Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 342 (2005) 28–55
Theoretical Computer Science
www.elsevier.com/locate/tcs

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 [M.J. Gabbay, A.M. Pitts, A new approach to abstract syntax with variable binding, Formal Aspects Comput. 13 (2002) 341–363], is shown to provide a model of dynamic allocation of fresh names that is both simple and useful. In particular, it is used to prove that the powerful facilities for manipulating fresh names and binding operations provided by the “Fresh” series of metalanguages [M.R. Shinwell, Swapping the atom: Programming with binders in Fresh O’Caml, Proc. MERλIN, 2003; M.R. Shinwell, A.M. Pitts, Fresh O’Caml User Manual, Cambridge University Computer Laboratory, September 2003, available at ⟨http://www.freshml.org/foc/⟩; M.R. Shinwell, A.M. Pitts, M.J. Gabbay, FreshML: Programming with binders made simple, in: Proc. ICFP ’03, ACM Press, 2003, pp. 263–274] respect α-equivalence of object-level languages up to meta-level contextual equivalence.
© 2005 Elsevier B.V. All rights reserved.

1. Introduction

Moggi’s use of category-theoretic monads to structure various notions of computational effect [7] is by now a standard technique in denotational semantics; and thanks to the work of Wadler [21] and others, monads are the accepted way of “tackling the awkward squad” [8] of side-effects within pure functional programming. Of Moggi’s examples of monads, we are here concerned with those for modelling *dynamic allocation of fresh resources.*[1]

* Corresponding author.
E-mail address: mrs30@cam.ac.uk (M.R. Shinwell).

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

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.06.003

Since these are not so well-known,[2] 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 (i.e. the finite sets $n = \{0, \ldots, n-1\}$ for $n = 0, 1, 2, \ldots$) and injective functions between them. Thus an object A of this functor category gives us a family of sets $A(n)$ of "A-values in world n", where n is the number of names created dynamically 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 $TA(n) \overset{\text{def}}{=} \sum_{m\in\mathbb{I}} A(n+m) = \{(m, x) \mid m \in \mathbb{I} \wedge x \in A(n+m)\}$; such "computations" simply create some number m of fresh names and then return an A-value in the appropriate world, $n+m$. The action of T on a natural transformation $\alpha : A \longrightarrow A'$ produces the natural transformation $T\alpha : TA \longrightarrow TA'$ whose component at $n \in \mathbb{I}$ is the function $(T\alpha)_n : TA(n) \longrightarrow TA'(n)$ mapping (m, x) to $(m, \alpha_{n+m}(x))$. When A is the object of names itself, given by $A(n) = n = \{0, \ldots, n-1\}$, there is a distinguished global element $\textbf{new} : 1 = \mathbb{I}(0, -) \longrightarrow TA$ corresponding under the Yoneda Lemma to the element $(1, 0) \in \sum_{m\in\mathbb{I}} m = TA(0)$; this represents the computation whose evaluation creates a name that is fresh with respect to the current world.

Although this is an attractive notion that has had nice applications (see [19], 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—those for modelling recursively defined higher-order functions and algebraic identities. The difficulty with higher-order functions is that while domains in functor categories do have exponentials, they are quite complicated things to work with in practice because of the indexing over "possible worlds". The difficulty with algebraic identities, such as

$$(\textbf{let } x \Leftarrow \textbf{new in } e) = e \quad \text{if } x \text{ not free in } e, \tag{1}$$

$$(\textbf{let } x \Leftarrow \textbf{new};\ x' \Leftarrow \textbf{new in } e) = (\textbf{let } x' \Leftarrow \textbf{new};\ x \Leftarrow \textbf{new in } e) \tag{2}$$

is that quotienting dynamic allocation monads in order to force such identities interacts badly with the order-theoretic completeness properties used to model recursive definitions. In this paper we get past these problems with recursively defined higher-order functions and algebraic identities in two steps, both of which turn out to greatly simplify matters.

First, we replace use of functor categories with the category of *FM-sets* [4].[3] Although this is equivalent to a category of functors,[4] working with it is almost entirely like working in the familiar category of sets: in particular exponentials are straightforward, as is the basic theory of domains in FM-sets [18,16]. FM-sets are certain sets equipped with an action of the group of permutations of a fixed, countably infinite set $\mathbb{A}$ of *atoms*; the key property of FM-sets is that their elements have *finite support*, a notion which provides a syntax-free notion of "set of free names". The existence of finite supports enables the dependence of semantic objects upon parameterising names to be left implicit—a convenient simplification compared with the explicit passing of parameterising name sets inherent in the "possible worlds"/functor category approach.

[2] Dynamic allocation monads are not mentioned in [7], but do appear in [6, Section 4.1.4].

[3] Also known as *nominal sets* in [11].

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

Secondly, we feed back into denotational semantics the operational insight of [13] that in the presence of fixpoint recursion, it is easier to validate contextual equivalences like (1) (and many other more subtle ones that do not concern us here) 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 [22], except that we formulate Felleisen's "evaluation contexts" as frame-stacks: see [10] for a recent 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 continuous 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 following *continuation monad*[5]

$$(-)^{\perp\perp} \overset{\text{def}}{=} (- \multimap 1_\perp) \multimap 1_\perp . \tag{3}$$

The notion of "finite support" now enters the picture: within the world of FM-sets, the domain of names is simply a flat domain $\mathbb{A}_\perp$ on the FM-set $\mathbb{A}$ of atoms. We get an element **new** $\in (\mathbb{A}_\perp \multimap 1_\perp) \multimap 1_\perp$ that models dynamic allocation by defining **new** to send any $\alpha \in \mathbb{A}_\perp \multimap 1_\perp$ to $\alpha(a) \in 1_\perp$, where $a \in \mathbb{A}$ is some atom *not in the support* of the function σ. Not only do standard properties of support make this recipe well defined (the value of $\alpha(a)$ is independent of which a we use), but **new** turns out to have good properties, such as (1) (see Remark 4.5).[6] We review those parts of "FM-domain theory" that we need in Section 3.

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 to prove some extensionality properties of contextual equivalence for the "Fresh" series of metalanguages [15,17,18]. In particular we give the first correct proof of the main technical result of [18],[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. Section 2 introduces a small version of FreshML, called Mini-FreshML, and states the properties of contextual equivalence we wish to prove. Section 3 gives a monadic denotational semantics for Mini-FreshML using the monad (3) on the category of FM-cppos. We prove the adequacy of this denotational semantics for Mini-FreshML's operational semantics by extending some standard methods based on logical relations for relating semantics to syntax [9]. Section 4 uses the logical relation from the previous section to prove the desired extensionality and correctness properties for Mini-FreshML's representation of object-level syntax involving binders. Finally in Section 5 we draw some conclusions.

[5] It is possible to use other continuation monads, by replacing one or other uses of $\multimap$ in (3) by other kinds of function space, but this simple version is enough for our purposes here.

[6] **new** is closely related to the "freshness quantifier" $\reflectbox{N}$ introduced in [4].

[7] In [18] the authors attempted to use a direct- rather than continuation-based monadic semantics that turns out to have problematic order-theoretic completeness properties.

2. Mini-FreshML

We present a small, monomorphic language *Mini-FreshML* that encapsulates the core freshness features of FreshML [18] and Fresh O'Caml [15]; 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 `<<name>>`τ). Mini-FreshML types τ are given by the following grammar:

$$\tau ::= \mathtt{unit} \mid \mathtt{name} \mid \delta \mid \tau \times \tau \mid \mathtt{<<name>>}\tau \mid \tau \to \tau.$$

Here δ ranges over a finite set of datatype names and we assume each δ comes with a top-level, ML-style type declaration of the form

$$\delta \;\mathtt{=}\; \mathtt{C}_1 \;\mathtt{of}\; \sigma_1 \mathtt{|} \cdots \mathtt{|} \mathtt{C}_n \;\mathtt{of}\; \sigma_n, \tag{4}$$

where the $\mathtt{C}_k$ are *constructors* and the corresponding constructor types σ_k are generated from the same grammar as types τ and in particular may involve (simultaneous) recursive occurrences of the datatype names δ. Mini-FreshML expressions *e* are given by the following grammar, where *x* ranges over a denumerable set VId of value identifiers and *a* ranges over another denumerable set $\mathbb{A}$, disjoint from VId, whose elements we call *atoms* (these are the closed values of type `name`):

$$\begin{aligned} e ::= {} & x \mid \mathtt{()} \mid a \mid \mathtt{C}_k\mathtt{(}e\mathtt{)} \mid \mathtt{(}e, e\mathtt{)} \mid \mathtt{fresh} \mid \mathtt{<<}e\mathtt{>>}e \mid \mathtt{swap}\; e, e \;\mathtt{in}\; e \\ & \mid \mathtt{if}\; e \;\mathtt{=}\; e \;\mathtt{then}\; e \;\mathtt{else}\; e \mid \mathtt{fun}\, x\mathtt{(}x\mathtt{)} \;\mathtt{=}\; e \mid e\, e \mid \mathtt{let}\; x \;\mathtt{=}\; e \;\mathtt{in}\; e \\ & \mid \mathtt{let}\;\mathtt{(}x, x\mathtt{)} \;\mathtt{=}\; e \;\mathtt{in}\; e \mid \mathtt{let}\;\mathtt{<<}x\mathtt{>>}x \;\mathtt{=}\; e \;\mathtt{in}\; e \\ & \mid \mathtt{match}\; e \;\mathtt{with}\; \mathtt{(}\cdots \mathtt{|} \mathtt{C}_k\mathtt{(}x\mathtt{)} \;\mathtt{->}\; e \mathtt{|} \cdots\mathtt{)}. \end{aligned}$$

Note that local declarations of the form $\mathtt{let}\; x \;\mathtt{=}\; e \;\mathtt{in}\; e'$ are included more for convenience than necessity; since we have excluded ML-style polymorphism from Mini-FreshML (in order to keep things simple), this expression has the same typing and evaluation behaviour as the function application $\mathtt{(fun}\; f\mathtt{(}x\mathtt{)} \;\mathtt{=}\; e'\mathtt{)}e$ (where *f* is a value identifier that does not occur in e').

The *values* (i.e. expressions in canonical form) of Mini-FreshML, v, form the subset of expressions generated by

$$v ::= x \mid \mathtt{()} \mid a \mid \mathtt{C}_k\mathtt{(}v\mathtt{)} \mid \mathtt{(}v, v\mathtt{)} \mid \mathtt{<<}a\mathtt{>>}v \mid \mathtt{fun}\, x\mathtt{(}x\mathtt{)} \;\mathtt{=}\; e.$$

We identify expressions up to α-conversion of bound value identifiers; the binding forms are as follows (with binding positions underlined):

$$\begin{gathered} \mathtt{fun}\, \underline{x}\mathtt{(}\underline{x'}\mathtt{)} \;\mathtt{=}\; [-], \quad \mathtt{let}\; \underline{x} \;\mathtt{=}\; e \;\mathtt{in}\; [-], \quad \mathtt{let}\;\mathtt{(}\underline{x}, \underline{x'}\mathtt{)} \;\mathtt{=}\; e \;\mathtt{in}\; [-], \\ \mathtt{let}\;\mathtt{<<}\underline{x}\mathtt{>>}\underline{x'} \;\mathtt{=}\; e \;\mathtt{in}\; [-], \quad \mathtt{match}\; e \;\mathtt{with}\; \mathtt{(}\cdots \mathtt{|} \mathtt{C}_k\mathtt{(}\underline{x}\mathtt{)} \;\mathtt{->}\; [-] \mathtt{|} \cdots\mathtt{)}. \end{gathered}$$

We write $e[v/x]$ for the capture-avoiding substitution of a value v for all free occurrences of the value identifier *x* in the expression *e*. We say that *e* is *closed* if it has no free value identifiers. Even if *e* is closed, it may well have occurrences of atoms *a* in it; we write supp(e)

for the finite set of atoms occurring in e. [8] Note that there are no expression constructions that bind atoms; in particular, although abstraction expressions `<<`e`>>`e' are used to represent binders in object-level syntax, they are not binding forms in Mini-FreshML itself. [9] In what follows we make heavy use of the operation on expressions of *swapping atoms*: $(a\ a') \cdot e$ indicates the result of interchanging all occurrences of the atoms a and a' in the expression e.

We only consider expressions that are well-typed, given a typing context Γ consisting of a finite map from value identifiers to types. We write $\Gamma \vdash e : \tau$ to indicate that e is assigned type τ in such a typing context Γ (and omit mention of Γ when it is empty). This relation is inductively generated by rules that are mostly standard and which are given in Appendix A. Let us just mention here that atoms a are assigned type `name`; and that if e is an expression of type `name` and e' one of type τ, then the abstraction expression `<<`e`>>`e' has type `<<name>>`τ.

Evaluation of Mini-FreshML expressions can be formalised operationally using a "big-step" relation $\Downarrow$ on 4-tuples $(\overline{a}, e, v, \overline{a}')$, written $\overline{a}, e \Downarrow v, \overline{a}'$. Here e is a closed expression, v is a closed value, and $\overline{a} \subseteq \overline{a}'$ are finite sets of atoms with the atoms of e contained in $\overline{a}$. The intended meaning of this relation is that in the world with "allocated" atoms $\overline{a}$, the expression e evaluates to v and allocates the fresh atoms $\overline{a}' - \overline{a}$ (evaluation of `fresh` and `let <<`x`>>`x' `=` e `in` e' causes dynamic allocation of fresh atoms—see below). Further details of the relation are given elsewhere [18]. Instead, in this paper we use an equivalent operational semantics based on the notion of *frame stacks*, or "evaluation contexts" [22]; see [10] for a recent survey of this technique. 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

$$S ::= [\,] \mid S \circ \mathcal{F},$$

where $\mathcal{F}$ ranges over frames as follows:

$$\begin{aligned}\mathcal{F} ::= {} & \mathtt{C}_k\mathtt{(}[-]\mathtt{)} \mid \mathtt{(}[-], e\mathtt{)} \mid \mathtt{(}v, [-]\mathtt{)} \mid \mathtt{<<}[-]\mathtt{>>}e \mid \mathtt{<<}v\mathtt{>>}[-] \\ & \mid \mathtt{swap}\ [-], e\ \mathtt{in}\ e \mid \mathtt{swap}\ v, [-]\ \mathtt{in}\ e \mid \mathtt{swap}\ v, v\ \mathtt{in}\ [-] \\ & \mid \mathtt{if}\ [-]\ \mathtt{=}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{if}\ v\ \mathtt{=}\ [-]\ \mathtt{then}\ e\ \mathtt{else}\ e \\ & \mid [-]\, e \mid v\, [-] \mid \mathtt{let}\ x\ \mathtt{=}\ [-]\ \mathtt{in}\ e \\ & \mid \mathtt{let}\ \mathtt{(}x, x'\mathtt{)}\ \mathtt{=}\ [-]\ \mathtt{in}\ e \mid \mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ [-]\ \mathtt{in}\ e \\ & \mid \mathtt{match}\ [-]\ \mathtt{with}\ \mathtt{(}\cdots \mathtt{|} \mathtt{C}_k\mathtt{(}x\mathtt{)}\ \mathtt{->}\ e \mathtt{|} \cdots\mathtt{)}.\end{aligned}$$

[8] The reason for this notation is the fact that this set of atoms is the *support* of e in the technical sense introduced in Section 3.

