

## GENERATIVE UNBINDING OF NAMES\*

ANDREW M. PITTS<sup>a</sup> AND MARK R. SHINWELL<sup>b</sup>

<sup>a</sup> University of Cambridge Computer Laboratory, Cambridge CB3 0FD, UK  
*e-mail address:* Andrew.Pitts@cl.cam.ac.uk

<sup>b</sup> CodeSourcery, Ltd  
*e-mail address:* mark@three-tuns.net

---

**ABSTRACT.** This paper is concerned with the form of typed name binding 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. The paper proves a new result about what operations on names can co-exist with this construct. 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 alpha-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 alpha-equivalence precisely corresponds to contextual equivalence at the programming meta-level), so long as one takes the state of dynamically created names into account.

### 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 [PG00, SPG03] and has been realised in the Fresh patch of Objective Caml [Shi05b]. FreshML has also inspired Pottier’s Caml tool [Pot05] for Objective Caml and Cheney’s FreshLib library [Che05] 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  $\alpha$ -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 [Shi05a, SP05b] that its static and dynamic properties combine to guarantee a certain “correctness

---

*1998 ACM Subject Classification:* D.3.1, D.3.3, F.3.2.

*Key words and phrases:* Abstract syntax, binders, alpha-conversion, meta-programming.

\* This paper is a revised and expanded version of [PS07].

<sup>a</sup> Research supported by UK EPSRC grant EP/D000459/1.

```

type atm
type  $\alpha$  bnd
val fresh : unit  $\rightarrow$  atm
val bind : atm *  $\alpha$   $\rightarrow$   $\alpha$  bnd
val unbind :  $\alpha$  bnd  $\rightarrow$  atm *  $\alpha$ 
val (=) : atm  $\rightarrow$  atm  $\rightarrow$  bool

```

Figure 1: A signature for name binding.

of representation” property: data structures representing  $\alpha$ -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,  $\alpha$ -equivalence of name bindings is taken care of automatically by the programming language design.

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  `$\alpha$  bnd` classifying abstractions of single names over values of type  `$\alpha$` . To explain: both `atm` and  `$\alpha$  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  `$\tau$` , closed values of type  `$\tau$  bnd` are called *atom bindings* and are given by pairs  $\langle a \rangle v$  consisting of an atom  $a : \text{atm}$  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 avoid the use of pattern-matching and consider an equivalent mechanism for deconstructing atom binding via an `unbind` function carrying out generative unbinding: `unbind  $\langle a \rangle v$`  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  $\langle a \rangle v$`  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  $a'$ . Although implementing such an atom swapping operation on all types of values is the main extension that the Fresh patch makes to Objective Caml, we have not included a `swap : atm  $\rightarrow$  atm  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$`  operation in the signature of Figure 1. This is because it is possible for users to define atom swapping themselves for specific types on a case-by-case basis. Although this approach has some limitations, is enough for our purposes here. (The approach is more useful in the presence of Haskell-style type classes—see [Che05].)

The type  `$\alpha$  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  $\lambda$ -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

$$\begin{aligned} \text{type term} &= \text{V of atm} & (1.1) \\ &| \text{L of term bnd} \\ &| \text{A of term * term .} \end{aligned}$$

The value  $\ulcorner t \urcorner$  : term representing a  $\lambda$ -term  $t$  is defined by

$$\begin{aligned} \ulcorner a \urcorner &\triangleq \text{V } a \\ \ulcorner \lambda a.t \urcorner &\triangleq \text{L } \langle\langle a \rangle\rangle \ulcorner t \urcorner \\ \ulcorner t_1 t_2 \urcorner &\triangleq \text{A}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \end{aligned} \quad (1.2)$$

and satisfies:

**Correctness of Representation:** *two  $\lambda$ -terms are  $\alpha$ -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  $\lambda$ -term  $t$ , it follows that there is a bijection between  $\alpha$ -equivalence classes of  $\lambda$ -terms and contextual equivalence classes of closed values of type term. The Correctness of Representation 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 [Shi05a, SP05b] using denotational methods that take permutations of atoms into account. The same methods can be used to generalise from the example of  $\lambda$ -terms to terms over any *nominal signature* in the sense of [UPG04].

**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 of Representation property? For example it would be very useful to have a linear order ( $<$ ) : atm  $\rightarrow$  atm  $\rightarrow$  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 of Representation property given in [Shi05a, SP05b] 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 of Representation 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* (we explain what is reasonable in Section 3), *without invalidating the Correctness of Representation property for any nominal signature*.

We have to work quite hard to get this result, which generalises the one announced in [SPG03] (with a flawed proof sketch) and finally proved in [SP05b, Shi05a]; but whereas those works uses denotational techniques, here we use an arguably more direct approach based on the operational semantics of the language. We obtain the correctness result (Theorem 5.3) as a corollary of more general result (Propositions 5.7 and 5.10) showing that, up to contextual equivalence, the type  $\tau$  `bnd` behaves like the atom-abstraction construct of [GP01, Sect. 5]. Along the way to these results we prove a Mason-Talcott-style “CIU” [MT91] characterisation of contextual equivalence for our language (Theorem 4.4). This is proved using Howe’s method [How96] applied to a formulation of the operational semantics with Felleisen-style evaluation contexts [FH92], via an abstract machine with frame stacks [Pit02]. 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 [GP01] and developed in [Pit03, UN05, Pit06].

## 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.

**Variable binding.** The syntax of expressions and frame stacks in Figure 2 involves some variable-binding constructs. Specifically:

- free occurrences of  $f$  and  $x$  in  $e$  are bound in `fun( $f\ x = e$ )`;
- free occurrences of  $x$  in  $e$  are bound in `let  $x = e'$  in  $e$` ;
- for  $i = 1..n$ , free occurrences of  $x_i$  in  $e_i$  are bound in `match  $v$  with ( $C\ x_1 \rightarrow e_1 \mid \dots \mid C\ x_n \rightarrow e_n$ )`;
- free occurrences of  $x$  in  $e$  are bound in  `$S \circ (x.e)$` .

As usual, *we identify expressions and frame stacks up to renaming of bound variables*. We write  $\text{fv}(e)$  for the finite set of free variables of an expression  $e$  (and similarly for frame stacks); and we write

$$e[v, \dots / x, \dots] \tag{2.1}$$

for the simultaneous, capture avoiding substitution of values  $v, \dots$  for all free occurrences of the corresponding variables  $x, \dots$  in the expression  $e$  (well-defined up to  $\alpha$ -equivalence of bound variables).

**Reduced form.** The expressions in Figure 2 are given in a “reduced” form (also called “A-normal” form [FSDF93]), in which the order of evaluation is made explicit through `let`-expressions. This is not essential: the use of reduced form makes the development of properties of the language’s dynamics more succinct and that is mostly what we are concerned with here. However, when giving example expressions it is convenient to use the “unreduced” forms given in Figure 3.

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

Figure 2: Language syntax.

$$\begin{array}{ll}
(e, e') \triangleq \text{let } x = e \text{ in let } x' = e' \text{ in } (x, x') & (x \notin \text{fv}(e'), x' \neq x) \\
\lambda x. e \triangleq \text{fun}(f x = e) & (f \notin \text{fv}(e), f \neq x) \\
k e \triangleq \text{let } x = e \text{ in } k x & (k = \text{C}, \text{fst}, \text{snd}) \\
\langle\langle e \rangle\rangle e' \triangleq \text{let } x = e \text{ in let } x' = e' \text{ in } \langle\langle x \rangle\rangle x' & (x \notin \text{fv}(e'), x' \neq x) \\
e e' \triangleq \text{let } x = e \text{ in let } x' = e' \text{ in } x x' & (x \notin \text{fv}(e'), x' \neq x) \\
\text{match } e \text{ with } (\dots) \triangleq \text{let } x = e \text{ in match } x \text{ with } (\dots) & (x \notin \text{fv}(\dots)) \\
\text{if } e \text{ then } e' \text{ else } e'' \triangleq \text{match } e \text{ with} & (x \notin \text{fv}(e'')) \\
\quad (\text{Zero}() \rightarrow e' \mid \text{Succ } x \rightarrow e'') & \\
\text{fresh } x \text{ in } e \triangleq \text{let } x = \text{fresh}() \text{ in } e & \\
\text{let } \langle\langle x_1 \rangle\rangle x_2 = e \text{ in } e' \triangleq \text{let } x = e \text{ in} & (x, x' \notin \text{fv}(e')) \\
\quad \text{let } x' = \text{unbind } x \text{ in} & x' \neq x, x_1 \neq x_2 \\
\quad \text{let } x_1 = \text{fst } x' \text{ in} & \\
\quad \text{let } x_2 = \text{snd } x' \text{ in } e' & \\
\text{obs } e_1 \dots e_n \triangleq \text{let } x_1 = e_1 \text{ in} & (x_1, \dots, x_n \notin \text{fv}(e_1, \dots, e_n)) \\
\quad \dots & x_1, \dots, x_n \text{ distinct).} \\
\quad \text{let } x_n = e_n \text{ in obs } x_1 \dots x_n &
\end{array}$$

Figure 3: Some “unreduced” forms of expression.

**Remark 2.1 (Object-level binding).** 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 to represent names in the object-level languages that are being represented as data in this programming meta-language. In particular a value of the form  $\langle\langle a \rangle\rangle v$  is used to represent the object-level binding of a name  $a$  in the value  $v$ . However, note that there are no atom-binding constructs at the programming meta-level. The reader (especially one used to using lambda-abstraction to represent all forms of statically-scoped binding) may well ask why? Why cannot we factor out by  $\langle\langle \rangle\rangle$ -bound atoms and thereby trivialise (one half of) the Correctness of Representation result referred to in the Introduction? The reason is that it does not make semantic sense to try to regard  $\langle\langle a \rangle\rangle(-)$  as a form of meta-level binding and identify all expressions up to an  $\alpha$ -equivalence involving renaming  $\langle\langle \rangle\rangle$ -bound atoms. For example, if  $a$  and  $a'$  are two different atoms, such an  $\alpha$ -equivalence would identify  $\text{fun}(f x = \langle\langle a \rangle\rangle x)$  with  $\text{fun}(f x = \langle\langle a' \rangle\rangle x)$ . However, these are two semantically different values: they are not contextually equivalent in the sense discussed in Section 4. For example, the operational semantics described below gives observably different results (0 and 1 respectively) when we place the two expressions in the context

$$\text{let } \langle\langle x_1 \rangle\rangle x_2 = [-] a \text{ in eq } x_1 x_2$$

