (\Gamma'[e/x]_{\Gamma}) . (A[e/x]_{\Gamma + m}) & (x \neq m,\ m \notin \mathsf{fv}(\Gamma)) \\
(\forall \delta . A)[e/x]_{\Gamma} & \stackrel{def}{=} & \forall \delta . (A[e/x]_{\Gamma + \delta}) & \\
\Gamma'[e/x]_{\Gamma} & \stackrel{def}{=} & \begin{cases} \Gamma'_e \text{ s.t. } \mathsf{dom}(\Gamma'_e) = \mathsf{fv}(e) \cup \mathsf{dom}(\Gamma' \backslash x), \Gamma \Vdash \Gamma'_e & x \in \mathsf{dom}(\Gamma') \\ \Gamma' & x \notin \mathsf{dom}(\Gamma') \end{cases} &
\end{array}$$

Type context substitution $A[\Gamma_0/\delta]_{\Gamma}$ instantiates δ with Γ_0 in A, similar to classical substitution. We often write $[\Gamma_0/\delta]$ for $[\Gamma_0/\delta]_{\Gamma}$ as Γ is used for ordering and is obvious. As above, the omitted cases are straightforward and the auxiliary operation on LTCs is included.

$$\begin{array}{rcll}
(x \# \Gamma')[\Gamma_0/\delta]_{\Gamma} & \stackrel{def}{=} & x \# (\Gamma'[\Gamma_0/\delta]_{\Gamma}) & \\
(e_1 \bullet e_2 = m\{A\})[\Gamma_0/\delta]_{\Gamma} & \stackrel{def}{=} & e_1 \bullet e_2 = m\{A[\Gamma_0/\delta]_{\Gamma + m}\} & (m \notin \mathsf{dom}(\Gamma_0)) \\
(\forall x \in (\Gamma') . A)[\Gamma_0/\delta]_{\Gamma} & \stackrel{def}{=} & \forall x \in (\Gamma'[\Gamma_0/\delta]_{\Gamma}) . (A[\Gamma_0/\delta]_{\Gamma + x}) & (x \notin \mathsf{dom}(\Gamma_0)) \\
(\forall \delta' . A)[\Gamma_0/\delta]_{\Gamma} & \stackrel{def}{=} & \begin{cases} (\forall \delta' . A[\Gamma_0/\delta]_{\Gamma + \delta'}) & \delta \neq \delta' \\ \forall \delta . A & \text{otherwise} \end{cases} & (\delta' \notin \mathsf{dom}(\Gamma_0)) \\
\Gamma'[\Gamma_0/\delta]_{\Gamma} & \stackrel{def}{=} & \begin{cases} \Gamma_1 \text{ s.t. } \mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_0, \Gamma'), \Gamma \Vdash \Gamma_1 & \delta \in \mathsf{dom}(\Gamma') \\ \Gamma' & \delta \notin \mathsf{dom}(\Gamma') \end{cases} &
\end{array}$$

4 Model

We define a *model* ξ as a finite (possibly empty) map from variables and TCV to closed values and TCV-free LTCs respectively.

$$\xi \ ::= \ \emptyset \ \mid \ \xi \cdot x : V \ \mid \ \xi \cdot \delta : \mathit{\Gamma}'$$

Standard actions on models ξ are defined as expected and include: variable mappings to values, $\xi(x)$, or TCV mapping to LTC, $\xi(\delta)$; removal of variable x as $\xi \backslash x$ (with $(\xi \cdot \delta : \mathit{\Gamma}_1)\backslash x = (\xi \backslash x) \cdot \delta : (\mathit{\Gamma}_1 \backslash x)$); removal of TCV δ as $\xi \backslash \delta$; removal of all TCVs as $\xi \backslash_{-TCV}$; and defining all names in ξ as $\mathring{\mathsf{a}}(\xi)$ noting that $\mathring{\mathsf{a}}(\mathit{\Gamma}) = \emptyset$.

A model ξ is typed by a LTC $\mathit{\Gamma}$ written $\xi^{\mathit{\Gamma}}$, if $\mathit{\Gamma} \Vdash \xi$ as defined below, were $\mathit{\Gamma}_d = \mathit{\Gamma}_d \backslash_{-TCV}$ formalises that $\mathit{\Gamma}_d$ is TCV-free.

$$\frac{-}{\emptyset \Vdash \emptyset} \qquad \frac{\mathit{\Gamma} \Vdash \xi \quad \emptyset \vdash V : \alpha}{\mathit{\Gamma}{+}x : \alpha \Vdash \xi \cdot x : V} \qquad \frac{\mathit{\Gamma} \Vdash \xi \quad \mathit{\Gamma} \Vdash \mathit{\Gamma}_d \quad \mathit{\Gamma}_d = \mathit{\Gamma}_d \backslash_{-TCV}}{\mathit{\Gamma}{+}\delta \Vdash \xi \cdot \delta : \mathit{\Gamma}_d}$$

The *closure* of a term M by a model ξ, written $M\xi$ is defined as standard with the additions, $\mathsf{gensym}\xi \stackrel{def}{=} \mathsf{gensym}$ and $r\xi \stackrel{def}{=} r$. Noting that $M\xi\backslash_{-TCV} = M\xi = M\xi \cdot \delta : \mathit{\Gamma}'$ holds for all δ and $\mathit{\Gamma}'$ as $\mathit{\Gamma} \downarrow_{-TC} \vdash M : \alpha$.