[9] It is one of the main results of this paper (Theorem 2.3) that the properties of Mini-FreshML contextual equivalence are such that atoms in e occurring in e' behave up to contextual equivalence as though they are bound in `<<`e`>>`e'; for example for atoms a, b then `<<`a`>>`a turns out to be contextually equivalent to `<<`b`>>`b.

Then the *termination relation* $\langle S, e\rangle\downarrow$ (read "*e* terminates when evaluated with stack *S*") can be inductively defined by rules that follow the structure of *e* and then the structure of *S*. For example:

- $\langle S, \texttt{fresh}\rangle\downarrow$ holds if $\langle S, a\rangle\downarrow$ does for some (or indeed as it turns out, for every) $a \in \mathbb{A} - \mathrm{supp}(S)$, i.e. for some atom *a* not occurring in the frame stack *S*.
- $\langle S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = [-] in } e, \texttt{<<}a\texttt{>>}v\rangle\downarrow$ holds if $\langle S, e[a'/x, ((a\ a')\cdot v)/x']\rangle\downarrow$ does for some (or indeed every) $a' \in \mathbb{A} - \mathrm{supp}(S, v, e)$.

The complete definition of the termination relation is given in Appendix B. Since we have not defined the "big-step" relation $\Downarrow$ here, we state the following relationship between it and the termination relation without proof; the details can be found in [16].

Fact 2.1. *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 $\overline{a}' \supseteq \overline{a}$.*

Just as we only use well-typed expressions, we only consider well-typed frame stacks: we write $\Gamma \vdash S : \tau \multimap \_\_$ to mean that in typing context Γ, the frame stack *S* takes expressions *e* of type τ (in context Γ) and produces a well-typed result (of some type that we do not need to name, since we only care about the termination of *e* when evaluated with stack *S*). This judgement is defined by induction on the length of the stack *S* by

$$\frac{}{\Gamma \vdash [] : \tau \multimap \_\_} \qquad \frac{\Gamma, [-] : \tau \vdash \mathcal{F} : \tau' \qquad \Gamma \vdash S : \tau' \multimap \_\_}{\Gamma \vdash S \circ \mathcal{F} : \tau \multimap \_\_},$$

where in the hypothesis $\Gamma, [-] : \tau \vdash \mathcal{F} : \tau'$ of the second rule, we regard $[-]$ as a special value identifier and type $\mathcal{F}$ using the typing rules for expressions given in Appendix A.

In [18], it is claimed that the features of Mini-FreshML that are novel compared with ML can be used to represent and to manipulate the terms of languages involving binding operators in ways that are guaranteed to respect α-equivalence between those terms. That paper shows that a wide range of syntax-manipulating functions can be very conveniently expressed using the new features. Here we wish to give a formal proof of the fact that α-equivalence between the terms of an "object language" is respected by Mini-FreshML when we represent those terms as expressions of a suitable Mini-FreshML datatype. For simplicity we use the untyped λ-calculus as a running example of an object language involving binding operators. [10] Write Λ for the set of *λ-terms t*, by which we mean abstract syntax trees (not identified up to α-equivalence) given by

$$t ::= x \mid \lambda x.t \mid t\ t,$$

where for variables *x* we are using elements of the set VId of Mini-FreshML value identifiers. To represent such terms in Mini-FreshML we use a top-level type declaration containing:

$$\delta \texttt{ = Var of name|Lam of <<name>>}\delta\texttt{|App of } \delta \times \delta. \tag{5}$$

[10] However, our results easily extend to any language with binders specified by a *nominal signature* [20, Definition 2.1].

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

$$\left.\begin{aligned}[x]_\mathrm{e} &\overset{\mathrm{def}}{=} \mathtt{Var}(x)\\ [\lambda x.t]_\mathrm{e} &\overset{\mathrm{def}}{=} \mathtt{let}\ x\ \mathtt{=}\ \mathtt{fresh}\ \mathtt{in}\ \mathtt{Lam}(\mathtt{<<}x\mathtt{>>}[t]_\mathrm{e})\\ [t\ t']_\mathrm{e} &\overset{\mathrm{def}}{=} \mathtt{App}([t]_\mathrm{e}, [t']_\mathrm{e}).\end{aligned}\right\} . \tag{6}$$

Note that under this translation, free variables in λ-terms are represented by free value identifiers in Mini-FreshML: the set of free variables of t is the same as the set of free value identifiers of $[t]_\mathrm{e}$. Note also that in a typing context Γ that assigns type `name` to each of those free variables, we have $\Gamma \vdash [t]_\mathrm{e} : \delta$. We want to relate α-equivalence of λ-terms, $t \equiv_\alpha t'$, to the operational behaviour of the Mini-FreshML expressions $[t]_\mathrm{e}$ and $[t']_\mathrm{e}$ of type δ. To do so, we shall use the traditional notion of *contextual equivalence* given by the following definition. [11]

Definition 2.2 (*Contextual equivalence*). The type-respecting relation of *contextual preorder*, 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 $\leqslant_{\mathrm{ctx}}$. For closed typeable expressions e and e' we just write $e \approx_{\mathrm{ctx}} e'$ when $\emptyset \vdash e \approx_{\mathrm{ctx}} e' : \tau$ holds for some type τ (and similarly for $\leqslant_{\mathrm{ctx}}$).

In the next section we show how to formulate a denotational semantics for Mini-FreshML which we use in Section 4 to prove the following theorem (and other properties of Mini-FreshML contextual equivalence).

Theorem 2.3 (*Correctness for expressions*). *For any λ-terms t and t', with free variables contained in the set $\{x_0, \ldots, x_n\}$ say,*

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

If t and t' are α-equivalent, then their translations into Mini-FreshML only differ up to renaming bound value identifiers; so since we identify Mini-FreshML expressions up to α-equivalence, in this case $[t]_\mathrm{e}$ and $[t']_\mathrm{e}$ are equal Mini-FreshML expressions and in particular are contextually equivalent. Thus the left-to-right direction of the above theorem is straightforward and the force of the theorem lies in the right-to-left direction: if the termination behaviour of $[t]_\mathrm{e}$ and $[t']_\mathrm{e}$ in any context is the same, then t and t' must be α-equivalent.

Remark 2.4 (*Representing* $\equiv_\alpha$). Since α-equivalence is a decidable relation between λ-terms, it makes sense to ask whether, given a type declaration for booleans

```
    bool = True of unit|False of unit
```

[11] We have formulated the definition using the termination relation $\downarrow$; but note that in view of Fact 2.1, we could have used the big-step evaluation relation $\Downarrow$.

we can strengthen the above theorem and represent $\equiv_\alpha$ by a function expression aeq : $(\delta \times \delta) \to$ `bool` in Mini-FreshML. Such an expression aeq does indeed exist in Mini-FreshML. Rather than give it explicitly, it is clearer to give the Fresh O'Caml version of it, since Fresh O'Caml's richer syntax (in particular it's richer language of patterns and built-in boolean operations) enables one to express aeq more clearly: [12]

```
let rec aeq(t,t') = match t,t' with
    Var x, Var x' -> if x=x' then true else false
  | Lam(<<x>>y),Lam(<<x'>>y') -> aeq(y, swap x and x' in y')
  | App(x,y), App(x',y') -> aeq(x,x') && aeq(y,y').
```

The Mini-FreshML version of `aeq` has to use nested `match`-expressions and simple patterns to express the above more complicated patterns and also to express the boolean conjunction &&. The precise sense in which `aeq` represents $\equiv_\alpha$ is described in Section 4 (see Remark 4.11).

3. Denotational semantics with FM-cppos

The FreshML language design was driven by the ability of the Fraenkel-Mostowski permutation model of set theory with atoms to model binding, α-equivalence and freshness of names [4]. So to give a denotational semantics to Mini-FreshML we could develop the usual notion of pointed, chain-complete poset in the axiomatic FM-set theory of [4]. This FM-set theory is just classical ZF set theory with urelements and an axiom asserting a "finite support property" (that is incompatible with the axiom of choice, it should be noted). So the fundamental constructions of domain theory, such as limit-colimit solutions of recursive domain equations, can be carried out in that axiomatic theory. Such a change of mathematical foundations demands a certain meta-logical sophistication from the reader which can render the results somewhat inaccessible. So instead here we take a less sophisticated, but equivalent approach and work with domains in FM-set theory as ordinary (partially ordered) sets with extra structure giving the effect on their elements of permuting atoms. [13] Whichever approach one takes, the main point is that domains in this new setting admit some relatively simple, but novel constructions for names and name-binding with which we can give a meaning to the novel features of Mini-FreshML. We concentrate on describing those new constructs; a fuller development of FM-cppos is given in [16].

Recall from [11,18] that an *FM-set* is a set X equipped with an *action*

$$perm(\mathbb{A}) \times X \longrightarrow X, \quad \text{written as } (\pi, x) \mapsto \pi \cdot x,$$

of the group $perm(\mathbb{A})$ of permutations of the set $\mathbb{A}$ of atoms (thus $\iota \cdot x = x$, where ι is the identity permutation; and $(\pi \circ \pi') \cdot x = \pi \cdot (\pi' \cdot x)$, where $\circ$ is composition of permutations).

[12] Indeed, the user has no need to make this declaration of `aeq` in Fresh O'Caml, because the language has a built-in structural equality function =, which at the type δ declared in (5) already implements `aeq`; so one can just use `t = t'` instead of `aeq(t, t')`.

[13] Strictly speaking, what we call an FM-cppo below corresponds to an object in the universe of FM-sets *which has empty support* and is a cppo in the axiomatic FM-set theory.

Furthermore, it is required that every $x \in X$ is *finitely supported*—meaning that there exists a finite subset $\overline{a} \subseteq \mathbb{A}$ (called a finite *support* for x) such that $(a\ a') \cdot x = x$ holds for all $a, a' \in \mathbb{A} - \overline{a}$. (Here $(a\ a') \in \mathit{perm}(\mathbb{A})$ is the permutation just interchanging a and a'.) Each $x \in X$ in fact possesses a *least* finite support which we write as $\mathrm{supp}(x)$; thus if $a, a' \in \mathbb{A} - \mathrm{supp}(x)$, then $(a\ a') \cdot x = x$. A function f between FM-sets X and Y is called *equivariant* if $\pi \cdot (f(x)) = f(\pi \cdot x)$ holds for all $\pi \in \mathit{perm}(\mathbb{A})$ and $x \in X$. The category of FM-sets and equivariant functions is rich in properties, being in fact equivalent to a well-known Grothendieck topos (of continuous G-sets, when G is the topological group given by $\mathit{perm}(\mathbb{A})$ endowed with the finite information topology). Here we will just describe finite products, power-objects and exponentials in this topos, since the associated notions of finitely supported subset and function will be important in what follows.

Definition 3.1 (*Finite products*). The product of X and Y in the category of FM-sets and equivariant functions is given by the usual cartesian product of sets $X \times Y \overset{\text{def}}{=} \{(x, y) \mid x \in X \ \wedge\ y \in Y\}$ with permutation action given by $\pi \cdot (x, y) \overset{\text{def}}{=} (\pi \cdot x, \pi \cdot y)$. It is not hard to see that with this action (x, y) is finitely supported because x and y are, and that $\mathrm{supp}(x, y) = \mathrm{supp}(x) \cup \mathrm{supp}(y)$. The projection functions $X \longleftarrow X \times Y \longrightarrow Y$ are equivariant and make $X \times Y$ into the categorical product of X and Y. The terminal object in this category is just a one-element set $1 = \{0\}$ endowed with the unique permutation action $\pi \cdot 0 \overset{\text{def}}{=} 0$.

Definition 3.2 (*Finitely supported subsets and functions*). A subset $S \subseteq X$ of an FM-set X is defined to be finitely supported if there is a finite set of atoms $\overline{a} \subseteq \mathbb{A}$ such that for all $a, a' \in \mathbb{A} - \overline{a}$ and all $x \in S$, $(a\ a') \cdot x \in S$. The set of all finitely supported subsets of X becomes an FM-set, denoted $\mathcal{P}X$, once we endow it with the permutation action given by $\pi \cdot S = \{\pi \cdot x \mid x \in S\}$. The *equivariant* subsets $S \subseteq X$ are by definition those finitely supported subsets for which we can take $\overline{a}$ to be empty (so that $x \in S$ implies $(a\ a') \cdot x \in S$ for all $a, a' \in \mathbb{A}$). (It is not hard to see that the subobjects of X in the topos of FM-sets and equivariant functions are naturally in bijection with the equivariant subsets of X, with inclusion of subobjects corresponding to inclusion of subsets; and $\mathcal{P}X$ is indeed the powerobject of X in this topos.) A function f between two FM-sets X and Y is defined to be finitely supported if its graph is a finitely supported subset of $X \times Y$; it is not hard to see that this is equivalent to requiring that there be a finite subset $\overline{a} \subseteq \mathbb{A}$ such that for all $a, a' \in \mathbb{A} - \overline{a}$ and all $x \in X$, $(a\ a') \cdot (f(x)) = f((a\ a') \cdot x)$ (i.e. f is "equivariant away from $\overline{a}$"). The set of all such functions becomes an FM-set, denoted Y^X, once we endow it with the permutation action given by $\pi \cdot f \overset{\text{def}}{=} \lambda x \in X. \pi \cdot (f(\pi^{-1} \cdot x))$, where π^{-1} is the inverse of the permutation π. (This is indeed the exponential of X and Y in the topos of FM-sets.) Note that the morphisms from X to Y in the category of FM-sets, i.e. the equivariant functions from X to Y, are precisely the elements of Y^X that have empty support.

