On a Monadic Semantics for Freshness

Mark R. Shinwell Andrew M. Pitts

University of Cambridge Computer Laboratory, Cambridge, CB3 0FD, UK

Abstract

A standard monad of continuations, when constructed with domains in the world of FM-sets [3], is shown to provide a model of dynamic allocation of fresh names that is both simple and useful. In particular, it is used to give the first correct proof of the fact that the powerful facilities for manipulating fresh names and binding operations provided by the 'Fresh' series of metalanguages [13–15] respect α-equivalence of object-level languages up to meta-level contextual equivalence.

1 Introduction

Moggi's use of category-theoretic monads to structure various notions of computational effect [6] is by now a standard technique in denotational semantics; and thanks to the work of Wadler [18] and others, monads are the accepted way of "tackling the awkward squad" [7] of side-effects within pure functional programming. Of Moggi's examples of monads, those for modelling *dynamic allocation of fresh resources* [1] are not so well-known. [2] So let us recall a simple example of such a monad, T. It is defined on the category of $\mathcal{S}et$-valued functors from the category $\mathbb{I}$ of finite cardinals and injective functions. Thus an object A of this functor-category gives us a family of sets $A(n)$ of "values in world n", where n is the number of names dynamically created so far; and each injection of n into a larger "world" n' gives rise to a coercion from $A(n)$ to $A(n')$. Then the monad T builds from A an object TA of "computations of A-values" whose value at each n is the dependent sum $TA(n) \stackrel{\text{def}}{=} \Sigma_{m\in\mathbb{I}} A(n+m)$. Such computations simply create some number m of fresh names and then return an A-value in the appropriate world, $n+m$. When A is the object of names itself, given by $A(n) = n$, there is a distinguished global element $\mathbf{new} : 1 \longrightarrow TA$ (given by the element $(1, 0)$ of the set $TA(0) = \Sigma_{m\in\mathbb{I}}\, m$) representing the computation whose evaluation creates a name that is fresh with respect to the current world.

[1] In this paper the only type of resource we consider are freshly generated *names*.

[2] Dynamic allocation monads are not mentioned in [6], but do appear in [5, Sect. 4.1.4].

Although an attractive notion that has had nice applications (see [16], for example), such dynamic allocation monads on functor-categories have proved at best difficult and at worst impossible to combine with some other important denotational techniques—for modelling higher-order functions, fixpoint definitions and algebraic identities. The difficulty with higher-order functions is that whilst functor-categories have exponentials, they are not so easy to work with in practice. The difficulty with fixpoints is finding a workable notion of "domain" in functor-categories (out of the several possibilities that present themselves). The difficulty with algebraic identities, such as

$$\begin{aligned}
(\mathbf{let}\ x \Leftarrow \mathbf{new}\ \mathbf{in}\ e) &= e, \quad \text{if } x \text{ not free in } e\\
(\mathbf{let}\ x \Leftarrow \mathbf{new};\ x' \Leftarrow \mathbf{new}\ \mathbf{in}\ e) &= (\mathbf{let}\ x' \Leftarrow \mathbf{new};\ x \Leftarrow \mathbf{new}\ \mathbf{in}\ e)
\end{aligned} \tag{1}$$

is that quotienting dynamic allocation monads in order to force such identities interacts badly with order-theoretic completeness properties needed for fixpoints. [3] We get past these problems with higher-order functions, fixpoints and algebraic identities in two steps, both of which turn out to hugely simplify matters.

First, we replace use of functor-categories with the category of *FM-sets* [3]. [4] Although this is equivalent to a category of functors, [5] working with it is almost entirely like working in the familiar category of sets: in particular exponentials are straightforward, as is the theory of domains in FM-sets [15,11]. The key property of FM-sets is that their elements have a notion of *finite support* that provides a syntax-free notion of "set of free names": it enables us to *make implicit all dependencies upon parameterising names*—a huge simplification compared with the explicit passing of parameterising name sets inherent in the functor-category approach.

Secondly, we feed back into denotational semantics the operational insight of [12] that in the presence of fixpoint recursion, it is easier to validate contextual equivalences like (1) (and many other more subtle ones) by forgetting about evaluation's properties of intermediate name-creation in favour of its simple termination properties. This leads to use of a Felleisen-style operational semantics with frame-stacks (evaluation contexts): see [9] for a survey. If D is the domain of denotations of values of some type, then frame-stacks can be modelled simply by elements of the strict function space $D \multimap 1_\perp$ where $1_\perp = \{\perp, \top\}$ (one element for non-termination, the other for termination); and since expressions are identified if they have the same termination behaviour with respect to all frame-stacks, we can take $(D \multimap 1_\perp) \multimap 1_\perp$ as the domain for interpreting expressions. Thus we are led to the use of the

---

3 Similar problems arise in connection with powerdomains.

4 Also known as *nominal sets* in [10].

5 The ones from $\mathbb{I}$ to $\mathcal{S}et$ that preserve pullbacks.

simple *continuation monad*

$$(-)^{\perp\perp} \stackrel{\text{def}}{=} (- \multimap 1_\perp) \multimap 1_\perp. \tag{2}$$

The notion of "finite support" now enters the picture: the domain A of names is simply a flat domain $\mathbb{A}_\perp$ on the FM-set $\mathbb{A}$ of atoms. We get an element $\mathbf{new} \in (\mathbb{A}_\perp \multimap 1_\perp) \multimap 1_\perp$ that models dynamic allocation by defining **new** to send any $\sigma \in \mathbb{A}_\perp \multimap 1_\perp$ to $\sigma(a) \in 1_\perp$, where $a \in \mathbb{A}$ is some (or indeed, any) atom *not in the support* of the function σ. Not only do standard properties of support make this recipe well defined,[6] but **new** turns out to have good properties, such as (1).

It might seem that the continuation monad $(- \multimap 1_\perp) \multimap 1_\perp$ on FM-domains is too simple to be useful. We show this is not so by using it, together with some standard methods based on logical relations for relating semantics to syntax [8], to prove some extensionality properties of contextual equivalence for the 'Fresh' series of metalanguages [13–15]. In particular we give the first correct proof of the main technical result of [15],[7] which shows that FreshML's powerful facilities for manipulating fresh names and binding operations do indeed respect α-equivalence of object-level languages up to meta-level contextual equivalence.