The *interpretation of expression* e in a model $\xi^{\mathit{\Gamma}}$, written $[\![e]\!]_\xi$, is standard, e.g. $[\![c]\!]_\xi \stackrel{def}{=} c$, $[\![x]\!]_\xi \stackrel{def}{=} \xi(x)$, $[\![\langle e, e' \rangle]\!]_\xi \stackrel{def}{=} \langle [\![e]\!]_\xi, [\![e']\!]_\xi \rangle$, etc.

The *interpretation of LTCs* $\mathit{\Gamma}_0$ in a model $\xi^{\mathit{\Gamma}}$, written $[\![\mathit{\Gamma}_0]\!]_\xi$, outputs a STC. It is assumed $\mathit{\Gamma} \Vdash$ the LTC in the following definition:

$$[\![\emptyset]\!]_\xi \stackrel{def}{=} \emptyset \qquad [\![\mathit{\Gamma}_0{+}x : \alpha]\!]_\xi \stackrel{def}{=} [\![\mathit{\Gamma}_0]\!]_\xi, x : \alpha \qquad [\![\mathit{\Gamma}_0{+}\delta : \mathbb{TC}]\!]_\xi \stackrel{def}{=} [\![\mathit{\Gamma}_0]\!]_\xi \cup [\![\xi(\delta)]\!]_\xi$$

Write $M \overset{[\mathit{\Gamma},\ \xi]}{\rightsquigarrow} V$ as the *derivation of a value* V from term M which is typed by the LTC $\mathit{\Gamma}$ and closed and evaluated in a model ξ. This ensures names are derived from actual reachable values in ξ as if they were programs closed by the model, hence not revealing hidden names from ξ. $M \overset{[\mathit{\Gamma},\ \xi]}{\rightsquigarrow} V$ holds exactly when:

- $\mathring{\mathsf{a}}(M) = \emptyset$
- $[\![\mathit{\Gamma}]\!]_\xi \vdash M : \alpha$
- $(\mathring{\mathsf{a}}(\xi), M\xi) \Downarrow (\mathring{\mathsf{a}}(\xi) \cup G',\ V)$

Model extensions aim to capture the fact that models represent real states of execution, by stating a model is only constructed by evaluating terms derivable from the model.

A model ξ' is a *single step model extension* to another model $\xi^{\mathit{\Gamma}}$, written $\xi \preccurlyeq \xi'$, if the single new value in ξ' is derived from ξ or the mapped LTC is $\mathit{\Gamma}$ with TCVs removed. Formally $\xi^{\mathit{\Gamma}} \preccurlyeq \xi'$ holds if either of the following hold:

- There is M_y^α such that $M_y \overset{[\mathit{\Gamma},\ \xi]}{\rightsquigarrow} V_y$ and $\xi'^{\mathit{\Gamma}+y:\alpha} = \xi \cdot y : V_y$.
- $\xi'^{\mathit{\Gamma}+\delta} = \xi \cdot \delta : \mathit{\Gamma}\backslash_{-TCV}$ for some δ.

We write $\preccurlyeq^\star$ for the transitive, reflexive closure of $\preccurlyeq$. If $\xi \preccurlyeq^\star \xi'$ we say ξ' is an *extension* of ξ and ξ is a *contraction* of ξ'.

A model ξ is *constructed by* $\mathit{\Gamma}$, written $\mathit{\Gamma} \triangleright \xi$, if any TCV represents a model extension. Formally we define $\mathit{\Gamma} \triangleright \xi$ by the following rules:

$$\frac{-}{\emptyset \triangleright \emptyset} \qquad \frac{\mathit{\Gamma} \triangleright \xi \quad \text{exists } M^{\alpha}.M \overset{[\mathit{\Gamma},\ \xi]}{\rightsquigarrow} V}{\mathit{\Gamma}+x : \alpha \triangleright \xi \cdot x : V} \qquad \frac{\mathit{\Gamma} \triangleright \xi_0 \quad \xi_0 \preccurlyeq^{\star} \xi^{\mathit{\Gamma}_2} \quad \mathit{\Gamma}_1 = \mathit{\Gamma}_2 \backslash_{-TCV}}{\mathit{\Gamma}+\delta \triangleright \xi \cdot \delta : \mathit{\Gamma}_1}$$

A model $\xi^{\mathit{\Gamma}}$ is *well constructed* if there exists an LTC, $\mathit{\Gamma}'$, such that $\mathit{\Gamma}' \triangleright \xi$, noting that $\mathit{\Gamma} \Vdash \mathit{\Gamma}'$.

Model extensions and well constructed models represent models derivable by ν-calculus programs, ensuring names cannot be revealed by later programs. Consider the basic model: $y : \lambda a.\mathsf{if}\ a = r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$, if r_1 could be added to the model, this clearly reveals access to r_2 otherwise r_2 is hidden. Hence the assumption that all models are well constructed from here onwards.