Remark 3.3. The finitely supported subsets of an FM-set are closed under the usual boolean operations. In particular, if a finite set of atoms $\overline{a} \subseteq \mathbb{A}$ witnesses that $S \subseteq X$ is finitely supported, then it also witnesses that the complement $(X - S) \subseteq X$ is finitely supported.

We will make use of a version of Tarski's fixed point theorem in the category of FM-sets:

Lemma 3.4. *An* FM-complete lattice *is an FM-set L equipped with an equivariant partial order relation $\sqsubseteq$ such that every finitely supported subset has a greatest lower bound. Given such an L, every element $f \in L^L$ which is monotone possesses a least* (*pre-*)*fixed point*.

Proof. The subset $\{x \in L \mid f(x) \sqsubseteq x\}$ is supported by the same finite set of atoms that supports f and therefore has a greatest lower bound. As usual, this is the least (pre-)fixed point of f. □

Definition 3.5 (*FM-cpos and FM-cppos*). An *FM-cpo* is an FM-set D equipped with an equivariant partial order $\sqsubseteq$ that possesses 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 there is a finite subset $\overline{a} \subseteq \mathbb{A}$ such that $\forall a, a' \in \mathbb{A} - \overline{a}. \forall n. (a\ a') \cdot d_n = d_n$. (This is equivalent to requiring that the subset $\{d_n \mid n \geqslant 0\} \subseteq D$ be finitely supported in the sense of Definition 3.2.) An *FM-cppo* is an FM-cpo with a least element $\perp$; note that since $\perp \sqsubseteq (a\ a') \cdot \perp$ (since $\perp$ is least) and hence $(a\ a') \cdot \perp \sqsubseteq (a\ a') \cdot (a\ a') \cdot \perp = \perp$, we have $\mathrm{supp}(\perp) = \emptyset$. A morphism f of FM-cpos is an equivariant function which is monotone and preserves lubs of finitely supported ω-chains. A morphism of FM-cppos, written $f : D \multimap E$, has the same properties but is also strict ($f(\perp) = \perp$). FM-cpos (respectively FM-cppos) and their morphisms form a category **FM-Cpo** (respectively $\mathbf{FM\text{-}Cpo}_{\perp}$).

Lemma 3.6 (*Least fixed points*). *Given an FM-cppo D, every function f from D to D that is finitely supported* (*Definition* 3.2), *monotone and preserves lubs of finitely-supported ω-chains possesses a least* (*pre-*)*fixed point* $\mathbf{fix}(f) \in D$.

Proof. Just note that the classical construction of $\mathbf{fix}(f)$ as the lub of the chain $\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \cdots$ can be used here, because this chain is finitely supported (by any $\overline{a}$ that finitely supports f, since as we noted above, $\perp$ always has empty support). □

To each Mini-FreshML type τ we assign an FM-cppo $[\![\tau]\!]$. To do so we make use of the following constructions on FM-cppos: smash product $(- \otimes -)$, coalesced sum $(- \oplus -)$, lifting $(-_{\perp})$, function space $(- \to -)$, strict function space $(- \multimap -)$, and atom-abstraction $([\mathbb{A}]-)$. All but the last three are just as for classical domain theory [2]. The FM-cppo $D \to D'$ is given by the FM-set of finitely supported functions f from D to D' (Definition 3.2) that preserve the partial order and lubs of finitely supported ω-chains; as usual, the partial order on $D \to D'$ is inherited from D' argument-wise. The FM-cppo $D \multimap D'$ is the sub-FM-cppo of $D \to D'$ consisting of those functions that also preserve $\perp$. The FM-cppo $[\mathbb{A}]D$ generalises to domain theory the atom-abstraction construct of [4, Section 5] and is defined as follows.

Definition 3.7 (*Atom-abstraction*). Given an FM-cpo D, the FM-cpo $[\mathbb{A}]D$ consists of 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 atom a'' not in $\{a\} \cup \mathrm{supp}(d) \cup \{a'\} \cup \mathrm{supp}(d')$; the permutation action is $\pi \cdot [a]d \overset{\text{def}}{=} [\pi(a)](\pi \cdot d)$ and the partial order is induced by the above pre-order. The elements of $[\mathbb{A}]D$ are indeed finitely supported: one can calculate that $\mathrm{supp}([a]d) = \mathrm{supp}(d) - \{a\}$. Finitely supported ω-chains in $[\mathbb{A}]D$ possess lubs, which can be calculated as follows: given a chain $[a_0]d_0 \sqsubseteq [a_1]d_1 \sqsubseteq \cdots$ supported by a finite set of atoms $\overline{a}$, picking any $a \in \mathbb{A} - \overline{a}$ one can show that $(a_o\ a) \cdot d_0 \sqsubseteq (a_1\ a) \cdot d_1 \sqsubseteq \cdots$ is an ω-chain in D supported by $\overline{a} \cup \{a\}$; taking its lub, d say, then $[a]d$ is a lub for the original chain $[a_0]d_0 \sqsubseteq [a_1]d_1 \sqsubseteq \cdots$. If D has a least element $\bot$, then so does $[\mathbb{A}]D$, namely $[a]\bot$ (for any $a \in \mathbb{A}$).

As may be expected, all these constructions are functorial. Lifting and atom-abstraction determine functors $\mathbf{FM\text{-}Cpo}_\bot \longrightarrow \mathbf{FM\text{-}Cpo}_\bot$; the smash product and sum determine functors $\mathbf{FM\text{-}Cpo}_\bot \times \mathbf{FM\text{-}Cpo}_\bot \longrightarrow \mathbf{FM\text{-}Cpo}_\bot$ and the function and strict function spaces determine functors $\mathbf{FM\text{-}Cpo}_\bot^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_\bot \longrightarrow \mathbf{FM\text{-}Cpo}_\bot$. In fact the action of these constructs on morphisms enriches to locally continuous functors in the following sense. We say that a functor $F : \mathbf{FM\text{-}Cpo}_\bot \longrightarrow \mathbf{FM\text{-}Cpo}_\bot$ is *locally FM-continuous* if its action on morphisms is induced by equivariant functions $F_{D,E} : (D \multimap E) \to (FD \multimap FE)$ that are monotonic and preserve least upper bounds of finitely-supported chains. For example when $F = [\mathbb{A}](-)$, $F_{D,E}$ sends $f \in (D \multimap E)$ to the element $[\mathbb{A}]f \in ([\mathbb{A}]D \multimap [\mathbb{A}]E)$ that maps $[a]d$ to $[a']f((a\ a') \cdot d)$ where a' is any atom not in $\mathrm{supp}(f) \cup \{a\} \cup \mathrm{supp}(d)$ (the result is independent of which such a' we choose).

For simplicity, we assume there is a single declaration (4) of a datatype δ (and later take the declaration to be (5)). [14] Following [9,2], the denotation of δ is the minimally invariant FM-cppo associated with a locally FM-continuous functor $F : \mathbf{FM\text{-}Cpo}_\bot^{\mathrm{op}} \times \mathbf{FM\text{-}Cpo}_\bot \longrightarrow \mathbf{FM\text{-}Cpo}_\bot$:

$$F(-,+) \overset{\text{def}}{=} F_{\sigma_1}(-,+) \oplus \cdots \oplus F_{\sigma_n}(-,+), \tag{7}$$

where for each type τ the functor F_τ is defined by induction on the structure of τ as follows:

$$\begin{aligned}
F_{\mathtt{unit}}(D^-, D^+) &\overset{\text{def}}{=} 1_\bot,\\
F_{\mathtt{name}}(D^-, D^+) &\overset{\text{def}}{=} \mathbb{A}_\bot,\\
F_\delta(D^-, D^+) &\overset{\text{def}}{=} D^+,\\
F_{\mathtt{<<name>>}\tau}(D^-, D^+) &\overset{\text{def}}{=} [\mathbb{A}]F_\tau(D^-, D^+),\\
F_{\tau\times\tau'}(D^-, D^+) &\overset{\text{def}}{=} F_\tau(D^-, D^+) \otimes F_{\tau'}(D^-, D^+),\\
F_{\tau\to\tau'}(D^-, D^+) &\overset{\text{def}}{=} F_\tau(D^+, D^-) \multimap (F_{\tau'}(D^-, D^+))^{\bot\bot}.
\end{aligned}$$

Here $(-)^{\bot\bot}$ is the continuation monad (3) defined in the Introduction; $1_\bot$ and $\mathbb{A}_\bot$ are flat FM-cppos on the FM-sets $1 \overset{\text{def}}{=} \{\top\}$ (trivial action: $\pi \cdot \top \overset{\text{def}}{=} \top$) and $\mathbb{A}$ (canonical action: $\pi \cdot a \overset{\text{def}}{=} \pi(a)$). Just as Lemma 3.6 shows that least fixed points can be constructed in the usual way, so can minimally invariant solutions to such domain equations be constructed in

[14] For finitely many datatypes one just has to solve a finite set of simultaneous domain equations rather than a single one.

this setting using the normal technique of embedding-projection pairs [9,2] adapted to FM-cppos, using finitely supported ω-chains where classically one uses arbitrary ω-chains.[15] So let D be an FM-cppo which is a minimal invariant solution to the recursive domain equation $D = F(D, D)$. Thus D comes equipped with an isomorphism

$$i : F(D, D) \cong D \tag{8}$$

and (D, i) is uniquely determined by the fact that the identity on D is $\mathbf{fix}(\phi)$, where $\phi : (D \multimap D) \to (D \multimap D)$ is given by $\phi(f) = i \circ F(f, f) \circ i^{-1}$.

We may now define the denotation $[\![\tau]\!]$ of a type τ as $[\![\tau]\!] \overset{\text{def}}{=} F_\tau(D, D)$. Denotations of typing contexts are given using a finite smash product: $[\![\Gamma]\!] \overset{\text{def}}{=} \bigotimes_{x \in \text{dom}(\Gamma)} [\![\Gamma(x)]\!]$. The denotations 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 finitely supported functions[16] of the following kinds:

$$\begin{aligned} \mathcal{V}[\![\Gamma \vdash v : \tau]\!] &\in [\![\Gamma]\!] \multimap [\![\tau]\!], \\ \mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!] &\in [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp}, \\ \mathcal{E}[\![\Gamma \vdash e : \tau]\!] &\in [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp\perp}, \end{aligned}$$

where for each FM-cppo D we define $D^\perp \overset{\text{def}}{=} D \multimap 1_\perp$. Intuitively, an element of $[\![\tau]\!]^\perp$ models a frame stack accepting a value of type τ and returning $\top$ for termination, or $\perp$ for divergence. Just as the behaviour of expressions is determined by any enclosing frame stack, the denotation of some expression in context is then a function in $[\![\tau]\!]^{\perp\perp}$ that accepts the denotation of a frame stack in context and returns either $\perp$ or $\top$. Thus, the denotations of expressions in context make use of the *continuation monad* $(-)^{\perp\perp}$ based on an FM-cppo of "answers" given by $1_\perp$. We have the usual two monad operations for $(-)^{\perp\perp}$, namely the unit **return** $\in D \multimap D^{\perp\perp}$ given by

$$\mathbf{return}(d) \overset{\text{def}}{=} \lambda\delta \in D^\perp.\, \delta(d) \in D^{\perp\perp} \tag{9}$$

and the Kleisli lifting operation **lift** $\in (D \multimap E^{\perp\perp}) \multimap (D^{\perp\perp} \multimap E^{\perp\perp})$ that sends $f \in (D \multimap E^{\perp\perp})$ and $e \in D^{\perp\perp}$ to

$$\mathbf{lift}(f)(e) \overset{\text{def}}{=} \lambda\varepsilon \in E^\perp.\, e(\lambda d \in D.\, f(d)(\varepsilon)) \in E^{\perp\perp}. \tag{10}$$

We use the informal notation **let** $d \Leftarrow e$ **in** $e'[d]$ for $\mathbf{lift}(f)(e)$ when f is given by some expression $e'[d]$ (involving d strict continuously).

The denotation of recursive function values makes use of the least fixed point operation **fix** $\in (D \to D) \multimap D$ from Lemma 3.6. The denotation of the `fresh` expression makes use of the element **new** $\in (\mathbb{A}_\perp)^{\perp\perp}$ mentioned in the Introduction:

$$\mathcal{E}[\![\Gamma \vdash \texttt{fresh} : \texttt{name}]\!] \overset{\text{def}}{=} \lambda\rho \in [\![\Gamma]\!].\, \mathbf{new}.$$

[15] A logically more sophisticated viewpoint is that we are carrying out the usual construction, but in the axiomatic FM-set theory [4] rather than in usual axiomatic ZFC set theory.

[16] Note that these functions do not necessarily have empty support (consider $\mathcal{V}[\![\emptyset \vdash a : \texttt{name}]\!]$ for example, where $a \in \mathbb{A}$) and are thus not necessarily morphisms in the category $\mathbf{FM\text{-}Cpo}_\perp$.