(where  $\text{eq} \in \mathcal{O}$  is the observation for atom-equality that we always assume is present—see Remark 3). The reason for this behaviour is that variables in FreshML-like languages stand for unknown values that may well involve atoms free at the object level. We may get capture of such atoms within the scope of an atom-binding  $\langle\langle a \rangle\rangle(-)$  during evaluation. In the example, we replaced the hole in  $[-] a$  with  $\text{fun}(f x = \langle\langle a \rangle\rangle x)$  and  $\text{fun}(f x = \langle\langle a' \rangle\rangle x)$  respectively, yielding expressions that evaluate to  $\langle\langle a \rangle\rangle a$  and  $\langle\langle a' \rangle\rangle a$ —the first involving capture and the second not; and such capturing substitution does not respect naive  $\alpha$ -equivalence. So the

$$\begin{array}{c}
 \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 * \tau_2} \quad \frac{\Gamma, f : \tau \rightarrow \tau', x : \tau \vdash e : \tau'}{\Gamma \vdash \text{fun}(f \ x = e) : \tau \rightarrow \tau'} \\
 \\
 \frac{C : \tau \rightarrow \delta \quad \Gamma \vdash v : \tau}{\Gamma \vdash C v : \delta} \quad \frac{a \in \mathbb{A}}{\Gamma \vdash a : \text{atm}} \quad \frac{\Gamma \vdash v_1 : \text{atm} \quad \Gamma \vdash v_2 : \tau}{\Gamma \vdash \langle\langle v_1 \rangle\rangle v_2 : \tau \text{ bnd}} \\
 \\
 \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'} \quad \frac{\Gamma \vdash v : \tau_1 * \tau_2}{\Gamma \vdash \text{fst } v : \tau_1} \quad \frac{\Gamma \vdash v : \tau_1 * \tau_2}{\Gamma \vdash \text{snd } v : \tau_2} \quad \frac{\Gamma \vdash v_1 : \tau \rightarrow \tau' \quad \Gamma \vdash v_2 : \tau}{\Gamma \vdash v_1 v_2 : \tau'} \\
 \\
 \frac{\delta = C_1 \text{ of } \tau_1 \mid \cdots \mid C_n \text{ of } \tau_n \quad \Gamma \vdash v : \delta \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \cdots \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \text{match } v \text{ with } (C_1 x_1 \rightarrow e_1 \mid \cdots \mid C_n x_n \rightarrow e_n) : \tau} \\
 \\
 \frac{}{\Gamma \vdash \text{fresh}() : \text{atm}} \quad \frac{\Gamma \vdash v : \tau \text{ bnd}}{\Gamma \vdash \text{unbind } v : \text{atm} * \tau} \quad \frac{\text{arity}(\text{obs}) = k \quad \Gamma \vdash v_1 : \text{atm} \cdots \Gamma \vdash v_k : \text{atm}}{\Gamma \vdash \text{obs } v_1 \dots v_k : \text{nat}} \\
 \\
 \frac{}{\Gamma \vdash \text{Id} : \tau \rightarrow \tau} \quad \frac{\Gamma, x : \tau \vdash e : \tau' \quad \Gamma \vdash S : \tau' \rightarrow \tau''}{\Gamma \vdash S \circ (x.e) : \tau \rightarrow \tau''}
 \end{array}$$

Notation:

- $\Gamma, x : \tau$  indicates the typing environment obtained by extending the finite partial function  $\Gamma$  by mapping a variable  $x$  to the type  $\tau$  (we always assume that  $x \notin \text{dom}(\Gamma)$ ).
- In the typing rule for match-expressions, the hypothesis “ $\delta = C_1 \text{ of } \tau_1 \mid \cdots \mid C_n \text{ of } \tau_n$ ” refers to the top-level data type declaration (2.2); in other words, the only constructors whose result type is  $\delta$  are  $C_1, \dots, C_n$  and  $\tau_i$  is the argument type of  $C_i$  (for  $i = 1..n$ ).

Figure 4: Typing relation.

relation of contextual equivalence that we define in Section 4 does not contain this naive  $\alpha$ -equivalence that identifies all (open or closed) expressions up to renaming of « $\rangle$ »-bound atoms.<sup>1</sup> However, we will show (Theorem 5.3) that when we restrict to closed expressions representing object-level languages, then contextual equivalence does contain (indeed, coincides with) this form of  $\alpha$ -equivalence: this is the correctness of representation result referred to in the Introduction.

**Data types and observations.** 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 in Section 3, by a finite set  $\mathcal{D}$  of *data type* symbols, and by a finite set  $\mathcal{C}$  of *constructor* symbols. Each constructor  $C \in \mathcal{C}$  is assumed to come with a type,  $C : \tau \rightarrow \delta$ , where  $\tau \in \text{Typ}$  and  $\delta \in \mathcal{D}$ . The choice of  $\mathcal{D}$ ,  $\mathcal{C}$  and this typing information constitutes an

<sup>1</sup>Since the problematic possibly-capturing substitution is part of the dynamics of FreshML, there remains the possibility that the end results in the dynamics of expression evaluation can be made more abstract by identifying them up to renaming bound atoms: see Remark 2.5. There are also less naive versions of object-level  $\alpha$ -equivalence that respect possibly-capturing substitution, such as the one developed in [UPG04] involving hypothetical judgements about freshness of atoms for variables; contextual equivalence and “contextual freshness” should form a model of this notion, but we do not pursue this here.

ML-style top-level declaration of some (possibly mutually recursive) data types:

$$\begin{aligned} \text{type } \delta_1 &= C_{1,1} \text{ of } \tau_{1,1} \mid \cdots \mid C_{1,n_1} \text{ of } \tau_{1,n_1} \\ &\vdots \\ \text{and } \delta_m &= C_{m,1} \text{ of } \tau_{m,1} \mid \cdots \mid C_{m,n_m} \text{ of } \tau_{m,n_m} . \end{aligned} \tag{2.2}$$

Here  $\delta_i$  (for  $i = 1..m$ ) are the distinct elements of the set  $\mathcal{D}$  of data type symbols and  $C_{i,j}$  (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  $C : \tau \rightarrow \delta$  that comes with each constructor, grouped by result types:  $\delta_i$  appears as the result type of precisely the constructors  $C_{i,1}, \dots, C_{i,n_i}$  and their argument types are  $\tau_{i,1}, \dots, \tau_{i,n_i}$ . For the moment we place no restriction on these types  $\tau_{i,j}$ : they can be any element of the set  $\text{Typ}$  whose grammar is given in Figure 2. However, when we consider representation of object-level languages up to  $\alpha$ -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. (The effect of admitting some other types of operation on atoms is discussed in Section 6.2.) So we assume  $\mathcal{D}$  always contains a distinguished data type  $\text{nat}$  for the type of natural numbers and that correspondingly  $\mathcal{C}$  contains constructors  $\text{Zero} : \text{unit} \rightarrow \text{nat}$  and  $\text{Succ} : \text{nat} \rightarrow \text{nat}$  for zero and successor. Each  $\text{obs} \in \mathcal{O}$  denotes a numerical function on atoms. We assume it comes with an *arity*, specifying the number of arguments it takes: so if  $\text{arity}(\text{obs}) = k$  and  $(v_1, \dots, v_k)$  is a  $k$ -tuple of values of type  $\text{atm}$ , then  $\text{obs } v_1 \dots v_k$  is an expression of type  $\text{nat}$ . The typing of the language's values, expressions and frame stacks takes place in the presence of typing environments,  $\Gamma$ , each assigning types to finitely many variables. The rules in Figure 4 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  $\text{obs} \in \mathcal{O}$  comes with a specified interpretation: the form this takes is discussed in Section 3.

**Example 2.2 (Swapping atoms).** Examples of programming in FreshML using its characteristic feature of generatively unbinding atom-binding values may be found in [SPG03, SP05a]. Another feature of FreshML, the operation of swapping atoms, has been left out of the grammar in Figure 2. However, as we mentioned in the introduction, there is a type-directed definition of swapping,  $\text{swap}_\tau : \text{atm} \rightarrow \text{atm} \rightarrow \tau \rightarrow \tau$ , for this language. For example, when  $\tau$  is the type  $\text{atm}$  of atoms we can make use of the observation  $\text{eq} \in \mathcal{O}$  for atom-equality that we always assume is present (see Remark 3) together with the abbreviations in Figure 3 and define

$$\text{swap}_{\text{atm}} \triangleq \lambda x. \lambda y. \lambda z. \text{if eq } z \ x \ \text{then } y \ \text{else if eq } z \ y \ \text{then } x \ \text{else } z. \tag{2.3}$$

At unit, product, function and atom-binding types we can make use of standard definitions of permutation action for these types of data (see [Pit06, Section 3], for example):

$$\text{swap}_{\text{unit}} \triangleq \lambda x. \lambda y. \lambda z. z \tag{2.4}$$

$$\text{swap}_{\tau_1 * \tau_2} \triangleq \lambda x. \lambda y. \lambda z. (\text{swap}_{\tau_1} x \ y \ (\text{fst } z), \text{swap}_{\tau_2} x \ y \ (\text{snd } z)) \tag{2.5}$$

$$\text{swap}_{\tau_1 \rightarrow \tau_2} \triangleq \lambda x. \lambda y. \lambda z. \lambda x_1. \text{swap}_{\tau_2} x \ y \ (z \ (\text{swap}_{\tau_1} x \ y \ x_1)) \tag{2.6}$$

$$\text{swap}_{\tau \text{ bnd}} \triangleq \lambda x. \lambda y. \lambda z. \text{let } z = \langle\langle z_1 \rangle\rangle z_2 \ \text{in } \langle\langle \text{swap}_{\text{atm}} x \ y \ z_1 \rangle\rangle (\text{swap}_\tau x \ y \ z_2). \tag{2.7}$$

At data types we have to make recursive definitions corresponding to the inductive nature of the data types. For example, if we assume that in addition to the data type  $\text{nat}$  for

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

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

Notation:

- $v\{a'/a\}$  is the result of replacing all occurrences of an atom  $a$  by an atom  $a'$  in the value  $v$ ;
- $\text{atom}(\_)$  is the finite set of all atoms occurring in  $\_$ ;
- $\vec{a} \otimes a'$  is the state obtained by appending an atom  $a'$  not in  $\text{atom}(\vec{a})$  to the right of the finite list of distinct atoms  $\vec{a}$ ;
- $\ulcorner m \urcorner$  is the the closed value of type  $\text{nat}$  corresponding to  $m \in \mathbb{N}$ :  $\ulcorner 0 \urcorner \triangleq \text{Zero}()$  and  $\ulcorner m+1 \urcorner \triangleq \text{Succ} \ulcorner m \urcorner$ ;
- $\llbracket \text{obs} \rrbracket$  is the meaning of  $\text{obs}$ : see Section 3.

Figure 5: Transition relation.

natural numbers we just have a data type term as in (1.1), then we can define

$$\text{swap}_{\text{nat}} \triangleq \lambda x. \lambda y. \text{fun}(f z = \text{match } z \text{ with } (\text{Zero}() \rightarrow \text{Zero}() \mid \text{Succ } z_1 \rightarrow \text{Succ}(f z_1))) \quad (2.8)$$

$$\begin{aligned}
 \text{swap}_{\text{term}} \triangleq \lambda x. \lambda y. \text{fun}(f z = \text{match } z \text{ with } (\text{V } z_1 \rightarrow \text{V}(\text{swap}_{\text{atm}} x y z_1) & \quad (2.9) \\
 \quad \mid \text{L } z_1 \rightarrow \text{let } \langle z_2 \rangle z_3 = z_1 \text{ in} & \\
 \quad \quad \text{L}(\langle \text{swap}_{\text{atm}} x y z_2 \rangle (f z_3)) & \\
 \quad \mid \text{A } z_1 \rightarrow \text{A}(f(\text{fst } z_1), f(\text{snd } z_1))) & ))).
 \end{aligned}$$

(The fact that values of type  $\text{nat}$  do not involve atoms means that the above systematic definition of  $\text{swap}_{\text{nat}}$  is in fact contextually equivalent to  $\lambda x. \lambda y. \lambda z. z$ .)