Contextual equivalence of two terms requires them to be contextually indistinguishable in all variable-closing single holed contexts of Boolean type in any valid configuration, as is standard [1, 17]. When M_1 and M_2 are closed terms of type α and $\mathring{\mathsf{a}}(M_1) \cup \mathring{\mathsf{a}}(M_2) \subseteq G$, we write $M_1 \cong_{\alpha}^{G} M_2$ to be equivalent to $G, \emptyset \vdash M_1 \equiv M_2 : \alpha$ from [1].

4.1 Semantics

The *satisfaction relation* for formula A in a well constructed model $\xi^{\mathit{\Gamma}}$, written $\xi \models A$, assumes $\mathit{\Gamma} \Vdash A$, and is defined as follows:

- $\xi \models e = e'$ if $[\![e]\!]_{\xi} \cong_{\alpha}^{\mathring{\mathsf{a}}(\xi)} [\![e']\!]_{\xi}$.
- $\xi \models \neg A$ if $\xi \not\models A$.
- $\xi \models A \wedge B$ if $\xi \models A$ and $\xi \models B$.
- $\xi \models e \bullet e' = m\{A\}$ if $[\![e]\!]_{\xi}[\![e']\!]_{\xi} \overset{[\emptyset,\ \xi]}{\rightsquigarrow} V$ and $\xi \cdot m : V \models A$
- $\xi \models \forall x^{\alpha} \in (\mathit{\Gamma}').A$ if for all M. $M \overset{[\mathit{\Gamma}',\ \xi]}{\rightsquigarrow} V$ implies $\xi \cdot x : V \models A$
- $\xi \models \forall \delta.A$ if forall $\xi'^{\mathit{\Gamma}'}.\xi \preccurlyeq^{\star} \xi'$ implies $\xi' \cdot \delta : (\mathit{\Gamma}' \backslash_{-TCV}) \models A$
- $\xi \models x \# \mathit{\Gamma}_0$ if there is no M_x such that $M_x \overset{[\mathit{\Gamma}_0,\ \xi]}{\rightsquigarrow} [\![x]\!]_{\xi}$

In first-order logic, if a formula is satisfied by a model, then it is also satisfied by extensions of that model, and vice-versa (as long as all free variables of the formula remain in the model). This can no longer be taken for granted in our logic. Consider the formula $\forall \delta.\exists z \in (\delta).(z \# \mathit{\Gamma} \wedge \neg z \# \delta)$ Validity of this formula depends on how many names exist in the ambient model: it may become invalid under contracting the model. Fortunately, such formulae are rarely needed when reasoning about programs. In order to simplify our soundness proofs we will therefore restrict some of our axioms and rules to formulae that are stable under model extension and contractions. Sometimes we need a weaker property, where formulae preserve their validity when a variable is removed from a model. Both concepts are defined semantically next.

We define formula A as *model extensions independent*, short Ext-Ind, if for all $\mathit{\Gamma}, \xi^{\mathit{\Gamma}}, \xi'$ such that $\mathit{\Gamma} \Vdash A$ and $\xi \preccurlyeq^{\star} \xi'$ we have: $\xi \models A$ iff $\xi' \models A$.

We define formula A as *thin* w.r.t. x, written A thin w.r.t x^{α}, if for all $\mathit{\Gamma}$ such that $\mathit{\Gamma} \backslash x \Vdash A$ and $x^{\alpha} \in \mathsf{dom}(\mathit{\Gamma})$ we have for all well constructed models $\xi^{\mathit{\Gamma}}$ and $\xi \backslash x$ that: $\xi \models A$ implies $\xi \backslash x \models A$.

5 Axioms

Axioms and axiom schemas are similar in intention to those of the logic for the STLC, but expressed within the constraints of our logic. Axiom schemas are indexed by the LTC that types them and the explicit types where noted. We introduce the interesting axioms (schemas) and those used in Sec. 7.

Equality axioms are standard where $(eq1)$ allows for substitution. Most axioms for universal quantification over LTCs $(u1)$-$(u5)$ are inspired by those of first order logic. The exceptions are $(u2)$ which allows for the reduction of LTCs and $(u5)$ which holds only on Nm-free types. Axioms for existential quantification over LTCs $(ex1)$-$(ex3)$ are new aside from $(ex1)$ which is the dual of $(u1)$. Axiom $(ex2)$ introduces existential quantification from evaluation formulae that produce a fixed result. Reducing $\mathit{\Gamma}$ in $\exists x \in (\mathit{\Gamma}).A$ is possible via $(ex3)$ for a specific structure. We use base types $\alpha_\mathsf{b} ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \alpha_\mathsf{b} \times \alpha_\mathsf{b}$ as core lambda calculus types excluding functions. Freshness axioms $(f1)$-$(f2)$ show instances LTCs can be extended, whereas $(f3)$-$(f4)$ reduce the LTC. Axiom $(f1)$ holds due to f being derived from $\mathit{\Gamma}{+}x$, and the rest are trivial.