Here **new** is the element of $(\mathbb{A}_\perp)^{\perp\perp}$ that sends each $\alpha \in (\mathbb{A}_\perp)^{\perp}$ to $\alpha(a) \in 1_\perp$ where a is any element of $\mathbb{A} - \mathrm{supp}(\alpha)$ (for each α, there are infinitely many such a because $\mathbb{A}$ is infinite and $\mathrm{supp}(\alpha)$ is finite); this gives a well-defined (strict, continuous) function because for any other $a' \in \mathbb{A} - \mathrm{supp}(\alpha)$ we have $(a\ a') \cdot \alpha = \alpha$ (since neither a nor a' are in the support of α) and hence $\alpha(a) = ((a\ a') \cdot \alpha)(a) = (a\ a') \cdot (\alpha((a'\ a) \cdot a)) = (a\ a') \cdot (\alpha(a')) = \alpha(a')$ (where in the last step we use the fact that any $x \in 1_\perp$ satisfies $(a\ a') \cdot x = x$). The denotation of `let <<`x`>>`x `=` e `in` e' expressions involves a similar use of choosing some fresh $a \in \mathbb{A}$ (mirroring the dynamic allocation involved in the evaluation of such expressions), noting that the result is independent of which fresh a is chosen.[17] The full definition of $\mathcal{E}[\![-]\!]$ by induction on the structure of expressions is given in Appendix C; the definition of $\mathcal{V}[\![-]\!]$ by induction on the structure of values and making use of $\mathcal{E}[\![-]\!]$ is given in Appendix D; the definition of $\mathcal{S}[\![-]\!]$ by induction on the length of frame stacks and making us of both $\mathcal{E}[\![-]\!]$ and $\mathcal{V}[\![-]\!]$ is given in Appendix E. The "continuation-passing style" of these definitions is self-evident. Note that since a value is in particular an expression, it has a denotation *qua* value, $\mathcal{V}[\![\Gamma \vdash v : \tau]\!]$, and *qua* expression, $\mathcal{E}[\![\Gamma \vdash v : \tau]\!]$. The two denotations are related via the unit (9) of the continuation monad:

Lemma 3.8. *If v is a value satisfying $\Gamma \vdash v : \tau$, then* $\mathcal{E}[\![\Gamma \vdash v : \tau]\!] = \mathbf{return} \circ \mathcal{V}[\![\Gamma \vdash v : \tau]\!] \in [\![\Gamma]\!] \multimap [\![\tau]\!]^{\perp\perp}$.

For closed values v of type τ, we write $\mathcal{V}[\![v]\!]$ for the element $\mathcal{V}[\![\vdash v : \tau]\!](\emptyset)$ of the FM-cppo $[\![\tau]\!]$ and use a similar convention for closed frame stacks and expressions.

Remark 3.9 (*FM-sets of syntax*). Note that the expressions of Mini-FreshML form an FM-set. The action of a permutation of atoms on an expression e is given by applying the permutation to the atoms occurring in any syntax tree representing e (recall that we identify expressions up to α-conversion of bound value identifiers); and then the support of an expression is in fact the finite set of atoms occurring in the expression. Furthermore, it is easy to prove that the denotational semantics gives equivariant functions on syntax, so that, for example $(a\ a') \cdot \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) = \mathcal{E}[\![\Gamma \vdash (a\ a') \cdot e : \tau]\!]((a\ a') \cdot \rho)$. In particular it is the case that $\mathrm{supp}(\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho)) \subseteq \mathrm{supp}(e) \cup \mathrm{supp}(\rho)$.

We wish to use our denotational semantics to prove operational properties of Mini-FreshML expressions. An important stepping-stone in this process is the construction of certain type-indexed *logical relations* which relate domain elements to values, frame stacks and expressions respectively:

$$\lhd_\tau^{\mathrm{val}} \subseteq [\![\tau]\!] \times \mathrm{Val}_\tau, \qquad \lhd_\tau^{\mathrm{stk}} \subseteq [\![\tau]\!]^{\perp} \times \mathrm{Stack}_\tau, \qquad \lhd_\tau^{\mathrm{exp}} \subseteq [\![\tau]\!]^{\perp\perp} \times \mathrm{Exp}_\tau,$$

where Val_τ is the set of closed Mini-FreshML values of type τ, Stack_τ is the set of well-typed frame stacks expecting an argument of type τ and Exp_τ is the set of closed expressions of type τ. These relations are all required to be equivariant subsets in the sense of Definition 3.2. We also require them to be suitably admissible; for example, for each

[17] This is just a manifestation of the "some/any" property of fresh names [4, Proposition 4.10].

$v \in \mathrm{Val}_\tau$, we require that $\{d \mid d \lhd^{\mathrm{val}}_\tau v\}$ to contain $\bot$ and be closed under lubs of finitely supported ω-chains in $[\![\tau]\!]$ (and similarly for $\lhd^{\mathrm{stk}}_\tau$ and $\lhd^{\mathrm{exp}}_\tau$). Finally, the relations should satisfy the following properties that follow the structure of types:

$$d \lhd^{\mathrm{val}}_{\mathtt{unit}} () \tag{11}$$

$$d \lhd^{\mathrm{val}}_{\mathtt{name}} a \Leftrightarrow d \neq \bot \Rightarrow d = a, \tag{12}$$

$$d \lhd^{\mathrm{val}}_\delta \mathtt{C}_k(v) \Leftrightarrow \exists\, d_k \in [\![\sigma_k]\!].d = (i \circ \mathrm{in}_k)(d_k) \;\wedge\; d_k \lhd^{\mathrm{val}}_{\sigma_k} v, \tag{13}$$

$$[a]\, d \lhd^{\mathrm{val}}_{\mathtt{<<name>>}\tau} \mathtt{<<}a'\mathtt{>>}v \Leftrightarrow (a\ a'') \cdot d \lhd^{\mathrm{val}}_\tau (a'\ a'') \cdot v \quad \text{for some } a \in \mathbb{A} - \mathrm{supp}(a, d, a', v), \tag{14}$$

$$(d_1, d_2) \lhd^{\mathrm{val}}_{\tau \times \tau'} (v_1, v_2) \Leftrightarrow d_1 \lhd^{\mathrm{val}}_\tau v_1 \;\wedge\; d_2 \lhd^{\mathrm{val}}_{\tau'} v_2, \tag{15}$$

$$d \lhd^{\mathrm{val}}_{\tau \to \tau'} v \Leftrightarrow \forall\, d' \lhd^{\mathrm{val}}_\tau v'.d(d') \lhd^{\mathrm{exp}}_{\tau'} v\ v', \tag{16}$$

$$\sigma \lhd^{\mathrm{stk}}_\tau S \Leftrightarrow \forall\, d \lhd^{\mathrm{val}}_\tau v.\sigma(d) = \top \Rightarrow \langle S, v\rangle{\downarrow}, \tag{17}$$

$$\varepsilon \lhd^{\mathrm{exp}}_\tau e \Leftrightarrow \forall\, \sigma \lhd^{\mathrm{stk}}_\tau S.\varepsilon(\sigma) = \top \Rightarrow \langle S, e\rangle{\downarrow}. \tag{18}$$

In clause (13), i is the isomorphism from (8) and $\mathrm{in}_k \in D_k \multimap D_1 \oplus \cdots \oplus D_n$ is the kth injection into a coalesced sum. Clause (14) makes use of the support of a tuple; as in Definition 3.1, $\mathrm{supp}(a, d, a', v) = \{a\} \cup \mathrm{supp}(d) \cup \{a'\} \cup \mathrm{supp}(v)$ (and $\mathrm{supp}(v)$ is just the finite set of atoms occurring in the value v—see Remark 3.9). In clauses (16) and (17), the notation $\forall d \lhd^{\mathrm{val}}_\tau v.(-)$ stands for $\forall d \in [\![\tau]\!], v \in \mathrm{Val}_\tau.d \lhd^{\mathrm{val}}_\tau v \Rightarrow (-)$ (and similarly for $\lhd^{\mathrm{stk}}_\tau$ in (18)). Clauses (17) and (18) define the logical relations for frame stacks and for expressions in terms of that for values. Clauses (11)–(16) serve to define $\lhd^{\mathrm{val}}_\tau$ at compound types τ in terms of $\lhd^{\mathrm{val}}_\delta$; and $\lhd^{\mathrm{val}}_\delta = F(\lhd^{\mathrm{val}}_\delta, \lhd^{\mathrm{val}}_\delta)$ is a fixed point of a certain operator acting on relations (whose definition we give in detail below). Unfortunately, due to the negative occurrence of $\lhd^{\mathrm{val}}_\tau$ on the right-hand side of the clause (16) for function types, this operator is non-monotonic; so it is non-trivial to deduce the existence of a suitable relation $\lhd^{\mathrm{val}}_\delta$. We do so by adapting the techniques of [9] to the world of FM-sets, as follows.

For each type τ, let $\mathcal{R}_\tau$ be the set of finitely supported subsets $R \subseteq [\![\tau]\!] \times \mathrm{Val}_\tau$ with the desired admissibility property, namely that for each $v \in \mathrm{Val}_\tau$, the subset $\{d \mid (d, v) \in R\}$ contains $\bot$ and is closed under lubs of finitely supported ω-chains in $[\![\tau]\!]$. This becomes an FM-set if we define the permutation action of $\pi \in \mathit{perm}(\mathbb{A})$ on $R \in \mathcal{R}_\tau$ to be $\pi \cdot R \stackrel{\mathrm{def}}{=} \{(\pi \cdot d, \pi \cdot v) \mid (d, v) \in R\}$. Partially ordering its elements by inclusion, it is not hard to see that $\mathcal{R}_\tau$ is in fact an FM-complete lattice (cf. Lemma 3.4), the greatest lower bound of a finitely supported subset of $\mathcal{R}_\tau$ just being given by intersection. Given $R^-, R^+ \in \mathcal{R}_\delta$, define $F_\tau(R^-, R^+) \in \mathcal{R}_\tau$ by induction on the structure of the type τ, as follows:

$$F_{\mathtt{unit}}(R^-, R^+) \stackrel{\mathrm{def}}{=} \{(d, ()) \mid d \in 1_\bot\},$$

$$F_{\mathtt{name}}(R^-, R^+) \stackrel{\mathrm{def}}{=} \{(\bot, a) \mid a \in \mathbb{A}\} \cup \{(a, a) \mid a \in \mathbb{A}\},$$

$$F_\delta(R^-, R^+) \stackrel{\mathrm{def}}{=} R^+,$$

$$F_{\mathtt{<<name>>}\tau}(R^-, R^+) \stackrel{\mathrm{def}}{=} \{([a]\, d, \mathtt{<<}a'\mathtt{>>}v) \mid \exists a'' \in \mathbb{A} - \mathrm{supp}(R^-, R^+, a, d, a', v). \\ ((a\ a'') \cdot d, (a'\ a'') \cdot v) \in F_\tau(R^-, R^+)\},$$

$$F_{\tau\times\tau'}(R^-, R^+) \overset{\text{def}}{=} \{(\langle d, d'\rangle, (v, v')) \mid (d, v) \in F_\tau(R^-, R^+) \ \wedge \\ (d', v') \in F_{\tau'}(R^-, R^+)\},$$

$$F_{\tau\to\tau'}(R^-, R^+) \overset{\text{def}}{=} \{(d, \texttt{fun}\ f(x)\ \texttt{=}\ e) \mid \\ \forall (d', v') \in F_\tau(R^+, R^-), \sigma \in [\![\tau']\!]^\perp, S \in \mathrm{Stack}_{\tau'}. \\ (\forall (d'', v'') \in F_{\tau'}(R^-, R^+).\sigma(d'') = \top \Rightarrow \langle S, v''\rangle\downarrow) \\ \Rightarrow d(d')(\sigma) = \top \Rightarrow \langle S, (\texttt{fun}\ f(x)\ \texttt{=}\ e)\ v'\rangle\downarrow\}.$$

(The notation "$\langle d, d'\rangle$" in the clause for product types indicates the smash pair such that $\langle d_1, d_2\rangle \overset{\text{def}}{=} \perp_{[\![\tau_1]\!]\otimes[\![\tau_2]\!]}$ when either of $d_1 \in [\![\tau_1]\!]$ and $d_2 \in [\![\tau_2]\!]$ are bottom). Assuming the single datatype δ has a top-level declaration as in (4), we define $F(R^-, R^+) \in \mathcal{R}_\delta$ by

$$F(R^-, R^+) \overset{\text{def}}{=} \{(\mathrm{in}_k(d), \mathtt{C}_k(v)) \mid 1 \leqslant k \leqslant n \ \wedge \ (d, v) \in F_{\sigma_k}(R^-, R^+)\}.$$

Then the relation we seek is a fixed point $\lhd_\delta^{\mathrm{val}} = F(\lhd_\delta^{\mathrm{val}}, \lhd_\delta^{\mathrm{val}})$, with the value logical relation at other types given by $\lhd_\tau^{\mathrm{val}} \overset{\text{def}}{=} F_\tau(\lhd_\delta^{\mathrm{val}}, \lhd_\delta^{\mathrm{val}})$.

The definition of $R^-, R^+ \mapsto F(R^-, R^+)$ implies that it is an equivariant function that is order-reversing in its first argument and order-preserving in its second. Therefore $F^\S(R^-, R^+) \overset{\text{def}}{=} (F(R^+, R^-), F(R^-, R^+))$ determines a monotone equivariant function from the FM-complete lattice $\mathcal{R}_\delta^{\mathrm{op}} \times \mathcal{R}_\delta$ to itself. Therefore we can apply Lemma 3.4 to deduce that it has a least fixed point, $(\varDelta^-, \varDelta^+)$ say. Thus $\varDelta^-, \varDelta^+ \in \mathcal{R}_\delta$ satisfy

- $\varDelta^- = F(\varDelta^+, \varDelta^-)$ and $F(\varDelta^-, \varDelta^+) = \varDelta^+$.
- For any $R^-, R^+ \in \mathcal{R}_\delta$, if $R^- \subseteq F(R^+, R^-)$ and $F(R^-, R^+) \subseteq R^+$, then $R^- \subseteq \varDelta^-$ and $\varDelta^+ \subseteq R^+$.
- $\mathrm{supp}(\varDelta^-) = \emptyset = \mathrm{supp}(\varDelta^+)$.

From this it follows that $\varDelta^+ \subseteq \varDelta^-$. So to construct $\lhd_\delta^{\mathrm{val}}$, it suffices to see that $\varDelta^- \subseteq \varDelta^+$, so that we can take $\lhd_\delta^{\mathrm{val}} = \varDelta^- = \varDelta^+$. To prove that inclusion holds, we appeal to the minimal invariance property of the FM-cppo $[\![\delta]\!] = D$ and the isomorphism i in (8). First, one can prove from the definition of F that the subset $\{f \in (D \multimap D) \mid \forall (d, v) \in \varDelta^-.(f(d), v) \in \varDelta^+\}$ is mapped to itself by the function $\phi = i \circ F(f, f) \circ i^{-1} : (D \multimap D) \to (D \multimap D)$ whose least fixed point is the identity on D. Since that subset contains $\perp$ and is closed under lubs of finitely supported ω-chains, it follows from the construction of $\mathbf{fix}(\phi)$ in Lemma 3.6 that the subset contains the identity on D—which means that $\varDelta^- \subseteq \varDelta^+$, as required.