**Operational semantics.** The abstract machine that we use to define the language's dynamics has configurations of the form  $\langle \vec{a}, S, e \rangle$ . Here  $e$  is the expression to be evaluated,  $S$  is a stack of evaluation frames and  $\vec{a}$  is a finite list of distinct atoms that have been allocated so far. Figure 5 defines the transition relation between configurations that we use to give the language's operational semantics. 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  $\text{atom}(\_)$  mentioned in Figure 5, we write this condition as

$$\text{atom}(S, e) \subseteq \text{atom}(\vec{a}). \quad (2.10)$$

Theorem 2.4 shows that this property of configurations is invariant under transitions, as is well-typedness. Before stating this theorem we introduce some useful terminology.

**Definition 2.3 (Worlds).** A (*possible*) *world*  $w$  is just a finite subset of the the fixed set  $\mathbb{A}$  of atoms. We write  $\text{World}$  for the set of all worlds.

In what follows we will index various relations associated with the language we are considering by worlds  $w \in \text{World}$  that make explicit the atoms involved in the relation. Sometimes (as in the following theorem) this is merely a matter of notational convenience; world-indexing will be more crucial when we consider program equivalence: see Remark 4.7 below.

**Theorem 2.4 (Type Safety).** *Write  $\vdash_w \langle \vec{a}, S, e \rangle : \tau$  to mean that  $\text{atom}(S, e) \subseteq \text{atom}(\vec{a}) = w$  and that there is some type  $\tau'$  with  $\emptyset \vdash S : \tau' \rightarrow \tau$  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  $\text{atom}(\vec{a}') = w'$  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  $S = \text{Id}$  and  $e \in \text{Val}$ , or  $\langle \vec{a}, S, e \rangle \longrightarrow \langle \vec{a}', S', e' \rangle$  holds for some  $\vec{a}', S'$  and  $e'$ .*

*Proof.* The proof of these properties is routine and is omitted. □

**Remark 2.5 (Alternative operational semantics).** It is worth remarking that there are alternative approaches to representing object-level binding of a name  $a$  in a value  $v$  in FreshML-like languages. In the original paper on FreshML [PG00], the authors make a distinction between non-canonical expressions  $a.v$  for atom-binding and the “semantic values”  $\text{abs}(a, \text{val})$  to which they evaluate. That paper gives an operational semantics in the style of the Definition of Standard ML [MTHM97] in which programming language expressions are separate from semantic values. It is possible to identify such semantic values up to  $\alpha$ -equivalence of  $\text{abs}(a, -)$ -bound atoms without the kind of inconsistency illustrated in Remark 2.1. (Such semantic values in which  $\text{abs}(a, -)$  is a binder are used by Pottier [Pot07], albeit for first-order values.) However, this does not help to simplify the type of Correctness of Representation result in which we are interested here, because programs are written using expressions, not semantic values. For example, identifying semantic values in this way,  $\text{abs}(a, a)$  and  $\text{abs}(a', a')$  are identical and hence trivially contextually equivalent; however the expressions  $a.a$  and  $a'.a'$  (that here we write as  $\llbracket a \rrbracket a$  and  $\llbracket a' \rrbracket a'$ ) are not equal and there is something to be done to prove that they are contextually equivalent. In the operational semantics of [PG00] these expressions evaluate to the same semantic value up to  $\alpha$ -equivalence; so one would need to prove that contextual equivalence for that language contains “Kleene equivalence”—for example by proving a “CIU” theorem like our Theorem 4.4 below. So it is probably possible to develop the results of this paper using this slightly more abstract style of operational semantics with semantic values identified up to  $\alpha$ -equivalence of bound atoms. However our experience is that the style of operational semantics we use here, in which semantic values are identified with certain canonical expressions (but necessarily not identified up  $\alpha$ -equivalence of bound atoms, for the reasons discussed in Remark 2.1) leads to a simpler technical development overall.

$$\begin{array}{c}
 \boxed{\langle \vec{a}, S, e \rangle \downarrow_n \quad \langle \vec{a}, S, e \rangle \downarrow} \\
 \hline
 \langle \vec{a}, S, e \rangle \longrightarrow \langle \vec{a}', S', e' \rangle \quad \langle \vec{a}', S', e' \rangle \downarrow_n \quad \langle \vec{a}, S, e \rangle \downarrow_n \\
 \hline
 \langle \vec{a}, \text{Id}, v \rangle \downarrow_0 \quad \langle \vec{a}, S, e \rangle \downarrow_{n+1} \quad \langle \vec{a}, S, e \rangle \downarrow
 \end{array}$$

Figure 6: Termination relations.

### 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  $\text{obs} \in \mathcal{O}$  comes with a specified meaning  $\llbracket \text{obs} \rrbracket$ . 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  $\text{arity}(\text{obs}) = k$ , for each  $\vec{a} \in \text{State}$  we assume given a function  $\llbracket \text{obs} \rrbracket_{\vec{a}} : \text{atom}(\vec{a})^k \rightarrow \mathbb{N}$  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 5. Not every such family ( $\llbracket \text{obs} \rrbracket_{\vec{a}} \mid \vec{a} \in \text{State}$ ) 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 3.1 (Permutations).** A finite *permutation* of atoms is a bijection  $\pi$  from the set  $\mathbb{A}$  of atoms onto itself such that  $\text{supp}(\pi) \triangleq \{a \in \mathbb{A} \mid \pi(a) \neq a\}$  is a finite set. We write  $\mathbb{P}$  for the set of all such permutations. If  $\pi \in \mathbb{P}$  and  $\vec{a} \in \text{State}$ , then  $\pi \cdot \vec{a}$  denotes the finite list of distinct atoms obtained by mapping  $\pi$  over the list  $\vec{a}$ ; if  $e$  is an expression, then  $\pi \cdot e$  denotes the expression obtained from it by applying  $\pi$  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 ( $\llbracket \text{obs} \rrbracket_{\vec{a}} \mid \vec{a} \in \text{State}$ ) associated with each  $\text{obs} \in \mathcal{O}$  to satisfy an *equivariance* property: for all  $\pi \in \mathbb{P}$ ,  $\vec{a} \in \text{State}$  and  $(a_1, \dots, a_k) \in \text{atom}(\vec{a})^k$  (where  $k$  is the arity of  $\text{obs}$ )

$$\llbracket \text{obs} \rrbracket_{\vec{a}}(a_1, \dots, a_k) = \llbracket \text{obs} \rrbracket_{\pi \cdot \vec{a}}(\pi(a_1), \dots, \pi(a_k)). \quad (3.1)$$

We impose condition (3.1) for the following reason. In Figure 5, the side conditions on transitions of types 7 and 8 do not specify which of the infinitely many atoms in  $\mathbb{A} - \text{atom}(\vec{a})$  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. So properties of configurations should be equivariant: if  $\langle \vec{a}, S, e \rangle$  has the property, then so should  $\langle \pi \cdot \vec{a}, \pi \cdot S, \pi \cdot e \rangle$  for any  $\pi \in \mathbb{P}$ . The main property of configurations we need is *termination*, defined in Figure 6, since as we see in the next section this determines contextual equivalence of expressions. With condition (3.1) we have:

**Lemma 3.2.** *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 6, it suffices to show that the transition relation is equivariant:

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

$$\begin{aligned}
& \text{Equality, eq (arity = 2):} \\
\llbracket \text{eq} \rrbracket_{\vec{a}}(a, a') & \triangleq \begin{cases} 0 & \text{if } a = a', \\ 1 & \text{otherwise.} \end{cases} \\
& \text{Linear order, lt (arity = 2):} \\
\llbracket \text{lt} \rrbracket_{\vec{a}}(a, a') & \triangleq \begin{cases} 0 & \text{if } a \text{ occurs to the left of } a' \text{ in the list } \vec{a}, \\ 1 & \text{otherwise.} \end{cases} \\
& \text{Ordinal, ord (arity = 1):} \\
\llbracket \text{ord} \rrbracket_{\vec{a}}(a) & \triangleq n, \text{ if } a \text{ is the } n\text{th element of the list } \vec{a}. \\
& \text{State size, card (arity = 0):} \\
\llbracket \text{card} \rrbracket_{\vec{a}}() & \triangleq \text{length of the list } \vec{a}.
\end{aligned}$$

Figure 7: Examples of observations on atoms.

This can be proved by cases from the definition of  $\longrightarrow$  in Fig 5. 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 (3.1).  $\square$

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 3.3.** *If  $\langle \vec{a}, S, \text{fresh} \rangle \downarrow_{n+1}$  with  $\text{atom}(S) \subseteq \text{atom}(\vec{a})$ , then for all  $a' \notin \text{atom}(\vec{a})$ , it is the case that  $\langle \vec{a} \otimes a', S, a' \rangle \downarrow_n$ . Similarly, if  $\langle \vec{a}, S, \text{unbind} \langle a \rangle v \rangle \downarrow_{n+1}$  with  $\text{atom}(S, a, v) \subseteq \text{atom}(\vec{a})$ , then for all  $a' \notin \text{atom}(\vec{a})$ , it is the case that  $\langle \vec{a} \otimes a', S, (a', v\{a'/a\}) \rangle \downarrow_n$ .  $\square$*

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  $\llbracket \text{obs} \rrbracket_{\vec{a}}(a) = \alpha^{-1}(a)$  fails to satisfy (3.1). Nevertheless, there is a wide range of functions that do have this property. Figure 7 gives some examples.

**Remark 3.4 (Atom-equality test).** The first observation on atoms given in Figure 7, eq, combined with the usual arithmetic operations for nat that are already definable in the language, gives us the effect of the function  $(=) : \text{atm} \rightarrow \text{atm} \rightarrow \text{bool}$  from the signature in Figure 1; so we assume that the set  $\mathcal{O}$  of observations on atoms always contains eq.

**Remark 3.5 (Fresh Atoms Largest).** Note that in the operational semantics of Figure 5 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 7).

## 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  $\langle \vec{a}, \text{Id}, v \rangle$ . 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 4.5 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 4.1 (Operational Equivalence of Closed Expressions).**  $\vdash_w e \cong e' : \tau$  is defined to hold if

- $\text{atom}(e, e') \subseteq w$ ;
- $\emptyset \vdash e : \tau$  and  $\emptyset \vdash e' : \tau$ ; and
- for all  $\vec{a}$ ,  $S$  and  $\tau'$  with  $w \cup \text{atom}(S) \subseteq \text{atom}(\vec{a})$  and  $\emptyset \vdash S : \tau \rightarrow \tau'$ , 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 [Pit05, 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 [Gor98] and Lassen [Las98] that 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 4.1*. A congruence relation is an expression relation that is an equivalence, compatible and substitutive, in the following sense.

**Definition 4.2 (Expression Relations).** An *expression relation*  $\mathcal{E}$  is a set of tuples  $(\Gamma, w, e, e', \tau)$  (made up of a typing context, a world, two expressions and a type) satisfying  $\text{atom}(e, e') \subseteq w$ ,  $\Gamma \vdash e : \tau$  and  $\Gamma \vdash e' : \tau$ . We write  $\Gamma \vdash_w e \mathcal{E} e' : \tau$  to indicate that  $(\Gamma, w, e, e', \tau)$  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 ( $\text{atom}(e) \subseteq w \wedge \Gamma \vdash e : \tau \Rightarrow \Gamma \vdash_w e \mathcal{E} e : \tau$ ), 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 8.