2 Mini-FreshML

We present a small language *Mini-FreshML* that encapsulates the core freshness features of FreshML [15] and Fresh O'Caml [13]; the reader is referred to those papers for motivation of the novel language features for manipulating bindable *names* (expressions of type `name`) and *name-abstractions* (expressions of type $\texttt{<<name>>}\tau$). Mini-FreshML types τ are given by the following grammar: $\tau \ ::= \texttt{unit} \mid \texttt{name} \mid \delta \mid \tau \times \tau \mid \texttt{<<name>>}\tau \mid \tau \to \tau$. Each data type δ has a top-level type declaration of the form $\delta\ \texttt{=}\ \texttt{C}_1\ \texttt{of}\ \sigma_1\ \texttt{|}\ \cdots\ \texttt{|}\ \texttt{C}_{C_{max}}\ \texttt{of}\ \sigma_{C_{max}}$, where the constructor types σ_k are generated from the same grammar as types τ and in particular may involve recursive occurrences of δ. Mini-FreshML expressions e are given by the following grammar, where x ranges over value identifiers, a over the denumerable set $\mathbb{A}$ of *atoms*[8] and $1 \leqslant k \leqslant C_{max}$.

$$
\begin{aligned}
e ::= {} & \\
& x \mid a \mid \texttt{C}_k(e) \mid (e\texttt{,}\ e) \mid \texttt{fresh} \mid \texttt{<<}e\texttt{>>}e \mid \texttt{swap}\ e\texttt{,}\ e\ \texttt{in}\ e \mid \texttt{fun}\ x(x)\ \texttt{=}\ e \mid \\
& e\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{let}\ (x\texttt{,}\ x)\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{let}\ \texttt{<<}x\texttt{>>}x\ \texttt{=}\ e\ \texttt{in}\ e \mid \\
& \texttt{match}\ e\ \texttt{with}\ \texttt{C}_1(x)\ \texttt{->}\ e \texttt{|}\ \cdots \texttt{|}\ \texttt{C}_{C_{max}}(x)\ \texttt{->}\ e.
\end{aligned}
$$

[6] **new** is closely related to the "freshness quantifier" И introduced in [3].

[7] In [15] the authors attempted to use a direct- rather than continuation-based monadic semantics that turns out to have problematic order-theoretic completeness properties.

[8] They are the closed values of type `name`.

The canonical forms ('*values*') of Mini-FreshML, v, form the subset of expressions generated by: $v ::= x \mid a \mid \mathtt{C}_k\mathtt{(}v\mathtt{)} \mid \mathtt{(}v\mathtt{,}\ v\mathtt{)} \mid \mathtt{<<}a\mathtt{>>}v \mid \mathtt{fun}\ x\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e$. We identify expressions up to α-conversion of bound value identifiers; the binding forms are as follows (with binding positions underlined): $\mathtt{fun}\ \underline{x}\mathtt{(}\underline{x'}\mathtt{)}\ \mathtt{=}\ [-]$, $\mathtt{let}\ \underline{x}\ \mathtt{=}\ e\ \mathtt{in}\ [-]$, $\mathtt{let}\ \mathtt{(}\underline{x}\mathtt{,}\ \underline{x'}\mathtt{)}\ \mathtt{=}\ e\ \mathtt{in}\ [-]$, $\mathtt{let}\ \mathtt{<<}\underline{x}\mathtt{>>}\underline{x'}\ \mathtt{=}\ e\ \mathtt{in}\ [-]$, $\mathtt{match}\ e\ \mathtt{with}\ \cdots \mathtt{|}\ \mathtt{C}_k\mathtt{(}\underline{x}\mathtt{)}\ \mathtt{->}\ [-]\ \mathtt{|}\ \cdots$. Note that an abstraction expression $\mathtt{<<}e\mathtt{>>}e'$ is not a binding form in this sense.[9] In typing contexts Γ (finite maps from value identifiers to types), expressions e are assigned types τ by a typing relation: we write $\Gamma \vdash e : \tau$ iff (Γ, e, τ) is in this relation (the Γ is omitted if it is empty). The details of the definition are all quite standard and we omit them in this extended abstract (see [15]).

Evaluation of Mini-FreshML expressions may be formalised operationally using a big-step relation $\Downarrow$ on 4-tuples $(\overline{a}, e, v, \overline{a}')$ where e is an expression, v is a canonical form and $\overline{a} \subseteq \overline{a}'$ are finite sets of atoms with the atoms of e contained in $\overline{a}$. We write $\overline{a}, e \Downarrow v, \overline{a}'$ to indicate that in the world with "allocated" atoms $\overline{a}$, the expression e evaluates to v and allocates the fresh atoms $\overline{a}' \setminus \overline{a}$ (evaluation of `fresh` and $\mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ e\ \mathtt{in}\ e'$ causes dynamic allocation of fresh atoms—see below). Further details of this relation are given elsewhere [15]. Instead, in this paper we use an equivalent operational semantics based on the notion of *frame stacks* (or "*evaluation contexts*")—see [9] for a survey. This abstracts away from the details of *which* particular atoms and values have been allocated and instead concentrates on the single notion of *termination*. In this formulation, as evaluation proceeds a stack of *evaluation frames* is built up. Each of these frames is a basic evaluation context: inside is a hole $[-]$ for which may be substituted another frame (as when *composing* frames to form a frame stack) or an expression, which may or may not be in canonical form. Formally then, a frame stack S consists of a possibly-empty list of evaluation frames, thus: $S ::= [] \mid S \circ \mathcal{F}$. Stacks are assigned types by a typing relation $\vdash_s$ whose details we omit. $\mathcal{F}$ ranges over frames as follows:

$$
\begin{array}{l}
\mathcal{F} ::= \\
\quad \mathtt{let}\ x\ \mathtt{=}\ [-]\ \mathtt{in}\ e \mid \mathtt{let}\ \mathtt{(}x\mathtt{,}\ x'\mathtt{)}\ \mathtt{=}\ [-]\ \mathtt{in}\ e \mid \mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ [-]\ \mathtt{in}\ e \mid \\
\quad \mathtt{(}[-]\mathtt{,}\ e\mathtt{)} \mid \mathtt{(}v\mathtt{,}\ [-]\mathtt{)} \mid \mathtt{<<}[-]\mathtt{>>}e \mid \mathtt{<<}v\mathtt{>>}[-] \mid \mathtt{swap}\ [-]\mathtt{,}\ e'\ \mathtt{in}\ e'' \mid \\
\quad \mathtt{swap}\ v\mathtt{,}\ [-]\ \mathtt{in}\ e'' \mid \mathtt{swap}\ v\mathtt{,}\ v'\ \mathtt{in}\ [-] \mid [-]\ e \mid v\ [-] \mid \\
\quad \mathtt{match}\ [-]\ \mathtt{with}\ \mathtt{C}_1\mathtt{(}x_1\mathtt{)}\ \mathtt{->}\ e_1\mathtt{|}\ \cdots \mathtt{|}\ \mathtt{C}_{C_{max}}\mathtt{(}x_{C_{max}}\mathtt{)}\ \mathtt{->}\ e_{C_{max}}.
\end{array}
$$

We define a *termination relation* $\langle S, e\rangle{\downarrow}$ (read "e terminates when evaluated in stack S") by induction on the structure of e and then on the structure of S. For example:

- $\langle S, \mathtt{fresh}\rangle{\downarrow}$ holds if $\langle S, a\rangle{\downarrow}$ does for some (or indeed as it turns out, for any) $a \in \mathbb{A}$ not occurring in S.

[9] Nevertheless the properties of Mini-FreshML contextual equivalence will be such that any *atoms* in e behave up to contextual equivalence as though they are bound in e'; for example $\mathtt{<<}a\mathtt{>>}a$ turns out to be contextually equivalent to $\mathtt{<<}b\mathtt{>>}b$.

- $\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = } [-] \texttt{ in } e, \texttt{<<}a\texttt{>>}v\rangle\!\downarrow$ holds if $\langle S, e[a'/x, ((a\ a')\cdot v)/x']\rangle\!\downarrow$ does for some (or indeed any) $a' \in \mathbb{A}$ not equal to a and not occurring in S or v. [10]

The complete definition of the termination relation is given in Appendix D. It is related to the big-step semantics as follows.