We next give the "fundamental property" of the logical relations we have just constructed. To state the property we need to introduce some terminology for *value-substitutions*, ψ, which are finite partial functions from value identifiers to values. Given such a ψ, we write $e[\psi]$ for the result of the capture-avoiding simultaneous substitution of $\psi(x)$ for x in e as x ranges over $\mathrm{dom}(\psi)$; similarly for value-substitutions into values $v[\psi]$, and into frame stacks $S[\psi]$. Given a typing context Γ, let Subst_Γ be the set of all value-substitutions ψ with domain $\mathrm{dom}(\Gamma)$ and such that for each $x \in \mathrm{dom}(\psi)$, $\psi(x)$ is closed. Given $\psi \in \mathrm{Subst}_\Gamma$ and $\rho \in [\![\Gamma]\!]$, write $\rho \lhd_\Gamma \psi$ to mean that for each $x \in \mathrm{dom}(\rho)$, $\rho(x) \lhd_{\Gamma(x)}^{\mathrm{val}} \psi(x)$.

Lemma 3.10 (*Fundamental property of the logical relations*). *For all typing contexts Γ, values v, frame stacks S and expressions e, we have that*

$$\begin{aligned}
&\Gamma \vdash v : \tau &&\Rightarrow \forall\, \rho \lhd_\Gamma \psi . \mathcal{V}[\![\Gamma \vdash v : \tau]\!]\rho \lhd^{\mathrm{val}}_\tau v[\psi],\\
&\Gamma \vdash S : \tau \multimap \_ &&\Rightarrow \forall\, \rho \lhd_\Gamma \psi . \mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_]\!]\rho \lhd^{\mathrm{stk}}_\tau S[\psi],\\
&\Gamma \vdash e : \tau &&\Rightarrow \forall\, \rho \lhd_\Gamma \psi . \mathcal{E}[\![\Gamma \vdash e : \tau]\!]\rho \lhd^{\mathrm{exp}}_\tau e[\psi].
\end{aligned}$$

Proof. These properties follow by induction on the derivation of the typing judgements, using the definitions of $\mathcal{V}[\![-]\!]$, $\mathcal{S}[\![-]\!]$, $\mathcal{E}[\![-]\!]$ and the properties (11)–(18) of the logical relations. $\square$

Theorem 3.11 (*Computational adequacy*). *Given $\Gamma \vdash e : \tau$, $\psi \in Subst_\Gamma$ and $S \in Stack_\tau$, then*

$$\langle S, e[\psi]\rangle\downarrow \;\Leftrightarrow\; \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\mathcal{V}[\![\psi]\!])(\mathcal{S}[\![S]\!]) = \top,$$

where $\mathcal{V}[\![\psi]\!] \in [\![\Gamma]\!]$ maps each $x \in dom(\psi)$ to $\mathcal{V}[\![\psi(x)]\!]$. In particular for all closed typeable expressions $e \in Exp_\tau$, values $v \in Val_\tau$ and frame stacks $S \in Stack_\tau$, we have: $\langle S, e\rangle\downarrow \Leftrightarrow \mathcal{E}[\![e]\!](\mathcal{S}[\![S]\!]) = \top$ and $\langle S, v\rangle\downarrow \Leftrightarrow \mathcal{S}[\![S]\!](\mathcal{V}[\![v]\!]) = \top$.

Proof. The first sentence follows from the second one using a substitutivity property of the denotational semantics

$$\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\mathcal{V}[\![\psi]\!]) = \mathcal{E}[\![e[\psi]]\!] \tag{19}$$

that is proved by induction on the structure of e (and similarly for values and frame stacks). The computational adequacy property for closed expressions is established by first proving a *soundness* property

$$\langle S, e\rangle\downarrow \Rightarrow \mathcal{E}[\![e]\!](\mathcal{S}[\![S]\!]) = \top \tag{20}$$

by induction on the derivation of $\langle S, e\rangle\downarrow$. The reverse implication is a corollary of Lemma 3.10: by the fundamental property of the logical relation we have $\mathcal{E}[\![e]\!] \lhd^{\mathrm{exp}}_\tau e$ and $\mathcal{S}[\![S]\!] \lhd^{\mathrm{stk}}_\tau S$; then properties (17) and (18) give the required implication. $\square$

4. Extensionality and correctness results

We now examine how our denotational semantics of Mini-FreshML can be used to prove the correctness result stated at the end of Section 2 (Theorem 2.3), which we recall centres around the notion of contextual equivalence. The quantification over all contexts that is part of the definition of contextual equivalence makes it hard to work with directly. Instead we make use of an alternative characterisation in terms of Mason and Talcott's notion of *CIU-equivalence* [5]. [18] We prove that this coincides with Mini-FreshML contextual equivalence using the logical relation from the previous section.

[18] CIU = "Closed Instances of all Uses".

Definition 4.1 (*CIU-equivalence*). We write $\Gamma \vdash e \approx_{\mathrm{ciu}} e' : \tau$ to indicate that the typeable expressions e and e' of type τ (in context Γ) are CIU-equivalent. This equivalence relation is the symmetrisation of the *CIU-pre-order* relation, written $\Gamma \vdash e \leqslant_{\mathrm{ciu}} e' : \tau$, which by definition holds if $\Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, and for all closing substitutions $\psi \in \mathrm{Subst}_\Gamma$ and all closed frame stacks S, $\langle S, e[\psi]\rangle{\downarrow}$ implies $\langle S, e'[\psi]\rangle{\downarrow}$. We write $e \leqslant_{\mathrm{ciu}} e'$ (respectively $\approx_{\mathrm{ciu}}$) when e and e' are closed expressions and $\emptyset \vdash e \leqslant_{\mathrm{ciu}} e' : \tau$ holds for some τ.

To show that CIU-equivalence coincides with contextual equivalence we need to turn frame stacks into (evaluation) contexts, as follows. The lemma is proved by a routine induction on the structure of frame stacks, S.

Lemma 4.2. *Define an operation mapping frame stacks S to contexts $\mathcal{T}(S)$ by induction on the structure of S*:

$$\mathcal{T}([]) \overset{\mathrm{def}}{=} [-] \qquad \mathcal{T}(S \circ \mathcal{F}) \overset{\mathrm{def}}{=} (\mathcal{T}(S))[\mathcal{F}].$$

Then for all stacks S and expressions e, $\langle [], \mathcal{T}(S)[e]\rangle{\downarrow} \Leftrightarrow \langle S, e\rangle{\downarrow}$.

Theorem 4.3 (*Coincidence of $\approx_{\mathrm{ctx}}$ with $\approx_{\mathrm{ciu}}$*). *For any typing context Γ and expressions e, e' it is the case that $\Gamma \vdash e \leqslant_{\mathrm{ctx}} e' : \tau$ iff $\Gamma \vdash e \leqslant_{\mathrm{ciu}} e' : \tau$. Thus the relations $\approx_{\mathrm{ctx}}$ and $\approx_{\mathrm{ciu}}$ coincide.*

Proof. We prove that $\leqslant_{\mathrm{ctx}}$ and $\leqslant_{\mathrm{ciu}}$ both coincide with the relation $\leqslant_{\mathrm{e}}$ defined from the denotational semantics and the logical relation as follows:

$$\Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau \overset{\mathrm{def}}{\Leftrightarrow} \Gamma \vdash e, e' : \tau \ \wedge\ \forall \rho \lhd_\Gamma \psi . \mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) \lhd_\tau^{\mathrm{exp}} e'[\psi],$$

where $\Gamma \vdash e, e' : \tau$ is the obvious conjunction of typing judgements. From the fundamental property (Lemma 3.10) we have $\Gamma \vdash e : \tau$ implies $\Gamma \vdash e \leqslant_{\mathrm{e}} e : \tau$; and from property (18) of the logical relation for expressions and the definition of $\leqslant_{\mathrm{ciu}}$ we have that $\leqslant_{\mathrm{e}}$ is closed under composition with $\leqslant_{\mathrm{ciu}}$ on the right. Therefore

$$\Gamma \vdash e \leqslant_{\mathrm{ciu}} e' : \tau \Rightarrow \Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau. \tag{21}$$

The compositional nature of the denotational semantics and the fundamental property of the logical relation ensure that if $\Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau$ holds, then so does $C[e] \leqslant_{\mathrm{e}} C[e']$, for any context $C[-]$ for which $C[e]$ and $C[e']$ are closed well-typed expressions. Then by computational adequacy (Theorem 3.11) and property (18) of the logical relation we have that $\langle [], C[e]\rangle{\downarrow}$ implies $\langle [], C[e']\rangle{\downarrow}$. Therefore

$$\Gamma \vdash e \leqslant_{\mathrm{e}} e' : \tau \Rightarrow \Gamma \vdash e \leqslant_{\mathrm{ctx}} e' : \tau. \tag{22}$$

To complete a circle of implications we just have to prove that the contextual pre-order is contained within the CIU-pre-order. To do so, we first have to show that the “instantiation” part of CIU, i.e. applying a value-substitution to an expression, is contextual. But we now know from (21) and (22) that every CIU-equivalence is also a contextual equivalence. In particular we have β-value conversion

$$\Gamma \vdash (\mathtt{fun}\ f(x)\ \mathtt{=}\ e)(v) \approx_{\mathrm{ctx}} e[v/x] \tag{23}$$

since the corresponding CIU-equivalence is immediate from the definitions of $\approx_{\text{ciu}}$ and the termination relation $\langle -, - \rangle{\downarrow}$. Because of the way they are defined, $\leqslant_{\text{ctx}}$ and $\approx_{\text{ctx}}$ are compatible with the various expression-forming constructs of Mini-FreshML, i.e. whenever $e \leqslant_{\text{ctx}} e'$, then $C[e] \leqslant_{\text{ctx}} C[e']$ for any context C (and similarly for $\approx_{\text{ctx}}$). Thus if $\Gamma, x : \tau \vdash e \leqslant_{\text{ctx}} e' : \tau'$ and $\Gamma \vdash v : \tau$, then $\Gamma \vdash (\texttt{fun}\ f\,(x)\ \texttt{=}\ e) v \leqslant_{\text{ctx}} (\texttt{fun}\ f\,(x)\ \texttt{=}\ e') v : \tau'$; and so by (23), $\Gamma \vdash e[v/x] \leqslant_{\text{ctx}} e'[v/x] : \tau'$. From this it follows that we have

$$\Gamma \vdash e \leqslant_{\text{ctx}} e' : \tau \Rightarrow \forall \psi \in \text{Subst}_\Gamma . e[\psi] \leqslant_{\text{ctx}} e'[\psi]. \tag{24}$$

So if $\Gamma \vdash e \leqslant_{\text{ctx}} e' : \tau$, then for all closing value-substitutions $\psi \in \text{Subst}_\Gamma$ and frame stacks $S \in \text{Stack}_\tau$, using the congruence property of $\leqslant_{\text{ctx}}$ and (24) we have $\mathcal{T}(S)[e[\psi]] \leqslant_{\text{ctx}} \mathcal{T}(S)[e'[\psi]]$; hence $\langle [], \mathcal{T}(S)[e[\psi]] \rangle{\downarrow}$ implies that $\langle [], \mathcal{T}(S)[e'[\psi]] \rangle{\downarrow}$ and so by Lemma 4.2, $\langle S, e[\psi] \rangle{\downarrow}$ implies $\langle S, e'[\psi] \rangle{\downarrow}$. Therefore

$$\Gamma \vdash e \leqslant_{\text{ctx}} e' : \tau \Rightarrow \Gamma \vdash e \leqslant_{\text{ciu}} e' : \tau \tag{25}$$

and the circle of implications is complete. □

Combining Theorems 3.11 and 4.3, we have:

Corollary 4.4 (*Equality of denotation*). *If* $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] = \mathcal{E}[\![\Gamma \vdash e' : \tau]\!]$, *then* $\Gamma \vdash e \approx_{\text{ctx}} e' : \tau$. *In particular, if* e *and* e' *are closed expressions of the same type, then* $\mathcal{E}[\![e]\!] = \mathcal{E}[\![e']\!]$ *implies* $e \approx_{\text{ctx}} e'$.

Remark 4.5. This result can be used to verify some algebraic identities such as (1) and (2). For example, if $\Gamma \vdash e : \tau$ and x is an identifier not occurring free in e, then it is straightforward to prove (by induction on the structure of e) that

$$\mathcal{E}[\![\Gamma \vdash e : \tau]\!](\rho) = \mathcal{E}[\![\Gamma, x \mapsto \tau' \vdash e : \tau]\!](\rho[x \mapsto d])$$

for any $\rho \in [\![\Gamma]\!]$, type τ' and $d \in [\![\tau']\!]$. Hence for any $\rho \in [\![\Gamma]\!]$ and $\sigma \in [\![\tau]\!]^\perp$

$$\begin{array}{lll} & \mathcal{E}[\![\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{fresh}\ \texttt{in}\ e : \tau]\!]\rho\sigma & \\ = & \mathcal{E}[\![\Gamma \vdash \texttt{fresh} : \texttt{name}]\!]\rho(\underline{\lambda} a \in [\![\texttt{name}]\!]. & \\ & \quad \mathcal{E}[\![\Gamma, x : \texttt{name} \vdash e : \tau]\!](\rho[x \mapsto a])\sigma) & \text{by definition of } \mathcal{E}[\![-]\!] \\ = & \mathcal{E}[\![\Gamma, x : \texttt{name} \vdash e : \tau]\!](\rho[x \mapsto a])\sigma & \text{for some } a \in \mathbb{A} - \text{supp}(e, \rho, \sigma) \\ = & \mathcal{E}[\![\Gamma \vdash e : \tau]\!]\rho\,\sigma & \text{from above.} \end{array}$$

Thus by Corollary 4.4, $e \approx_{\text{ctx}} \texttt{let}\ x\ \texttt{=}\ \texttt{fresh}\ \texttt{in}\ e$ holds when x is an identifier not occurring free in e. The identity (2) is similarly straightforward to verify.

