Generative Unbinding of Names

Andrew M. Pitts *
University of Cambridge Computer Laboratory
Andrew.Pitts@cl.cam.ac.uk

Mark R. Shinwell
CodeSourcery, Ltd
shinwell@codesourcery.com

Abstract

This paper is concerned with a programming language construct for typed name binding that enforces α-equivalence. It proves a new result about what operations on names can co-exist with this construct. The particular form of typed name binding studied is that used by the FreshML family of languages. Its characteristic feature is that a name binding is represented by an abstract (name,value)-pair that may only be deconstructed via the generation of fresh bound names. In FreshML the only observation one can make of names is to test whether or not they are equal. This restricted amount of observation was thought necessary to ensure that there is no observable difference between α-equivalent name binders. Yet from an algorithmic point of view it would be desirable to allow other operations and relations on names, such as a total ordering. This paper shows that, contrary to expectations, one may add not just ordering, but almost any relation or numerical function on names without disturbing the fundamental correctness result about this form of typed name binding (that object-level α-equivalence precisely corresponds to contextual equivalence at the programming meta-level), so long as one takes the state of dynamically created names into account.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms Languages, Theory

Keywords Abstract syntax, binders, alpha-conversion, meta-programming

1. Introduction

FreshML and the language systems that it has inspired provide some user friendly facilities within the context of strongly typed functional programming for computing with syntactical data structures involving names and name binding. The underlying theory was presented in [20, 26] and has been realised in the Fresh patch of Objective Caml [24]. FreshML has also inspired Pottier's Cαml tool [21] for Objective Caml and Cheney's FreshLib library [3] for Haskell. The approach taken to binding in all these works is "nominal" in that the user is given access to the names of bound entities and can write syntax manipulating programs that follow the informal practice of referring to α-equivalence classes of terms via representatives. However, in FreshML the means of access to bound names is carefully controlled by the type system. It has been shown [23, 25] that its static and dynamic properties combine to guarantee a certain "correctness of representation" property: data structures representing α-equivalent syntactical terms (that is, ones differing only in the names of bound entities) always behave the same in any program. So even though programs can name names, as it were, α-equivalence of name bindings is taken care of automatically by the programming language design.

* Research supported by UK EPSRC grant EP/D000459/1

POPL'07 January 17–19, 2007, Nice, France.


```
type  atm
type  α bnd
val   fresh : unit → atm
val   bind : atm * α → α bnd
val   unbind : α bnd → atm * α
val   (=) : atm → atm → bool
```

Figure 1. A signature for name binding.

Of course such a correctness of representation property depends rather delicately upon which operations on bound names are allowed. At the heart of this approach to binding is an operation that we call *generative unbinding*. To explain what it involves, consider a simplified version of Fresh Objective Caml with a single type atm of bindable names and a parametric family of types α bnd classifying abstractions of single names over values of type α. To explain: both atm and α bnd are abstract types that come with the signature of operations shown in Figure 1. The closed values of type atm are drawn from a countably infinite set $\mathbb{A}$ of symbols that we call *atoms*. Programs only get access to atoms by evaluating the expression fresh() to get a fresh one; and hence program execution depends upon a state recording the atoms that have been created so far. Given a type τ, closed values of type τ bnd are called *atom bindings* and are given by pairs $\langle\!\langle a \rangle\!\rangle v$ consisting of an atom a : atm and a closed value $v : \tau$. Atom bindings are constructed by evaluating bind$(a\,, v)$. Fresh Objective Caml provides a very convenient form of generative pattern matching for deconstructing atom bindings. To keep things simple, here we will consider an equivalent mechanism for deconstructing atom binding via an unbind function carrying out generative unbinding: unbind $\langle\!\langle a \rangle\!\rangle v$ evaluates by first evaluating fresh() to obtain a fresh atom a' and then returning the pair $(a'\,, v\{a'/a\})$, where in general $v\{a'/a\}$ denotes the value obtained from v by renaming all occurrences of a to be a'. The instance of renaming that arises when evaluating unbind $\langle\!\langle a \rangle\!\rangle v$ is special: the fresh atom a' does not occur in v and so $v\{a'/a\}$ is equivalent to the result of applying to v the semantically better behaved operation of *swapping* a and a'. Implementing such an atom

swapping operation on all types of values is the main extension that the Fresh patch makes to Objective Caml. A language extension is not needed if users define atom swapping themselves, on a case-by-case basis; this more limited approach is quite workable in the presence of Haskell-style type classes—see [3].

The type $\alpha\ \mathsf{bnd}$ is used in data type declarations in the argument type of value constructors representing binders. To take a familiar example, the terms of the untyped λ-calculus (all terms, whether open or closed, with variables given by atoms $a \in \mathbb{A}$)

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

can be represented by closed values of the type term given by the declaration

$$\begin{array}{rcl} \mathsf{type\ \ term} & = & \mathsf{V\ of\ atm} \\ & \mid & \mathsf{L\ of\ term\ bnd} \\ & \mid & \mathsf{A\ of\ term * term\ .} \end{array} \tag{1}$$

The value $\ulcorner t \urcorner : \mathsf{term}$ representing a λ-term t is defined by

$$\begin{array}{rcl} \ulcorner a \urcorner & \triangleq & \mathsf{V}\,a \\ \ulcorner \lambda a.t \urcorner & \triangleq & \mathsf{L}\,\langle\!\langle a \rangle\!\rangle \ulcorner t \urcorner \\ \ulcorner t_1\,t_2 \urcorner & \triangleq & \mathsf{A}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \end{array} \tag{2}$$

and satisfies:

> **Correctness of Representation**: *two λ-terms are α-equivalent, $t_1 =_\alpha t_2$, iff $\ulcorner t_1 \urcorner$ and $\ulcorner t_2 \urcorner$ are contextually equivalent closed values of type* term*, i.e. can be used interchangeably in any well-typed Fresh Objective Caml program without affecting the observable results of program execution.*

Since it is also the case that every closed value of type term is of the form $\ulcorner t \urcorner$ for some λ-term t, it follows that there is a bijection between α-equivalence classes of λ-terms and contextual equivalence classes of closed values of type term. The Correctness property is not easy to prove because of the nature of contextual equivalence, with its quantification over all possible program contexts. It was established in [23, 25] using denotational methods that take permutations of atoms into account. The same methods can be used to generalise from the example of λ-terms to terms over any *nominal signature* in the sense of [29].

Contribution of this paper. For the signature in Figure 1, the only operation on atoms apart from bind is a test for equality: $a = a'$ evaluates to true if a and a' are the same atom and to false otherwise. Adding extra operations and relations for atoms may well change which program phrases are contextually equivalent. Is it possible to have some relations or operations on atoms in addition to equality without invalidating the above Correctness property? For example it would be very useful to have a linear order $(<) : \mathsf{atm} \to \mathsf{atm} \to \mathsf{bool}$, so that values of type atm could be used as keys in efficient data structures for finite maps and the like. We show that this is possible, and more. This is a rather unexpected result, for the following reason.

The proof of the Correctness property given in [23, 25] relies upon *equivariant* properties of the semantics, in other words ones whose truth is invariant under permuting atoms. Atom equality is equivariant: since a permutation is in particular bijective, it preserves and reflects the value of $a = a'$. At first it seems that a linear order on atoms cannot be equivariant, since if $a < a'$ is true, then applying the permutation swapping a and a' we get $a' < a$, which is false. However, equivariance is a global property: when considering invariance of the truth of a property under permutations, it is crucial to take into account all the parameters upon which the property depends. Here there is a hidden parameter: *the current state of dynamically created atoms*. So we should permute the atoms in this state as well as the arguments of the relation. We shall see that it is perfectly possible to have a state-dependent equivariant ordering for the type atm without invalidating the Correctness property. Indeed we prove that *one can add any n-ary function from* atm *to numbers* (or to booleans, for that matter) *whose semantics is reasonable,*[1] *without invalidating the Correctness property for any nominal signature.*

We have to work quite hard to get this result, which generalises the one announced in [26] (with a flawed proof sketch) and finally proved in [25, 23]; but whereas those works use denotational techniques, here we use an arguably more direct approach based on the operational semantics of the language. Along the way to the main result (Theorem 14) we prove a Mason-Talcott-style "CIU" [12] characterisation of contextual equivalence for our language (The-

Variables	$f, x \in \mathbb{V}$	countably infinite set (fixed)
Atoms	$a \in \mathbb{A}$	countably infinite set (fixed)
Data types	$\delta \in \mathcal{D}$	finite set (variable)
Constructors	$\mathsf{C} \in \mathcal{C}$	finite set (variable)
Observations	$\mathsf{obs} \in \mathcal{O}$	finite set (variable)
Values	$v \in \mathit{Val}$	::=
	variable	x
	unit	$()$
	pair	$(v\,,v)$
	recursive function	$\mathsf{fun}(f\,x = e)$
	data construction	$\mathsf{C}\,v$
	atom	a
	atom binding	$\langle\!\langle v \rangle\!\rangle v$
Expressions	$e \in \mathit{Exp}$	::=
	value	v
	sequencing	$\mathsf{let}\,x = e\,\mathsf{in}\,e$
	first projection	$\mathsf{fst}\,v$
	second projection	$\mathsf{snd}\,v$
	function application	$v\,v$
	data deconstruction	$\mathsf{match}\,v\,\mathsf{with}\,(\mathsf{C}\,x \to e \mid \cdots)$
	fresh atom	$\mathsf{fresh}()$
	generative unbinding	$\mathsf{unbind}\,v$
	atom observation	$\mathsf{obs}\,v \cdots v$
Frame stacks	$S \in \mathit{Stk}$	::=
	empty	Id
	non-empty	$S \circ (x.e)$
States	$\vec{a} \in \mathit{State}$	$\triangleq$ finite lists of distinct atoms
Machine configurations		$\langle \vec{a}, S, e \rangle$
Types	$\tau \in \mathit{Typ}$	::=
	unit	unit
	pairs	$\tau * \tau$
	functions	$\tau \to \tau$
	data type	δ
	atoms	atm
	atom bindings	$\tau\ \mathsf{bnd}$
Typing environments	$\Gamma \in \mathbb{V} \xrightarrow{\text{fin}} \mathit{Typ}$	
Typing judgements		
	expressions & values	$\Gamma \vdash e : \tau$
	frame stacks	$\Gamma \vdash S : \tau \to \tau'$
Initial basis		
	natural numbers	$\mathsf{nat} \in \mathcal{D}$
	zero	$(\mathsf{Zero} : \mathsf{unit} \to \mathsf{nat}) \in \mathcal{C}$
	successor	$(\mathsf{Succ} : \mathsf{nat} \to \mathsf{nat}) \in \mathcal{C}$
	atom equality	$\mathsf{eq} \in \mathcal{O} \quad (\mathit{arity} = 2)$

Figure 2. Language syntax.