Theorem. *For any closed Mini-FreshML expression* e, $\langle [], e\rangle\!\downarrow$ *holds iff for any finite set* $\overline{a} \subseteq \mathbb{A}$ *containing the atoms of* e, *the relation* $\overline{a}, e \Downarrow v, \overline{a}'$ *holds for some value* v *and set of atoms* $\overline{a}' \supseteq \overline{a}$. $\square$

We wish to consider correctness properties of Mini-FreshML expressions of type δ, where δ is an algebraic data type [11] corresponding to the syntax of some object language. For simplicity, we use the untyped λ-calculus as an example object language; the results easily extend to any such language specified by a certain notion of *nominal signature* [17, Definition 2.1]. We thus use a top-level type declaration just containing:

$$\delta \texttt{ = Var of name | Lam of <<name>>}\delta \texttt{ | App of } \delta \times \delta.$$

For each λ-term t, define a Mini-FreshML expression $[t]_{\mathrm{e}}$ by induction on the structure of t as follows.

$$\begin{aligned}
[x]_{\mathrm{e}} &\stackrel{\text{def}}{=} \texttt{Var}(x) \\
[\lambda x\,.\,t]_{\mathrm{e}} &\stackrel{\text{def}}{=} \texttt{let } x \texttt{ = fresh in Lam(<<}x\texttt{>>}[t]_{\mathrm{e}}) \\
[t\ t']_{\mathrm{e}} &\stackrel{\text{def}}{=} \texttt{App}([t]_{\mathrm{e}}, [t']_{\mathrm{e}}).
\end{aligned}$$

Thus in a typing context Γ that assigns type `name` to each of the free variables of t, we have $\Gamma \vdash [t]_{\mathrm{e}} : \delta$.

We want to relate α-equivalence of λ-terms, $t \equiv_\alpha t$, to the operational equivalence of the Mini-FreshML expressions $[t]_{\mathrm{e}}$ and $[t']_{\mathrm{e}}$ of type δ. We shall use the traditional notion of *contextual equivalence* given in the following definition.

Definition (Contextual equivalence). The type-respecting relation of *contextual pre-order*, written $\Gamma \vdash e \leqslant_{\mathrm{ctx}} e' : \tau$, is defined to hold if $\Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, and for all closed, well-typed expressions $C[e]$ containing occurrences of e, if $\langle [], C[e]\rangle\!\downarrow$ holds, then so does $\langle [], C[e']\rangle\!\downarrow$ (where $C[e']$ is the expression obtained from $C[e]$ by replacing the occurrences of e with e'). The relation of *contextual equivalence*, $\approx_{\mathrm{ctx}}$ is the symmetrisation of $\leqslant_{\mathrm{ctx}}$.

[10] In general $(a\ a')\cdot v$ indicates the value obtained from v by interchanging all occurrences of a and a'; since here a' does not occur in v, this is the same as replacing occurrences of a by a' in this case.

[11] That is, one where the constructor types σ_k do not involve function types.

Theorem (Correctness of representation). *For any* λ*-terms* t *and* t'*, with free variables among* $\{x_0, \ldots, x_n\}$ *say, then*

$$t \equiv_\alpha t' \Leftrightarrow \{x_0 : \mathtt{name}, \cdots, x_n : \mathtt{name}\} \vdash [t]_{\mathrm{e}} \approx_{\mathrm{ctx}} [t']_{\mathrm{e}} : \delta. \qquad \square$$

We now show how to formulate a denotational semantics for Mini-FreshML which we can use to prove this theorem (and other properties of Mini-FreshML contextual equivalence).

3 Denotational semantics

To give a denotational semantics to Mini-FreshML we use *FM-cpos* [11,15]. An FM-cpo D is specified by an *FM-set* and an ordering, satisfying certain properties. Recall from [3,15] that an FM-set is a set D equipped with an *action* $\pi \in \mathit{perm}(\mathbb{A}), d \in D \mapsto \pi \cdot d \in D$ of the group $\mathit{perm}(\mathbb{A})$ of permutations of the set $\mathbb{A}$ of atoms, with the property that every $d \in D$ is *finitely supported*—meaning that $(a\ a') \cdot d = d$ holds for all but finitely many $a, a' \in \mathbb{A}$. (Here $(a\ a') \in \mathit{perm}(\mathbb{A})$ is the permutation interchanging a and a'.) To qualify as an FM-cpo, D must be equipped with a partial order $\sqsubseteq$ that is *equivariant*, (i.e. $d \sqsubseteq d'$ implies $\pi \cdot d \sqsubseteq \pi \cdot d'$); furthermore, D must possess least upper bounds (lubs) for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ that are finitely supported, in the sense that $\forall n.\ (a\ a') \cdot d_n = d_n$ holds for all but finitely many $a, a' \in \mathbb{A}$. An *FM-cppo* is an FM-cpo with a distinguished least element $\perp$. A morphism f of FM-cpos is a monotone function which preserves lubs of finitely-supported chains and is equivariant (i.e. $(a\ a') \cdot (f(d)) = f((a\ a') \cdot d)$). A morphism of FM-cppos has the same properties but is also strict ($f(\perp) = \perp$). FM-cpos (resp. FM-cppos) and their morphisms form a category **FM-Cpo** (resp. $\mathbf{FM\text{-}Cpo}_\perp$). Least fixed points may be constructed in FM-cppos via the familiar Tarski construction, since that construction only requires the use of finitely supported chains.

To each Mini-FreshML type τ we assign an FM-cppo $[\![\tau]\!]$. To do so we make use of the smash product $(- \otimes -)$, sum $(- \oplus -)$, lifting $(-_\perp)$, strict function space $(- \multimap -)$ and atom-abstraction $([\mathbb{A}]-)$ constructions. All but the last two are just as for classical domain theory [2]. The FM-cppo $D \multimap D'$ is given by the FM-set of finitely supported [12] functions from D to D' that preserve the partial order, lubs of finitely supported ω-chains and $\perp$; as usual, the partial order on $D \multimap D'$ is inherited from D' argument-wise. The FM-cppo $[\mathbb{A}]D$ generalises to domain theory the atom-abstraction construct of [3, Sect. 5]; its elements are equivalence classes $[a]d$ of pairs $(a, d) \in \mathbb{A} \times D$ for the equivalence relation induced by the pre-order: $(a, d) \sqsubseteq (a', d')$ iff $(a\ a'') \cdot d = (a'\ a'') \cdot d'$ for some/any atom a'' not in the support of d and d'; the permutation action is

[12] Finitely supported with respect to the usual permutation action for functions, given by $(\pi \cdot f)(d) = \pi \cdot (f(\pi^{-1} \cdot d))$.