$$
\begin{array}{lrcll}
(eq1) & \mathit{\Gamma} \Vdash A(x) \wedge x = e & \leftrightarrow & A(x)[e/x]_{\mathit{\Gamma}} & \\
(u1) & \mathit{\Gamma} \Vdash \forall x^{\alpha} \in (\mathit{\Gamma}_0).A & \rightarrow & A[e/x]_{\mathit{\Gamma}} & \mathit{\Gamma}_0 \vdash e : \alpha \\
(u2) & \forall x \in (\mathit{\Gamma}_0{+}\mathit{\Gamma}_1).A & \rightarrow & (\forall x \in (\mathit{\Gamma}_0).A) \wedge (\forall x \in (\mathit{\Gamma}_1).A) & \\
(u3) & A^{-x} & \leftrightarrow & \forall x \in (\mathit{\Gamma}_0).A^{-x} & A - \mathsf{Ext\text{-}Ind} \\
(u4) & \forall x \in (\mathit{\Gamma}_0).(A \wedge B) & \leftrightarrow & (\forall x \in (\mathit{\Gamma}_0).A) \wedge (\forall x \in (\mathit{\Gamma}_0).B) & \\
(u5) & \forall x^{\alpha} \in (\mathit{\Gamma}_0).A & \leftrightarrow & \forall x^{\alpha} \in (\emptyset).A & \alpha \text{ is } \mathsf{Nm}\text{-free}
\end{array}
$$

$$
\begin{array}{lrcll}
(ex1) & \mathit{\Gamma} \Vdash A[e/x]_{\mathit{\Gamma}} & \rightarrow & \exists x' \in (\mathit{\Gamma}_0).A & \mathit{\Gamma} \Vdash \mathit{\Gamma}_0 \text{ and } \mathit{\Gamma}_0 \vdash e : \alpha \\
(ex2) & \mathit{\Gamma}{+}x{+}\mathit{\Gamma}_0 \Vdash a \bullet b = c\{c = x\} & \rightarrow & \exists x' \in (\mathit{\Gamma}_0).x = x' & \{a, b\} \subseteq \mathsf{dom}(\mathit{\Gamma}_0) \\
(ex3) & \mathit{\Gamma}{+}x \Vdash \forall y \in (\emptyset).\exists z^{\mathsf{Nm}} \in (\mathit{\Gamma}_0{+}y).x = z & \rightarrow & \exists z \in (\mathit{\Gamma}_0).x = z &
\end{array}
$$

$$
\begin{array}{lrcll}
(f1) & \mathit{\Gamma}{+}x{+}f : \alpha \rightarrow \alpha_\mathsf{b} \Vdash x\#\mathit{\Gamma} & \rightarrow & x\#\mathit{\Gamma}{+}f : \alpha \rightarrow \alpha_\mathsf{b} & \\
(f2) & \mathit{\Gamma} \Vdash x\#\mathit{\Gamma}_0 \wedge \forall y^{\alpha} \in (\mathit{\Gamma}_0).A & \leftrightarrow & \forall y^{\alpha} \in (\mathit{\Gamma}_0).(x\#(\mathit{\Gamma}_0{+}y) \wedge A) & \\
(f3) & x\#\mathit{\Gamma}_0 & \rightarrow & x \neq e & \mathit{\Gamma}_0 \vdash e : \mathsf{Nm} \\
(f4) & x\#(\mathit{\Gamma}_0{+}\mathit{\Gamma}_1) & \rightarrow & x\#\mathit{\Gamma}_0 \wedge x\#\mathit{\Gamma}_1 &
\end{array}
$$

Axioms for quantification over LTCs are also similar to those for the classical universal quantifier except (utc2) which extends the restricted quantifier to any future LTC which can only mean adding fresh names.

$$
\begin{array}{lrcll}
(utc1) & \mathit{\Gamma} \Vdash \forall \delta.A & \rightarrow & A[\mathit{\Gamma}/\delta]_{\mathit{\Gamma}} & \\
(utc2) & \mathit{\Gamma} \Vdash \forall x^{\mathsf{Nm}} \in (\mathit{\Gamma}).A^{-\delta} & \leftrightarrow & \forall \delta.\forall x^{\mathsf{Nm}} \in (\mathit{\Gamma}{+}\delta).A & A - \mathsf{Ext\text{-}Ind} \\
(utc3) & A^{-\delta} & \leftrightarrow & \forall \delta.A^{-\delta} & A - \mathsf{Ext\text{-}Ind} \\
(utc4) & \forall \delta.(A \wedge B) & \leftrightarrow & (\forall \delta.A) \wedge (\forall \delta.B) &
\end{array}
$$

Axioms for the evaluation formulae are similar to those of [2]. The interaction between evaluation formulae and the new constructors are shown. All STLC

values are included in the variables of Nm-free type, and if we let $\mathsf{Ext}(e_2, e_2)$ stand for $\forall x \in (\emptyset).e_1 \bullet x = m_1\{e_2 \bullet x = m_2\{m_1 = m_2\}\}$ then (ext) maintains extensionality in this logic for the STLC terms. Typing restrictions require $m \notin \mathsf{fv}(A)$ in $(e1)$ and $\mathsf{fv}(e_1, e_2, m) \cap \mathsf{dom}(\mathit{\Gamma} + x) = \emptyset$ in $(e2)$.