Although equality of denotation implies contextual equivalence, we do not believe that the converse is always true. In other words the denotational semantics is not “fully abstract”, not only for the usual reasons concerning sequentiality [14], but also because of the subtle examples of contextual equivalence that hold when dynamically allocated names are combined with higher order functions: see [12,13]. We do not settle this question here, because to do so would require the development of more subtle techniques for calculating with our continuation-based denotational semantics. Instead we concentrate on using the denotational semantics as a tool for establishing extensionality and correctness properties of Mini-FreshML contextual equivalence. We now have all the tools needed to prove these properties.

Corollary 4.6 (*Extensionality*).

(i) *For unit values*: $\vdash v \approx_{\mathrm{ctx}} v' : \mathtt{unit}$ *iff* $v = v' = \mathtt{()}$.

(ii) *For name values*: $\vdash a \approx_{\mathrm{ctx}} a' : \mathtt{name}$ *iff* $a = a' \in \mathbb{A}$.

(iii) *For data values*: $\vdash \mathtt{C}_k\mathtt{(}v\mathtt{)} \approx_{\mathrm{ctx}} \mathtt{C}_k\mathtt{(}v'\mathtt{)} : \delta$ *iff* $\vdash v \approx_{\mathrm{ctx}} v' : \sigma_k$.

(iv) *For pair values*: $\vdash \mathtt{(}v_1, v_2\mathtt{)} \approx_{\mathrm{ctx}} \mathtt{(}v_1', v_2'\mathtt{)} : \tau_1 \times \tau_2$ *iff* $\vdash v_1 \approx_{\mathrm{ctx}} v_1' : \tau_1$ *and* $\vdash v_2 \approx_{\mathrm{ctx}} v_2' : \tau_2$.

(v) *For name-abstraction values*: $\vdash \mathtt{<<}a\mathtt{>>}v \approx_{\mathrm{ctx}} \mathtt{<<}a'\mathtt{>>}v' : \mathtt{<<name>>}\tau$ *iff* $\vdash (a\ a'') \cdot v \approx_{\mathrm{ctx}} (a'\ a'') \cdot v' : \tau$ *for some* (*or indeed, for every*) $a'' \in \mathbb{A} - \mathit{supp}(a, v, a', v')$.

(vi) *For function values*: $\vdash f \approx_{\mathrm{ctx}} f' : \tau \to \tau'$ *iff for all closed* v *of type* τ, $\vdash f\ v \approx_{\mathrm{ctx}} f'\ v : \tau'$.

Proof. First note that by Theorem 4.3, it suffices to prove these extensionality properties hold with respect to $\approx_{\mathrm{ciu}}$. In each case, the left-to-right implications can be proved directly from the definition of CIU-equivalence. Using this fact, together with properties (11)–(16) of the logical relation for values, one can show by induction on the structure of values that the relation

$$\Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau \overset{\text{def}}{\Leftrightarrow} \Gamma \vdash v, v' : \tau \ \wedge\ \forall \rho \lhd_{\Gamma} \psi . \mathcal{V}[\![\Gamma \vdash v : \tau]\!](\rho) \lhd_{\tau}^{\mathrm{val}} v'[\psi]$$

is closed under composition with $\leqslant_{\mathrm{ciu}}$ on the right. It follows from this and the reflexivity of $\leqslant_{\mathrm{v}}$ (Lemma 3.10) that

$$\Gamma \vdash v \leqslant_{\mathrm{ciu}} v' : \tau \Rightarrow \Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau.$$

Properties (17) and (18) together with Lemma 3.8 ensure that $\leqslant_{\mathrm{v}}$ is contained in $\leqslant_{\mathrm{e}}$; and we know from the proof of Theorem 4.3 that $\leqslant_{\mathrm{e}}$ coincides with $\leqslant_{\mathrm{ciu}}$. Therefore all in all, we have $\Gamma \vdash v \leqslant_{\mathrm{v}} v' : \tau$ holds iff $\Gamma \vdash v \leqslant_{\mathrm{ciu}} v' : \tau$. Using this, each of the right-to-left implications in the extensionality properties then follows from those required of the logical relation in (11)–(16). □

We now turn to the issue of relating object language and metalanguage behaviours as discussed at the end of Section 2, using the example of λ-terms for the object language and the Mini-FreshML datatype δ declared in (5).

Lemma 4.7. *For each λ-term t, define a Mini-FreshML value* $[t]_{\mathrm{v}}$ *by induction on the structure of t as follows*:

$$[x]_{\mathrm{v}} \overset{\text{def}}{=} \mathtt{Var}(x),$$
$$[\lambda x.t]_{\mathrm{v}} \overset{\text{def}}{=} \mathtt{Lam(<<}x\mathtt{>>}[t]_{\mathrm{v}}\mathtt{)},$$
$$[t\ t']_{\mathrm{v}} \overset{\text{def}}{=} \mathtt{App(}[t]_{\mathrm{v}}, [t']_{\mathrm{v}}\mathtt{)}.$$

Then for any λ-terms t, t′ and any value-substitution ψ that maps the free variables of t and t′ to atoms t injectively (*i.e.* $\psi(x) = \psi(x') \Rightarrow x = x'$), *we have* $[t]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t']_{\mathrm{v}}[\psi] \Leftrightarrow t \equiv_{\alpha} t'$.

Proof. We make use of the fact [4, Proposition 2.2] that α-equivalence for λ-terms $t \in \Lambda$ can be inductively defined by the following rules:

$$\frac{x \in \mathrm{VId}}{x \equiv_\alpha x} \qquad \frac{\begin{array}{c}(x\ x'') \cdot t \equiv_\alpha (x'\ x'') \cdot t' \\ x'' \in \mathrm{VId} - \mathrm{supp}(x, t, x', t')\end{array}}{\lambda x.t \equiv_\alpha \lambda x'.t'} \qquad \frac{t_1 \equiv_\alpha t_1' \quad t_2 \equiv_\alpha t_2'}{t_1\ t_2 \equiv_\alpha t_1'\ t_2'}.$$

Then the lemma is proved by induction on the size of t, making use of the extensionality properties of Corollary 4.6. □

Now consider translating a λ-term t into an expression $[t]_\mathrm{e}$ as in (6), then applying an injective value-substitution of atoms for free identifiers to get a closed expression $[t]_\mathrm{e}[\psi]$ and finally evaluating it. Bound variables in t get translated into identifiers bound to `fresh`, which give rise to fresh atoms in the result of evaluating $[t]_\mathrm{e}[\psi]$. So we can expect that result to be contextually equivalent to the value $[t]_\mathrm{v}[\psi]$ provided the bound variables of t are distinct from each other and from the free variables—in other words, provided the "Barendregt variable convention" [3, Section 2.1.13] holds for t. It is convenient to formalise that convention via a structurally inductive definition. For disjoint finite subsets $\overline{x}, \overline{x}'$ of VId we define a subset $\Lambda(\overline{x}; \overline{x}') \subseteq \Lambda$ inductively by the following rules.

$$\frac{x \in \overline{x}}{x \in \Lambda(\overline{x}, \emptyset)} \qquad \frac{t \in \Lambda(\{x\} \cup \overline{x}, \overline{x}') \quad x \notin \overline{x}}{\lambda x.t \in \Lambda(\overline{x}, \{x\} \cup \overline{x}')}$$

$$\frac{t \in \Lambda(\overline{x}, \overline{x}_1') \quad t' \in \Lambda(\overline{x}, \overline{x}_2') \quad \overline{x}_1' \cap \overline{x}_2' = \emptyset}{t\ t' \in \Lambda(\overline{x}, \overline{x}_1' \cup \overline{x}_2')}.$$

If $t \in \Lambda(\overline{x}, \overline{x}')$ then: the free variables of t are contained within $\overline{x}$; the occurrences of bound variables of t are mutually distinct and are contained within $\overline{x}'$; the sets of free and bound variables of t are disjoint; and the support of—i.e. the set of all variables within—the term t is contained within $\overline{x} \cup \overline{x}'$. Note that each term $t \in \Lambda$ is α-equivalent to a term in $\Lambda(\overline{x}, \overline{x}')$ for some $\overline{x}, \overline{x}'$. One can show by induction on the derivation from the above rules that if $t \in \Lambda(\overline{x}, \overline{x}')$, then for any injective substitution $\psi : \mathrm{VId} \to \mathbb{A}$ with $\mathrm{dom}(\psi) = \overline{x} \cup \overline{x}'$ it is the case that $\mathcal{E}[\![[t]_\mathrm{e}[\psi]]\!] = \mathcal{E}[\![[t]_\mathrm{v}[\psi]]\!]$. Hence by Corollary 4.4 we have

Lemma 4.8. *For $t \in \Lambda(\overline{x}, \overline{x}')$ and any injective substitution $\psi : VId \to \mathbb{A}$ with $dom(\psi) = \overline{x} \cup \overline{x}'$, it is the case that $\vdash [t]_\mathrm{e}[\psi] \approx_\mathrm{ctx} [t]_\mathrm{v}[\psi] : \delta$.*

We are now in a position to prove the correctness theorem.

Proof of Theorem 2.3. As we observed earlier, one can show by induction over the rules defining α-equivalence of λ-terms (given in the proof of Lemma 4.7) that if $t \equiv_\alpha t'$ then $[t]_\mathrm{e}$ and $[t']_\mathrm{e}$ are the same Mini-FreshML expression (since we identify Mini-FreshML expressions up to α-equivalence of bound value identifiers). So we just have to show that $\{x_0 : \mathtt{name}, \ldots, x_n : \mathtt{name}\} \vdash [t]_\mathrm{e} \approx_\mathrm{ctx} [t']_\mathrm{e} : \delta$ implies $t \equiv_\alpha t'$. By suitably renaming bound variables we can find a finite set $\overline{x}'$ and terms $t_1, t_1' \in \Lambda(\overline{x}, \overline{x}')$ such that $t_1 \equiv_\alpha t$ and $t_1' \equiv_\alpha t'$; and hence $[t_1]_\mathrm{e} = [t]_\mathrm{e}$ and $[t_1']_\mathrm{e} = [t']_\mathrm{e}$. So if $\{x_0 : \mathtt{name}, \ldots, x_n : \mathtt{name}\} \vdash [t]_\mathrm{e} \approx_\mathrm{ctx} [t']_\mathrm{e} : \delta$, then $\{x_0 : \mathtt{name}, \ldots, x_n : \mathtt{name}\} \vdash [t_1]_\mathrm{e} \approx_\mathrm{ctx} [t_1']_\mathrm{e} : \delta$. Then choosing some injective substitution $\psi : \mathrm{VId} \to \mathbb{A}$ with domain $\overline{x} \cup \overline{x}'$, we can apply Lemma 4.8 to conclude that $\vdash [t_1]_{\mathrm{v}}[\psi] \approx_{\mathrm{ctx}} [t_1']_{\mathrm{v}}[\psi] : \delta$. Finally, we apply Lemma 4.7 to obtain $t \equiv_\alpha t_1 \equiv_\alpha t_1' \equiv_\alpha t'$. □

Fix a bijection $\psi : \mathrm{VId} \cong \mathbb{A}$ between the countably infinite sets of value identifiers and of atoms. Lemma 4.7 tells us that the mapping $t \mapsto [t]_{\mathrm{v}}[\psi]$ induces an injective function from α-equivalence classes of λ-terms to contextual equivalence classes of closed values of type δ. In fact this function is a bijection: from the typing rules of Mini-FreshML (see Appendix A) it is not hard to see that every closed value of type δ must be of the form $[t]_{\mathrm{v}}[\psi]$ for some λ-term t. The contextual equivalence classes of *non-value* expressions of type δ are more complicated; but as the final theorem shows, a closed expression of type δ is either divergent or contextually equivalent to the "restriction" of some value. To prove it we need the following property of divergent terms, which is a corollary of Theorems 3.11 and 4.3.

Lemma 4.9 (*Divergent terms*). *For a closed expression e of type δ and the divergent term $\Omega \stackrel{\text{def}}{=} (\texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ f\texttt{(}x\texttt{))()}$,*

$$e \approx_{\mathrm{ctx}} \Omega \iff \forall S.\mathcal{E}[\![e]\!](\mathcal{S}[\![S]\!]) = \bot \Leftrightarrow \forall S.\langle S, e\rangle \not\downarrow .$$

Theorem 4.10 (*Form of expressions*). *For a closed Mini-FreshML expression e of the type δ declared in* (5), *either $e \approx_{\mathrm{ctx}} \Omega$ or*

$$e \approx_{\mathrm{ctx}} \texttt{let}\ x_1\ \texttt{= fresh in}\ \cdots\ \texttt{let}\ x_n\ \texttt{= fresh in}\ v$$

for some value v of type δ.