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\Gamma \vdash_w x \widehat{\mathcal{E}} x : \tau} \quad \frac{}{\Gamma \vdash_w () \widehat{\mathcal{E}} () : \text{unit}} \quad \frac{\Gamma \vdash_w v_1 \mathcal{E} v'_1 : \tau_1 \quad \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 \rightarrow \tau', x : \tau \vdash_w e \mathcal{E} e' : \tau'}{\Gamma \vdash_w \text{fun}(f x = e) \widehat{\mathcal{E}} \text{fun}(f x = e') : \tau \rightarrow \tau'} \quad \frac{C : \tau \rightarrow \delta \quad \Gamma \vdash_w v \mathcal{E} v' : \tau}{\Gamma \vdash_w C v \widehat{\mathcal{E}} C v' : \delta} \quad \frac{a \in w}{\Gamma \vdash_w a \widehat{\mathcal{E}} a : \text{atm}} \\
\\
\frac{\Gamma \vdash_w v_1 \mathcal{E} v'_1 : \text{atm} \quad \Gamma \vdash_w v_2 \mathcal{E} v'_2 : \tau}{\Gamma \vdash_w \langle v_1 \rangle v_2 \widehat{\mathcal{E}} \langle v'_1 \rangle v'_2 : \tau \text{ bnd}} \quad \frac{\Gamma \vdash_w e_1 \mathcal{E} e'_1 : \tau \quad \Gamma, x : \tau \vdash_w e_2 \mathcal{E} e'_2 : \tau'}{\Gamma \vdash_w \text{let } x = e_1 \text{ in } e_2 \widehat{\mathcal{E}} \text{let } x = e'_1 \text{ in } e'_2 : \tau'} \\
\\
\frac{\Gamma \vdash_w v \mathcal{E} v' : \tau_1 * \tau_2}{\Gamma \vdash_w \text{fst } v \widehat{\mathcal{E}} \text{fst } v' : \tau_1} \quad \frac{\Gamma \vdash_w v \mathcal{E} v' : \tau_1 * \tau_2}{\Gamma \vdash_w \text{snd } v \widehat{\mathcal{E}} \text{snd } v' : \tau_2} \quad \frac{\Gamma \vdash_w v_1 \mathcal{E} v'_1 : \tau \rightarrow \tau' \quad \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{\Gamma \vdash_w v \mathcal{E} v' : \delta \quad \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}{\Gamma \vdash_w \text{match } v \text{ with } (C_1 x_1 \rightarrow e_1 \mid \cdots \mid C_n x_n \rightarrow e_n) \widehat{\mathcal{E}} \text{match } v' \text{ with } (C_1 x_1 \rightarrow e'_1 \mid \cdots \mid C_n x_n \rightarrow e'_n) : \tau} \\
\text{where } \delta = C_1 \text{ of } \tau_1 \mid \cdots \mid C_n \text{ of } \tau_n \\
\\
\frac{}{\Gamma \vdash_w \text{fresh}() \widehat{\mathcal{E}} \text{fresh}() : \text{atm}} \quad \frac{\Gamma \vdash_w v \mathcal{E} v' : \tau \text{ bnd}}{\Gamma \vdash_w \text{unbind } v \widehat{\mathcal{E}} \text{unbind } v' : \text{atm} * \tau} \\
\\
\frac{\text{arity}(\text{obs}) = k \quad \Gamma \vdash_w v_1 \mathcal{E} v'_1 : \text{atm} \cdots \Gamma \vdash_w v_k \mathcal{E} v'_k : \text{atm}}{\Gamma \vdash_w \text{obs } v_1 \dots v_k \widehat{\mathcal{E}} \text{obs } v'_1 \dots v'_k : \text{nat}} \\
\\
\frac{}{\Gamma \vdash_w \text{Id } \widehat{\mathcal{E}} \text{Id} : \tau \rightarrow \tau} \quad \frac{\Gamma, x : \tau \vdash_w e \mathcal{E} e' : \tau' \quad \Gamma \vdash_w S \widehat{\mathcal{E}} S' : \tau' \rightarrow \tau''}{\Gamma \vdash_w S \circ (x.e) \widehat{\mathcal{E}} S' \circ (x.e') : \tau \rightarrow \tau''}
\end{array}$$

Figure 8: Compatible refinement  $\widehat{\mathcal{E}}$  of an expression relation  $\mathcal{E}$ .

- $\mathcal{E}$  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  $\emptyset \vdash_w e \mathcal{E} e' : \tau \Rightarrow \vdash_w e \cong e' : \tau$ .

We extend operational equivalence (Definition 4.1) to an expression relation,  $\Gamma \vdash_w e \cong^\circ e' : \tau$ , by instantiating free variables with closed values:

**Definition 4.3** ( $\cong^\circ$ ). Supposing  $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ , we define  $\Gamma \vdash_w e \cong^\circ e' : \tau$  to hold if

- $\text{atom}(e, e') \subseteq w$ ;
- $\Gamma \vdash e : \tau$  and  $\Gamma \vdash e' : \tau$ ; and
- for all  $w' \supseteq w$  and all closed values  $v_i$  with  $\text{atom}(v_i) \subseteq w'$  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  $\cong$ :

$$\emptyset \vdash_w e \cong^\circ e' : \tau \Leftrightarrow \vdash_w e \cong e' : \tau. \quad (4.1)$$

**Theorem 4.4 (CIU).** *Operational equivalence of possibly open expressions,  $\cong^\circ$ , is a compatible, substitutive and adequate equivalence. It is the largest such expression relation. It is also equivariant.*

*Proof.* The fact that  $\cong^\circ$  is equivariant follows from Lemma 3.2. The fact that it is an equivalence and adequate is immediate from its definition; as is the fact that it contains any expression relation that is adequate, substitutive and reflexive. So the main difficulty is to show that it is compatible and substitutive. One can do this by adapting a construction due to Howe [How96]; see Appendix A.  $\square$

**Definition 4.5 (Contextual Equivalence).** In view of the discussion at the beginning of this section, Theorem 4.4 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*.

**Remark 4.6 (Uses of closed instantiations).** We labelled the above theorem “CIU” because it is analogous to a theorem of that name due to Mason and Talcott [MT91]. 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* [FH92]. 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

$$\langle \vec{a}, S, e \rangle \downarrow \Leftrightarrow \langle \vec{a}, \text{Id}, S[e] \rangle \downarrow \quad (4.2)$$

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

$$\begin{aligned} \text{Id}[e] &\triangleq e \\ S \circ (x.e')[e] &\triangleq S[\text{let } x = e \text{ in } e'] . \end{aligned} \quad (4.3)$$

Theorem 4.4 serves to establish some basic properties of contextual equivalence, such as the fact that the state-independent transitions in Figure 5 (types 1–6 and 9) give rise to contextual equivalences. For example,  $\Gamma \vdash_w \text{let } x = v \text{ in } e \cong^\circ e[v/x] : \tau'$  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,  $\tau \text{ bnd}$ . We address this in the next section.

**Remark 4.7 (Possible Worlds).** It is immediate from the definition of  $\cong^\circ$  that it satisfies a weakening property:

$$\Gamma \vdash_w e \cong^\circ e' : \tau \wedge w \subseteq w' \Rightarrow \Gamma \vdash_{w'} e \cong^\circ e' : \tau . \quad (4.4)$$

If it also satisfied a strengthening property

$$\Gamma \vdash_{w'} e \cong^\circ e' : \tau \wedge \text{atom}(e, e') \subseteq w \subseteq w' \Rightarrow \Gamma \vdash_w e \cong^\circ e' : \tau \quad (4.5)$$

then we could make the indexing of contextual equivalence by “possible worlds”  $w$  implicit by taking  $w = \text{atom}(e, e')$ . When  $\mathcal{O}$  just contains eq, property (4.5) does hold; this is why there is no need for indexing by possible worlds in [Shi05a, SP05b]. However, it is not hard to see that the presence of some observations on atoms, such as the function card in Figure 7, can cause (4.5) to fail. It is for this reason that we have built indexing by possible worlds into expression relations (Definition 4.2).

$$\begin{array}{c}
\frac{}{\vdash_w () =_\alpha () : \text{unit}} \qquad \frac{\vdash_w v_1 =_\alpha v'_1 : \sigma_1 \quad \vdash_w v_2 =_\alpha v'_2 : \sigma_2}{\vdash_w (v_1, v_2) =_\alpha (v'_1, v'_2) : \sigma_1 * \sigma_2} \qquad \frac{C : \sigma \rightarrow \delta \quad \vdash_w v =_\alpha v' : \sigma}{\vdash_w C v =_\alpha C v' : \delta} \\
\\
\frac{a \in w}{\vdash_w a =_\alpha a : \text{atm}} \qquad \frac{a'' \notin w \supseteq \text{atom}(a, v, a', v') \quad \vdash_{w \cup \{a''\}} v \{a''/a\} =_\alpha v' \{a''/a'\} : \sigma}{\vdash_w \langle\langle a \rangle\rangle v =_\alpha \langle\langle a' \rangle\rangle v' : \sigma \text{ bnd}}
\end{array}$$

Figure 9:  $\alpha$ -Equivalence.

## 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:

$$\begin{array}{l}
\text{type } \delta_1 = C_{1,1} \text{ of } \tau_{1,1} \mid \cdots \mid C_{1,n_1} \text{ of } \tau_{1,n_1} \\
\quad \vdots \\
\text{and } \delta_m = C_{m,1} \text{ of } \tau_{m,1} \mid \cdots \mid C_{m,n_m} \text{ of } \tau_{m,n_m} .
\end{array} \tag{5.1}$$

If we restrict attention to declarations in which the argument types  $\tau_{i,j}$  of the constructors  $C_{i,j}$  are just finite products of the declared data types  $\delta_1 \dots, \delta_m$ , 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 [UPG04] and more fully developed in [Pit06]. 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 5.1 (Nominal Signatures).** The subset  $\text{Arity} \subseteq \text{Typ}$  is given by the grammar

$$\sigma \in \text{Arity} ::= \text{unit} \mid \sigma * \sigma \mid \delta \mid \text{atm} \mid \sigma \text{ bnd} \tag{5.2}$$

where  $\delta$  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,  $\rightarrow$ .) The elements of the set Arity are called *nominal arities*. (The notation  $\langle\langle \text{atm} \rangle\rangle \sigma$  is used in [UPG04, Pit06] for what we here write as  $\sigma \text{ bnd}$ .) A *nominal signature* with a single sort of atoms, atm, is specified by a data type declaration (5.1) in which the argument types  $\tau_{i,j}$  of the constructors  $C_{i,j}$  are all nominal arities.