$$
\begin{array}{llcll}
(e1) & e_1 \bullet e_2 = m\{A \wedge B\} & \leftrightarrow & (A \wedge e_1 \bullet e_2 = m\{B\}) & A - \mathsf{Ext\text{-}Ind} \\
(e2) & e_1 \bullet e_2 = m\{\forall x \in (\mathit{\Gamma}).A\} & \leftrightarrow & \forall x \in (\mathit{\Gamma}).e_1 \bullet e_2 = m\{A\}) & \\
(e3) & e_1 \bullet e_2 = m^{\alpha_\flat}\{\forall \delta.A\} & \leftrightarrow & \forall \delta.e_1 \bullet e_2 = m^{\alpha_\flat}\{A\} & A - \mathsf{Ext\text{-}Ind} \\
(ext) & \mathsf{Ext}(e_1, e_2) & \leftrightarrow & e_1 =^{\alpha_1 \to \alpha_2} e_2 & \alpha_1 \to \alpha_2 \text{ is } \mathsf{Nm}\text{-free}
\end{array}
$$

6 Rules

Our logic uses standard *triples* $\{A\}\ M :_m \{B\}$ where in this logic, the program M is restricted to compile-time syntax. Triples are typed by the rule in Fig. 1. Semantics of triples is standard: if the *pre-condition* A holds and the value derived from M is assigned to the *anchor* m then the *post-condition* B holds. In detail: let $\xi^{\mathit{\Gamma}}$ be a model.

$$\xi^{\mathit{\Gamma}} \models \{A\}\ M :_m \{B\} \qquad \text{iff} \qquad \xi \models A \text{ implies } (M \overset{[\mathit{\Gamma},\ \xi]}{\rightsquigarrow} V \text{ and } \xi \cdot m : V \models B)$$

The triple is *valid*, written $\models \{A\}\ M :_m \{B\}$, if for all $\mathit{\Gamma}$ and $\xi_0^{\mathit{\Gamma}_0}$ we have

$$\mathit{\Gamma} \Vdash \{A\}\ M :_m \{B\} \text{ and } \mathit{\Gamma} \triangleright \xi_0 \text{ together imply: } \xi_0 \models \{A\}\ M :_m \{B\}$$

From here on we will assume all models are well constructed, noting that the construction of models is the essence of $\forall\delta.$ as it allows for all possible future names generated. Variables occurring in $\mathsf{dom}(\xi_0) - \mathsf{dom}(\mathit{\Gamma})$ may never occur directly in the triple, but their mapped values will have an effect.

The rules of inference can be found in Fig. 2 and Fig. 3. We write $\vdash \{A\}\ M :_m \{B\}$ to indicate that $\{A\}\ M :_m \{B\}$ can be derived from these rules. Our rules are similar to those of [9] for vanilla λ-calculi, but suitably adapted to the effectful nature of the ν-calculus. All rules are typed. The typing of rules follows the corresponding typing of the programs occurring in the triples, but with additions to account for auxiliary variables. We have two substantially new rules: [Gensym] and [Let]. The former lets us reason about fresh name creation by gensym, the latter about the let $x = M$ in N. Operationally, let $x = M$ in N is often just an abbreviation for $(\lambda x.N)M$, but we have been unable to derive [Let] using the remaining rules and axioms. Any syntactic proof of [Let] requires [Lam] and [App], which requires the postcondition: C thin w.r.t p for p the anchor of the [Lam] rule. We have not been able to prove this thinness for all models of the relevant type.

In comparison with [9, 19], the primary difference with our rules is our substitution. Our changes to substitution only affects [Eq] and [Proj(i)] which are reduced in strength by the new definition of substitution as more constraints

are placed on the formulae to ensure correct substitution occurs. All other rules remain equally strong. The other difference from [9] is the need for thinness to replace the standard 'free from', which is discussed above. Removal of variables via thinness is required in the proof of soundness, for example [App], which produces u from m and n, hence Ext-Ind is insufficient given the order of $\mathit{\Gamma}{+}m{+}n{+}u \Vdash C$, i.e. u introduced after m and n. We explain the novelty in the rules in more detail below.

$$\frac{-}{\{A[x/m]\}\ x :_m \{A\}}\ [\textsc{Var}] \qquad \frac{-}{\{\mathsf{T}\}\ \mathsf{gensym} :_u \{\forall \delta.u \bullet () = m\{m\#\delta\}\}}\ [\textsc{Gensym}]$$

$$\frac{-}{\{A[\mathsf{c}/m]\}\ \mathsf{c} :_m \{A\}}\ [\textsc{Const}] \qquad \frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{C[m = n/u]\}}{\{A\}\ M = N :_u \{C\}}\ [\textsc{Eq}]$$