Proof. Using Lemma 4.9 we see that if $\vdash e \approx_{\mathrm{ctx}} \Omega$ does *not* hold, then $\langle [], e\rangle\downarrow$. We can now apply the forwards direction of Fact 2.1 to deduce that there exists some closed value v' of type δ and some finite set of atoms $\overline{a}$ such that $\emptyset, e \Downarrow \overline{a}, v'$ with $\mathrm{supp}(v') \subseteq \overline{a}$. Pick a bijection $\psi : \overline{x} \cong \overline{a}$, where $\overline{x} = \{x_1, \ldots, x_n\}$ is a set of value identifiers, and replace each occurrence of an atom $a \in \overline{a}$ in v' with $\psi^{-1}(a)$ to obtain a (possibly open) value v. Thus $v' = v[\psi]$ and it is not hard to see that $e \approx_{\mathrm{ciu}} \texttt{let}\ x_1\ \texttt{= fresh in}\ \cdots\ \texttt{let}\ x_n\ \texttt{= fresh in}\ v$. Now apply Theorem 4.3. □

Remark 4.11 (*Representing* $\equiv_\alpha$). In Remark 2.4 we mentioned that $\equiv_\alpha$ can be represented in Mini-FreshML, in a certain sense, by the function expression $\texttt{aeq} : (\delta \times \delta) \to \texttt{bool}$ described there. We can now make the nature of the representation precise. One can prove by induction on the structure of λ-terms t and t' for any injective substitution $\psi : \mathrm{VId} \to \mathbb{A}$ whose domain contains the free variables of t, t' and whose image is the finite set of atoms $\overline{a}$ say, that

$$t \equiv_\alpha t' \Rightarrow \exists\, \overline{a}' \supseteq \overline{a}.(\overline{a}, \texttt{aeq}([t]_{\mathrm{v}}[\psi], [t']_{\mathrm{v}}[\psi]) \Downarrow \texttt{True()}, \overline{a}'),$$

$$t \not\equiv_\alpha t' \Rightarrow \exists\, \overline{a}' \supseteq \overline{a}.(\overline{a}, \texttt{aeq}([t]_{\mathrm{v}}[\psi], [t']_{\mathrm{v}}[\psi]) \Downarrow \texttt{False()}, \overline{a}').$$

It follows from Theorem 4.3 and Lemma 4.8 that

$$t \equiv_\alpha t' \Rightarrow \texttt{aeq}([t]_e[\psi], [t']_e[\psi]) \approx_{ctx} \texttt{True()} : \texttt{bool}$$
$$t \not\equiv_\alpha t' \Rightarrow \texttt{aeq}([t]_e[\psi], [t']_e[\psi]) \approx_{ctx} \texttt{False()} : \texttt{bool}.$$

5. Conclusion

In this paper we have begun to develop domain theory in the world of FM-sets. Rather than change foundation and work in FM-set theory, we took a concrete approach and developed FM-cppos as ordinary sets equipped with extra structure. Really the only change from classical domain theory is that one must restrict to "finitely supported" functions and subsets. What one gains is new constructs for fresh names and name-binding that can be combined with familiar domain-theoretic constructs for modelling recursion both at the level of terms and of types, to give the kind of refined semantics of fresh names and binders previously associated with more complicated (we would claim) functor category techniques. We applied the new approach, using a continuation monad with a very simple domain of "results" ($1_\perp$) to prove properties of FreshML. Variations on this theme seem very promising; for example, replacing $1_\perp$ by $S \multimap 1_\perp$ 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. Finally we should mention that game semantics can also make good use of FM-sets to achieve new full abstraction results: see [1].

Appendix A. Typing relation

The Mini-FreshML typing relation for expressions, $\Gamma \vdash e : \tau$, is inductively defined by the following axioms and rules.

$$\frac{}{\Gamma \vdash x : \tau}\ (x \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(x) = \tau) \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash a : \texttt{name}}\ (a \in \mathbb{A}) \qquad \frac{\Gamma \vdash e : \sigma_k}{\Gamma \vdash \texttt{C}_k\texttt{(}e\texttt{)} : \delta}\ (\delta\ \texttt{=}\ \texttt{C}_1\ \texttt{of}\ \sigma_1 \texttt{|} \cdots \texttt{|} \texttt{C}_n\ \texttt{of}\ \sigma_n)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \texttt{(}e, e'\texttt{)} : \tau \times \tau'} \qquad \frac{}{\Gamma \vdash \texttt{fresh} : \texttt{name}}$$

$$\frac{\Gamma \vdash e : \texttt{name} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \texttt{<<}e\texttt{>>}e' : \texttt{<<name>>}\tau} \qquad \frac{\Gamma \vdash e : \texttt{name} \quad \Gamma \vdash e' : \texttt{name} \quad \Gamma \vdash e'' : \tau}{\Gamma \vdash \texttt{swap}\ e, e'\ \texttt{in}\ e'' : \tau}$$

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau'}{\Gamma \vdash \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma, x : \tau' \vdash e' : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau}$$

$$\frac{\Gamma \vdash e : \tau' \times \tau'' \qquad \Gamma, x : \tau', x' : \tau'' \vdash e' : \tau}{\Gamma \vdash \mathtt{let}\ (x, x')\ \mathtt{=}\ e\ \mathtt{in}\ e' : \tau}$$

$$\frac{\Gamma \vdash e : \mathtt{<<name>>}\tau' \qquad \Gamma, x : \mathtt{name}, x' : \tau' \vdash e' : \tau}{\Gamma \vdash \mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ e\ \mathtt{in}\ e' : \tau}$$

$$\frac{\Gamma \vdash e : \mathtt{name} \qquad \Gamma \vdash e' : \mathtt{name} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{=}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \delta \\ \forall k \in \{1, \ldots, n\}.\ \Gamma, x : \sigma_k \vdash e_k : \tau\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ (\mathtt{C}_1(x_1)\ \mathtt{->}\ e_1 \mathtt{|} \cdots \\ \qquad\qquad\qquad \mathtt{|}\ \mathtt{C}_n(x_n)\ \mathtt{->}\ e_n)\ : \tau\end{array}} \ (\delta\ \mathtt{=}\ \mathtt{C}_1\ \mathtt{of}\ \sigma_1 \mathtt{|} \cdots \mathtt{|} \mathtt{C}_n\ \mathtt{of}\ \sigma_n)$$

Appendix B. Termination relation

$\langle S, e\rangle\!\downarrow$ is inductively defined by the following axiom and rules, where S ranges over frame stacks, $e, e', \ldots$ over expressions, $v, v', \ldots$ over values, and $a, a', \ldots$ over atoms. The definition is split into two parts for clarity.

Part 1: *$\langle S, v\rangle\!\downarrow$ where v is a value.*

$$\frac{}{\langle [], v\rangle\!\downarrow} \qquad \frac{\langle S, \mathtt{C}_k(v)\rangle\!\downarrow}{\langle S \circ \mathtt{C}_k([-]), v\rangle\!\downarrow} \qquad \frac{\langle S \circ (v, [-]), e\rangle\!\downarrow}{\langle S \circ ([-], e), v\rangle\!\downarrow}$$

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

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

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

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

$$\frac{\langle S, e[v/x]\rangle\!\downarrow}{\langle S \circ \mathtt{let}\ x\ \mathtt{=}\ [-]\ \mathtt{in}\ e, v\rangle\!\downarrow}$$

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

$$\frac{a' \in \mathbb{A} - \mathrm{supp}(S, v, e) \qquad \langle S, e[a'/x, ((a\ a') \cdot v)/x']\rangle\!\downarrow}{\langle S \circ \mathtt{let}\ \mathtt{<<}x\mathtt{>>}x'\ \mathtt{=}\ [-]\ \mathtt{in}\ e, \mathtt{<<}a\mathtt{>>}v\rangle\!\downarrow}$$

$$\frac{\langle S \circ \mathtt{if}\ a\ \mathtt{=}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, e'\rangle\!\downarrow}{\langle S \circ \mathtt{if}\ [-]\ \mathtt{=}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, a\rangle\!\downarrow}$$

$$\frac{\langle S, e_1\rangle\!\downarrow}{\langle S \circ \mathtt{if}\ a\ \mathtt{=}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, a'\rangle\!\downarrow}\ \text{if } a = a'$$

$$\frac{\langle S, e_2\rangle{\downarrow}}{\langle S \circ \texttt{if } a \texttt{ = [-] then } e_1 \texttt{ else } e_2, a'\rangle{\downarrow}} \text{ if } a \neq a'$$

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

Part 2: $\langle S, e\rangle{\downarrow}$ *where e is non-value expression.*

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

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

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

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

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

$$\frac{\langle S \circ \texttt{if [-] = } e' \texttt{ then } e_1 \texttt{ else } e_2, e\rangle{\downarrow}}{\langle S, \texttt{if } e \texttt{ = } e' \texttt{ then } e_1 \texttt{ else } e_2\rangle{\downarrow}}$$

$$\frac{\langle S \circ \texttt{match [-] with C}_1\texttt{(}x_1\texttt{) -> } e_1 \texttt{|} \cdots \texttt{|C}_n\texttt{(}x_n\texttt{) -> } e_n, e\rangle{\downarrow}}{\langle S, \texttt{match } e \texttt{ with C}_1\texttt{(}x_1\texttt{) -> } e_1 \texttt{|} \cdots \texttt{|C}_n\texttt{(}x_n\texttt{) -> } e_n\ \rangle{\downarrow}}.$$

Appendix C. Denotation of expressions

Notation: In this and the following appendices, write $\underline{\lambda} x.t$ for the strict function that maps non-bottom elements x to t. Extend this notation in the obvious way to write $\underline{\lambda}\langle d_1, d_2\rangle.t$ for strict functions $D_1 \otimes D_2 \multimap D$ and $\underline{\lambda}\,[a]\,d.t$ for strict functions $[\mathbb{A}]D \multimap D'$. (Note that this notation imposes no conditions as to which particular representative in $[\mathbb{A}]D$ is chosen: the semantics below makes this explicit.) We also write $\langle d_1, d_2\rangle$ to indicate a smash pair (such that $\langle d_1, d_2\rangle \overset{\text{def}}{=} \bot_{D_1 \otimes D_2}$ when either of $d_1 \in D_1$ and $d_2 \in D_2$ are bottom). The notation *if* $a = a'$ *then* d *else* d' means d if a and a' are equal and d' otherwise.

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

- $\mathcal{E}[\![\Gamma \vdash x : \tau]\!]\rho \overset{\text{def}}{=} \lambda\sigma \in [\![\Gamma(x)]\!]^{\bot}.\sigma(\rho(x))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{()} : \texttt{unit}]\!]\rho \overset{\text{def}}{=} \lambda\sigma \in [\![\texttt{unit}]\!]^{\bot}.\sigma(\top)$
- $\mathcal{E}[\![\Gamma \vdash a : \texttt{name}]\!]\rho \overset{\text{def}}{=} \lambda\sigma \in [\![\texttt{name}]\!]^{\bot}.\sigma(a)$
- $\mathcal{E}[\![\Gamma \vdash \texttt{C}_k\texttt{(}e\texttt{)} : \delta]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\delta]\!]^{\bot}.\mathcal{E}[\![\Gamma \vdash e : \sigma_k]\!]\rho(\lambda d \in [\![\sigma_k]\!].\sigma((i \circ \mathrm{in}_k)d))$

- $\mathcal{E}[\![\Gamma \vdash \texttt{(}e, e'\texttt{)} : \tau \times \tau']\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau \times \tau']\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \tau]\!]\rho(\lambda d \in [\![\tau]\!].$
  $\mathcal{E}[\![\Gamma \vdash e' : \tau']\!]\rho(\lambda d' \in [\![\tau']\!].\sigma\langle d, d'\rangle))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{fresh} : \texttt{name}]\!]\rho \overset{\text{def}}{=} \textbf{new} \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\texttt{name}]\!]^{\perp}.\sigma(a)$ (any $a \in \mathbb{A} - \text{supp}(\sigma)$)
- $\mathcal{E}[\![\Gamma \vdash \texttt{<<}e\texttt{>>}e' : \texttt{<<name>>}\tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\texttt{<<name>>}\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \texttt{name}]\!]\rho(\underline{\lambda} a \in [\![\texttt{name}]\!].$
  $\mathcal{E}[\![\Gamma \vdash e' : \tau]\!]\rho(\lambda d \in [\![\tau]\!].\sigma([a]\,d)))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{swap}\ e, e'\ \texttt{in}\ e'' : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \texttt{name}]\!]\rho(\underline{\lambda} a \in [\![\texttt{name}]\!].$
  $\mathcal{E}[\![\Gamma \vdash e' : \texttt{name}]\!]\rho(\underline{\lambda} a' \in [\![\texttt{name}]\!].\mathcal{E}[\![\Gamma \vdash e'' : \tau]\!]\rho(\lambda d \in [\![\tau]\!].$
  $\sigma((a\ a') \cdot d))))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{fun}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e : \tau \to \tau']\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau \to \tau']\!]^{\perp}.\sigma(\textbf{fix}(\underline{\lambda} d \in [\![\tau \to \tau']\!].\underline{\lambda} d' \in [\![\tau]\!].$
  $\mathcal{E}[\![\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau']\!](\rho[f \mapsto d, x \mapsto d'])))$
- $\mathcal{E}[\![\Gamma \vdash e\ e' : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \tau \to \tau']\!]\rho(\lambda d \in [\![\tau \to \tau']\!].$
  $\mathcal{E}[\![\Gamma \vdash e' : \tau]\!]\rho(\lambda d' \in [\![\tau]\!].d\ d'\ \sigma))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \tau']\!]\rho(\underline{\lambda} d' \in [\![\tau']\!].$
  $\mathcal{E}[\![\Gamma, x : \tau' \vdash e' : \tau]\!](\rho[x \mapsto d'])\sigma).$
- $\mathcal{E}[\![\Gamma \vdash \texttt{let (}x, x'\texttt{)}\ \texttt{=}\ e\ \texttt{in}\ e' : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\rho(\underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau_1 \times \tau_2]\!].$
  $\mathcal{E}[\![\Gamma, x : \tau_1, x' : \tau_2 \vdash e' : \tau]\!](\rho[x \mapsto d_1, x' \mapsto d_2])\sigma)$
- $\mathcal{E}[\![\Gamma \vdash \texttt{let <<}x\texttt{>>}x'\ \texttt{=}\ e\ \texttt{in}\ e' : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \texttt{<<name>>}\tau']\!]\rho(\underline{\lambda}\,[a]\,d' \in [\![\texttt{<<name>>}\tau']\!].$
  $\mathcal{E}[\![\Gamma, x : \texttt{name}, x' : \tau' \vdash e' : \tau]\!](\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d'])\sigma)$
  (any $a' \in \mathbb{A} - \text{supp}(e, e', \rho, \sigma, a, d')$)
- $\mathcal{E}[\![\Gamma \vdash \texttt{if}\ e\ \texttt{=}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.[\![\Gamma \vdash e : \texttt{name}]\!]\rho(\underline{\lambda} a \in [\![\texttt{name}]\!].[\![\Gamma \vdash e' : \texttt{name}]\!]\rho(\underline{\lambda} a' \in [\![\texttt{name}]\!].$
  if $a = a'$ *then* $\mathcal{E}[\![\Gamma \vdash e_1 : \tau]\!]\rho\,\sigma$ *else* $\mathcal{E}[\![\Gamma \vdash e_2 : \tau]\!]\rho\,\sigma))$
- $\mathcal{E}[\![\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \cdots \texttt{|C}_k\texttt{(}x_k\texttt{)}\ \texttt{->}\ e_k \texttt{|} \cdots : \tau]\!]\rho \overset{\text{def}}{=}$
  $\lambda\sigma \in [\![\tau]\!]^{\perp}.\mathcal{E}[\![\Gamma \vdash e : \delta]\!]\rho(\underline{\lambda} d' \in [\![\delta]\!].\mathcal{E}[\![\Gamma, x_k : \sigma_k \vdash e_k : \tau]\!](\rho[x_k \mapsto d_k])\sigma)$
  (for the unique k and d_k such that $d' = (i \circ \text{in}_k)d_k$).