The occurrences of  $\sigma \text{ bnd}$  in a nominal signature (5.1) indicate arguments with bound atoms. In particular, we can associate with each such signature a notion of  $\alpha$ -equivalence,  $=_\alpha$ , that identifies closed values of nominal arity up to renaming bound atoms. The inductive definition of  $=_\alpha$  is given in Figure 9. It generalises to an arbitrary nominal signature the syntax-directed characterisation of  $\alpha$ -equivalence of  $\lambda$ -terms given in [Gun92, p. 36]. The definition in Figure 9 is essentially that given in [Pit06], 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 “ $a'' \notin w \supseteq \text{atom}(a, v, a', v')$ ” in the rule for  $\alpha$ -equivalence of values of atom binding type would be replaced by “ $a'' \notin \text{atom}(a, v, a', v')$ ”.

**Remark 5.2 (The role of closed values).** For each  $\sigma \in \text{Arity}$ , the *closed* values (that is, ones with no free variables) of that type,  $\emptyset \vdash_w v : \sigma$ , correspond precisely to the ground

terms (with arity  $\sigma$  and atoms in  $w$ ) over the given nominal signature, as defined in [UPG04]. For example, the declaration (1.1) corresponds to the nominal signature for  $\lambda$ -calculus; and closed values of type term correspond as in (1.2) to the abstract syntax trees for  $\lambda$ -terms—open or closed ones, with  $\lambda$ -calculus variables represented by atoms. For other examples of nominal signatures, with more complicated patterns of binding, see [Pit06, Section 2.2].

Note that the definition of  $=_\alpha$  in Figure 9 cannot be extended naively to *open* values with free variables, for the reasons discussed in Remark 2.1. Free variables stand for unknown values that may well involve atoms that get captured by « $\gg$ »-binders upon substitution. So as we saw in that remark, it does not make semantic sense to say, for example, that « $a$ » $x$  and « $a$ » $x$  are  $\alpha$ -equivalent without putting some restrictions on the kind of value  $x$  stands for. In [UPG04], Urban *et al* consider such restrictions consisting of assumptions about the freshness of atoms for variables; they generalise Figure 9 to a hypothetical notion of  $\alpha$ -equivalence between open values<sup>2</sup>, with hypotheses consisting of such freshness assumptions. It may be possible to relate the validity of this general form of  $\alpha$ -equivalence to contextual equivalence, but here we content ourselves with the following result about the straightforward notion of  $\alpha$ -equivalence on closed values given by Figure 9.

**Theorem 5.3 (Correctness of Representation).** *Suppose that all the observations on atoms  $\text{obs}$  in  $\mathcal{O}$  satisfy the equivariance property (3.1). For each nominal signature, two closed values  $v, v'$  of the same nominal arity  $\sigma$  (with atoms contained in the finite set  $w$ , say) are  $\alpha$ -equivalent if and only if they are contextually equivalent:*

$$\vdash_w v =_\alpha v' : \sigma \Leftrightarrow \vdash_w v \cong v' : \sigma. \quad (5.3)$$

The rest of this section is devoted to the proof of the bi-implication in (5.3). Before commencing the proof we make some remarks about the relative difficulty of each half of the bi-implication and about alternative approaches to the proof than the one we take.

**Remark 5.4** ( $\vdash_w v =_\alpha v' : \sigma \Rightarrow \vdash_w v \cong v' : \sigma$ ). At first sight it might seem that this implication is trivial: since we identify expressions up to  $\alpha$ -equivalence of bound variables, contextual equivalence automatically contains that notion of equivalence. However,  $=_\alpha$  is not that meta-level  $\alpha$ -equivalence, it is  $\alpha$ -equivalence at the object-level for « $\gg$ »-bound atoms. As we noted in Remark 2.1, identifying all (open or closed) expressions up to renaming « $\gg$ »-bound atoms is incompatible with contextual equivalence: so we cannot trivialise the left-to-right implication in (5.3) by factoring out in this way. Note that the restriction to nominal arities in Figure 9 means that we do not have to consider  $=_\alpha$  for values of the form  $\text{fun}(f x = e)$  and hence for open expressions  $e$  where the naive definition of  $=_\alpha$  would encounter the semantic problems discussed in Remarks 2.1 and 5.2.

So there really is something to do to establish the left-to-right implication in (5.3). However, we will see that we have already done most of the heavy lifting for this half of the theorem by establishing the CIU Theorem 4.4.

**Remark 5.5** ( $\vdash_w v \cong v' : \sigma \Rightarrow \vdash_w v =_\alpha v' : \sigma$ ). This is equivalent to showing that if two closed values  $v$  and  $v'$  of nominal arity  $\sigma$  are not  $\alpha$ -equivalent, then they are not contextually equivalent. Proving contextual inequivalence is much easier than proving contextual equivalence, since one just has to construct a context in which the two values have different operational behaviour. In this case it would suffice to exhibit a closed expression  $\text{aeq}_\sigma : \sigma \rightarrow \sigma \rightarrow \text{nat}$  correctly implementing  $=_\alpha$ , in the sense that for all  $v$  and  $v'$

<sup>2</sup>This is a slight over-simplification, since their “nominal terms” are not just the open values considered here: they involved explicit atom-permutations as well.

$$\begin{aligned} \vdash_w v =_\alpha v' : \sigma &\Rightarrow \forall \vec{a}. w \subseteq \text{atom}(\vec{a}) \Rightarrow \exists \vec{a}'. \langle \vec{a}, \text{Id}, \text{aeq}_\sigma v v' \rangle \longrightarrow^* \langle \vec{a}', \text{Id}, \text{Zero}() \rangle \\ \vdash_w v \neq_\alpha v' : \sigma &\Rightarrow \forall \vec{a}. w \subseteq \text{atom}(\vec{a}) \Rightarrow \exists \vec{a}'. \langle \vec{a}, \text{Id}, \text{aeq}_\sigma v v' \rangle \longrightarrow^* \langle \vec{a}', \text{Id}, \text{Succ}(\text{Zero}()) \rangle. \end{aligned}$$

It is indeed possible to construct such an expression  $\text{aeq}_\sigma$  by induction on the structure of  $\sigma$ , by a definition that mimics the rules in Figure 9, using the definition of atom-swapping from Example 2.2 in the case of an atom-binding arity and using recursively defined functions at data types. The proof of the above properties of  $\text{aeq}_\sigma$  is relatively straightforward if tedious; one first has to prove suitable correctness properties for the swapping expressions  $\text{swap}_\sigma$  from Example 2.2.

This is not the route to the right-to-left implication in (5.3) that we take. Instead we deduce it from a general “extensionality” property of atom-binding types  $\tau \text{ bind}$  that holds for all types  $\tau$ , including ones that are not nominal arities, that is, ones involving function types. This property (Propositions 5.7 and 5.10) shows that, up to contextual equivalence, the type  $\tau \text{ bind}$  behaves like the atom-abstraction construct of [GP01, Sect. 5]. It seems interesting in its own right. We are able to prove this property of general atom-binding types  $\tau \text{ bind}$  only under a restriction on observations on atoms over and above the equivariance property (3.1) that we always assume they possess. This is the “affineness” property given in Definition 5.8 below. The equality test  $\text{eq}$  (Figure 7) is affine and we will see that this fact is enough to prove Theorem 5.3 as stated, that is, without any restriction on the observations present other than equivariance.

We now begin the proof of Theorem 5.3.

**Proposition 5.6.**

- (i)  $\vdash_w () \cong () : \text{unit}$ .
- (ii) For all types  $\tau_1, \tau_2 \in \text{Typ}$ ,  $\vdash_w (v_1, v_2) \cong (v'_1, v'_2) : \tau_1 * \tau_2$  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  $\delta_i$  in the declaration (5.1),  $\vdash_w C_{i,j} v \cong C_{i,j'} v' : \delta_i$  iff  $j = j'$  and  $\vdash_w v \cong v' : \tau_{i,j}$ .
- (iv)  $\vdash_w a \cong a' : \text{atm}$  iff  $a = a' \in w$ .

*Proof.* Part (i) and the “if” directions of (ii)–(iv) are consequences of the fact (Theorem 4.4) 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  $\vdash_w \text{fst}(v_1, v_2) \cong v_1 : \tau_1$  and  $\vdash_w \text{snd}(v_1, v_2) \cong v_2 : \tau_2$  that are consequences of the definitions of  $\cong$  and the termination relation. Similarly, part (iii) follows from the easily established operational (in)equivalences

$$\begin{aligned} \vdash_w \text{diverge} \not\cong v : \tau \\ \vdash_w \text{proj}_{i,j}(C_{i,j} v) \cong v : \tau_{i,j} \\ \vdash_w \text{proj}_{i,j}(C_{i,j'} v) \cong \text{diverge} : \tau_{i,j} \quad \text{if } j \neq j' \end{aligned}$$

which make use of the following expressions

$$\begin{aligned} \text{diverge} &\triangleq \text{fun}(f x = f x)() \\ \text{proj}_{i,j} v &\triangleq \text{match } v \text{ with } (C_{i,1} x_1 \rightarrow d_{j,1} \mid \dots \mid C_{i,n_i} x_{n_i} \rightarrow d_{j,n_i}) \end{aligned}$$

where

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

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 7: see Lemma A.4(i) in Appendix A.  $\square$

This proposition tells us that  $\cong$  has properties mirroring those of  $\alpha$ -equivalence given by the first four rules in Figure 9. To complete the proof of the correctness theorem, we need to prove a property of  $\cong$  at atom binding arities  $\sigma$  bnd that mirrors the fifth rule in that figure. We split this into two parts, Propositions 5.7 and 5.10.

**Proposition 5.7.** *For any type  $\tau \in \text{Typ}$ , suppose we are given closed, well-typed atom binding values  $\emptyset \vdash_w \langle a \rangle v : \tau$  bnd and  $\emptyset \vdash_w \langle a' \rangle v' : \tau$  bnd. If for some atom  $a'' \notin w$  we have*

$$\vdash_{w \cup \{a''\}} v \{a''/a\} \cong v' \{a''/a'\} : \tau \quad (5.4)$$

then

$$\vdash_w \langle a \rangle v \cong \langle a' \rangle v' : \tau \text{ bnd} . \quad (5.5)$$

*Proof.* Unlike the previous proposition, this result is not just a simple consequence of the congruence properties of operational equivalence. It can be proved via an induction over the rules defining termination: see Appendix B.  $\square$

Next we need to prove the converse of the above proposition, namely that (5.5) implies (5.4) for any  $a'' \notin w$ . The difficulty is that in verifying (5.4) we have to consider the termination behaviour of  $v \{a''/a\}$  and  $v' \{a''/a'\}$  in all states  $\vec{a}$  with  $\text{atom}(\vec{a}) \supseteq w \cup \{a''\}$ . The atom  $a''$  may occur at *any* position in  $\vec{a}$  and not necessarily at its right-hand end; whereas in assuming (5.5), 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 3.5). 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  $a' \otimes \vec{a}$  for the state obtained from  $\vec{a} \in \text{State}$  by appending an atom  $a'$  not in  $\text{atom}(\vec{a})$  to the *left* of the finite list of distinct atoms  $\vec{a}$  (cf.  $\vec{a} \otimes a'$  defined in Figure 5).

**Definition 5.8 (Affine Observations).** An observation on atoms,  $\text{obs} \in \mathcal{O}$ , is *affine* if it is equivariant (3.1) and satisfies: for all  $\vec{a} \in \text{State}$ , all  $a' \notin \text{atom}(\vec{a})$  and all  $(a_1, \dots, a_k) \in \text{atom}(\vec{a})^k$  (where  $k$  is the arity of obs)

$$\llbracket \text{obs} \rrbracket_{a' \otimes \vec{a}}(a_1, \dots, a_k) = \llbracket \text{obs} \rrbracket_{\vec{a}}(a_1, \dots, a_k) . \quad (5.6)$$

For example, of the observations defined in Figure 7, eq and lt are affine, whereas ord and card are not.