$\pi \cdot [a]d = [\pi(a)](\pi \cdot d)$ and the partial order is induced by the above pre-order. Each FM-cppo $\llbracket \tau \rrbracket$ is defined by induction on the structure of τ as follows.

$$\begin{array}{lll}
\llbracket \texttt{unit} \rrbracket \stackrel{\text{def}}{=} 1_\bot & \llbracket \texttt{name} \rrbracket \stackrel{\text{def}}{=} \mathbb{A}_\bot & \llbracket \delta \rrbracket \stackrel{\text{def}}{=} D \\
\llbracket \tau \times \tau' \rrbracket \stackrel{\text{def}}{=} \llbracket \tau \rrbracket \otimes \llbracket \tau' \rrbracket & \llbracket \texttt{<<name>>}\tau \rrbracket \stackrel{\text{def}}{=} [\mathbb{A}]\llbracket \tau \rrbracket & \llbracket \tau \to \tau' \rrbracket \stackrel{\text{def}}{=} (\llbracket \tau \rrbracket \multimap \llbracket \tau' \rrbracket^{\bot\bot})_\bot.
\end{array}$$

Here $(-)^{\bot\bot}$ is the continuation monad (2) defined in the Introduction; $1_\bot$ and $\mathbb{A}_\bot$ are flat FM-cppos on the FM-sets $1 \stackrel{\text{def}}{=} \{\top\}$ (trivial action: $\pi \cdot \top = \top$) and $\mathbb{A}$ (canonical action: $\pi \cdot a = \pi(a)$); and D is the minimal invariant solution to a certain recursive domain equation on FM-cppos corresponding to the top-level data type declaration. Such solutions may be constructed in this setting using the normal technique of chains of embedding-projection pairs [8,2] and come equipped with an isomorphism $i \,:\, \llbracket \sigma_1 \rrbracket \oplus \cdots \oplus \llbracket \sigma_{C_{max}} \rrbracket \cong \llbracket \delta \rrbracket$.

Denotations of typing contexts are given using a finite smash product: $\llbracket \Gamma \rrbracket \stackrel{\text{def}}{=} \bigotimes_{x \in \mathrm{dom}(\Gamma)} \llbracket \Gamma(x) \rrbracket$. The denotation of values v (of type τ in context Γ), of frame stacks S (of argument type τ in context Γ) and expressions e (of type τ in context Γ) are given by morphisms in $\mathbf{FM\text{-}Cpo}_\bot$ of the following kinds

$$\begin{aligned}
\mathcal{V}\llbracket \Gamma \vdash v \,:\, \tau \rrbracket \;&: \llbracket \Gamma \rrbracket \multimap \llbracket \tau \rrbracket \\
\mathcal{S}\llbracket \Gamma \vdash_s S \,:\, \tau \multimap ? \rrbracket \;&: \llbracket \Gamma \rrbracket \multimap \llbracket \tau \rrbracket^\bot \\
\mathcal{E}\llbracket \Gamma \vdash e \,:\, \tau \rrbracket \;&: \llbracket \Gamma \rrbracket \multimap \llbracket \tau \rrbracket^{\bot\bot}
\end{aligned}$$

where for each FM-cppo D we define $D^\bot \stackrel{\text{def}}{=} D \multimap 1_\bot$. Intuitively, an element of $\llbracket \tau \rrbracket^\bot$ corresponds to a stack accepting a value of type τ and returning $\top$ for termination, or $\bot$ for divergence. Just as the behaviour of expressions is determined by any enclosing frame stack, the denotation $\epsilon \in \llbracket \tau \rrbracket^{\bot\bot}$ of some expression in context is then a function that accepts the denotation of a frame stack in context and returns either $\bot$ or $\top$. Thus, the denotations of expressions in context lie in the *underlying set of the continuation monad*, where the 'result' set is $1_\bot$. We have the usual two monad operations for $(-)^{\bot\bot}$, $\mathbf{return}(d)$ and $\mathbf{let}\ x \Leftarrow e\ \mathbf{in}\ e'$, whose standard definitions we omit in this abstract. We also have the element $\mathbf{new} \in (\mathbb{A}_\bot)^{\bot\bot}$ defined in the Introduction: this gives us the denotation of the `fresh` expression:

$$\mathcal{E}\llbracket \Gamma \vdash \texttt{fresh} \,:\, \texttt{name} \rrbracket(\rho) \stackrel{\text{def}}{=} \mathbf{new}.$$

The full definitions of $\mathcal{V}\llbracket - \rrbracket$, $\mathcal{S}\llbracket - \rrbracket$ and $\mathcal{E}\llbracket - \rrbracket$ are given in the Appendix: the "continuation-passing style" of them is self-evident. For closed values v of type τ, write $\mathcal{V}\llbracket v \rrbracket$ to stand for $\mathcal{V}\llbracket \vdash v \,:\, \tau \rrbracket(\emptyset)$. We use a similar convention for closed frame stacks and expressions.

An important stepping-stone from denotational semantics to prove properties of operational semantics is the construction of certain type-indexed *logical relations* which relate domain elements to values, frame stacks and expressions respectively:

$$\lhd^{\text{val}}_{\tau} \subset [\![\tau]\!]_{\downarrow} \times \text{Val}_{\tau} \qquad \lhd^{\text{stk}}_{\tau} \subset [\![\tau]\!]^{\perp} \times \text{Stack}_{\tau} \qquad \lhd^{\text{exp}}_{\tau} \subset [\![\tau]\!]^{\perp\perp} \times \text{Exp}_{\tau}.$$

(Here $D_{\downarrow}$ denotes the non-bottom elements of the FM-cppo D.) For values v of type τ, frame stacks S and expressions e we require that $\{d \mid d \lhd^{\text{val}}_{\tau} v\}$ is closed under lubs of finitely supported ω-chains, is equivariant ($d \lhd^{\text{val}}_{\tau} v$ implies $\pi \cdot d \lhd^{\text{val}}_{\tau} \pi \cdot v$) and satisfies:

$$\begin{aligned}
\top \lhd^{\text{val}}_{\texttt{unit}} ()& \\
a \lhd^{\text{val}}_{\texttt{name}} v &\Leftrightarrow a = v \\
d \lhd^{\text{val}}_{\delta} \texttt{C}_k(v) &\Leftrightarrow \exists d_k \in [\![\sigma_k]\!] \,.\, d = (i \circ \text{in}_k)(d_k) \ \wedge\ d_k \lhd^{\text{val}}_{\sigma_k} v \\
[a_1]\, d \lhd^{\text{val}}_{\texttt{<<name>>}\tau} \texttt{<<}a_2\texttt{>>}v &\Leftrightarrow (a_1\ a) \cdot d \lhd^{\text{val}}_{\tau} (a_2\ a) \cdot v \\
&\qquad \text{for some/any } a \notin \{a_1, a_2\} \cup \text{supp}(d) \cup \text{supp}(v) \\
(d_1, d_2) \lhd^{\text{val}}_{\tau \times \tau'} (v_1\texttt{,}\ v_2) &\Leftrightarrow d_1 \lhd^{\text{val}}_{\tau} v_1 \ \wedge\ d_2 \lhd^{\text{val}}_{\tau'} v_2 \\
d \lhd^{\text{val}}_{\tau \to \tau'} v &\Leftrightarrow \forall d' \lhd^{\text{val}}_{\tau} v' \,.\, d(d') \lhd^{\text{exp}}_{\tau'} v\ v' \\
\sigma \lhd^{\text{stk}}_{\tau} S &\Leftrightarrow \forall d \lhd^{\text{val}}_{\tau} v \,.\, \sigma(d) = \top \Rightarrow \langle S, v \rangle{\downarrow} \\
\epsilon \lhd^{\text{exp}}_{\tau} e &\Leftrightarrow \forall \sigma \lhd^{\text{stk}}_{\tau} S \,.\, \epsilon(\sigma) = \top \Rightarrow \langle S, e \rangle{\downarrow}
\end{aligned}$$

where the meta-notation $\forall d \lhd^{\text{val}}_{\tau} v$ stands for $\forall d \in [\![\tau]\!], v \in \text{Val}_{\tau} \,.\, d \lhd^{\text{val}}_{\tau} v$ (and similarly for $\lhd^{\text{stk}}_{\tau}$); and where $\text{supp}(d)$ is the support of d and $\text{supp}(v)$ the finite set of atoms occurring in v.[13] It is not straightforward to deduce that such relations even *exist*, but this can be proved by using the techniques of [8], which easily adapt to the world of FM-sets. The following key result is proved by induction on the derivation of typing judgements.