$$\frac{A - \mathsf{Ext\text{-}Ind} \quad \mathit{\Gamma}{+}\delta{+}x : \alpha \vdash \{A^{-x} \wedge B\}\ M :_m \{C\}}{\mathit{\Gamma} \vdash \{A\}\ \lambda x^{\alpha}.M :_u \{\forall \delta. \forall x^{\alpha} \in (\delta).(B \to u \bullet x = m\{C\})\}}\ [\textsc{Lam}]$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{m \bullet n = u\{C\}\}}{\{A\}\ MN :_u \{C\}}\ [\textsc{App}]$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B[b_i/m]\}\ N_i :_u \{C\} \quad b_1 = \mathsf{true} \quad b_2 = \mathsf{false} \quad i = 1,2}{\{A\}\ \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 :_u \{C\}}\ [\textsc{If}]$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{C[\langle m, n\rangle/u]\}}{\{A\}\ \langle M, N\rangle :_u \{C\}}\ [\textsc{Pair}] \qquad \frac{\{A\}\ M :_m \{C[\pi_i(m)/u]\}}{\{A\}\ \pi_i(M) :_u \{C\}}\ [\textsc{Proj}(i)]$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_u \{C\}}{\{A\}\ \mathsf{let}\ m^{\alpha} = M\ \mathsf{in}\ N :_u \{C\}}\ [\textsc{Let}]$$

Fig. 2. Rules for the core language, cf. [2,9,19]. We require C thin w.r.t m in [Proj(i), Let] and C thin w.r.t m, n in [Eq, App, Pair]. We omit LTCs where not essential.

In [Gensym], $u \bullet () = m\{m\#\delta\}$ indicates the name produced by $u()$ and stored at m is not derivable from the LTC δ. If there were no quantification over LTCs prior to the evaluation we could only say m is fresh from the current typing context, however we want to say that even if there is a future typing context with new names and we evaluate $u()$, this will still produce a fresh name. Hence we introduce the $\forall\delta.$ to quantify over all future LTCs (and hence all future names). Elsewhere in reasoning it is key that the post-condition of [Gensym] is Ext-Ind and hence holds in all extending and contracting models (assuming the anchor for gensym is present), reinforcing the re-applicability of gensym in any context.

Rules for λ-abstraction in previous logics for lambda-calculi [9,19] universally quantify over all possible arguments. Our corresponding [Lam] rule refines this and quantifies over current or future values that do not contain hidden names. Comparing the two LTCs typing the assumption and conclusion implies $\mathit{\Gamma}{+}\delta{+}x$ extends $\mathit{\Gamma}$ to any possible extension assigned to δ, and extends to x a value

derived from $\mathit{\Gamma}+\delta$. Hence the typing implies precisely what is conveyed in the post-condition of the conclusion: '$\forall\delta.\forall x \in (\delta).$'. Constraints on δ and x are introduced by B, and $A - \mathsf{Ext\text{-}Ind}$ implies A still holds in all extensions of $\mathit{\Gamma}$ including $\mathit{\Gamma}+\delta+x$. The rest is trivial when we consider $((\lambda x.M)x)\xi \cong_{\alpha}^{\mathring{\mathsf{a}}(\xi)} M\xi$.

The STLC's [Let] rule introduces x in the post-condition by means of an '$\exists x.C$'. This fails here as x may be unreachable, hence not derivable from any extending or contracting LTC. The requirement that C thin $\mathsf{w.r.t}$ x ensures x is not critical to C so can either be derived from the current LTC or is hidden. Thinness ensures no reference to the variable m is somehow hidden under quantification over LTCs.

The [Invar] rule is standard with the constraint that $C - \mathsf{Ext\text{-}Ind}$ to ensure C holds in the extension where m has been assigned. The [LetFRESH] rule is commonly used and hence included for convenience, but it is entirely derivable from the other rules.

$$\frac{A \to A' \quad \{A'\}\ M :_m \{B'\} \quad B' \to B}{\{A\}\ M :_m \{B\}}\ [\textsc{Conseq}] \qquad \frac{C - \mathsf{Ext\text{-}Ind} \quad \{A\}\ M :_m \{B\}}{\{A \wedge C\}\ M :_m \{B \wedge C\}}\ [\textsc{Invar}]$$

$$\frac{A - \mathsf{Ext\text{-}Ind} \quad \mathit{\Gamma}+m : \mathsf{Nm} \vdash \{A \wedge m\#\mathit{\Gamma}\}\ M :_u \{C\}}{\mathit{\Gamma} \vdash \{A\}\ \mathsf{let}\ m = \mathsf{gensym}()\ \mathsf{in}\ M :_u \{C\}}\ [\textsc{LetFresh}]$$

Fig. 3. Key structural rules [Conseq] and [Invar] and for convenience the derived [LetFRESH] rule where C thin $\mathsf{w.r.t}$ m is required.

Theorem 1. *All axioms and rules are sound.*

Theorem 2. *The logic for the ν-calculus is a conservative extension of the logic [9] for the STLC.*

All proofs can be found in the first author's forthcoming dissertation [5].

7 Reasoning Examples

Example 1. We reason about the core construct $\mathsf{gensym}()$ in an LTC $\mathit{\Gamma}$. In Line 2, $(utc1)$ instantiates the postcondition to $b \bullet () = a\{a\#\mathit{\Gamma}+b\}$ and $(f4)$ removes the b from the LTC to ensure the postcondition satisfies the thin $\mathsf{w.r.t}$ b requirement in [App].