[1] We explain what is reasonable in Section 3.

orem 10). This is proved using Howe's method [10] applied to a formulation of the operational semantics with Felleisen-style evaluation contexts [4], via an abstract machine with frame stacks [16]. We also prove "extensionality" laws for the atom binding type construction $\alpha\,\mathsf{bnd}$ (Propositions 16 and 19). The proof technique underlying our work is rule based induction, but with the novel twist that we exploit semantic properties of freshness of names that are based on the use of name permutations and that were introduced in [6] and developed in [17, 28, 19].

2. Generative Unbinding

We use a version of FreshML that provides the signature in Figure 1 in the presence of higher order recursively defined functions on user declared data structures. Its syntax is given in Figure 2. As usual,

$$\begin{array}{l}\mathsf{fun}(f\,x = -)\\ \mathsf{let}\,x = e\,\mathsf{in}\,-\\ \mathsf{match}\,v\,\mathsf{with}\,(\mathsf{C}\,x \to - \mid \cdots)\\ S \circ (x.-)\end{array}$$

are all variable-binding constructs and we identify expressions and frame stacks up to renaming of bound variables. As well as variables (standing for unknown values), the language's expressions and frame stacks may contain *atoms* drawn from a fixed, countably infinite set $\mathbb{A}$. As discussed in the introduction, atoms are used in FreshML to represent names in object-level languages. Note that even though there are variable-binding constructs in FreshML, none of the language constructs in Figure 1 involve binding atoms. So we do not identify expressions up to renaming atoms; for example, if $a \neq a'$, then $\langle\!\langle a \rangle\!\rangle(\mathsf{C}\,a)$ and $\langle\!\langle a' \rangle\!\rangle(\mathsf{C}\,a')$ are different expressions (that turn out to be contextually equivalent). We write

$$atom(e) \tag{3}$$

for the finite set of atoms that occur anywhere in the expression e. The same notation $atom(-)$ is used for the finite set of atoms in a frame stack and, more generally, in a finite list of expressions, frame stacks, atoms, and so on.

The language defined in Figure 1 is parameterised by the choice of a finite set $\mathcal{O}$ of function symbols (that we call *observations on atoms* and whose role is discussed below), by a finite set $\mathcal{D}$ of *data type* symbols, and by a finite set $\mathcal{C}$ of *constructor* symbols. Each constructor $\mathsf{C} \in \mathcal{C}$ is assumed to come with a type, $\mathsf{C} : \tau \to \delta$, where $\tau \in \mathit{Typ}$ and $\delta \in \mathcal{D}$. The choice of $\mathcal{D}$, $\mathcal{C}$ and this typing information constitutes an ML-style top-level declaration of some (possibly mutually recursive) data types:

$$\begin{array}{lll}\mathsf{type} & \delta_1 = \mathsf{C}_{1,1}\,\mathsf{of}\,\tau_{1,1} \mid \cdots \mid \mathsf{C}_{1,n_1}\,\mathsf{of}\,\tau_{1,n_1}\\ & \vdots\\ \mathsf{and} & \delta_m = \mathsf{C}_{m,1}\,\mathsf{of}\,\tau_{m,1} \mid \cdots \mid \mathsf{C}_{m,n_m}\,\mathsf{of}\,\tau_{m,n_m}\ .\end{array} \tag{4}$$

Here δ_i (for $i = 1..m$) are the distinct elements of the set $\mathcal{D}$ of data type symbols and $\mathsf{C}_{i,j}$ (for $i = 1..m$ and $j = 1..n_i$) are the distinct elements of the set $\mathcal{C}$ of constructor symbols. The above declaration just records the typing information $\mathsf{C} : \tau \to \delta$ that comes with each constructor, grouped by result types: δ_i appears as the result type of precisely the constructors $\mathsf{C}_{i,1}, \ldots, \mathsf{C}_{i,n_i}$ and their argument types are $\tau_{i,1}, \ldots, \tau_{i,n_i}$. For the moment we place no restriction on these types $\tau_{i,j}$: they can be any element of the set Typ whose grammar is given in Figure 2. However, when we consider representation of object-level languages up to α-equivalence in Section 5, we will restrict attention to top-level data type declarations where the types $\tau_{i,j}$ do not involve function types.

We consider observations on atoms that return natural numbers.[2] So we assume $\mathcal{D}$ always contains a distinguished data type

[2] The effect of admitting some other types of operation on atoms is discussed in Section 6.2.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1\,, v_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau'}{\Gamma \vdash \mathsf{fun}(f\,x = e) : \tau \to \tau'} \qquad \frac{\mathsf{C} : \tau \to \delta \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \mathsf{C}\,v : \delta}$$

$$\frac{a \in \mathbb{A}}{\Gamma \vdash a : \mathsf{atm}} \qquad \frac{\Gamma \vdash v_1 : \mathsf{atm} \qquad \Gamma \vdash v_2 : \tau}{\Gamma \vdash \langle\!\langle v_1 \rangle\!\rangle v_2 : \tau\,\mathsf{bnd}}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'} \qquad \frac{\Gamma \vdash v : \tau_1 * \tau_2}{\Gamma \vdash \mathsf{fst}\,v : \tau_1}$$

$$\frac{\Gamma \vdash v : \tau_1 * \tau_2}{\Gamma \vdash \mathsf{snd}\,v : \tau_2} \qquad \frac{\Gamma \vdash v_1 : \tau \to \tau' \qquad \Gamma \vdash v_2 : \tau}{\Gamma \vdash v_1\,v_2 : \tau'}$$

$$\frac{\begin{array}{c}\delta = \mathsf{C}_1\,\mathsf{of}\,\tau_1 \mid \cdots \mid \mathsf{C}_n\,\mathsf{of}\,\tau_n\\ \Gamma \vdash v : \delta \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \ \cdots\ \Gamma, x_n : \tau_n \vdash e_n : \tau\end{array}}{\Gamma \vdash \mathsf{match}\,v\,\mathsf{with}\,(\mathsf{C}_1\,x_1 \to e_1 \mid \cdots \mid \mathsf{C}_n\,x_n \to e_n) : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{fresh}() : \mathsf{atm}} \qquad \frac{\Gamma \vdash v : \tau\,\mathsf{bnd}}{\Gamma \vdash \mathsf{unbind}\,v : \mathsf{atm} * \tau}$$

$$\frac{\mathit{arity}(\mathsf{obs}) = k \qquad \Gamma \vdash v_1 : \mathsf{atm}\ \cdots\ \Gamma \vdash v_k : \mathsf{atm}}{\Gamma \vdash \mathsf{obs}\,v_1 \ldots v_k : \mathsf{nat}}$$

$$\frac{}{\Gamma \vdash \mathit{Id} : \tau \to \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau' \qquad \Gamma \vdash S : \tau' \to \tau''}{\Gamma \vdash S \circ (x.e) : \tau \to \tau''}$$

Notation:

- $\Gamma, x : \tau$ indicates the typing environment obtained by extending the finite partial function Γ by mapping a variable x to the type τ (we always assume that $x \notin \mathit{dom}(\Gamma)$).
- In the typing rule for match-expressions, the hypothesis "$\delta = \mathsf{C}_1\,\mathsf{of}\,\tau_1 \mid \cdots \mid \mathsf{C}_n\,\mathsf{of}\,\tau_n$" refers to the top-level data type declaration (4); in other words, the only constructors whose result type is δ are $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and τ_i is the argument type of C_i (for $i = 1..n$).

Figure 3. Typing relation.

nat for the type of natural numbers and that correspondingly $\mathcal{C}$ contains constructors $\mathsf{Zero} : \mathsf{unit} \to \mathsf{nat}$ and $\mathsf{Succ} : \mathsf{nat} \to \mathsf{nat}$ for zero and successor. Each $\mathsf{obs} \in \mathcal{O}$ denotes a numerical function on atoms. We assume it comes with an *arity*, specifying the number of arguments it takes: so if $\mathit{arity}(\mathsf{obs}) = k$ and $(v_1, \ldots, v_k)$ is a k-tuple of values of type atm, then $\mathsf{obs}\,v_1 \ldots v_k$ is an expression of type nat. The typing of the language's values, expressions and frame stacks takes place in the presence of typing environments, Γ, each assigning types to finitely many variables. The rules in Figure 3 for the inductively defined typing relation are entirely standard, given that we are following the signature in Fig 1.

As well as an arity, we assume that each $\mathsf{obs} \in \mathcal{O}$ comes with a specified interpretation: the form this takes is discussed in the next section.

The expressions of the language are given in a "reduced" (or "A-normal" [5]) form in which the order of evaluation is made explicit through let-expressions. Hence there is a single form of frame on the evaluation stack S in configurations $\langle \vec{a}, S, e \rangle$ of the abstract machine that we use for defining the language's dynamics. The other two components of machine configurations are the state $\vec{a}$,

$$\boxed{\langle \vec{a}, S, e\rangle \longrightarrow \langle \vec{a}', S', e'\rangle}$$

1. $\langle \vec{a}, S \circ (x.e), v\rangle \longrightarrow \langle \vec{a}, S, e[v/x]\rangle$
2. $\langle \vec{a}, S, \mathsf{let}\, x = e_1\ \mathsf{in}\ e_2\rangle \longrightarrow \langle \vec{a}, S \circ (x.e_2), e_1\rangle$
3. $\langle \vec{a}, S, \mathsf{match}\ \mathsf{C}\, v\ \mathsf{with}(\cdots \mid \mathsf{C}\, x \to e \mid \cdots)\rangle \longrightarrow \langle \vec{a}, S, e[v/x]\rangle$
4. $\langle \vec{a}, S, \mathsf{fst}(v_1\ , v_2)\rangle \longrightarrow \langle \vec{a}, S, v_1\rangle$
5. $\langle \vec{a}, S, \mathsf{snd}(v_1\ , v_2)\rangle \longrightarrow \langle \vec{a}, S, v_2\rangle$
6. $\langle \vec{a}, S, v_1\, v_2\rangle \longrightarrow \langle \vec{a}, S, e[v_1, v_2/f, x]\rangle$ if $v_1 = \mathsf{fun}(f\, x = e)$
7. $\langle \vec{a}, S, \mathsf{fresh}()\rangle \longrightarrow \langle \vec{a} \circledcirc a', S, a'\rangle$ if $a' \notin atom(\vec{a})$
8. $\langle \vec{a}, S, \mathsf{unbind}\, \langle\!\langle a\rangle\!\rangle v\rangle \longrightarrow \langle \vec{a} \circledcirc a', S, (a'\ , v\{a'/a\})\rangle$ if $a' \notin atom(\vec{a})$
9. $\langle \vec{a}, S, \mathsf{obs}\, a_1 \ldots a_k\rangle \longrightarrow \langle \vec{a}, S, \ulcorner m \urcorner\rangle$ if $arity(\mathsf{obs}) = k$, $(a_1, \ldots, a_k) \in atom(\vec{a})^k$ and $[\![\mathsf{obs}]\!]_{\vec{a}}(a_1, \ldots, a_k) = m$

Notation:

- $e[v, \ldots/x, \ldots]$ is the simultaneous, capture avoiding substitution of values $v, \ldots$ for all free occurrences of the corresponding variables $x, \ldots$ in the expression e;
- $v\{a'/a\}$ is the result of replacing all occurrences of an atom a by an atom a' in the value v;
- $\vec{a} \circledcirc a'$ is the state obtained by appending an atom a' not in $atom(\vec{a})$ to the right of the finite list of distinct atoms $\vec{a}$;
- $\ulcorner m \urcorner$ is the the closed value of type nat corresponding to $m \in \mathbb{N}$: $\ulcorner 0 \urcorner \triangleq \mathsf{Zero}()$ and $\ulcorner m + 1 \urcorner \triangleq \mathsf{Succ}\ulcorner m \urcorner$;
- $[\![\mathsf{obs}]\!]$ is the meaning of obs: see Section 3.