Appendix D. Denotation of values (expressions in canonical form)

The function $\mathcal{V}[\![\Gamma \vdash v : \tau]\!] \in [\![\Gamma]\!] \multimap [\![\tau]\!]$ maps $\perp$ to itself and for non-bottom arguments ρ is defined by induction on the structure of the canonical form v as given below.

- $\mathcal{V}\llbracket \Gamma \vdash x : \tau \rrbracket \rho \overset{\text{def}}{=} \rho(x)$
- $\mathcal{V}\llbracket \Gamma \vdash \mathtt{()} : \mathtt{unit} \rrbracket \rho \overset{\text{def}}{=} \top$
- $\mathcal{V}\llbracket \Gamma \vdash a : \mathtt{name} \rrbracket \rho \overset{\text{def}}{=} a$
- $\mathcal{V}\llbracket \Gamma \vdash \mathtt{C}_k\mathtt{(}v\mathtt{)} : \delta \rrbracket \rho \overset{\text{def}}{=} (i \circ \mathrm{in}_k)(\mathcal{V}\llbracket \Gamma \vdash v : \sigma_k \rrbracket \rho)$
- $\mathcal{V}\llbracket \Gamma \vdash \mathtt{(}v, v'\mathtt{)} : \tau \times \tau' \rrbracket \rho \overset{\text{def}}{=} \langle \mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket \rho, \mathcal{V}\llbracket \Gamma \vdash v' : \tau' \rrbracket \rho \rangle$
- $\mathcal{V}\llbracket \Gamma \vdash \mathtt{<<}a\mathtt{>>}v : \mathtt{<<name>>}\tau \rrbracket \rho \overset{\text{def}}{=} [a]\,(\mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket \rho)$
- $\mathcal{V}\llbracket \Gamma \vdash \mathtt{fun}\ f\mathtt{(}x\mathtt{)}\ \mathtt{=}\ e : \tau \to \tau' \rrbracket \rho \overset{\text{def}}{=}$
  $\mathbf{fix}(\underline{\lambda} d \in \llbracket \tau \to \tau' \rrbracket . \underline{\lambda} d' \in \llbracket \tau \rrbracket .$
  $\quad \mathcal{E}\llbracket \Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau' \rrbracket (\rho[f \mapsto d, x \mapsto d'])).$

Appendix E. Denotation of frame stacks

The function $\mathcal{S}\llbracket \Gamma \vdash S : \tau \multimap \_\_ \rrbracket \in \llbracket \Gamma \rrbracket \multimap \llbracket \tau \rrbracket^{\perp}$ maps $\perp$ to itself and for non-bottom arguments ρ is defined by induction on the structure of S as follows. (The notation *let* $a = d$ *in* $d'[a]$ means $d'[a]$ if $d \in \mathbb{A}_{\perp}$ is the non-bottom element given by $a \in \mathbb{A}$ and $\perp$ otherwise.)

- $\mathcal{S}\llbracket \Gamma \vdash [] : \tau \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=} \underline{\lambda} x \in \llbracket \tau \rrbracket . \top$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{C}_k\mathtt{(}[-]\mathtt{)} : \sigma_k \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=} \lambda v \in \llbracket \sigma_k \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \delta \multimap \_\_ \rrbracket \rho ((i \circ \mathrm{in}_k) v)$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{(}[-], e\mathtt{)} : \tau \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\lambda d \in \llbracket \tau \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e : \tau' \rrbracket \rho (\lambda d' \in \llbracket \tau' \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \tau \times \tau' \rrbracket \rho \langle d, d' \rangle)$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{(}v, [-]\mathtt{)} : \tau' \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\lambda d \in \llbracket \tau' \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \tau \times \tau' \rrbracket \rho \langle \mathcal{V}\llbracket \Gamma \vdash v : \tau \rrbracket \rho, d \rangle$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{<<}[-]\mathtt{>>}e : \mathtt{name} \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\underline{\lambda} a \in \llbracket \mathtt{name} \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e : \tau \rrbracket \rho (\lambda d \in \llbracket \tau \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \mathtt{<<name>>}\tau \rrbracket \rho ([a]\,d))$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{<<}v\mathtt{>>}[-] : \tau \multimap \_\_ \rrbracket (\rho) \overset{\text{def}}{=}$
  $\lambda d \in \llbracket \tau \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \mathtt{<<name>>}\tau \rrbracket \rho (\big[\mathcal{V}\llbracket \Gamma \vdash v : \mathtt{name} \rrbracket \rho\big]\, d)$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{swap}\ [-], e'\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\underline{\lambda} a \in \llbracket \mathtt{name} \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e' : \mathtt{name} \rrbracket \rho (\underline{\lambda} a' \in \llbracket \mathtt{name} \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e'' : \tau \rrbracket \rho (\lambda d \in \llbracket \tau \rrbracket .$
  $\quad \mathcal{S}\llbracket \Gamma \vdash S : \tau \multimap \_\_ \rrbracket \rho ((a\ a') \cdot d)))$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{swap}\ v, [-]\ \mathtt{in}\ e'' : \mathtt{name} \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  let $a = \mathcal{V}\llbracket \Gamma \vdash v : \mathtt{name} \rrbracket \rho$ *in* $\underline{\lambda} a' \in \llbracket \mathtt{name} \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e'' : \tau \rrbracket \rho (\lambda d \in \llbracket \tau \rrbracket .$
  $\quad \mathcal{S}\llbracket \Gamma \vdash S : \tau \multimap \_\_ \rrbracket \rho ((a\ a') \cdot d))$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ \mathtt{swap}\ v, v'\ \mathtt{in}\ [-] : \tau \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  let $a = \mathcal{V}\llbracket \Gamma \vdash v : \mathtt{name} \rrbracket \rho$ *in let* $a' = \mathcal{V}\llbracket \Gamma \vdash v' : \mathtt{name} \rrbracket \rho$ *in*
  $\quad \lambda d \in \llbracket \tau \rrbracket . \mathcal{S}\llbracket \Gamma \vdash S : \tau \multimap \_\_ \rrbracket \rho ((a\ a') \cdot d)$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ [-]\ e : (\tau \to \tau') \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\lambda d \in \llbracket \tau \to \tau' \rrbracket . \mathcal{E}\llbracket \Gamma \vdash e : \tau \rrbracket \rho (\lambda d' \in \llbracket \tau \rrbracket . d\ d' (\mathcal{S}\llbracket \Gamma \vdash S : \tau' \multimap \_\_ \rrbracket \rho))$
- $\mathcal{S}\llbracket \Gamma \vdash S \circ v\ [-] : \tau \multimap \_\_ \rrbracket \rho \overset{\text{def}}{=}$
  $\lambda d \in \llbracket \tau \rrbracket . (\mathcal{V}\llbracket \Gamma \vdash v : \tau \to \tau' \rrbracket \rho\ d)(\mathcal{S}\llbracket \Gamma \vdash S : \tau' \multimap \_\_ \rrbracket \rho)$

- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{let } x \texttt{ = [-] in } e : \tau \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda} d \in [\![\tau]\!].\mathcal{E}[\![\Gamma, x : \tau \vdash e : \tau']\!](\rho[x \mapsto d])(\mathcal{S}[\![\Gamma \vdash S : \tau' \multimap \_\_]\!]\rho)$
- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{let (}x, x'\texttt{) = [-] in } e : \tau \times \tau' \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda}\langle d_1, d_2\rangle \in [\![\tau \times \tau']\!].$
  $\mathcal{E}[\![\Gamma, x : \tau, x' : \tau' \vdash e : \tau'']\!](\rho[x \mapsto d_1, x' \mapsto d_2])(\mathcal{S}[\![\Gamma \vdash S : \tau'' \multimap \_\_]\!]\rho)$
- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{let <<}x\texttt{>>}x' \texttt{ = [-] in } e : \texttt{<<name>>}\tau \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda}\,[a]\,d \in [\![\texttt{<<name>>}\tau]\!].\mathcal{E}[\![\Gamma, x : \texttt{name}, x' : \tau \vdash e : \tau']\!]$
  $(\rho[x \mapsto a', x' \mapsto (a\ a') \cdot d])(\mathcal{S}[\![\Gamma \vdash S : \tau' \multimap \_\_]\!]\rho)$
  (any $a' \in \mathbb{A} - \mathrm{supp}(S, e, \rho, a, d)$)
- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{if [-] = } e' \texttt{ then } e_1 \texttt{ else } e_2 : \tau \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda} a \in [\![\texttt{name}]\!].\mathcal{E}[\![\Gamma \vdash e' : \texttt{name}]\!]\rho(\underline{\lambda} a' \in [\![\texttt{name}]\!].$
  $\textit{if } a = a' \textit{ then } \mathcal{E}[\![\Gamma \vdash e_1 : \tau]\!]\rho(\mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!]\rho)$
  $\textit{else } \mathcal{E}[\![\Gamma \vdash e_2 : \tau]\!]\rho(\mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!]\rho))$
- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{if } v \texttt{ = [-] then } e_1 \texttt{ else } e_2 : \tau \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda} a' \in [\![\texttt{name}]\!].\textit{if } \mathcal{V}[\![\Gamma \vdash v : \texttt{name}]\!](\rho) = a'$
  $\textit{then } \mathcal{E}[\![\Gamma \vdash e_1 : \tau]\!]\rho(\mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!]\rho)$
  $\textit{else } \mathcal{E}[\![\Gamma \vdash e_2 : \tau]\!]\rho(\mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!]\rho)$
- $\mathcal{S}[\![\Gamma \vdash S \circ \texttt{match [-] with } \cdots \texttt{|C}_k\texttt{(}x_k\texttt{) -> } e_k \texttt{|} \cdots : \delta \multimap \_\_]\!]\rho \overset{\text{def}}{=}$
  $\underline{\lambda} d \in [\![\delta]\!].\mathcal{E}[\![\Gamma, x_k : \sigma_k \vdash e_k : \tau]\!](\rho[x_k \mapsto d_k])(\mathcal{S}[\![\Gamma \vdash S : \tau \multimap \_\_]\!]\rho)$
  (for the unique k and d_k such that $d = (i \circ \mathrm{in}_k)d_k$).

References

[1] S. Abramsky, D.R. Ghica, A.S. Murowski, C.-H.L. Ong, I.D.B. Stark, Nominal games and full abstraction for the nu-calculus, in: 19th Annu. Symp. Logic in Computer Science, IEEE Computer Society Press, Washington, 2004.

[2] S. Abramsky, A. Jung, Domain theory, in: Handbook of Logic in Computer Science, vol. 3, Clarendon Press, 1994, pp. 1–168.

[3] H.P. Barendregt, The Lambda Calculus: Its Syntax and Semantics, revised ed., North-Holland, Amsterdam, 1984.

[4] M.J. Gabbay, A.M. Pitts, A new approach to abstract syntax with variable binding, Formal Aspects of Comput. 13 (2002) 341–363.

[5] I.A. Mason, C.L. Talcott, Equivalence in functional languages with effects, J. Function. Program. 1 (3) (1991) 287–327.

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

[7] E. Moggi, Notions of computation and monads, Inform. Comput. 93 (1) (1991) 55–92.

[8] S.L. Peyton Jones, Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, in: C.A.R. Hoare, M. Broy, R. Steinbruggen (Eds.), Engineering Theories of Software Construction, IOS Press, 2001, pp. 47–96.

[9] A.M. Pitts, Relational properties of domains, Inform. Comput. 127 (1996) 66–90.

[10] A.M. Pitts, Operational semantics and program equivalence, in: Applied Semantics, Advanced Lectures, Lecture Notes in Computer Science Tutorial, vol. 2395, Springer, Berlin, 2002, pp. 378–412.

[11] A.M. Pitts, Nominal logic a first order theory of names and binding, Inform. Comput. 186 (2003) 165–193.

[12] A.M. Pitts, I.D.B. Stark, Observable properties of higher order functions that dynamically create local names, or: What's new? in: Mathematical Foundations of Computer Science, Proc. 18th Internat. Symp., Gdańsk, 1993, Lecture Notes in Computer Science, vol. 711, Springer, Berlin, 1993, pp. 122–141.

[13] A.M. Pitts, I.D.B. Stark, Operational reasoning for functions with local state, in: A.D. Gordon, A.M. Pitts (Eds.), Higher Order Operational Techniques in Semantics, Cambridge University Press, Cambridge, 1998, pp. 227–273.
[14] G.D. Plotkin, LCF considered as a programming language, Theor. Comput. Sci. 5 (1977) 223–255.
[15] M.R. Shinwell, Swapping the atom: programming with binders in Fresh O'Caml, Proc. MERλIN, 2003.
[16] M.R. Shinwell, The Fresh Approach: Functional Programming with Names and Binders, Ph.D. Thesis, University of Cambridge Computer Laboratory, 2005, in preparation.
[17] M.R. Shinwell, A.M. Pitts, Fresh O'Caml User Manual, Cambridge University Computer Laboratory, September. Available at ⟨`http://www.freshml.org/foc/`⟩.
[18] M.R. Shinwell, A.M. Pitts, M.J. Gabbay, FreshML: programming with binders made simple, in: Proc. ICFP '03, ACM Press, 2003, pp. 263–274.
[19] I.D.B. Stark, Categorical models for local names, Lisp Symbol. Comput. 9 (1) (1996) 77–107.
[20] C. Urban, A.M. Pitts, M.J. Gabbay, Nominal unification, in: Proc. CSL'03 & KGC, Lecture Notes in Computer Science, vol. 2803, Springer, Berlin, 2003, pp. 513–527.
[21] P. Wadler, Comprehending monads, Math. Struct. Computer Sci. 2 (1992) 461–493.
[22] A.K. Wright, M. Felleisen, A syntactic approach to type soundness, Inform. Comput. 115 (1994) 38–94.