1	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{gensym} :_b \{\forall\delta.b \bullet () = a\{a\#\delta\}\}$	[Gensym]
2	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{gensym} :_b \{b \bullet () = a\{a\#\mathit{\Gamma}\}\}$	[Conseq], *(utc1), (f4), 1*
3	$\mathit{\Gamma}+b \Vdash \{b \bullet () = a\{a\#\mathit{\Gamma}\}\}\ () :_c \{b \bullet c = a\{a\#\mathit{\Gamma}\}\}$	[Const]
4	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{gensym}() :_a \{a\#\mathit{\Gamma}\}$	[App], *2, 3*

Example 2. We reason about the comparison of two fresh names, clearly returning false, by applying Example 1 in the relevant LTCs.

1	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{gensym}() :_a \{a\#\mathit{\Gamma}\}$	*See Example 1*
2	$\mathit{\Gamma}+a \Vdash \{\mathsf{T}\}\ \mathsf{gensym}() :_b \{b\#\mathit{\Gamma}+a\}$	*See Example 1*
3	$\mathit{\Gamma}+a \Vdash \{a\#\mathit{\Gamma}\}\ \mathsf{gensym}() :_b \{a \neq b\}$	[Conseq], *(f3), 2*
4	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{gensym}() = \mathsf{gensym}() :_u \{u = \mathsf{false}\}$	[Eq], *3*

Example 3. Placing name generation inside an abstraction halts the production of fresh names until the function is applied. When y is of type Unit then this specification is identical to that of gensym.

1	$\mathit{\Gamma}+\delta+y \Vdash \{\mathsf{T}\}\ \mathsf{gensym}() :_m \{m\#\mathit{\Gamma}+\delta+y\}$	*See Example 1*
2	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \lambda y.\mathsf{gensym}() :_u \{\forall\delta.\forall y \in (\delta).u \bullet y = m\{m\#\mathit{\Gamma}+\delta+y\}\}$	[Lam], *2*

Example 4. Generating a name outside an abstraction and returning that same name in the function is often compared to Example 3 [1,17]. We reason as follows: letting $A_4(p) \stackrel{def}{=} \forall\delta.\forall y \in (\delta).u \bullet y = m\{m\#\mathit{\Gamma} \wedge p = m\}$.

1	$\{x\#\mathit{\Gamma}\}\ x :_m \{m\#\mathit{\Gamma} \wedge x = m\}$	[Var]
2	$\{x\#\mathit{\Gamma}\}\ \lambda y.x :_u \{A_4(x)\}$	[Lam], *1*
3	$\{x\#\mathit{\Gamma}\}\ \lambda y.x :_u \{\exists x' \in (u).A_4(x')\}$	[Conseq], *2*
4	$\mathit{\Gamma} \Vdash \{\mathsf{T}\}\ \mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \lambda y.x :_u \{\exists x' \in (u).A_4(x')\}$	[LetFRESH], *3*

Proof of line 3 above, essentially proves x is derivable from u:

5	$A_4(x) \wedge \forall y \in (\emptyset).u \bullet y = m\{x = m\}$	*(utc1), (u2), FOL*
6	$A_4(x) \wedge \exists x' \in (u).x = x'$	*(ex2), (ex3)*
7	$\exists x' \in (u).(A_4(x) \wedge x = x')$	*(u3), (u4)*
8	$\exists x' \in (u).A_4(x')$	*(eq1)*

Example 5. In order to demonstrate the subtlety of hidden names, the Introduction used Program (2), which was $M \stackrel{def}{=} \mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \lambda y.x = y$. We now use our logic to reason about M.

1	$\Gamma{+}x{+}\delta{+}y \Vdash \{\mathsf{T}\}\ x = y :_m \{m = (x = y)\}$	[Eq]
2	$\Gamma{+}x \Vdash \{\mathsf{T}\}\ \lambda y.x = y :_u \{\forall\delta.\forall y \in (\delta).u \bullet y = (x = y)\}$	[Lam], *1*
3	$\Gamma{+}x \Vdash \{x\#\Gamma\}\ \lambda y.x = y :_u \{x\#\Gamma \wedge \forall\delta.\forall y \in (\delta).u \bullet y = (x = y)\}$	[Invar], *2*
4	$\Gamma{+}x \Vdash \{x\#\Gamma\}\ \lambda y.x = y :_u \{\forall y \in (\Gamma{+}u).u \bullet y = \mathsf{false}\}$	[Conseq], *3*
5	$\Gamma \Vdash \{\mathsf{T}\}\ M :_u \{\forall y \in (\Gamma{+}u).u \bullet y = \mathsf{false}\}$	[LetFresh]
6	$\Gamma \Vdash \{\mathsf{T}\}\ M :_u \{\forall\delta.\forall y^{\mathsf{Nm}} \in (\delta).u \bullet y = \mathsf{false}\}$	*(utc2)*

To prove line 4 above we apply the axioms as follows:

7	$\Gamma{+}x{+}u \Vdash x\#\Gamma \wedge \forall\delta.\forall y \in (\delta).u \bullet y = (x = y)$	
8	$x\#\Gamma \wedge \forall y \in (\Gamma{+}x{+}u).u \bullet y = (x = y)$	*(utc1)*
9	$x\#\Gamma{+}u \wedge \forall y \in (\Gamma{+}x{+}u).\ u \bullet y = (x = y)$	*(f1)*
10	$x\#\Gamma{+}u \wedge \forall y \in (\Gamma{+}u).\ u \bullet y = (x = y)$	*(u2)*
11	$\forall y \in (\Gamma{+}u).\ x\#\Gamma{+}u{+}y \wedge u \bullet y = (x = y)$	*(f2)*
12	$\forall y \in (\Gamma{+}u).\ x \neq y \wedge u \bullet y = (x = y)$	*(f3)*
13	$\forall y \in (\Gamma{+}u).u \bullet y = \mathsf{false}$	*(e1)*