Figure 4. Transition relation.

consisting of the finite list of distinct atoms that have been allocated so far, and the expression e to be evaluated. The use of reduced form is a common device to ease development of properties of the language's dynamics. Those dynamics are given by the transition relation in Figure 4. The first six types of transition are all quite standard. Transition 7 defines the dynamic allocation of a fresh atom and transition 8 defines generative unbinding using a freshly created atom; we discuss transition 9 for observations on atoms in the next section. For the atom a' in 7 to really be fresh, we need to know that it does not occur in S; similarly, in 8 we need to know that a' does not occur in (S, a, v). These requirements are met if configurations $\langle \vec{a}, S, e\rangle$ satisfy that all the atoms occurring in the frame stack S or the expression e occur in the list $\vec{a}$. Using the notation $atom(-)$ mentioned above (3), we can write this condition as

$$atom(S, e) \subseteq atom(\vec{a})\ .$$

Theorem 2 shows that this property of configurations is invariant under transitions, as is well-typedness. Before stating this theorem we introduce some useful terminology.

Definition 1 (Worlds). A (*possible*) *world* w is just a finite subset of the the fixed set $\mathbb{A}$ of atoms. We write $World$ for the set of all worlds.

In what follows we will index various relations associated with the language we are considering by worlds $w \in World$ that make explicit the atoms involved in the relation. Sometimes (as in the following theorem) this is a merely a matter of notational convenience; world-indexing will be more crucial when we consider program equivalence: see Remark 12 below.

$$\boxed{\langle \vec{a}, S, e\rangle\downarrow_n} \qquad \boxed{\langle \vec{a}, S, e\rangle\downarrow}$$

$$\frac{}{\langle \vec{a}, Id, v\rangle\downarrow_0} \qquad \frac{\langle \vec{a}, S, e\rangle \longrightarrow \langle \vec{a}', S', e'\rangle \qquad \langle \vec{a}', S', e'\rangle\downarrow_n}{\langle \vec{a}, S, e\rangle\downarrow_{n+1}}$$

$$\frac{\langle \vec{a}, S, e\rangle\downarrow_n}{\langle \vec{a}, S, e\rangle\downarrow}$$

Figure 5. Termination relations.

Theorem 2 (Type Safety). *Write* $\vdash_w \langle \vec{a}, S, e\rangle : \tau$ *to mean that* $atom(S, e) \subseteq atom(\vec{a}) = w$ *and that there is some type* τ' *with* $\emptyset \vdash S : \tau' \to \tau$ *and* $\emptyset \vdash e : \tau'$. *The type system has the following properties.*

Preservation: *if* $\vdash_w \langle \vec{a}, S, e\rangle : \tau$ *and* $\langle \vec{a}, S, e\rangle \longrightarrow \langle \vec{a}', S', e'\rangle$, *with* $atom(\vec{a}') = w'$ *say, then* $w \subseteq w'$ *and* $\vdash_{w'} \langle \vec{a}', S', e'\rangle : \tau$.

Progress: *if* $\vdash_w \langle \vec{a}, S, e\rangle : \tau$, *then either* $S = Id$ *and* $e \in Val$, *or* $\langle \vec{a}, S, e\rangle \longrightarrow \langle \vec{a}', S', e'\rangle$ *holds for some* $\vec{a}'$, S' *and* e'. □

3. Observations on Atoms

The language we are considering is parameterised by a choice of a finite set $\mathcal{O}$ of numerical functions on atoms. We assume that each $\mathsf{obs} \in \mathcal{O}$ comes with a specified meaning $[\![\mathsf{obs}]\!]$. As mentioned in the introduction, we should allow these meanings to be dependent on the current state (the list of distinct atoms that have been created so far). So if $arity(\mathsf{obs}) = k$, for each $\vec{a} \in State$ we assume given a function $[\![\mathsf{obs}]\!]_{\vec{a}} : atom(\vec{a})^k \to \mathbb{N}$ mapping k-tuples of atoms occurring in the state $\vec{a}$ to natural numbers. These functions are used in the transitions of type 9 in Figure 4. Not every such family $([\![\mathsf{obs}]\!]_{\vec{a}} \mid \vec{a} \in State)$ of functions is acceptable as an observation on atoms: we require that the family be *equivariant*. To explain what this means we need the following definition.

Definition 3 (Permutations). A finite *permutation* of atoms is a bijection π from the set $\mathbb{A}$ of atoms onto itself such that $supp(\pi) \triangleq \{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is a finite set. We write $\mathbb{P}$ for the set of all such permutations. If $\pi \in \mathbb{P}$ and $\vec{a} \in State$, then $\pi \cdot \vec{a}$ denotes the finite list of distinct atoms obtained by mapping π over the list $\vec{a}$; if e is an expression, then $\pi \cdot e$ denotes the expression obtained from it by applying π to the atoms in e; and similarly for other syntactical structures involving finitely many atoms, such as values and frame stacks.

We require the functions $([\![\mathsf{obs}]\!]_{\vec{a}} \mid \vec{a} \in State)$ associated with each $\mathsf{obs} \in \mathcal{O}$ to satisfy an *equivariance* property: for all $\pi \in \mathbb{P}$, $\vec{a} \in State$ and $(a_1, \ldots, a_k) \in atom(\vec{a})^k$ (where k is the arity of obs)

$$[\![\mathsf{obs}]\!]_{\vec{a}}(a_1, \ldots, a_k) = [\![\mathsf{obs}]\!]_{\pi \cdot \vec{a}}(\pi(a_1), \ldots, \pi(a_k))\ . \qquad (5)$$

We impose condition (5) for the following reason. In Figure 4, the side conditions on transitions of types 7 and 8 do not specify which of the infinitely many atoms in $\mathbb{A} - atom(\vec{a})$ should be chosen as the fresh atom a'. Any particular implementation of the language will make such choices in some specific way, for example by implementing atoms as numbers and incrementing a global counter to get the next fresh atom. We wish to work at a level of abstraction that is independent of such implementation details. We can do so by ensuring that we only use properties of machine configurations $\langle \vec{a}, S, e\rangle$ that depend on the relative positions of atoms in the list $\vec{a}$, rather than upon their identities. In other words, properties should respect α-equivalence of configurations when the state component is regarded as binding atoms in the stack

Equality, eq ($arity = 2$):

$$[\![\mathsf{eq}]\!]_{\vec{a}}(a, a') \triangleq \begin{cases} 0 & \text{if } a = a', \\ 1 & \text{otherwise.} \end{cases}$$

Linear order, lt ($arity = 2$):

$$[\![\mathsf{lt}]\!]_{\vec{a}}(a, a') \triangleq \begin{cases} 0 & \text{if } a \text{ occurs to the left of } a' \text{ in the list } \vec{a}, \\ 1 & \text{otherwise.} \end{cases}$$

Ordinal, ord ($arity = 1$):

$$[\![\mathsf{ord}]\!]_{\vec{a}}(a) \triangleq n, \text{ if } a \text{ is the } n\text{th element of the list } \vec{a}.$$

State size, card ($arity = 0$):

$$[\![\mathsf{card}]\!]_{\vec{a}}() \triangleq \text{length of the list } \vec{a}.$$

Figure 6. Examples of observations on atoms.

and expression components. For the reasons given in the previous section we only use configurations that are closed in the sense of satisfying $atom(S, e) \subseteq atom(\vec{a})$ (cf. Theorem 2); and α-equivalence for such configurations relates $\langle \vec{a}, S, e\rangle$ with $\langle \pi \cdot \vec{a}, \pi \cdot S, \pi \cdot e\rangle$ for any $\pi \in \mathbb{P}$. So properties of configurations should be equivariant: if $\langle \vec{a}, S, e\rangle$ has the property, then so should $\langle \pi \cdot \vec{a}, \pi \cdot S, \pi \cdot e\rangle$. The main property of configurations we need is *termination*, defined in Figure 5, since as we see in the next section this determines contextual equivalence of expressions. With condition (5) we have:

Lemma 4. *If $\langle \vec{a}, S, e\rangle\downarrow_n$, then $\langle \pi \cdot \vec{a}, \pi \cdot S, \pi \cdot e\rangle\downarrow_n$ for any $\pi \in \mathbb{P}$.*

Proof. In view of the definition of termination in Figure 5, it suffices to show that the transition relation is equivariant:

$$\langle \vec{a}, S, e\rangle \longrightarrow \langle \vec{a}', S', e'\rangle \;\Rightarrow\; \langle \pi \cdot \vec{a}, \pi \cdot S, \pi \cdot e\rangle \longrightarrow \langle \pi \cdot \vec{a}', \pi \cdot S', \pi \cdot e'\rangle\,.$$

This can be proved by cases from the definition of $\longrightarrow$ in Fig 4. Cases 1–8 follow from general properties of the action of permutations on syntactical structures (such as the fact that $\pi \cdot (e[v/x])$ equals $(\pi \cdot e)[\pi \cdot v/x]$); case 9 uses property (5). □

As a corollary we find that termination is indeed independent of the choice of fresh atom in transitions of the form 7 or 8.