Theorem (Fundamental property of the logical relations). *Write ψ for finite maps from value identifiers to values, thought of as* substitutions *which may be applied in a capture-avoiding manner (written using square brackets) to values, frame stacks and expressions; let $Subst_{\Gamma}$ be the set of all ψ with domain $dom(\Gamma)$. Given $\psi \in Subst_{\Gamma}$ and $\rho \in [\![\Gamma]\!]$, write $\rho \lhd_{\Gamma} \psi$ to mean that the domains of ρ and ψ are equal and for each $x \in dom(\rho)$, $\rho(x) \lhd^{\text{val}}_{\Gamma(x)} \psi(x)$. Then for typing contexts Γ, values v, frame stacks S and expressions e, we have that*

$$\begin{array}{lll}
\Gamma \vdash v : \tau & \Rightarrow & \forall \rho \lhd_{\Gamma} \psi \,.\, \mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd^{\text{val}}_{\tau} v[\psi] \\
\Gamma \vdash_s S : \tau \multimap ? & \Rightarrow & \forall \rho \lhd_{\Gamma} \psi \,.\, \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap ?]\!](\rho) \lhd^{\text{stk}}_{\tau} S[\psi] \\
\Gamma \vdash e : \tau & \Rightarrow & \forall \rho \lhd_{\Gamma} \psi \,.\, \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) \lhd^{\text{exp}}_{\tau} e[\psi]. \qquad \square
\end{array}$$

As an immediate corollary we get

Theorem (Computational adequacy). *Suppose v, S and e are closed typeable values, frame stacks and expressions. Then $\langle S, e \rangle{\downarrow} \Leftrightarrow \mathcal{E}[\![e]\!](\mathcal{S}[\![S]\!]) = \top$ and $\langle S, v \rangle{\downarrow} \Leftrightarrow \mathcal{S}[\![S]\!](\mathcal{V}[\![v]\!]) = \top$.* $\square$

[13] Like other sets of syntax, values form an FM-set in which "support of" coincides with "set of atoms of".

4 Extensionality and correctness results

We now examine how the denotational semantics can be used to prove the correctness theorem given at the end of Sect. 2.

Definition (CIU-equivalence [4]). The *CIU-pre-order* relation $\Gamma \vdash e \leqslant_{\text{ciu}} e' : \tau$ holds iff $\Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, and for all closing substitutions $\psi \in \mathrm{Subst}_\Gamma$ and all closed frame stacks S, $\langle S, e[\psi]\rangle\!\downarrow$ implies $\langle S, e'[\psi]\rangle\!\downarrow$. The symmetrisation is called *CIU-equivalence* and written $\Gamma \vdash e \approx_{\text{ciu}} e' : \tau$.

Combining the compositionality properties of the denotational semantics with the computational adequacy theorem and the properties of the logical relation given above, we obtain:

Theorem (Coincidence). *For closed expressions:* $\vdash e \leqslant_{ctx} e' : \tau$ *iff* $\vdash e \leqslant_{ciu} e' : \tau$ *iff* $\mathcal{E}[\![e]\!] \lhd^{\text{exp}}_\tau e'$*; for closed values we also have* $\mathcal{E}[\![v]\!] \lhd^{\text{exp}}_\tau v'$ *iff* $\mathcal{V}[\![v]\!] \lhd^{\text{val}}_\tau v'$*. For open expressions:* $\Gamma \vdash e \leqslant_{ctx} e' : \tau$ *iff* $\Gamma \vdash e \leqslant_{ciu} e' : \tau$*.* $\square$

Then using properties of the relation $\lhd^{\text{val}}_\tau$ we get:

Corollary (Extensionality).
For unit values: $\vdash v \approx_{\text{ctx}} v' : \texttt{unit}$ *iff* $v = v' = ()$.
For name values: $\vdash a \approx_{\text{ctx}} a' : \texttt{name}$ *iff* $a = a' \in \mathbb{A}$.
For data values: $\vdash \mathtt{C}_k\texttt{(}v\texttt{)} \approx_{\text{ctx}} \mathtt{C}_k\texttt{(}v'\texttt{)} : \delta$ *iff* $\vdash v \approx_{\text{ctx}} v' : \sigma_k$.
For pair values: $\vdash \texttt{(}v_1\texttt{,}\ v_2\texttt{)} \approx_{\text{ctx}} \texttt{(}v'_1\texttt{,}\ v'_2\texttt{)} : \tau_1 \times \tau_2$ *iff* $\vdash v_1 \approx_{\text{ctx}} v'_1 : \tau_1$ *and* $\vdash v_2 \approx_{\text{ctx}} v'_2 : \tau_2$.
For name-abstraction values: $\vdash \texttt{<<}a\texttt{>>}v \approx_{\text{ctx}} \texttt{<<}a'\texttt{>>}v' : \texttt{<<name>>}\tau$ *iff* $\vdash (a\ a'')\cdot v \approx_{\text{ctx}} (a'\ a'')\cdot v' : \tau$ *for some/any* $a'' \in \mathbb{A}$ *not equal to* a *or* a' *and not occurring in* v *or* v'.
For function values: $\vdash f \approx_{\text{ctx}} f' : \tau \to \tau'$ *iff for all closed* v *of type* τ, $\vdash f\ v \approx_{\text{ctx}} f'\ v : \tau'$. $\square$

Finally, the correctness of representation theorem in Sect. 2 follows from this Extensionality property of names, data, pair and name-abstraction values, using the characterisation of α-equivalence for λ-terms from [3, Proposition 2.2].

5 Conclusion

Doing domain theory in the world of FM-sets seems to combine the familiarity and power of classical domain theory with a refined semantics of fresh names that simplifies and extends what has previously been achieved for freshness with functor category semantics. Here we applied this new approach using a continuation monad with a very simple domain of "results" ($1_\bot$) to prove properties of FreshML. Variations on this theme seem very promising; for example, replacing $1_\bot$ by $S \multimap 1_\bot$ for a suitable (recursively defined) FM-cppo of "states" should give a useful denotational semantics of ML-style references with no restriction on the type of value stored—we plan to explore this elsewhere. A somewhat different application of "FM domain theory" appears in [11]. Finally we should mention that game semantics can also make good use of FM-sets to achieve new full abstraction results: see [1].

References

[1] S. Abramsky, D. R. Ghica, A. S. Murowski, C.-H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. Submitted, 2004.

[2] S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

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

[4] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[5] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. Computer Science, Univ. Edinburgh, 1989.