Example 6. To demonstrate the release of a hidden variable using Program (3), which was $M \stackrel{def}{=} \mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \langle x, \lambda y.x = y\rangle$, we reason as follows, with $A_6(p,q) \stackrel{def}{=} p\#\Gamma \wedge \forall\delta.\forall y \in (\delta).q \bullet y = (p = y)$:

1	$\{x\#\Gamma\}\ x :_b \{x = b \wedge x\#\Gamma\}$	[Var]
2	$\{\mathsf{T}\}\ \lambda y.x = y :_c \{\forall\delta.\forall y \in (\delta).c \bullet y = (x = y)\}$	*See Example 4, lines 1-2*
3	$\{x = b \wedge x\#\Gamma\}\ \lambda y.x = y :_c \{x = \pi_1(\langle b, c\rangle) \wedge C(x,c)\}$	[Conseq], [Invar], *2*
4	$\{x = b \wedge x\#\Gamma\}\ \lambda y.x = y :_c \{A_6(\pi_1(a), \pi_2(a))[\langle b, c\rangle/a]\}$	[Conseq], *(eq1)*
5	$\{x\#\Gamma\}\ \langle x, \lambda y.x = y\rangle :_a \{A_6(\pi_1(a), \pi_2(a))\}$	[Pair], *1, 4*
6	$\Gamma \Vdash \{\mathsf{T}\}\ M :_a \{A_6(\pi_1(a), \pi_2(a))\}$	[LetFresh], *5*

8 Conclusion

We have presented the first program logic for the ν-calculus, a variant of the STLC with names as first class values. Our logic is a conservative extension with two new universal quantifiers of the logic in [9] for the STLC. We provide axioms

and proof rules for the logic, prove their soundness, and show its expressive power by reasoning about well-known difficult examples from the literature.

We are currently unable to reason about this example from [16]:

$$\begin{array}{l}\mathsf{let}\ F = (\mathsf{let}\ x, y = \mathsf{gensym}()\ \mathsf{in}\ \lambda f^{\mathsf{Nm}\to\mathsf{Nm}}.fx = fy)\ \mathsf{in}\\ \mathsf{let}\ G = \lambda v^{\mathsf{Nm}}.F(\lambda u^{\mathsf{Nm}}.v = u)\ \mathsf{in}\ FG\end{array}$$

Is this because our logic is too inexpressive, or did we simply fail to find the right proof? Another open question is whether our logic's approach to freshness is independent of the ν-calculus's lack of integers and recursion, or not? For both questions we conjecture the former, and leave them as future work.

References

1. Benton, N., Koutavas, V.: A Mechanized Bisimulation for the Nu-Calculus. Tech. Rep. MSR-TR-2008-129, Microsoft (2008)
2. Berger, M., Tratt, L.: Program Logics for Homogeneous Generative Run-Time Meta-Programming. Logical Methods in Computer Science (LMCS) **11**(1:5) (2015)
3. Berger, M., Tratt, L., Urban, C.: Modelling Homogeneous Generative Meta-Programming. In: Proc. ECOOP. pp. 5:1–5:23 (2017)
4. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. In: Proc. ICFP. p. 143–156 (2010)
5. Eliott, H.P.: Program Logic for Fresh Name Generation. Ph.D. thesis, University of Sussex (expected 2021), draft.
6. Fernández, M., Gabbay, M.J., Mackie, I.: Nominal Rewriting Systems. In: Proc. PPDP. p. 108–119 (2004)
7. Gabbay, M.J., Pitts, A.M.: A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing **13**, 341–363 (2001)
8. Honda, K.: Elementary structures in process theory (1): Sets with renaming. MSCS **10**(5), 617–663 (2000)
9. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: Proc. PPDP'04. pp. 191–202. ACM Press (2004)
10. Mendelson, E.: Introduction to Mathematical Logic. Wadsworth Inc. (1987)
11. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Information and Computation **100**(1) (1992)
12. Odersky, M.: A Functional Theory of Local Names. In: Proc. POPL. pp. 48–59 (1994)
13. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
14. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation **186**, 165–193 (2003)
15. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. CUP (2013)
16. Pitts, A.M., Stark, I.D.B.: Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's new? In: MFCS (1993)
17. Stark, I.: Names and Higher-Order Functions. Ph.D. thesis, University of Cambridge (1994), technical report 363, Univ. of Cambridge Computer Laboratory
18. Urban, C., Tasson, C.: Nominal Techniques in Isabelle/HOL. In: Proc. CADE. p. 38–53 (2005)
19. Yoshida, N., Honda, K., Berger, M.: Logical Reasoning for Higher-Order Functions with Local State. Logical Methods in Computer Science **4**(2) (2008)