Corollary 5. *If $\langle \vec{a}, S, \mathsf{fresh}\rangle\downarrow_{n+1}$ with $atom(S) \subseteq atom(\vec{a})$, then for all $a' \notin atom(\vec{a})$, it is the case that $\langle \vec{a} \circledcirc a', S, a'\rangle\downarrow_n$. Similarly, if $\langle \vec{a}, S, \mathsf{unbind}\,\langle\!\langle a\rangle\!\rangle v\rangle\downarrow_{n+1}$ with $atom(S, a, v) \subseteq atom(\vec{a})$, then for all $a' \notin atom(\vec{a})$, it is the case that $\langle \vec{a} \circledcirc a', S, (a', v\{a'/a\})\rangle\downarrow_n$.* □

There are observations on atoms that are not equivariant, that is, whose value on some atoms in a particular state does not depend just upon the relative position of those atoms in the state. For example, if we fix some enumeration of the set of atoms, $\alpha : \mathbb{N} \cong \mathbb{A}$, it is easy to see that the unary observation given by $[\![\mathsf{obs}]\!]_{\vec{a}}(a) = \alpha^{-1}(a)$ fails to satisfy (5). Nevertheless, there is a wide range of functions that do have this property. Figure 6 gives some examples. The first one, eq, combined with the usual arithmetic operations for nat that are already definable in the language, gives us the effect of the function $(=) : \mathsf{atm} \to \mathsf{atm} \to \mathsf{bool}$ from the signature in Figure 1; so we assume that the set $\mathcal{O}$ of observations on atoms always contains eq.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_w x \;\widehat{\mathcal{E}}\; x : \tau} \qquad \frac{}{\Gamma \vdash_w () \;\widehat{\mathcal{E}}\; () : \mathsf{unit}}$$

$$\frac{\Gamma \vdash_w v_1 \;\mathcal{E}\; v_1' : \tau_1 \qquad \Gamma \vdash_w v_2 \;\mathcal{E}\; v_2' : \tau_2}{\Gamma \vdash_w (v_1\,, v_2) \;\widehat{\mathcal{E}}\; (v_1'\,, v_2') : \tau_1 * \tau_2}$$

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash_w e \;\mathcal{E}\; e' : \tau'}{\Gamma \vdash_w \mathsf{fun}(f\,x = e) \;\widehat{\mathcal{E}}\; \mathsf{fun}(f\,x = e') : \tau \to \tau'}$$

$$\frac{\mathsf{C} : \tau \to \delta \qquad \Gamma \vdash_w v \;\mathcal{E}\; v' : \tau}{\Gamma \vdash_w \mathsf{C}\,v \;\widehat{\mathcal{E}}\; \mathsf{C}\,v' : \delta} \qquad \frac{a \in w}{\Gamma \vdash_w a \;\widehat{\mathcal{E}}\; a : \mathsf{atm}}$$

$$\frac{\Gamma \vdash_w v_1 \;\mathcal{E}\; v_1' : \mathsf{atm} \quad \Gamma \vdash_w v_2 \;\mathcal{E}\; v_2' : \tau}{\Gamma \vdash_w \langle\!\langle v_1\rangle\!\rangle v_2 \;\widehat{\mathcal{E}}\; \langle\!\langle v_1'\rangle\!\rangle v_2' : \tau\,\mathsf{bnd}}$$

$$\frac{\Gamma \vdash_w e_1 \;\mathcal{E}\; e_1' : \tau \qquad \Gamma, x : \tau \vdash_w e_2 \;\mathcal{E}\; e_2' : \tau'}{\Gamma \vdash_w \mathsf{let}\; x = e_1 \;\mathsf{in}\; e_2 \;\widehat{\mathcal{E}}\; \mathsf{let}\; x = e_1' \;\mathsf{in}\; e_2' : \tau'}$$

$$\frac{\Gamma \vdash_w v \;\mathcal{E}\; v' : \tau_1 * \tau_2}{\Gamma \vdash_w \mathsf{fst}\,v \;\widehat{\mathcal{E}}\; \mathsf{fst}\,v' : \tau_1} \qquad \frac{\Gamma \vdash_w v \;\mathcal{E}\; v' : \tau_1 * \tau_2}{\Gamma \vdash_w \mathsf{snd}\,v \;\widehat{\mathcal{E}}\; \mathsf{snd}\,v' : \tau_2}$$

$$\frac{\Gamma \vdash_w v_1 \;\mathcal{E}\; v_1' : \tau \to \tau' \qquad \Gamma \vdash_w v_2 \;\mathcal{E}\; v_2' : \tau}{\Gamma \vdash_w v_1\,v_2 \;\widehat{\mathcal{E}}\; v_1'\,v_2' : \tau'}$$

$$\frac{\begin{array}{c}\delta = \mathsf{C}_1\;\mathsf{of}\;\tau_1 \mid \cdots \mid \mathsf{C}_n\;\mathsf{of}\;\tau_n \qquad \Gamma \vdash_w v \;\mathcal{E}\; v' : \delta \\ \Gamma, x_1 : \tau_1 \vdash_w e_1 \;\mathcal{E}\; e_1' : \tau \;\cdots\; \Gamma, x_n : \tau_n \vdash_w e_n \;\mathcal{E}\; e_n' : \tau\end{array}}{\begin{array}{l}\Gamma \vdash_w \mathsf{match}\;v\;\mathsf{with}\;(\mathsf{C}_1\,x_1 \to e_1 \mid \cdots \mid \mathsf{C}_n\,x_n \to e_n) \;\widehat{\mathcal{E}} \\ \quad \mathsf{match}\;v'\;\mathsf{with}\;(\mathsf{C}_1\,x_1 \to e_1' \mid \cdots \mid \mathsf{C}_n\,x_n \to e_n') : \tau\end{array}}$$

$$\frac{}{\Gamma \vdash_w \mathsf{fresh}() \;\widehat{\mathcal{E}}\; \mathsf{fresh}() : \mathsf{atm}}$$

$$\frac{\Gamma \vdash_w v \;\mathcal{E}\; v' : \tau\,\mathsf{bnd}}{\Gamma \vdash_w \mathsf{unbind}\,v \;\widehat{\mathcal{E}}\; \mathsf{unbind}\,v' : \mathsf{atm} * \tau}$$

$$\frac{arity(\mathsf{obs}) = k \quad \Gamma \vdash_w v_1 \;\mathcal{E}\; v_1' : \mathsf{atm} \;\cdots\; \Gamma \vdash_w v_k \;\mathcal{E}\; v_k' : \mathsf{atm}}{\Gamma \vdash_w \mathsf{obs}\,v_1 \ldots v_k \;\widehat{\mathcal{E}}\; \mathsf{obs}\,v_1' \ldots v_k' : \mathsf{nat}}$$

$$\frac{}{\Gamma \vdash_w Id \;\widehat{\mathcal{E}}\; Id : \tau \to \tau}$$

$$\frac{\Gamma, x : \tau \vdash_w e \;\mathcal{E}\; e' : \tau' \qquad \Gamma \vdash_w S \;\widehat{\mathcal{E}}\; S' : \tau' \to \tau''}{\Gamma \vdash_w S \circ (x.e) \;\widehat{\mathcal{E}}\; S' \circ (x.e') : \tau \to \tau''}$$

Figure 7. Compatible refinement $\widehat{\mathcal{E}}$ of an expression relation $\mathcal{E}$.

Remark 6 (Fresh Atoms Largest). Note that in the operational semantics of Figure 4 we have chosen to make "fresh atoms largest", in the sense that the fresh atom a' in transitions 7 and 8 is added to the right-hand end of the list $\vec{a}$ representing the current state. In the presence of observations on atoms other than equality, such a choice may well affect the properties of the notion of program equivalence that we explore in the next section. Other choices are possible, but to insist that program equivalence is independent of any such choice would rule out many useful observations on atoms (such as lt or ord in Figure 6).

4. Contextual Equivalence

We wish to prove that the language we have described satisfies "Correctness of Representation" properties of the kind mentioned in the introduction. To do so, we first have to be more precise about what it means for two expressions to be *contextually equivalent*, that is, to be interchangeable in any program without affecting the observable results of executing that program. What is a program, what does it mean to execute it, and what results of execution do we observe? The answers we take to these questions are: programs are closed well-typed expressions; execution means carrying out a sequence of transitions of the abstract machine from an initial machine configuration consisting of a state (that is, a list of atoms containing those mentioned in the program), the empty frame stack and the program; and we observe whether execution reaches a terminal configuration, that is, one of the form $\langle \vec{a}, Id, v\rangle$. We need only observe termination because of the language's strict evaluation strategy: observing any (reasonable) properties of the final value v results in the same notion of contextual equivalence. Also, it is technically convenient to be a bit more liberal about what constitutes an initial configuration by allowing the starting frame stack to be non-empty: this does not change the notion of contextual equivalence because of the correspondence between frame stacks and "evaluation" contexts—see the remarks after Definition 11 below. So we can say that e and e' are contextually equivalent if for all program contexts $\mathcal{C}[-]$, the programs $\mathcal{C}[e]$ and $\mathcal{C}[e']$ are *operationally equivalent* in the following sense.

Definition 7 (Operational Equivalence of Closed Expressions). $\vdash_w e \cong e' : \tau$ is defined to hold if

- $atom(e, e') \subseteq w$;
- $\emptyset \vdash e : \tau$ and $\emptyset \vdash e' : \tau$; and
- for all $\vec{a}$, S and τ' with $w \cup atom(S) \subseteq atom(\vec{a})$ and $\emptyset \vdash S : \tau \to \tau'$, it is the case that $\langle \vec{a}, S, e\rangle\downarrow \;\Leftrightarrow\; \langle \vec{a}, S, e'\rangle\downarrow$.

However, for the reasons given in [18, Section 7.5], we prefer not to phrase the formal definition of contextual equivalence in terms of the inconveniently concrete operation of possibly capturing substitution of open expressions for the hole "$-$" in program contexts $\mathcal{C}[-]$. Instead we take the more abstract relational approach originally advocated by Gordon [7] and Lassen [11] which focuses upon the key features of contextual equivalence, namely that it is *the largest congruence relation for well-typed expressions that contains the relation of operational equivalence of Definition 7.* A congruence relation is an expression relation that is both an equivalence and compatible, in the following sense.

Definition 8 (Expression Relations). An *expression relation* $\mathcal{E}$ is a set of tuples (Γ, w, e, e', τ) (made up of a typing context, a world, two expressions and a type) satisfying $atom(e, e') \subseteq w$, $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. We write $\Gamma \vdash_w e \;\mathcal{E}\; e' : \tau$ to indicate that (Γ, w, e, e', τ) is a member of $\mathcal{E}$. We use the following terminology in connection with expression relations.

- $\mathcal{E}$ is an **equivalence** if it is reflexive ($atom(e) \subseteq w \;\wedge\; \Gamma \vdash e : \tau \;\Rightarrow\; \Gamma \vdash_w e \;\mathcal{E}\; e : \tau$), symmetric ($\Gamma \vdash_w e \;\mathcal{E}\; e' : \tau \;\Rightarrow\; \Gamma \vdash_w e' \;\mathcal{E}\; e : \tau$) and transitive ($\Gamma \vdash_w e \;\mathcal{E}\; e' : \tau \;\wedge\; \Gamma \vdash_w e' \;\mathcal{E}\; e'' : \tau \;\Rightarrow\; \Gamma \vdash_w e \;\mathcal{E}\; e'' : \tau$).
- $\mathcal{E}$ is **compatible** if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$, where $\widehat{\mathcal{E}}$ is the *compatible refinement* of $\mathcal{E}$, defined in Figure 7.
- $\mathcal{E}$ is **substitutive** if $\Gamma \vdash_w v \;\mathcal{E}\; v' : \tau \wedge \Gamma, x : \tau \vdash_w e \;\mathcal{E}\; e' : \tau' \;\Rightarrow\; \Gamma \vdash_w e[v/x] \;\mathcal{E}\; e'[v'/x] : \tau'$.
- $\mathcal{E}$ is **equivariant** if $\Gamma \vdash_w e \;\mathcal{E}\; e' : \tau \;\Rightarrow\; \Gamma \vdash_{\pi\cdot w} \pi\cdot e \;\mathcal{E}\; \pi\cdot e' : \tau$.
- $\mathcal{E}$ is **adequate** if $\vdash_w e \cong e' : \tau \;\Rightarrow\; \emptyset \vdash_w e \;\mathcal{E}\; e' : \tau$.