The following property of termination follows from its definition in Figures 5 and 6, using Corollary 3.3.

**Lemma 5.9.** *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  $\text{atom}(S, e) \subseteq \text{atom}(\vec{a})$  and all  $a' \notin \text{atom}(\vec{a})$ ,  $\langle a \otimes \vec{a}, S, e \rangle \downarrow_n \Leftrightarrow \langle \vec{a}, S, e \rangle \downarrow_n$ .  $\square$*

We now give a converse of Proposition 5.7, under the assumption that only affine observations are used. The proof is the technically most involved result in the paper.

**Proposition 5.10.** *Suppose that  $\mathcal{O}$  only contains affine observations. For any type  $\tau \in \text{Typ}$ , suppose we are given closed, well-typed atom binding values  $\emptyset \vdash_w \langle a \rangle v : \tau \text{ bnd}$  and  $\emptyset \vdash_w \langle a' \rangle v' : \tau \text{ bnd}$ . Then for all atoms  $a'' \notin w$  we have*

$$\vdash_w \langle a \rangle v \cong \langle a' \rangle v' : \tau \text{ bnd} \quad (5.7)$$

implies

$$\vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a'\} : \tau. \quad (5.8)$$

*Proof.* Suppose (5.7) holds and that  $a'' \notin w$ . To prove (5.8) we have to show for any  $w' \in \text{World}$ ,  $\vec{a} \in \text{State}$  and  $\tau' \in \text{Typ}$  with  $\text{atom}(\vec{a}) = w' \supseteq w \cup \{a''\}$  and  $\emptyset \vdash_{w'} S : \tau \rightarrow \tau'$  that

$$\langle \vec{a}, S, v\{a''/a\} \rangle \downarrow \Leftrightarrow \langle \vec{a}, S, v'\{a''/a'\} \rangle \downarrow. \quad (5.9)$$

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

$$\vec{a} = \vec{a}' \otimes a'' \otimes a_0 \otimes \cdots \otimes a_{n-1} \quad (5.10)$$

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

$$\begin{aligned} S' &\triangleq \text{Id} \circ (z. \text{let } \langle x \rangle y_0 = z \text{ in} \\ &\quad \text{let } \langle x_0 \rangle y_1 = \langle b_0 \rangle y_0 \text{ in} \\ &\quad \vdots \\ &\quad \text{let } \langle x_{n-1} \rangle y_n = \langle b_{n-1} \rangle y_{n-1} \text{ in} \\ &\quad S\{x, x_0, \dots, x_{n-1}/a'', a_0 \dots, a_{n-1}\}[y_n]) \end{aligned} \quad (5.11)$$

where  $z, x, x_0, \dots, x_{n-1}, y_0, \dots, y_n$  are distinct variables not occurring in  $S$ . Here we have used the notation “let  $\langle x_1 \rangle x_2 = e$  in  $e'$ ” from Figure 3, the notation “ $S[e]$ ” from (4.3) and the operation  $(-)\{x/a\}$  of replacing an atom  $a$  by a variable  $x$ .

Since  $\text{atom}(S) \subseteq w' = \text{atom}(\vec{a})$ , by definition of  $S'$  and from (5.10) we have  $\text{atom}(S') \subseteq \text{atom}(\vec{b}')$  where

$$\vec{b}' \triangleq b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}'. \quad (5.12)$$

Let  $\pi \in \mathbb{P}$  be the permutation swapping each  $a_i$  with  $b_i$  (for  $i = 0..n-1$ ). Since  $a'' \notin w \supseteq \text{atom}(a, v)$ , by definition of  $\vec{b}'$  we have  $\text{atom}(\pi \cdot \langle a \rangle v) \subseteq \text{atom}(\vec{b}')$ . Therefore the configuration  $\langle \vec{b}', S', \pi \cdot \langle a \rangle v \rangle$  satisfies the well-formedness condition needed to apply Corollary 3.3. Noting that  $\pi \cdot (\langle a \rangle v) = \langle \pi(a) \rangle (\pi \cdot v)$  and that  $\pi \cdot (v\{a''/a\}) = (\pi \cdot v)\{\pi(a'')/\pi(a)\} = (\pi \cdot v)\{a''/\pi(a)\}$ , from that corollary, property (4.2) and the definition of  $S'$  we get:

$$\begin{aligned} \langle \vec{b}', S', \pi \cdot (\langle a \rangle v) \rangle \downarrow &\Leftrightarrow \\ \langle \vec{b}' \otimes a'' \otimes a_0 \otimes \cdots \otimes a_{n-1}, S, (\pi \cdot (v\{a''/a\})) \rangle \{a_0, \dots, a_{n-1}/b_0, \dots, b_{n-1}\} \downarrow &. \end{aligned}$$

Note that by definition of  $\pi$

$$\begin{aligned} &(\pi \cdot (v\{a''/a\}))\{a_0, \dots, a_{n-1}/b_0, \dots, b_{n-1}\} \\ &= ((v\{a''/a\})\{b_0, \dots, b_{n-1}/a_0, \dots, a_{n-1}\})\{a_0, \dots, a_{n-1}/b_0, \dots, b_{n-1}\} \\ &= v\{a''/a\}; \end{aligned}$$

and  $\vec{b}' \otimes a'' \otimes a_0 \otimes \cdots \otimes a_{n-1} = b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}$  by (5.10) and (5.12). So altogether we have

$$\langle \vec{b}', S', \pi \cdot \langle a \rangle v \rangle \downarrow \Leftrightarrow \langle b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}, S, v\{a''/a\} \rangle \downarrow. \quad (5.13)$$

A similar argument gives

$$\langle \vec{b}', S', \pi \cdot \langle a' \rangle v' \rangle \downarrow \Leftrightarrow \langle b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}, S, v'\{a''/a'\} \rangle \downarrow. \quad (5.14)$$

We noted in Theorem 4.4 that operational equivalence is equivariant. So from (5.7) we have  $\vdash_{\text{atom}(\vec{b}')} \pi \cdot \langle a \rangle v \cong \pi \cdot \langle a' \rangle v' : \tau \text{bnd}$ . Since  $\emptyset \vdash_{\text{atom}(\vec{b}')} S' : \tau \text{bnd} \rightarrow \tau'$ , this operational equivalence gives

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

Combining this with (5.13) and (5.14) yields

$$\langle b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}, S, v\{a''/a\} \rangle \downarrow \Leftrightarrow \langle b_0 \otimes \cdots \otimes b_{n-1} \otimes \vec{a}, S, v'\{a''/a'\} \rangle \downarrow. \quad (5.15)$$

Since  $b_0, \dots, b_{n-1} \notin w' = \text{atom}(\vec{a}) \supseteq \text{atom}(S, a'', v, v')$  and  $\mathcal{O}$  only contains affine observations, we can now apply Lemma 5.9 to (5.15) to get (5.9), as required.  $\square$

**Example 5.11.** We conjecture that Proposition 5.10 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  $\text{ord}$  in Figure 7 and the values

$$\begin{aligned} v &\triangleq \text{fun}(f x = f x) \\ v' &\triangleq \text{fun}(f x = \text{match ord } a \text{ with } (\text{Zero} \rightarrow () \mid \text{Succ } y \rightarrow v())) \end{aligned}$$

where  $a$  is some atom. We claim that

$$\vdash_{\{a\}} \langle a \rangle v \cong \langle a \rangle v' : (\text{unit} \rightarrow \text{unit}) \text{bnd} \quad (5.16)$$

but that for any  $a' \neq a$

$$\vdash_{\{a, a'\}} v\{a'/a\} \not\cong v'\{a'/a\} : \text{unit} \rightarrow \text{unit}. \quad (5.17)$$

The operational inequivalence (5.17) is witnessed by the state  $\vec{a} \triangleq [a', a]$  and the frame stack  $S \triangleq \text{Id} \circ (x. x \text{ unit})$ , 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 (5.16), 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  $\langle \vec{a}, S, \langle a \rangle v' \rangle$  the occurrence of  $\text{ord } a$  in  $v'$  can only be renamed to  $\text{ord } a'$  for atoms  $a'$  at positions strictly greater than 0 in the current state; and hence  $\langle \vec{a}, S, \langle a \rangle v' \rangle$  has the same termination properties as  $\langle \vec{a}, S, \langle a \rangle v \rangle$ .

*Proof of Theorem 5.3.* One proves that  $\vdash_w v =_\alpha v' : \sigma$  implies  $\vdash_w v \cong v' : \sigma$  by induction on the the rules defining  $\alpha$ -equivalence in Figure 9, using Propositions 5.6 and 5.7.

To prove the converse implication, first note that if  $\emptyset \vdash v : \sigma$ , then  $v$  contains no instances of observations  $\text{obs} \in \mathcal{O}$ . The proof of this is by induction on the structure of the nominal arity  $\sigma$ ; the only way observations on atoms can appear in values of the language is via function values,  $\text{fun}(f x = e)$ , and the definition of “nominal arity” excludes function types. It follows from the definition of operational equivalence in Definition 4.1 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  $\{\text{eq}\}$ . 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  $\sigma$  using Propositions 5.6 and 5.10 (the latter applies because  $\text{eq}$  is affine).  $\square$

## 6. RELATED AND FURTHER WORK

**6.1. Correctness of Representation.** It is instructive to compare the Correctness of Representation property of FreshML (Theorem 5.3) with *adequacy* results for type-theoretic logical frameworks [Pfe01]. 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  $\alpha$ -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  $\alpha$ -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  $\alpha$ -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 [Pfe01] 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  $\beta$ -conversion at the meta-level. From the point of view of nominal signatures [Pit06], variables are just one kind of name. Properties of  $\alpha$ -conversion of all kinds of names are treated by the theory; but if one wants notions of substitution and  $\beta$ -conversion for a particular kind of name, one has to give a definition (an “ $\alpha$ -structural” recursive definition [Pit06]). For example in FreshML, using the data type (1.1) for  $\lambda$ -terms one can give an appealingly simple declaration for a function  $\text{subst} : \text{term} \rightarrow \text{atm} \rightarrow \text{term} \rightarrow \text{term}$  for capture-avoiding substitution; see [SPG03, p. 264]. Compositionality of the representation  $t \mapsto \ulcorner t \urcorner$  given in the introduction then becomes the contextual equivalence  $\vdash_w \ulcorner t_1[t_2/a] \urcorner \cong \text{subst} \ulcorner t_2 \urcorner a \ulcorner t_1 \urcorner : \text{term}$ . The CIU theorem (Theorem 4.4) provides the basis for proving such contextual equivalences. (We believe this particular equivalence is valid when  $\mathcal{O} = \{\text{eq}, \text{lt}\}$ , but not when  $\mathcal{O} = \{\text{eq}, \text{card}\}$ ; see Section 7.)