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

[7] S. L. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In R. Steinbruggen C. A. R. Hoare, M. Broy, editor, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[8] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[9] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS Tutorial*, pages 378–412. Springer, 2002.

[10] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[11] A. M. Pitts and T. Sheard. On the denotational semantics of staged execution of open code. Submitted, 2004.

[12] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[13] M. R. Shinwell. Swapping the atom: Programming with binders in Fresh O'Caml. Proc. MERλIN 2003.

[14] M. R. Shinwell and A. M. Pitts. *Fresh O'Caml User Manual*. Cambridge University Computer Laboratory, September 2003. Available at ⟨`http://www.freshml.org/foc/`⟩.

[15] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP '03*, pages 263–274. ACM Press, 2003.

[16] I. D. B. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.

[17] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In *Proc. CSL'03 & KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.

[18] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

A Denotation of values (expressions in canonical form)

The function $\mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket : \llbracket \Gamma \rrbracket \mathrel{\circ\!\!\rightarrow} \llbracket \tau \rrbracket$ maps $\bot$ to itself and for non-bottom arguments is defined by induction on the structure of the canonical form v as given below.

Notation. In this and the following appendices, write $\underline{\lambda} x \,.\, t$ for the function acting as $\lambda x.t$ except that $(\underline{\lambda} x.t)(\bot) \stackrel{\text{def}}{=} \bot$. Extend this notation in the obvious way to write $\underline{\lambda}\langle d_1, d_2\rangle.t$ for strict functions $D_1 \otimes D_2 \mathrel{\circ\!\!\rightarrow} D$ and $\underline{\lambda}\,[a]\,d.t$ for strict functions $[\mathbb{A}]D \mathrel{\circ\!\!\rightarrow} D'$. (Note that this notation imposes no conditions as to which particular representative in $[\mathbb{A}]D$ is chosen: the semantics below makes this explicit when required.) We also write $\langle d_1, d_2\rangle$ to indicate the construction of a smash pair (such that $\langle d_1, d_2\rangle \stackrel{\text{def}}{=} \bot_{D_1 \otimes D_2}$ when either of $d_1 \in D_1$ and $d_2 \in D_2$ are bottom).

$$\mathcal{V}\llbracket \Gamma \vdash x : \tau \rrbracket(\rho) \stackrel{\text{def}}{=} \rho(x). \quad \mathcal{V}\llbracket \Gamma \vdash () : \texttt{unit} \rrbracket(\rho) \stackrel{\text{def}}{=} \top. \quad \mathcal{V}\llbracket \Gamma \vdash a : \texttt{name} \rrbracket(\rho) \stackrel{\text{def}}{=} a.$$

$$\mathcal{V}\llbracket \Gamma \vdash \texttt{C}_k\texttt{(}v\texttt{)} : \delta \rrbracket(\rho) \stackrel{\text{def}}{=} (i \circ \mathrm{in}_k)(\mathcal{V}\llbracket \Gamma \vdash v : \sigma_k \rrbracket(\rho)).$$

$$\mathcal{V}\llbracket \Gamma \vdash \texttt{(}v\texttt{,}\ v'\texttt{)} : \tau \times \tau' \rrbracket(\rho) \stackrel{\text{def}}{=} \langle \mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket(\rho), \mathcal{V}\llbracket \Gamma \vdash v' : \tau' \rrbracket(\rho)\rangle.$$

$$\mathcal{V}\llbracket \Gamma \vdash \texttt{<<}a\texttt{>>}v : \texttt{<<name>>}\tau \rrbracket(\rho) \stackrel{\text{def}}{=} [a]\,(\mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket(\rho)).$$

$$\mathcal{V}\llbracket \Gamma \vdash \mathbf{fun}\ f(x) = e : \tau \to \tau' \rrbracket(\rho) \stackrel{\text{def}}{=} \mathrm{fix}(\lambda f' \in \llbracket \tau \to \tau' \rrbracket.$$
$$\quad \underline{\lambda} x' \in \llbracket \tau \rrbracket \,.\, \mathcal{E}\llbracket \Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau' \rrbracket(\rho[f \mapsto f', x \mapsto x'])).$$

B Denotation of frame stacks

The function $\mathcal{S}\llbracket \Gamma \vdash_s S : \tau \multimap ? \rrbracket : \llbracket \Gamma \rrbracket \mathrel{\circ\!\!\rightarrow} \llbracket \tau \rrbracket^{\bot}$ maps $\bot$ to itself and for non-bottom arguments is defined by induction on the structure of S as follows.

$$\mathcal{S}\llbracket \Gamma \vdash_s [] : \tau \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \underline{\lambda} x \in \llbracket \tau \rrbracket \,.\, \top.$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{let}\ x\ \texttt{=}\ [-]\ \texttt{in}\ e : \tau \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=}$$
$$\quad \underline{\lambda} d \in \llbracket \tau \rrbracket \,.\, \mathcal{E}\llbracket \Gamma, x : \tau \vdash e : \tau' \rrbracket(\rho[x \mapsto d])(\mathcal{S}\llbracket \Gamma \vdash_s S : \tau' \multimap ? \rrbracket(\rho)).$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{let (}x\texttt{,}\ x'\texttt{)}\ \texttt{=}\ [-]\ \texttt{in}\ e : \tau \times \tau' \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \underline{\lambda}\langle d_1, d_2\rangle \in \llbracket \tau \times \tau' \rrbracket.$$
$$\quad \mathcal{E}\llbracket \Gamma, x : \tau, x' : \tau' \vdash e : \tau'' \rrbracket(\rho[x \mapsto d_1, x' \mapsto d_2])(\mathcal{S}\llbracket \Gamma \vdash_s S : \tau'' \multimap ? \rrbracket(\rho)).$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ [-]\ \texttt{in}\ e : \texttt{<<name>>}\tau \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=}$$
$$\quad \underline{\lambda}\,[a]\,d \in \llbracket \texttt{<<name>>}\tau \rrbracket \,.\, \mathcal{E}\llbracket \Gamma, x : \texttt{name}, x' : \tau \vdash e : \tau' \rrbracket$$
$$\qquad (\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d])(\mathcal{S}\llbracket \Gamma \vdash_s S : \tau' \multimap ? \rrbracket(\rho))$$
$$\quad \text{for some/any } a' \in \mathbb{A} \setminus (\{a\} \cup \mathrm{supp}(S, d, e)).$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{(}[-]\texttt{,}\ e\texttt{)} : \tau \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \lambda d \in \llbracket \tau \rrbracket.$$
$$\quad \mathcal{E}\llbracket \Gamma \vdash e : \tau' \rrbracket(\rho)(\lambda d' \in \llbracket \tau' \rrbracket \,.\, \mathcal{S}\llbracket \Gamma \vdash_s S : \tau \times \tau' \rrbracket(\rho)\langle d, d'\rangle).$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{(}v\texttt{,}\ [-]\texttt{)} : \tau' \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \lambda d \in \llbracket \tau' \rrbracket.$$
$$\quad \mathcal{S}\llbracket \Gamma \vdash_s S : \tau \times \tau' \rrbracket(\rho)\langle \mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket(\rho), d\rangle.$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{<<}[-]\texttt{>>}e : \texttt{name} \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \underline{\lambda} a \in \llbracket \texttt{name} \rrbracket.$$
$$\quad \mathcal{E}\llbracket \Gamma \vdash e : \tau \rrbracket(\rho)(\lambda d \in \llbracket \tau \rrbracket \,.\, \mathcal{S}\llbracket \Gamma \vdash_s S : \texttt{<<name>>}\tau \rrbracket(\rho)([a]\,d)).$$

$$\mathcal{S}\llbracket \Gamma \vdash_s S \circ \texttt{<<}v\texttt{>>}[-] : \tau \multimap ? \rrbracket(\rho) \stackrel{\text{def}}{=} \lambda d \in \llbracket \tau \rrbracket.$$
$$\quad \mathcal{S}\llbracket \Gamma \vdash_s S : \texttt{<<name>>}\tau \rrbracket(\rho)([\mathcal{V}\llbracket \Gamma \vdash v : \texttt{name} \rrbracket(\rho)]\,d).$$