We extend operational equivalence (Definition 7) to an expression relation, $\Gamma \vdash_w e \cong^\circ e' : \tau$, by instantiating free variables with closed values:

Definition 9 ($\cong^\circ$). Supposing $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, we define $\Gamma \vdash_w e \cong^\circ e' : \tau$ to hold if

- $atom(e, e') \subseteq w$;
- $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$; and
- for all $w' \supseteq w$ and all closed values v_i with $atom(v_i) \subseteq w'$ and $\emptyset \vdash v_i : \tau_i$ (for $i = 1..n$), it is the case that $\vdash_{w'} e[\vec{v}/\vec{x}] \cong e'[\vec{v}/\vec{x}] : \tau$.

Note that for closed expressions, that is, in the case that $\Gamma = \emptyset$, the relation $\cong^\circ$ agrees with $\cong$:

$$\emptyset \vdash_w e \cong^\circ e' : \tau \;\Leftrightarrow\; \vdash_w e \cong e' : \tau\,. \tag{6}$$

Theorem 10 (CIU). *Operational equivalence of possibly open expressions, $\cong^\circ$, is a compatible adequate equivalence. It is the largest such expression relation. It is also substitutive and equivariant.*

Proof. The fact that $\cong^\circ$ is equivariant follows from Lemma 4; and the fact that it is an equivalence and adequate is immediate from its definition. So the main difficulty is to show that it is compatible and substitutive. One can do this by adapting a construction due to Howe [10], as follows. Let the expression relation $\cong^*$ be inductively defined from $\cong^\circ$ by the rule

$$\frac{\Gamma \vdash_w e \;\widehat{\cong^*}\; e' : \tau \qquad \Gamma \vdash_w e' \cong^\circ e'' : \tau}{\Gamma \vdash_w e \cong^* e'' : \tau}\,. \tag{7}$$

(In making this inductive definition, we are implicitly relying upon the easily proved fact that compatible refinement, $\mathcal{E} \mapsto \widehat{\mathcal{E}}$, is a monotone operation on expression relations, that is, $\mathcal{E}_1 \subseteq \mathcal{E}_2 \;\Rightarrow\; \widehat{\mathcal{E}_1} \subseteq \widehat{\mathcal{E}_2}$.) It is not hard to show that $\cong^*$ is compatible and substitutive and contains $\cong^\circ$. So the compatibility and substitutivity of $\cong^\circ$ follow once one proves $\cong^* \subseteq \cong^\circ$ and hence that these two expression relations coincide. This can be deduced by proving the following two properties, (8) and (9). In (8), $\widehat{\cong^*}$ is extended to a relation between frame stacks using the last two rules in Figure 7; and in (9), $(\cong^*)^+$ denotes the transitive closure of $\cong^*$.

$$\emptyset \vdash_w S \;\widehat{\cong^*}\; S' : \tau \to \tau' \;\wedge\; \emptyset \vdash_w e \cong^* e' : \tau \;\wedge\; atom(\vec{a}) = w \;\wedge\; \langle \vec{a}, S, e\rangle\downarrow_n \;\Rightarrow\; \langle \vec{a}, S', e'\rangle\downarrow \tag{8}$$

$$\Gamma \vdash_w e \cong^* e' : \tau \;\Rightarrow\; \Gamma \vdash_w e' \;(\cong^*)^+\; e : \tau\,. \tag{9}$$

If we have these two properties, then from the first one (using the fact that $\widehat{\cong^\circ}$ is closed under weakening from a world w to a larger one $w' \supseteq w$) we get

$$\emptyset \vdash_w e \cong^* e' : \tau \;\Rightarrow\; \forall \vec{a}, S, \tau'.\; w \cup atom(S) \subseteq atom(\vec{a}) \;\wedge\; \emptyset \vdash S : \tau \to \tau' \;\wedge\; \langle \vec{a}, S, e\rangle\downarrow \;\Rightarrow\; \langle \vec{a}, S, e'\rangle\downarrow\,.$$

Since the right-hand side of this implication is a transitive relation between e and e', it follows that

$$\emptyset \vdash_w e \cong^{*+} e' : \tau \;\Rightarrow\; \forall \vec{a}, S, \tau'.\, w \cup atom(S) \subseteq atom(\vec{a}) \wedge \emptyset \vdash S : \tau \to \tau' \;\wedge\; \langle \vec{a}, S, e \rangle\downarrow \;\Rightarrow\; \langle \vec{a}, S, e' \rangle\downarrow$$

and hence by property (9) that

$$\emptyset \vdash_w e \cong^{*} e' : \tau \;\Rightarrow\; \forall \vec{a}, S, \tau'.\, w \cup atom(S) \subseteq atom(\vec{a}) \wedge \emptyset \vdash S : \tau \to \tau' \;\wedge\; \langle \vec{a}, S, e' \rangle\downarrow \;\Rightarrow\; \langle \vec{a}, S, e \rangle\downarrow .$$

Therefore by definition of $\cong$ (Definition 7) we have

$$\emptyset \vdash_w e \cong^{*} e' : \tau \;\Rightarrow\; \vdash_w e \cong e' : \tau . \tag{10}$$

Since the open extension $\cong^\circ$ of $\cong$ is easily seen to be substitutive and reflexive, (10) implies that $\cong^* \subseteq \cong^\circ$, as required.

So to complete the proof, one just has to establish properties (8) and (9). The second is easy to prove, but property (8) requires a tricky induction on the derivation of $\langle \vec{a}, S, e \rangle\downarrow_n$ from the rules in Figure 5. The induction step corresponding to transitions of type 8 in Figure 4, that is, to generative unbinding, is quite delicate. It relies upon the fact that when $a' \notin atom(v)$, a renamed value $v\{a'/a\}$ is the same as a permuted value $(a\ a') \cdot v$ (where $(a\ a')$ denotes the permutation swapping a and a'). This allows one to use the equivariance of the termination relation, Lemma 4, to prove the induction step—the details appear in the full version of this paper. Of course, the validity of Lemma 4 depends crucially upon the equivariance condition (5) that we require of observations on atoms. □

Definition 11 (Contextual Equivalence). In view of the discussion at the beginning of this section, Theorem 10 tells us that $\cong^\circ$ coincides with a conventional notion of contextual equivalence defined using program contexts: so from now on we refer to $\cong^\circ$ as *contextual equivalence*.

We labelled the above theorem "CIU" because it is analogous to a theorem of that name due to Mason and Talcott [12]. CIU, after permutation, stands for "Uses of Closed Instantiations"; and the theorem tells us that to test open expressions for contextual equivalence it suffices to first close them by substituting closed values for free variables and then test the resulting closed expressions for termination when they are used in any *evaluation context* [4]. This follows from the definition of $\cong^\circ$ and the fact that termination in evaluation contexts corresponds to termination of machine configurations via the easily verified property

$$\langle \vec{a}, S, e \rangle\downarrow \;\Leftrightarrow\; \langle \vec{a}, Id, S[e] \rangle\downarrow \tag{11}$$

where the expression $S[e]$ is defined by recursion on the length of the stack S by:

$$\begin{array}{rcl} Id[e] & \triangleq & e \\ S \circ (x.e')[e] & \triangleq & S[\mathsf{let}\, x = e\, \mathsf{in}\, e'] \end{array} . \tag{12}$$

Theorem 10 serves to establish some basic properties of contextual equivalence, such as the fact that the state-independent transitions in Figure 4 (types 1–6 and 9) give rise to contextual equivalences. For example, $\Gamma \vdash_w \mathsf{let}\, x = v\, \mathsf{in}\, e \cong^\circ e[v/x] : \tau'$ holds if $\Gamma \vdash_w v : \tau$ and $\Gamma, x : \tau \vdash_w e : \tau'$. However, we have to work a bit harder to understand the consequences of transitions of types 7 and 8 for contextual equivalence at atom binding types, $\tau\,\mathsf{bnd}$. We address this in the next section.

Remark 12 (Possible Worlds). It is immediate from the definition of $\cong^\circ$ that it satisfies a weakening property:

$$\Gamma \vdash_w e \cong^\circ e' : \tau \;\wedge\; w \subseteq w' \;\Rightarrow\; \Gamma \vdash_{w'} e \cong^\circ e' : \tau . \tag{13}$$

If it also satisfied a strengthening property

$$\Gamma \vdash_{w'} e \cong^\circ e' : \tau \;\wedge\; atom(e, e') \subseteq w \subseteq w' \;\Rightarrow\; \Gamma \vdash_w e \cong^\circ e' : \tau \tag{14}$$

then we could make the indexing of contextual equivalence by "possible worlds" w implicit by taking $w = atom(e, e')$. When $\mathcal{O}$ just contains eq, property (14) does hold; this is why there is no need for indexing by possible worlds in [23, 25]. However, it is not hard to see that the presence of some observations on atoms, such as the function card in Figure 6, can cause (14) to fail. It is for this reason that we have built indexing by possible worlds into expression relations (Definition 8).

5. Correctness of Representation

Recall from Section 2 that the language we are considering is parameterised by a top-level declaration of some (possibly mutually recursive) data types:

$$\begin{array}{lll} \mathsf{type} & \delta_1 = \mathsf{C}_{1,1}\ \mathsf{of}\ \tau_{1,1} \mid \cdots \mid \mathsf{C}_{1,n_1}\ \mathsf{of}\ \tau_{1,n_1} & \\ & \vdots & \\ \mathsf{and} & \delta_m = \mathsf{C}_{m,1}\ \mathsf{of}\ \tau_{m,1} \mid \cdots \mid \mathsf{C}_{m,n_m}\ \mathsf{of}\ \tau_{m,n_m} & . \end{array} \tag{15}$$

If we restrict attention to declarations in which the argument types $\tau_{i,j}$ of the constructors $\mathsf{C}_{i,j}$ are just finite products of the declared data types $\delta_1 \ldots, \delta_m$, then the above declaration corresponds to a *many-sorted algebraic signature*; furthermore, in this case the language's values at each data type are just the abstract syntax trees of terms of the corresponding sort in the signature. By allowing atoms and atom bindings in addition to products in the argument types $\tau_{i,j}$, one arrives at the notion of "nominal signature", introduced in [29] and more fully developed in [19]. It extends the notion of many-sorted algebraic signature with names (of possibly many kinds) and information about name binding in constructors. Here, for simplicity, we are restricting to a single kind of name, represented by the type atm of atoms; but our results extend easily to the case of many kinds of name.