**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 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  $\text{lt}$  from Figure 7 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  $\text{atom}(S, e) \subseteq \text{atom}(\vec{a})$  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  $\text{succ} :$

$\text{atm} \rightarrow \text{atm}$  whose meaning is “successor function on atoms”, with transitions  $\langle \vec{a}, S, \text{succ } a \rangle \longrightarrow \langle \vec{a}, S, a' \rangle$  whenever  $a = \alpha(n)$  and  $a' = \alpha(n + 1)$  for some  $n \in \mathbb{N}$ . Then it may well be the case that  $a' \notin \text{atom}(\vec{a})$  even though  $a \in \text{atom}(\vec{a})$ .

So exposing the numerical representation of atoms involves giving up the invariant properties of the abstract semantics we have used here. 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 [Shi05a, SP05b]. 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” [ABF<sup>+</sup>05] 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 4.4), are quite complex logically speaking, compared with the kind of type safety results (like Theorem 2.4) that POPLMark has focused on so far. Systems like Isabelle/HOL [NPW02] 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 [UB06] 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” [Pit06] 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 [SPG03, Shi05b] 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 some theoretical properties of this design to do with data correctness. We showed that the addition of integer valued observations on names does not break FreshML’s fundamental Correctness of Representation property that  $\alpha$ -equivalence classes of abstract syntax trees (for any nominal signature) coincide with contextual equivalence classes of user declared data values. In particular, it is possible to give programmers access to a linear order on names without breaking the “up

to  $\alpha$ -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. 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 original version of FreshML [PG00] 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 [SPG03]. However, Pottier [Pot07] 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.

This paper has been concerned with data correctness, but what about general results on *program correctness*? The only restriction we placed on observations on atoms is that, as functions of both the state and the names they operate upon, they should be invariant under permuting names. We have seen that the Correctness of Representation property (Theorem 5.3) remains valid in the presence of any such observation. However, we are certainly not advocating that arbitrary equivariant observations be added to FreshML. This is because 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 5.7 and 5.10 to get an “extensionality” result explaining contextual equivalence at type  $\tau$  bnd in terms of contextual equivalence at  $\tau$ , for arbitrary higher types  $\tau$ .

More investigation of program correctness properties in the presence of particular observations on atoms is needed before one can advocate adding them to the FreshML design. The techniques we used in this paper could form the basis for such an investigation. 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 [GP01] and developed in [Pit03, UN05, Pit06].

#### ACKNOWLEDGEMENT.

The authors are grateful for the suggestions for improvement made by the anonymous referees.

#### REFERENCES

- [ABF<sup>+</sup>05] 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 J. Hurd and T. Melham, editors, *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. [www.cis.upenn.edu/group/proj/plclub/mmm/](http://www.cis.upenn.edu/group/proj/plclub/mmm/).
- [BL05] P. N. Benton and X. Leroy, editors. *ACM SIGPLAN Workshop on ML (ML 2005), Tallinn, Estonia*, Electronic Notes in Theoretical Computer Science. Elsevier, September 2005.
- [Che05] J. Cheney. Scrap your nameplate (functional pearl). In *Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP’05), Tallinn, Estonia*, pages 180–191. ACM Press, September 2005.
- [FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

- [FSDF93] 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.
- [Gor98] A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In Gordon and Pitts [GP98], pages 9–54.
- [GP98] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press, 1998.
- [GP01] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
- [Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
- [How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
- [Las98] S. B. Lassen. Relational reasoning about contexts. In Gordon and Pitts [GP98], pages 91–135.
- [MT91] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
- [MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [NPW02] 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.
- [Pfe01] 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.
- [PG00] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.
- [Pit02] A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *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, September 9–15, 2000.
- [Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
- [Pit05] 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.
- [Pit06] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006.
- [Pot05] F. Pottier. An overview of Caml. In Benton and Leroy [BL05], pages 27–52.
- [Pot07] F. Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, pages 356–365, Wroclaw, Poland, July 2007. IEEE Computer Society Press.
- [PS07] A. M. Pitts and M. R. Shinwell. Generative unbinding of names. In *34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007), Nice, France*, pages 85–95. ACM Press, January 2007.
- [Shi05a] 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.
- [Shi05b] M. R. Shinwell. Fresh O’Caml: Nominal abstract syntax for the masses. In Benton and Leroy [BL05], pages 53–76.
- [SP05a] M. R. Shinwell and A. M. Pitts. Fresh Objective Caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge Computer Laboratory, February 2005.
- [SP05b] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.
- [SPG03] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

- [UB06] 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.
- [UN05] C. Urban and M. Norrish. A formal treatment of the Barendregt Variable Convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN '05), Tallinn, Estonia*, pages 25–32. ACM Press, 2005.
- [UPG04] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

#### APPENDIX A. PROOF OF THEOREM 4.4

We wish to show that the expression relation  $\cong^\circ$  of Definition 4.3 is compatible and substitutive (see Definition 4.2). We use an adaptation of “Howe’s method” [How96] to do so. Let the expression relation  $\cong^*$  be inductively defined from  $\cong^\circ$  by the rule

$$\frac{\Gamma \vdash_w e \widehat{\cong}^* e' : \tau \quad \Gamma \vdash_w e' \cong^\circ e'' : \tau}{\Gamma \vdash_w e \cong^* e'' : \tau}. \quad (\text{A.1})$$

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,  $\mathcal{E}_1 \subseteq \mathcal{E}_2 \Rightarrow \widehat{\mathcal{E}}_1 \subseteq \widehat{\mathcal{E}}_2$ .

**Lemma A.1.**

- (i)  $\Gamma \vdash_w e \cong^* e' : \tau \wedge \Gamma \vdash_w e' \cong^\circ e'' : \tau \Rightarrow \Gamma \vdash_w e \cong^* e'' : \tau$ .
- (ii)  $\cong^*$  is compatible and substitutive.
- (iii)  $\text{atom}(e) \subseteq w \wedge \Gamma \vdash e : \tau \Rightarrow \Gamma \vdash_w e \cong^* e : \tau$ .
- (iv)  $\text{atom}(S) \subseteq w \wedge \Gamma \vdash S : \tau \rightarrow \tau' \Rightarrow \Gamma \vdash_w S \widehat{\cong}^* S : \tau \rightarrow \tau'$ .
- (v)  $\Gamma \vdash_w v \cong^* e' : \tau \Rightarrow \exists v'. \Gamma \vdash_w v \cong^* v' : \tau \wedge \Gamma \vdash_w v' \cong^\circ e' : \tau$ .

*Proof.* These properties of  $\cong^*$  are simple consequences of its definition and the definition of the extension of compatible refinement to a relation between frame stacks given by the last two rules in Figure 8. □

**Lemma A.2.**

- (i)  $\cong^*$  is equivariant.
- (ii)  $\Gamma \vdash_w e \cong^* e' : \tau \wedge w \subseteq w' \Rightarrow \Gamma \vdash_{w'} e \cong^* e' : \tau$ .
- (iii)  $\Gamma \vdash_w S \widehat{\cong}^* S' : \tau \rightarrow \tau' \wedge w \subseteq w' \Rightarrow \Gamma \vdash_{w'} S \widehat{\cong}^* S' : \tau \rightarrow \tau'$ .

*Proof.* Part (i) follows from the fact that  $\cong^\circ$  is equivariant, which in turn is a consequence of Lemma 3.2. Parts (ii) and (iii) are consequences of the fact that world weakening is built into the definition of operational equivalence in Definition 4.1. □

**Lemma A.3.**  $\Gamma \vdash_w e \cong^\circ e' : \tau \Rightarrow \Gamma \vdash_w e \cong^* e' : \tau$ .

*Proof.* If  $\Gamma \vdash_w e \cong^\circ e' : \tau$ , then in particular  $\text{atom}(e) \subseteq w$  and  $\Gamma \vdash e : \tau$ , so that by Lemma A.1(iii) we have  $\Gamma \vdash_w e \cong^* e : \tau$ ; so from part (i) of that lemma we get  $\Gamma \vdash_w e \cong^* e' : \tau$ . □

We wish to show that  $\cong^*$  coincides with  $\cong^\circ$ . In view of the previous lemma, it just remains to show that  $\cong^* \subseteq \cong^\circ$ . Lemma A.5 provides the key to this. Before stating that lemma we give some simple properties of  $\cong$  that are needed to prove it.

**Lemma A.4.**

- (i)  $\vdash_w a \cong a' : \text{atm} \Rightarrow a = a'$ .
- (ii)  $\vdash_w v \cong v' : \tau \text{ bnd} \Rightarrow \vdash_w \text{unbind } v \cong \text{unbind } v' : \text{atm} * \tau$ .
- (iii) *If  $\vdash_w v \cong v' : \tau_1 \rightarrow \tau_2$ , then for any world  $w' \supseteq w$  and value  $v_1$  with  $\text{atom}(v_1) \subseteq w'$  and  $\vdash_{w'} v_1 : \tau_1$ , it is the case that  $\vdash_{w'} v v_1 \cong v' v_1 : \tau_2$ .*

*Proof.* For part (i) we make use of the fact that  $\mathcal{O}$  always contains the atom equality function eq from Figure 7. Consider the frame stack

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

If  $a \neq a'$  are distinct elements of  $w$ , then choosing some  $\vec{a} \in \text{State}$  with  $\text{atom}(\vec{a}) = w$ , 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  $\vdash_w a \cong a' : \text{atm}$  it cannot be the case that  $a \neq a'$ .

For part (ii), given any  $\vec{a}$ ,  $S$  and  $\tau'$  with  $w \cup \text{atom}(S) \subseteq \text{atom}(\vec{a})$  and  $\emptyset \vdash S : \tau \rightarrow \tau'$ , then

$$\begin{aligned} \langle \vec{a}, S, \text{unbind } v \rangle \downarrow &\Leftrightarrow \langle \vec{a}, S \circ (x. \text{unbind } x), v \rangle \downarrow && \text{by definition of } \downarrow \\ &\Leftrightarrow \langle \vec{a}, S \circ (x. \text{unbind } x), v' \rangle \downarrow && \text{since } \vdash_w v \cong v' : \tau \text{ bnd} \\ &\Leftrightarrow \langle \vec{a}, S, \text{unbind } v' \rangle \downarrow && \text{by definition of } \downarrow \end{aligned}$$

and thus  $\vdash_w \text{unbind } v \cong \text{unbind } v' : \text{atm} * \tau$ .

The proof of part (iii) is similar to that for (ii), using the frame  $(x. x v_1)$  in place of  $(x. \text{unbind } x)$ .  $\square$

**Lemma A.5.** *For all  $n \geq 0$  and all  $w, S, S', \tau, \tau', e, e', \vec{a}$*

$$\begin{aligned} \emptyset \vdash_w S \widehat{\cong}^* S' : \tau \rightarrow \tau' \wedge \emptyset \vdash_w e \cong^* e' : \tau \wedge \text{atom}(\vec{a}) = w \wedge \langle \vec{a}, S, e \rangle \downarrow_n \\ \Rightarrow \langle \vec{a}, S', e' \rangle \downarrow . \end{aligned} \quad (\text{A.2})$$

*Proof.* The lemma is proved by induction on  $n$ . The base case  $n = 0$  follows from the definition of  $\widehat{\cong}^*$  (which implies that  $\emptyset \vdash_w \text{Id} \widehat{\cong}^* S' : \tau \rightarrow \tau'$  can only hold when  $S' = \text{Id}$ ), combined with Lemma A.1(v) and the definition of  $\cong^\circ$ . For the induction step, assume (A.2) holds and that

$$\emptyset \vdash_w S \widehat{\cong}^* S' : \tau \rightarrow \tau' \quad (\text{A.3})$$

$$\emptyset \vdash_w e \cong^* e' : \tau \quad (\text{A.4})$$

$$\text{atom}(\vec{a}) = w \quad (\text{A.5})$$

$$\langle \vec{a}, S, e \rangle \longrightarrow \langle \vec{a}_1, S_1, e_1 \rangle \quad (\text{A.6})$$

$$\langle \vec{a}_1, S_1, e_1 \rangle \downarrow^n \quad (\text{A.7})$$

We have to prove  $\langle \vec{a}, S', e' \rangle \downarrow$  and do so by an analysis of (A.6) against the possible cases 1–9 in the definition of the transition relation in Figure 5.