$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{swap [-], } e' \texttt{ in } e'' : \texttt{name} \multimap ?]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a \in [\![\texttt{name}]\!] .$
$\quad \mathcal{E}[\![\Gamma \vdash e' : \texttt{name}]\!](\rho)(\underline{\lambda} a' \in [\![\texttt{name}]\!] . \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)$
$\qquad (\lambda d \in [\![\tau]\!] . \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap ?]\!](\rho)((a\ a') \cdot d)))$.
$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{swap } v\texttt{, [-] in } e'' : \texttt{name} \multimap ?]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} a' \in [\![\texttt{name}]\!] .$
$\quad \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!] . \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap ?]\!](\rho)(((\mathcal{V}[\![\Gamma \vdash v : \texttt{name}]\!](\rho))\ a') \cdot d))$.
$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{swap } v\texttt{, } v' \texttt{ in [-]} : \tau \multimap ?]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!] .$
$\quad \mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap ?]\!](\rho)(((\mathcal{V}[\![\Gamma \vdash v : \texttt{name}]\!](\rho))\ (\mathcal{V}[\![\Gamma \vdash v' : \texttt{name}]\!](\rho))) \cdot d)$.
$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{[-] } e : (\tau \to \tau') \multimap ?]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau \to \tau']\!] .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)(\lambda d' \in [\![\tau]\!] . (d\ d')(\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap ?]\!](\rho)))$.
$\mathcal{S}[\![\Gamma \vdash_s S \circ v \texttt{ [-]} : \tau \multimap ?]\!](\rho) \stackrel{\text{def}}{=} \lambda d \in [\![\tau]\!] .$
$\quad ((\mathcal{V}[\![\Gamma \vdash v : \tau \to \tau']\!](\rho))\ d)(\mathcal{S}[\![\Gamma \vdash_s S : \tau' \multimap ?]\!](\rho))$.
$\mathcal{S}[\![\Gamma \vdash_s S \circ \texttt{match [-] with } \cdots \texttt{ | C}_k\texttt{(}x_k\texttt{) -> } e_k \texttt{ | } \cdots : \delta \multimap ?]\!] \stackrel{\text{def}}{=}$
$\quad \underline{\lambda} d \in [\![\delta]\!] . \mathcal{E}[\![\Gamma, x_k : \sigma_k \vdash e_k : \tau]\!](\rho[x_k \mapsto d_k])(\mathcal{S}[\![\Gamma \vdash_s S : \tau \multimap ?]\!](\rho))$
$\quad$ for the unique k and d_k such that $d = (i \circ \mathrm{in}_k)(d_k)$.

C Denotation of expressions

The function $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp\perp}$ maps $\perp$ to itself and for non-bottom arguments is defined by induction on the structure of e as follows.

$\mathcal{E}[\![\Gamma \vdash v : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!] . \sigma(\mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho)) \quad$ (v a canonical form).
$\mathcal{E}[\![\Gamma \vdash \texttt{C}_k\texttt{(}e\texttt{)} : \delta]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\delta]\!]^{\perp} . \mathcal{E}[\![\Gamma \vdash e : \sigma_k]\!](\rho)(\lambda d \in [\![\sigma_k]\!] . \sigma((i \circ \mathrm{in}_k)(d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{(}e\texttt{, } e'\texttt{)} : \tau \times \tau']\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau \times \tau']\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)(\lambda d \in [\![\tau]\!] . \mathcal{E}[\![\Gamma \vdash e' : \tau']\!](\rho)(\lambda d' \in [\![\tau']\!] . \sigma\langle d, d'\rangle))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{fresh} : \texttt{name}]\!](\rho) \stackrel{\text{def}}{=} \mathbf{new} \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\texttt{name}]\!] . \sigma(a) \quad$ (any $a \in \mathbb{A} \setminus \mathrm{supp}(\sigma)$).
$\mathcal{E}[\![\Gamma \vdash \texttt{<<}e\texttt{>>}e' : \texttt{<<name>>}\tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\texttt{<<name>>}\tau]\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!] . \mathcal{E}[\![\Gamma \vdash e' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!] . \sigma([a]\,d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let } x \texttt{ = } e \texttt{ in } e' : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau']\!](\rho)(\underline{\lambda} d' \in [\![\tau']\!] . \mathcal{E}[\![\Gamma, x : \tau' \vdash e' : \tau]\!](\rho[x \mapsto d'])(\lambda d \in [\![\tau]\!] . \sigma(d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let (}x\texttt{, } x'\texttt{) = } e \texttt{ in } e' : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!](\rho)(\underline{\lambda} \langle d_1, d_2\rangle \in [\![\tau_1 \times \tau_2]\!] .$
$\qquad \mathcal{E}[\![\Gamma, x : \tau_1, x' : \tau_2 \vdash e' : \tau]\!](\rho[x \mapsto d_1, x' \mapsto d_2])(\lambda d \in [\![\tau]\!] . \sigma(d)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{let <<}x\texttt{>>}x' \texttt{ = } e \texttt{ in } e' : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \texttt{<<name>>}\tau']\!](\rho)(\underline{\lambda}\, [a]\, d' \in [\![\texttt{<<name>>}\tau']\!] .$
$\qquad \mathcal{E}[\![\Gamma, x : \texttt{name}, x' : \tau' \vdash e' : \tau]\!](\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d'])(\lambda d \in [\![\tau]\!] . \sigma(d))$
$\quad$ (any $a' \in \mathbb{A} \setminus \mathrm{supp}(\sigma, a, d, e, e', \rho)$).
$\mathcal{E}[\![\Gamma \vdash \texttt{swap } e\texttt{, } e' \texttt{ in } e'' : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} . \mathcal{E}[\![\Gamma \vdash e : \texttt{name}]\!](\rho)(\underline{\lambda} a \in [\![\texttt{name}]\!] .$
$\quad \mathcal{E}[\![\Gamma \vdash e' : \texttt{name}]\!](\rho)(\underline{\lambda} a' \in [\![\texttt{name}]\!] . \mathcal{E}[\![\Gamma \vdash e'' : \tau]\!](\rho)(\lambda d \in [\![\tau]\!] . \sigma((a\ a') \cdot d))))$.
$\mathcal{E}[\![\Gamma \vdash e\ e' : \tau]\!](\rho) \stackrel{\text{def}}{=} \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} .$
$\quad \mathcal{E}[\![\Gamma \vdash e : \tau \to \tau']\!](\rho)(\lambda d \in [\![\tau \to \tau']\!] . \mathcal{E}[\![\Gamma \vdash e' : \tau]\!](\rho)(\lambda d' \in [\![\tau]\!] . (d\ d')(\sigma)))$.
$\mathcal{E}[\![\Gamma \vdash \texttt{match } e \texttt{ with } \cdots \texttt{ | C}_k\texttt{(}x_k\texttt{) -> } e_k \texttt{ | } \cdots : \tau]\!] \stackrel{\text{def}}{=}$
$\quad \underline{\lambda} \sigma \in [\![\tau]\!]^{\perp} . \mathcal{E}[\![\Gamma \vdash e : \delta]\!](\rho)(\underline{\lambda} d' \in [\![\delta]\!] .$
$\qquad \mathcal{E}[\![\Gamma, x_k : \sigma_k \vdash e_k : \tau]\!](\rho[x_k \mapsto d_k])(\lambda d \in [\![\tau]\!] . \sigma(d)))$
$\quad$ for the unique k and d_k such that $d' = (i \circ \mathrm{in}_k)(d_k)$ when $d' \neq \perp$.