Definition 13 (Nominal Signatures). The subset $Arity \subseteq Typ$ is given by the grammar

$$\sigma \in Arity ::= \mathsf{unit} \mid \sigma * \sigma \mid \delta \mid \mathsf{atm} \mid \sigma\,\mathsf{bnd} \tag{16}$$

where δ ranges over the finite set $\mathcal{D}$ of data type symbols. (In other words $Arity$ consists of those types of our language that do not involve any use of the function type construction, $\to$.) The elements of the set $Arity$ are called *nominal arities*.[3] A *nominal signature* (with a single sort of atoms, atm) is specified by a data type declaration (15) in which the argument types $\tau_{i,j}$ of the constructors $\mathsf{C}_{i,j}$ are all nominal arities.

For each $\sigma \in Arity$, the closed values of that type, $\emptyset \vdash_w v : \sigma$, correspond precisely to the ground terms (with arity σ and atoms in w) over the nominal signature, as defined in [29]. For example, the declaration (1) corresponds to the nominal signature for λ-calculus; and closed values of type term correspond as in (2) to the abstract syntax trees for λ-terms (open or closed ones, with variables represented by atoms). For other examples of nominal signatures, with more complicated patterns of binding, see [19, Section 2.2].

The occurrences of $\sigma\,\mathsf{bnd}$ in a nominal signature (15) indicate arguments with bound atoms. In particular, we can associate with each such signature a notion of *α-equivalence*, $=_\alpha$, that identifies values up to renaming bound atoms. The inductive definition of $=_\alpha$ is given in Figure 8. It generalises to an arbitrary nominal signature

[3] The notation $\langle\!\langle \mathsf{atm} \rangle\!\rangle \sigma$ is used in [29, 19] for what we here write as $\sigma\,\mathsf{bnd}$.

$$\frac{}{\vdash_w () =_\alpha () : \mathsf{unit}}$$

$$\frac{\vdash_w v_1 =_\alpha v_1' : \sigma_1 \qquad \vdash_w v_2 =_\alpha v_2' : \sigma_2}{\vdash_w (v_1 , v_2) =_\alpha (v_1' , v_2') : \sigma_1 * \sigma_2}$$

$$\frac{\mathsf{C} : \sigma \to \delta \qquad \vdash_w v =_\alpha v' : \sigma}{\vdash_w \mathsf{C}\, v =_\alpha \mathsf{C}\, v' : \delta} \qquad \frac{a \in w}{\vdash_w a =_\alpha a : \mathsf{atm}}$$

$$\frac{\begin{array}{c} a'' \notin w \supseteq atom(a, v, a', v') \\ \vdash_{w \cup \{a''\}} v\{a''/a\} =_\alpha v'\{a''/a'\} : \sigma \end{array}}{\vdash_w \langle\!\langle a \rangle\!\rangle v =_\alpha \langle\!\langle a' \rangle\!\rangle v' : \sigma\ \mathsf{bnd}}$$

Figure 8. α-Equivalence.

the syntax-directed characterisation of α-equivalence of λ-terms given in [9, p. 36]. The definition in Figure 8 is essentially that given in [19], except that we have included an indexing by possible worlds w, to chime with our form of judgement for contextual equivalence; without that indexing, the condition "$a'' \notin w \supseteq atom(a, v, a', v')$" in the rule for α-equivalence of values of atom binding type would be replaced by "$a'' \notin atom(a, v, a', v')$".

We can now state the main result of this paper. Recall from Definitions 9 and 11 that for closed expressions contextual equivalence is the same as the relation of operational equivalence given in Definition 7.

Theorem 14 (Correctness of Representation). *Suppose that all the observations on atoms* obs *in* $\mathcal{O}$ *satisfy the equivariance property* (5)*. For each nominal signature, two closed values* v, v' *of the same nominal arity* σ *(with atoms contained in the finite set* w*, say) are* α*-equivalent if and only if they are contextually equivalent:*

$$\vdash_w v =_\alpha v' : \sigma \;\Leftrightarrow\; \vdash_w v \cong v' : \sigma\ . \tag{17}$$

The rest of this section is devoted to the proof of this theorem. Proving the left-to-right implication in (17) is not too hard, given the simple nature of α-equivalence as defined in Figure 8. The right-to-left implication is much harder: see the discussion after Proposition 16. We get it as a corollary of a property of $\cong$ at atom binding arities $\sigma\ \mathsf{bnd}$ that mirrors the fifth rule in Figure 8. In fact we will prove this property of atom binding (the combination of Propositions 16 and 19) for all types $\tau\ \mathsf{bnd}$, that is, not just for nominal arities but also for types τ possibly involving the function type construct. We can only do so under a restriction on observations on atoms over and above the equivariance property (5) that we always assume they possess. This is the "affineness" property given in Definition 17 below. The equality test eq (Figure 6) is affine and we will see that this fact is enough to prove Theorem 14 as stated, that is, without any restriction on the observations present other than equivariance.

Proposition 15. *(i)* $\vdash_w () \cong () : \mathsf{unit}$.

(ii) For all types $\tau_1, \tau_2 \in Typ$, $\vdash_w (v_1 , v_2) \cong (v_1' , v_2') : \tau_1 * \tau_2$ *iff* $\vdash_w v_1 \cong v_1' : \tau_1$ *and* $\vdash_w v_2 \cong v_2' : \tau_2$.

(iii) For each data type δ_i *in the declaration* (15), $\vdash_w \mathsf{C}_{i,j}\, v \cong \mathsf{C}_{i,j'}\, v' : \delta_i$ *iff* $j = j'$ *and* $\vdash_w v \cong v' : \tau_{i,j}$.

(iv) $\vdash_w a \cong a' : \mathsf{atm}$ *iff* $a = a' \in w$.

Proof. Part (i) and the "if" directions of (ii)–(iv) are consequences of the fact (Theorem 10) that $\cong^\circ$ is a compatible equivalence. For the "only if" directions of (ii) and (iii) we apply suitably chosen destructors. Thus for part (ii) we use the operational equivalences $\vdash_w \mathsf{fst}(v_1 , v_2) \cong v_1 : \tau_1$ and $\vdash_w \mathsf{snd}(v_1 , v_2) \cong v_2 : \tau_2$ that are consequences of the definitions of $\cong$ and the termination relation. Similarly, part (iii) follows from the easily established operational (in)equivalences

$$\begin{array}{l} \vdash_w \mathsf{diverge} \ncong v : \tau \\ \vdash_w \mathsf{proj}_{i,j}\, (\mathsf{C}_{i,j}\, v) \cong v : \tau_{i,j} \\ \vdash_w \mathsf{proj}_{i,j}\, (\mathsf{C}_{i,j'}\, v) \cong \mathsf{diverge} : \tau_{i,j} \quad \text{if } j \neq j' \end{array}$$

which make use of the following expressions

$$\begin{array}{rcl} \mathsf{diverge} & \triangleq & \mathsf{fun}(f\, x = f\, x)() \\ \mathsf{proj}_{i,j}\, v & \triangleq & \mathsf{match}\ v\ \mathsf{with} \\ & & \quad (\mathsf{C}_{i,1} x_1 \to d_{j,1} \mid \cdots \mid \mathsf{C}_{i,n_i}\, x_{n_i} \to d_{j,n_i}) \end{array}$$

where $d_{j,j'} \triangleq \begin{cases} x_j & \text{if } j = j', \\ \mathsf{diverge} & \text{if } j \neq j'. \end{cases}$

Finally, for the "only if" direction of part (iv) we make use of the fact that $\mathcal{O}$ always contains the atom equality function eq from Figure 6. Consider the frame stack

$$S_a \triangleq Id \circ (x.\, \mathsf{let}\ y = \mathsf{eq}\, x\, a\ \mathsf{in} \\ \mathsf{match}\ y\ \mathsf{with}\ (\mathsf{Zero} \to () \mid \mathsf{Succ}\, z \to \mathsf{diverge}))\ .$$

If $a \neq a'$ are distinct elements of w, then choosing some $\vec{a} \in State$ with $atom(\vec{a}) = w$, it is not hard to see that $\langle \vec{a}, S_a, a \rangle\!\downarrow$ holds whereas $\langle \vec{a}, S_a, a' \rangle\!\downarrow$ does not hold. So if $\vdash_w a \cong a' : \mathsf{atm}$ it cannot be the case that $a \neq a'$. □

This proposition tells us that $\cong$ has properties mirroring those of α-equivalence given by the first four rules in Figure 8. To complete the proof of the correctness theorem, we need to prove a property of $\cong$ at atom binding arities $\sigma\ \mathsf{bnd}$ that mirrors the fifth rule in that figure. We split this into two parts, Propositions 16 and 19.

Proposition 16. *For any type* $\tau \in Typ$*, suppose we are given closed, well-typed atom binding values* $\emptyset \vdash_w \langle\!\langle a \rangle\!\rangle v : \tau\ \mathsf{bnd}$ *and* $\emptyset \vdash_w \langle\!\langle a' \rangle\!\rangle v' : \tau\ \mathsf{bnd}$*. If for some atom* $a'' \notin w$ *we have*

$$\vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a'\} : \tau \tag{18}$$

then

$$\vdash_w \langle\!\langle a \rangle\!\rangle v \cong \langle\!\langle a' \rangle\!\rangle v' : \tau\ \mathsf{bnd}\ . \tag{19}$$

Proof. Unlike the previous proposition, this result is not just a simple consequence of the congruence properties of operational equivalence. We establish it via a property analogous to (8) in the proof of Theorem 10, once again using an induction over the rules defining termination.

Let $\mathcal{E}$ be the closure under compatible refinement (Figure 7) of the pairs of closed atom binding values that we wish to show are operationally equivalent. In other words $\mathcal{E}$ is the expression relation inductively defined by the following two rules.

$$\frac{\begin{array}{c} a'' \notin w \supseteq atom(a, v, a', v') \\ \vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a'\} : \tau \end{array}}{\emptyset \vdash_w \langle\!\langle a \rangle\!\rangle v\ \mathcal{E}\ \langle\!\langle a' \rangle\!\rangle v' : \tau\ \mathsf{bnd}} \qquad \frac{\Gamma \vdash_w e\ \widehat{\mathcal{E}}\ e' : \tau}{\Gamma \vdash_w e\ \mathcal{E}\ e' : \tau}\ .$$

The analogue of property (8), with $\mathcal{E}$ replacing $\cong^*$, holds by induction on the derivation of $\langle \vec{a}, S, e \rangle\!\downarrow_n$ from the rules in Figure 5; the details of the induction can be found in the full version of this paper. If (18) holds, then by definition of $\mathcal{E}$ we have $\emptyset \vdash_w \langle\!\langle a \rangle\!\rangle v\ \mathcal{E}\ \langle\!\langle a' \rangle\!\rangle v' : \tau\ \mathsf{bnd}$; and $\emptyset \vdash_w S\ \widehat{\mathcal{E}}\ S : \tau \to \tau'$ always holds, by definition of the extension of $\mathcal{E}$ to a relation between frame stacks. So by the analogue of (8) for $\mathcal{E}$ we get

$$\langle \vec{a}, S, \langle\!\langle a \rangle\!\rangle v \rangle\!\downarrow \;\Rightarrow\; \langle \vec{a}, S, \langle\!\langle a' \rangle\!\rangle v' \rangle\!\downarrow$$

for any $\vec{a}$ with $atom(\vec{a}) \supseteq w$ and any suitably typed frame stack S. A symmetrical argument shows that the converse implication

holds. Thus (18) implies that $\langle\!\langle a \rangle\!\rangle v$ and $\langle\!\langle a' \rangle\!\rangle v'$ are operationally equivalent, as required. □

Next we need to prove the converse of the above proposition, namely that (19) implies (18) for any $a'' \notin w$. The difficulty is that in verifying (18) we have to consider the termination behaviour of $v\{a''/a\}$ and $v'\{a''/a'\}$ in all states $\vec{a}$ with $atom(\vec{a}) \supseteq w \cup \{a''\}$. The atom a'' may occur at *any* position in $\vec{a}$ and not necessarily at its right-hand end; whereas in assuming (19), all we appear to know about the termination behaviour of $v\{a''/a\}$ and $v'\{a''/a'\}$ is what happens when a fresh atom a'' is placed at the end of the state via generative unbinding (cf. Remark 6). In fact we are able to combine bind and unbind operations to rearrange atoms sufficiently to prove the result we want, but only in the presence of observations on atoms that are insensitive to atoms being added at the left-hand (that is, least) end of the state. The following definition makes this property of observations precise. It uses the notation $a' \circledcirc \vec{a}$ for the state obtained from $\vec{a} \in State$ by appending an atom a' not in $atom(\vec{a})$ to the *left* of the finite list of distinct atoms $\vec{a}$ (cf. $\vec{a} \circledcirc a'$ defined in Figure 4).

Definition 17 (Affine Observations). An observation on atoms, $\mathsf{obs} \in \mathcal{O}$, is *affine* if it is equivariant (5) and satisfies: for all $\vec{a} \in State$, all $a' \notin atom(\vec{a})$ and all $(a_1, \ldots, a_k) \in atom(\vec{a})^k$ (where k is the arity of obs)

$$[\![\mathsf{obs}]\!]_{a' \circledcirc \vec{a}}(a_1, \ldots, a_k) = [\![\mathsf{obs}]\!]_{\vec{a}}(a_1, \ldots, a_k)\ . \tag{20}$$

For example, of the observations defined in Figure 6, eq and lt are affine, whereas ord and card are not.

The following property of termination follows from its definition in Figures 4 and 5, using Corollary 5.

Lemma 18. *Given a frame stack S and an expression e, suppose that only affine observations on atoms occur in them. Then for all $\vec{a}$ with $atom(S, e) \subseteq atom(\vec{a})$ and all $a' \notin atom(\vec{a})$, $\langle a \circledcirc \vec{a}, S, e\rangle\downarrow_n \Leftrightarrow \langle \vec{a}, S, e\rangle\downarrow_n$.* □

Proposition 19. *Suppose that $\mathcal{O}$ only contains affine observations. Then for any type τ,* (19) *implies that* (18) *holds for any $a'' \notin w$.*

Proof. Suppose (19) holds and that $a'' \notin w$. We have to show for any $w' \in World$, $\vec{a} \in State$ and $\tau' \in Typ$ with $atom(\vec{a}) = w' \supseteq w \cup \{a''\}$ and $\emptyset \vdash_{w'} S : \tau \to \tau'$ that

$$\langle \vec{a}, S, v\{a''/a\}\rangle\downarrow \ \Leftrightarrow\ \langle \vec{a}, S, v'\{a''/a'\}\rangle\downarrow\ . \tag{21}$$

Since $a'' \in atom(\vec{a})$, we have

$$\vec{a} = \vec{a}' \circledcirc a'' \circledcirc a_0 \circledcirc \cdots \circledcirc a_{n-1} \tag{22}$$

for some state $\vec{a}'$ and atoms $a_0, \ldots, a_{n-1}$ ($n \geq 0$). Choose distinct atoms $b_0, \ldots, b_{n-1}$ not occurring in w' and consider the frame stack S' given by

$$\begin{array}{l} Id \circ (z.\, \mathsf{unbind}\ z\ \mathsf{as}\ (x\ ,\ y_0)\ \mathsf{in} \\ \qquad \mathsf{unbind}\ \langle\!\langle b_0 \rangle\!\rangle y_0\ \mathsf{as}\ (x_0\ ,\ y_1)\ \mathsf{in} \\ \qquad \vdots \\ \qquad \mathsf{unbind}\ \langle\!\langle b_{n-1} \rangle\!\rangle y_{n-1}\ \mathsf{as}\ (x_{n-1}\ ,\ y_n)\ \mathsf{in} \\ \qquad S\{x, x_0, \ldots, x_{n-1}/a'', a_0 \ldots, a_{n-1}\}[y_n]) \end{array} \tag{23}$$

where $z, x, x_0, \ldots, x_{n-1}, y_0, \ldots, y_n$ are distinct variables not occurring in S. Here we have used the notation

$$\begin{array}{rcl} \mathsf{unbind}\ v\ \mathsf{as}\ (x_1\ ,\ x_2)\ \mathsf{in}\ e & \triangleq & \mathsf{let}\ y = \mathsf{unbind}\ v\ \mathsf{in} \\ & & \mathsf{let}\ x_1 = \mathsf{fst}\ y\ \mathsf{in} \\ & & \mathsf{let}\ x_2 = \mathsf{snd}\ y\ \mathsf{in}\ e \end{array}$$

(where y does not occur in (v, x_1, x_2, e)); (23) also uses the notation "$S[e]$" from (12); and it uses the operation $(-)\{x/a\}$ of replacing an atom a by a variable x. Using Corollary 5 and property (11), a somewhat intricate calculation of transitions (given in more detail in the full version of this paper) yields

$$\langle \vec{b}', S', \pi \cdot \langle\!\langle a \rangle\!\rangle v\rangle\downarrow\ \Leftrightarrow\ \langle \vec{b}, S, v\{a''/a\}\rangle\downarrow \tag{24}$$

$$\text{and}\quad \langle \vec{b}', S', \pi \cdot \langle\!\langle a' \rangle\!\rangle v'\rangle\downarrow\ \Leftrightarrow\ \langle \vec{b}, S, v'\{a''/a'\}\rangle\downarrow \tag{25}$$

where $\vec{b} \triangleq b_0 \circledcirc \cdots \circledcirc b_{n-1} \circledcirc \vec{a}$, $\vec{b}' \triangleq b_0 \circledcirc \cdots \circledcirc b_{n-1} \circledcirc \vec{a}'$ and $\pi \in \mathbb{P}$ is the permutation swapping each a_i with b_i (for $i = 0..n-1$). We noted in Theorem 10 that operational equivalence is equivariant. So from (19) we have $\vdash_{atom(\vec{b}')} \pi \cdot \langle\!\langle a \rangle\!\rangle v \cong \pi \cdot \langle\!\langle a' \rangle\!\rangle v' : \tau\mathsf{bnd}$. Since $\emptyset \vdash_{atom(\vec{b}')} S' : \tau\,\mathsf{bnd} \to \tau'$, this operational equivalence gives

$$\langle \vec{b}', S', \pi \cdot \langle\!\langle a \rangle\!\rangle v\rangle\downarrow\ \Leftrightarrow\ \langle \vec{b}', S', \pi \cdot \langle\!\langle a' \rangle\!\rangle v'\rangle\downarrow\ .$$

Combining this with (24) and (25) yields

$$\langle \vec{b}, S, v\{a''/a\}\rangle\downarrow\ \Leftrightarrow\ \langle \vec{b}, S, v'\{a''/a'\}\rangle\downarrow\ . \tag{26}$$

Recall that $\vec{b} = b_0 \circledcirc \cdots \circledcirc b_{n-1} \circledcirc \vec{a}$ and $b_0, \ldots, b_{n-1} \notin w' = atom(\vec{a}) \supseteq atom(S, a'', v, v')$. So since $\mathcal{O}$ only contains affine observations, we can now apply Lemma 18 to (26) to get (21), as required. □

Example 20. We conjecture that Proposition 19 fails to hold if we drop the requirement that observations are affine (but still require them to be equivariant). For example consider the equivariant but non-affine observation ord in Figure 6 and the values

$$\begin{array}{rcl} v & \triangleq & \mathsf{fun}(f\ x = f\ x) \\ v' & \triangleq & \mathsf{fun}(f\ x = \mathsf{let}\ y = \mathsf{ord}\ a\ \mathsf{in} \\ & & \quad \mathsf{match}\ y\ \mathsf{with}\ (\mathsf{Zero} \to () \mid \mathsf{Succ}\, y' \to v())) \end{array}$$

where a is some atom. We claim that

$$\vdash_{\{a\}} \langle\!\langle a \rangle\!\rangle v \cong \langle\!\langle a \rangle\!\rangle v' : (\mathsf{unit} \to \mathsf{unit})\mathsf{bnd} \tag{27}$$

but that for any $a' \neq a$

$$\vdash_{\{a,a'\}} v\{a'/a\} \ncong v'\{a'/a\} : \mathsf{unit} \to \mathsf{unit}\ . \tag{28}$$

The operational inequivalence (28) is witnessed by the state $\vec{a} \triangleq [a', a]$ and the frame stack $S \triangleq Id \circ (x.\, x\,\mathsf{unit})$, for which one has $\langle \vec{a}, S, v'\{a'/a\}\rangle\downarrow$, but not $\langle \vec{a}, S, v\{a'/a\}\rangle\downarrow$. At the moment we lack a formal proof of the operational equivalence (27), but the intuitive justification for it is as follows. For any state $\vec{a}$ containing a and any frame stack S, we claim that in any sequence of transitions from $\langle \vec{a}, S, \langle\!\langle a \rangle\!\rangle v'\rangle$ the occurrence of $\mathsf{ord}\ a$ in v' can only be renamed to $\mathsf{ord}\ a'$ for atoms a' at positions strictly greater than 0 in the current state; and hence $\langle \vec{a}, S, \langle\!\langle a \rangle\!\rangle v'\rangle$ has the same termination properties as $\langle \vec{a}, S, \langle\!\langle a \rangle\!\rangle v\rangle$.

Proof of Theorem 14. One proves that $\vdash_w v =_\alpha v' : \sigma$ implies $\vdash_w v \cong v' : \sigma$ by induction on the the rules defining α-equivalence in Figure 8, using Propositions 15 and 16.