D Termination relation

$\langle S, e\rangle\downarrow$ is inductively defined by the following axiom and rules, where $e, e', \ldots$ range over expressions, $v, v', \ldots$ over expressions in canonical form, and $a, a', \ldots$ over atoms.

$$\frac{}{\langle [], v\rangle\downarrow}$$

$$\frac{\langle S, e[v/x]\rangle\downarrow}{\langle S \circ \texttt{let } x \texttt{ = } [-] \texttt{ in } e, v\rangle\downarrow} \qquad \frac{\langle S, e[v/x, v'/x']\rangle\downarrow}{\langle S\circ \texttt{let (}x\texttt{, }x'\texttt{) = }[-]\texttt{ in } e, \texttt{(}v\texttt{, }v'\texttt{)}\rangle\downarrow}$$

$$\frac{\begin{array}{c}\langle S, e[a'/x, ((a\ a')\cdot v)/x']\rangle\downarrow\\ \text{for some/any } a'\in\mathbb{A} \text{ such that } a'\notin \mathrm{supp}(S)\cup\{a\}\cup\mathrm{supp}(v)\end{array}}{\langle S\circ\texttt{let <<}x\texttt{>>}x'\texttt{ = }[-]\texttt{ in } e, \texttt{<<}a\texttt{>>}v\rangle\downarrow}$$

$$\frac{\langle S\circ \texttt{(}v\texttt{, }[-]\texttt{)}, e\rangle\downarrow}{\langle S\circ\texttt{(}[-]\texttt{, }e\texttt{)}, v\rangle\downarrow} \qquad \frac{\langle S, \texttt{(}v'\texttt{, }v\texttt{)}\rangle\downarrow}{\langle S\circ\texttt{(}v'\texttt{, }[-]\texttt{)}, v\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{<<}v\texttt{>>}[-], e\rangle\downarrow}{\langle S\circ\texttt{<<}[-]\texttt{>>}e, v\rangle\downarrow}$$

$$\frac{\langle S, \texttt{<<}v\texttt{>>}v'\rangle\downarrow}{\langle S\circ\texttt{<<}v\texttt{>>}[-], v'\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{swap } a\texttt{, } [-]\texttt{ in } e'', e'\rangle\downarrow}{\langle S\circ\texttt{swap } [-]\texttt{, } e'\texttt{ in } e'', a\rangle\downarrow}$$

$$\frac{\langle S\circ\texttt{swap } a\texttt{, } a'\texttt{ in } [-], e''\rangle\downarrow}{\langle S\circ\texttt{swap } a\texttt{, } [-]\texttt{ in } e'', a'\rangle\downarrow} \qquad \frac{\langle S, (a\ a')\cdot v\rangle\downarrow}{\langle S\circ\texttt{swap } a\texttt{, } a'\texttt{ in } [-], v\rangle\downarrow}$$

$$\frac{\langle S\circ v\ [-], e\rangle\downarrow}{\langle S\circ [-]\ e, v\rangle\downarrow} \qquad \frac{v = (\mathbf{fun}\ f(x) = e) \qquad \langle S, e[v/f, v'/x]\rangle\downarrow}{\langle S\circ v\ [-], v'\rangle\downarrow}$$

$$\frac{v = \texttt{C}_k\texttt{(}v_k\texttt{)}, \text{for some } 1\leqslant k\leqslant C_{max} \qquad \langle S, e_k[v_k/x_k]\rangle\downarrow}{\langle S\circ\texttt{match } [-] \texttt{ with C}_1\texttt{(}x_1\texttt{) -> } e_1 \texttt{|} \cdots \texttt{| C}_{C_{max}}\texttt{(}x_{C_{max}}\texttt{) -> } e_{C_{max}}, v\rangle\downarrow_v}$$

$$\frac{\langle S, e\rangle\downarrow}{\langle S, \texttt{C}_k\texttt{(}e\texttt{)}\rangle\downarrow} \qquad \frac{a\in\mathbb{A} \qquad a\notin\mathrm{supp}(S) \qquad \langle S, a\rangle\downarrow}{\langle S, \texttt{fresh}\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{(}[-]\texttt{, }e'\texttt{)}, e\rangle\downarrow}{\langle S, \texttt{(}e\texttt{, }e'\texttt{)}\rangle\downarrow}$$

$$\frac{\langle S\circ\texttt{<<}[-]\texttt{>>}e', e\rangle\downarrow}{\langle S, \texttt{<<}e\texttt{>>}e'\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{let } x\texttt{ = }[-]\texttt{ in } e', e\rangle\downarrow}{\langle S, \texttt{let } x \texttt{ = } e \texttt{ in } e'\rangle\downarrow}$$

$$\frac{\langle S\circ\texttt{let (}x\texttt{, }x'\texttt{) = }[-]\texttt{ in } e', e\rangle\downarrow}{\langle S, \texttt{let (}x\texttt{, }x'\texttt{) = } e\texttt{ in } e'\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{let <<}x\texttt{>>}x'\texttt{ = }[-]\texttt{ in } e', e\rangle\downarrow}{\langle S, \texttt{let <<}x\texttt{>>}x'\texttt{ = } e\texttt{ in } e'\rangle\downarrow}$$

$$\frac{\langle S\circ [-]\ e', e\rangle\downarrow}{\langle S, e\ e'\rangle\downarrow} \qquad \frac{\langle S\circ\texttt{swap } [-]\texttt{, } e'\texttt{ in } e'', e\rangle\downarrow}{\langle S, \texttt{swap } e\texttt{, } e'\texttt{ in } e''\rangle\downarrow}$$

$$\frac{\langle S\circ\texttt{match } [-] \texttt{ with C}_1\texttt{(}x_1\texttt{) -> } e_1 \texttt{|} \cdots \texttt{| C}_{C_{max}}\texttt{(}x_{C_{max}}\texttt{) -> } e_{C_{max}}, e\rangle\downarrow_e}{\langle S, \texttt{match } e \texttt{ with C}_1\texttt{(}x_1\texttt{) -> } e_1 \texttt{|} \cdots \texttt{| C}_{C_{max}}\texttt{(}x_{C_{max}}\texttt{) -> } e_{C_{max}}\rangle\downarrow_e}$$