To prove the converse implication, first note that if $\emptyset \vdash v : \sigma$, then v contains no instances of observations $\mathsf{obs} \in \mathcal{O}$ (proof by induction on the structure of the nominal arity σ).[4] It follows from the definition of operational equivalence in Definition 7 that if $\vdash_w v \cong v' : \sigma$ holds for a language with observation set $\mathcal{O}$, it also holds for the sub-language with minimal observation set $\{\mathsf{eq}\}$. Thus it suffices to prove the implication $\vdash_w v \cong v' : \sigma \ \Rightarrow\ \vdash_w v =_\alpha v' : \sigma$ for this minimal sub-language; and this can be done by induction on the structure of σ using Propositions 15 and 19 (the latter applies because eq is affine). □

[4] The only way observations on atoms can appear in values of the language is via function values, $\mathsf{fun}(f\ x = e)$, and the definition of "nominal arity" excludes function types.

6. Related and Further Work

6.1 Correctness of Representation

It is instructive to compare the Correctness of Representation property of FreshML (Theorem 14) with *adequacy* results for type-theoretic logical frameworks [15]. Both are concerned with the representation of expressions of some object-language in a meta-language. For logical frameworks the main issue is surjectivity: one wants every expression at the meta-level to be convertible to a normal form and for every normal form at certain types to be the representation of some object-level expression. The fact that α-equivalence of object-level expressions is preserved and reflected by the representation is a simple matter, because equivalence in the logical framework is taken to be $\alpha\beta\eta$-conversion, which specialises on normal forms to just α-equivalence. Contrast this with the situation for FreshML where surjectivity of the representation is straightforward, because values of the relevant FreshML data types *are* just first order abstract syntax trees; whereas the fact that α-equivalence of object-level expressions is preserved and reflected by the representation in FreshML is a non-trivial property. This is because we take equivalence of FreshML expressions to be contextual equivalence. This is the natural notion of equivalence from a programming point of view, but its properties are hard won.

One aspect of adequacy results for logical frameworks highlighted in [15] is *compositionality* of representations. Although important, this issue is somewhat orthogonal to our concerns here. It refers to the question of whether substitution of expressions for variables at the object-level is represented by β-conversion at the meta-level. From the point of view of nominal signatures [19], variables are just one kind of name. Properties of α-conversion of all kinds of names are treated by the theory; but if one wants notions of substitution and β-conversion for a particular kind of name, one has to give a definition (an "α-structural" recursive definition [19]). For example in FreshML, using the data type (1) for λ-terms one can give an appealingly simple declaration for a function $\mathsf{subst} : \mathsf{term} \to \mathsf{atm} \to \mathsf{term} \to \mathsf{term}$ for capture avoiding substitution; see [26, p. 264]. Compositionality of the representation $t \mapsto \ulcorner t \urcorner$ given in the introduction then becomes the contextual equivalence $\vdash_w \ulcorner t_1[t_2/a] \urcorner \cong \mathsf{subst} \ulcorner t_2 \urcorner a \ulcorner t_1 \urcorner : \mathsf{term}$. The CIU theorem (Theorem 10) provides the basis for proving such contextual equivalences.[5]

6.2 Concrete Semantics

We have explored some of the consequences of adding integer-valued "observations on atoms" to FreshML over and above the usual test for equality. Such functions may allow more efficient data structures to be used for algorithms involving atoms as keys. For example, binary search trees making use of the comparison function lt from Figure 6 could be used instead of association lists.

What about adding functions from numbers to atoms? An implementation of the language may well represent atoms by numbers, via some fixed enumeration of the set of atoms, $\alpha : \mathbb{N} \cong \mathbb{A}$. Can we give the programmer access to this bijection? Less radically, can we allow operations on atoms that make use of arithmetic properties of the underlying representation? Not without breaking the invariant $atom(S, e) \subseteq atom(\vec{a})$ of configurations $\langle \vec{a}, S, e \rangle$—the property of our operational semantics that ensures that an atom's freshness with respect to the current state really does mean that it is different from all other atoms in the current context. For example, suppose we add to the language an operation $\mathsf{suc} : \mathsf{atm} \to \mathsf{atm}$ whose meaning is "successor function on atoms", with transitions $\langle \vec{a}, S, \mathsf{suc}\, a \rangle \longrightarrow \langle \vec{a}, S, a' \rangle$ whenever $a = \alpha(n)$ and $a' = \alpha(n+1)$ for some $n \in \mathbb{N}$. Then it may well be the case that $a' \notin atom(\vec{a})$ even though $a \in atom(\vec{a})$.

So exposing the numerical representation of atoms involves giving up the invariant properties of the abstract semantics we have used here. One motivation for studying this extension of the language is the fact that some algorithms make use of concrete numerical representations of keys, for example ones using hash tables or Patricia trees [14]. It may still be the case that the Correctness of Representation property holds for this extension, even though our equivariant proof techniques are no longer applicable. However, contextual equivalence for this language probably satisfies few useful laws. Perhaps a more interesting alternative to actually exposing numerical representations of atoms would be to prove contextual equivalence of efficient and naive implementations of the abstract semantics extended with types of finite maps on atoms. Such abstract types form an addition to the signature in Figure 1 different from the kind we have considered here, but certainly one worthy of investigation.

6.3 Mechanising Meta-Theory

The techniques we used here to prove the Correctness of Representation property are operationally based, in contrast to the denotational techniques used in [23, 25]. The advantage of working directly with the syntax and operational semantics of the language is that there are lower mathematical "overheads"—various kinds of induction being the main techniques involved. The disadvantage is that to deploy such inductive techniques often involves great ingenuity choosing inductive hypotheses and much error prone tedium checking induction steps. Furthermore, with these methods it seems harder to predict the effect that a slight change in language or formalisation may have on a proof. Although ingenuity in choosing inductive hypotheses may always be the preserve of humans, machine assistance of the kind envisaged by the "POPLmark challenge" [1] seems a very good idea for the other aspects of the operationally based approach. The main results presented here are still a challenging target for fully formalised and machine checked proofs. We have taken some care with the formalisation (using a "relational" approach to contextual equivalence, for example); but results concerning coinductive equivalences, like the CIU theorem (Theorem 10), are quite complex logically speaking, compared with the kind of type safety results (like Theorem 2) that POPLMark has focused on so far. Systems like Isabelle/HOL [13] that develop proofs in full classical higher order logic seem appropriate to the task, in principle. But there is a gap between what is possible in principle for an expert of any particular system and what is currently practicable for a casual user. Urban and Berghofer [27] are developing a *Nominal Data Type Package* for Isabelle/HOL that may be very useful for narrowing this gap. The fact that FreshML and the Urban-Berghofer package both have to do with the same mathematical universe of "nominal sets" [19] is perhaps slightly confusing: their Nominal Data Type Package is useful for fully formalising proofs about names and name-binding in operational semantics whether or not those proofs have to do with the particular mechanism of generative unbinding that is the focus of this paper.

7. Conclusion

The FreshML [26, 24] approach to functional programming with binders combines abstract types for names and name binding with an unbinding operation that involves generation of fresh names. In this paper we have studied the theoretical properties of this design. We showed that the addition of integer valued observations on names does not break FreshML's fundamental Correctness of Representation property that α-equivalence classes of abstract syntax trees (for any nominal signature) coincide with contextual equivalence classes of user declared data values. In particular, it is pos-

[5] We believe this particular equivalence is valid when $\mathcal{O} = \{\mathsf{eq}, \mathsf{lt}\}$, but not when $\mathcal{O} = \{\mathsf{eq}, \mathsf{card}\}$; cf. Section 7.

sible to give programmers access to a linear order on names without breaking the "up to α-equivalence" representation of syntax. The simple insight behind this possibly surprising result has to do with the fact that FreshML is impure—program execution mutates the state of dynamically created names.[6] If the state is taken into account when giving the meaning of observations on names, then the permutation invariance properties that underly the correctness property can be retained.

The only restriction we placed on observations is that, as functions of both the state and the names they operate upon, they should be invariant under permuting names. The Correctness of Representation property (Theorem 14) remains valid in the presence of any such observation. However, some observations are better behaved than others and we are not advocating that arbitrary equivariant observations be added to FreshML. Some forms of observation may radically affect the general programming laws that contextual equivalence satisfies. We saw one example of this here: only for "affine" observations (which are insensitive to how many names have been created before the arguments to which they are applied) were we able to combine Propositions 16 and 19 to get an "extensionality" result explaining contextual equivalence at type τ bnd in terms of contextual equivalence at τ, for arbitrary higher types τ.

The techniques we used to establish these results are of independent interest. They combine the usual engine of structural operational semantics—namely syntax-directed, rule based induction—with the approach to freshness of names based on name permutations that was introduced in [6] and developed in [17, 28, 19].

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanised metatheory for the masses: The POPLmark challenge. In *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

[2] P. N. Benton and X. Leroy, editors. *ACM SIGPLAN Workshop on ML (ML 2005), Tallinn, Estonia*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[3] J. Cheney. Scrap your nameplate (functional pearl). In *10th ACM SIGPLAN Int. Conference on Functional Programming (ICFP'05), Tallinn, Estonia*, pages 180–191. ACM Press, 2005.

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

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA*, pages 237–247. ACM Press, June 1993.

[6] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[7] A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In Gordon and Pitts [8], pages 9–54.

[8] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press, 1998.

[6] The original version of FreshML [20] was pure by dint of the "freshness inference" included in its type system. Subsequent experience with the language showed that the form of freshness inference that was used there was overly restrictive from a programming point of view. So freshness inference was dropped in [26]. However, Pottier [22] has recently regained purity in a FreshML-like language through the use of user-provided assertions. We have not investigated whether results like those presented in this paper also apply to Pottier's language.

[9] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[10] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[11] S. B. Lassen. Relational reasoning about contexts. In Gordon and Pitts [8], pages 91–135.

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

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[14] C. Okasaki and A. Gill. Fast mergeable integer maps. In *ACM-SIGPLAN Workshop on ML, Baltimore, Maryland, USA*, pages 77–86. ACM Press, 1998.

[15] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[16] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, 2000.

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

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

[19] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006.

[20] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *5th Int. Conference on Mathematics of Program Construction (MPC2000), Ponte de Lima, Portugal*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[21] F. Pottier. An overview of Cαml. In Benton and Leroy [2], pages 27–52.

[22] F. Pottier. Static name control for FreshML. Draft, July 2006.

[23] M. R. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, University of Cambridge Computer Laboratory, 2005. Available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-618.

[24] M. R. Shinwell. Fresh O'Caml: Nominal abstract syntax for the masses. In Benton and Leroy [2], pages 53–76.

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

[26] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *8th ACM SIGPLAN Int. Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, 2003.

[27] C. Urban and S. Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In *3rd International Joint Conference on Automated Reasoning (IJCAR 2006), Seattle, USA*, volume 4130 of *Lecture Notes in Computer Science*, pages 498–512. Springer-Verlag, 2006.

[28] C. Urban and M. Norrish. A formal treatment of the Barendregt Variable Convention in rule inductions. In *3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN '05), Tallinn, Estonia*, pages 25–32. ACM Press, 2005.

[29] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.