**Case 1.** In this case  $S = S_1 \circ (x. e_2)$ ,  $e = v \in \text{Val}$ ,  $\vec{a}_1 = \vec{a}$ , and  $e_1 = e_2[v/x]$ , for some  $e_2$  and  $v$ . For (A.3) to hold, by definition of  $\widehat{\cong}^*$  it must be the case that  $S' = S'_1 \circ (x. e'_2)$  for some  $S'_1$  and  $e'_2$  with

$$\{x : \tau\} \vdash_w e_2 \widehat{\cong}^* e'_2 : \tau_2 \quad (\text{A.8})$$

$$\emptyset \vdash_w S_1 \widehat{\cong}^* S'_1 : \tau_2 \rightarrow \tau' \quad (\text{A.9})$$

for some type  $\tau_2$ . Since  $e = v$  is a value, applying Lemma A.1(v) to (A.4) we get

$$\emptyset \vdash_w v \widehat{\cong}^* v' : \tau \quad (\text{A.10})$$

$$\vdash_w v' \cong e' : \tau \quad (\text{A.11})$$

for some  $v' \in \text{Val}$ . Since  $\widehat{\cong}^*$  is substitutive (Lemma A.1(ii)), from (A.8) and (A.10) we get

$$\emptyset \vdash_w e_2[v/x] \widehat{\cong}^* e'_2[v'/x] : \tau_2. \quad (\text{A.12})$$

Applying the induction hypothesis (A.2) to (A.9), (A.12), (A.5) and to (A.7), we get  $\langle \vec{a}, S'_1, e'_2[v'/x] \rangle \downarrow$ ; hence  $\langle \vec{a}, S'_1 \circ (x. e'_2), v' \rangle \downarrow$ , that is,  $\langle \vec{a}, S', v' \rangle \downarrow$ ; and therefore by (A.11) we also have  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 2.** In this case we have  $e = \text{let } x = e_1 \text{ in } e_2$ ,  $\vec{a}_1 = \vec{a}$  and  $S_1 = S \circ (x. e_2)$  for some  $e_2$ . Since (A.4) holds, by definition of  $\widehat{\cong}^*$ , there must exist some  $e'_1, e'_2$  and  $\tau_1$  with

$$\emptyset \vdash_w e_1 \widehat{\cong}^* e'_1 : \tau_1 \quad (\text{A.13})$$

$$\{x : \tau_1\} \vdash_w e_2 \widehat{\cong}^* e'_2 : \tau \quad (\text{A.14})$$

$$\vdash_w (\text{let } x = e'_1 \text{ in } e'_2) \cong e' : \tau \quad (\text{A.15})$$

and then from (A.3) and (A.14) we get

$$\emptyset \vdash_w S \circ (x. e_2) \widehat{\cong}^* S' \circ (x. e'_2) : \tau_1 \rightarrow \tau'. \quad (\text{A.16})$$

The induction hypothesis (A.2) applied to (A.16), (A.13) and (A.5) gives  $\langle \vec{a}, S' \circ (x. e'_2), e'_1 \rangle \downarrow$  and hence  $\langle \vec{a}, S', \text{let } x = e'_1 \text{ in } e'_2 \rangle \downarrow$ . This and (A.15) then give  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 3.** This follows from the definition of  $\widehat{\cong}^*$  using its substitutivity property, much as for case 1.

**Case 4.** In this case  $\tau = \tau_1 * \tau_2$ ,  $e = (v_1, v_2)$ ,  $\vec{a}_1 = \vec{a}$  and  $e_1 = v_1$ , for some  $\tau_1, \tau_2 \in \text{Typ}$  and  $v_1, v_2 \in \text{Val}$ . By definition of  $\widehat{\cong}^*$ , for (A.4) to hold it must be the case that

$$\emptyset \vdash_w v_i \widehat{\cong}^* v'_i : \tau_i \quad (\text{for } i = 1, 2) \quad (\text{A.17})$$

$$\vdash_w (v'_1, v'_2) \cong e' : \tau_1 * \tau_2 \quad (\text{A.18})$$

for some  $v'_1$  and  $v'_2$ . By the induction hypothesis (A.2) applied to (A.3), (A.17), (A.5) and (A.7), we get  $\langle \vec{a}, S', v'_1 \rangle \downarrow$  and hence also  $\langle \vec{a}, S', \text{fst}(v'_1, v'_2) \rangle \downarrow$ . Hence by (A.18) we have  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 5.** This is similar to the previous case.

**Case 6.** In this case  $e = v_1 v_2$ ,  $\vec{a}_1 = \vec{a}$ ,  $S_1 = S$  and  $e_1 = e_2[v_1, v_2/f, x]$  for some  $v_1 = \text{fun}(f x = e_2)$  and  $v_2$ . Since (A.4) holds, by definition of  $\cong^*$  together with Lemma A.4(iii), there must exist some  $e'_2$ ,  $v'_2$  and  $\tau_1$  with

$$\{f : \tau_1 \rightarrow \tau, x : \tau_1\} \vdash_w e_2 \cong^* e'_2 : \tau \quad (\text{A.19})$$

$$\emptyset \vdash_w v_2 \cong^* v'_2 : \tau_1 \quad (\text{A.20})$$

$$\vdash_w \text{fun}(f x = e'_2) v'_2 \cong e' : \tau_1 \rightarrow \tau . \quad (\text{A.21})$$

Since  $\cong^*$  is compatible (Lemma A.1(ii)), from (A.19) we get  $\emptyset \vdash_w v_1 \cong^* \text{fun}(f x = e'_2) : \tau_1 \rightarrow \tau$ ; and since  $\cong^*$  is also substitutive, this together with (A.19) and (A.20) gives  $\emptyset \vdash_w e_2[v_1, v_2/f, x] \cong^* e'_2[\text{fun}(f x = e'_2), v'_2/f, x] : \tau$ . Therefore by the induction hypothesis (A.2) applied to (A.3), this, (A.5) and (A.7), we get  $\langle \vec{a}, S', e'_2[\text{fun}(f x = e'_2), v'_2/f, x] \rangle \downarrow$ . Hence  $\langle \vec{a}, S', \text{fun}(f x = e'_2) v'_2 \rangle \downarrow$  and thus by (A.21),  $\langle \vec{a}, S', e' \rangle \downarrow$  as required.

**Case 7.** In this case  $\tau = \text{atm}$ ,  $e = \text{fresh}()$ ,  $\vec{a}_1 = \vec{a} \otimes a$ ,  $S_1 = S$  and  $e_1 = a$ , for some  $a \notin \text{atom}(\vec{a}) = w$ . Since (A.4) holds, by definition of  $\cong^*$  we have

$$\vdash_w \text{fresh}() \cong e' : \text{atm} . \quad (\text{A.22})$$

By Lemma A.2(iii) applied to (A.3), we have  $\emptyset \vdash_{w \cup \{a\}} S \widehat{\cong}^* S' : \text{atm} \rightarrow \tau'$ ; and by Lemma A.1(iii) we also have  $\emptyset \vdash_{w \cup \{a\}} a \cong^* a : \text{atm}$ . So by the induction hypothesis (A.2) applied to these,  $\text{atom}(\vec{a} \otimes a) = w \cup \{a\}$  and (A.7), we get  $\langle \vec{a} \otimes a, S', a \rangle \downarrow$ . Hence  $\langle \vec{a}, S', \text{fresh}() \rangle \downarrow$  and hence from (A.22) we also have  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 8.** In this case  $\tau = \text{atm} * \tau_1$ ,  $e = \text{unbind} \langle a \rangle v$ ,  $\vec{a}_1 = \vec{a} \otimes a_1$ ,  $S_1 = S$ , and  $e_1 = (a_1, v\{a_1/a\})$ , for some  $\tau_1$ ,  $a$ ,  $v$  and  $a_1$  with  $a_1 \notin \text{atom}(\vec{a}) = w$ . Since (A.4) holds, by definition of  $\cong^*$  together with parts (i) and (ii) of Lemma A.4, there must exist some  $v'$  with

$$\emptyset \vdash_w v \cong^* v' : \tau_1 \quad (\text{A.23})$$

$$\vdash_w \text{unbind} \langle a \rangle v' \cong e' : \text{atm} * \tau_1 . \quad (\text{A.24})$$

We now appeal to the easily verified fact that since  $a_1 \notin w \supseteq \text{atom}(v, v')$ , the renamed values  $v\{a_1/a\}$  and  $v'\{a_1/a\}$  are respectively equal to the permuted values  $(a \ a_1) \cdot v$  and  $(a \ a_1) \cdot v'$  (where  $(a \ a_1)$  denotes the permutation swapping  $a$  and  $a'$ ). Therefore by parts (i) and (ii) of Lemma A.2 applied to (A.23) and by parts (ii) and (iii) of Lemma A.1, we have

$$\emptyset \vdash_{w \cup \{a_1\}} (a_1, v\{a_1/a\}) \cong^* (a_1, v'\{a_1/a\}) : \text{atm} * \tau_1 . \quad (\text{A.25})$$

By applying Lemma A.2(iii) to (A.3) we also have

$$\emptyset \vdash_{w \cup \{a_1\}} S \widehat{\cong}^* S' : \text{atm} * \tau_1 \rightarrow \tau' .$$

Then applying the induction hypothesis (A.2) to this, (A.25),  $\text{atom}(\vec{a} \otimes a_1) = w \cup \{a_1\}$  and (A.7) yields  $\langle \vec{a} \otimes a_1, S', (a_1, v'\{a_1/a\}) \rangle \downarrow$ . Therefore  $\langle \vec{a}, S', \text{unbind} \langle a \rangle v' \rangle \downarrow$ ; and hence by (A.24), we also have  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 9.** In this case  $\tau = \text{nat}$ ,  $e = \text{obs } a_1 \dots a_k$  for some  $a_1, \dots, a_k \in w$ ,  $\vec{a}_1 = \vec{a}$ ,  $S_1 = S$ , and  $e_1 = \ulcorner m \urcorner$  where  $m = \llbracket \text{obs} \rrbracket_{\vec{a}}(a_1, \dots, a_k)$ . Since (A.4) holds, by definition of  $\cong^*$  together with Lemma A.4(i), we must have

$$\vdash_w \text{obs } a_1 \dots a_k \cong e' : \text{nat} . \quad (\text{A.26})$$

Note that by Lemma A.1(iii) we also have  $\emptyset \vdash_w \ulcorner m \urcorner \cong^* \ulcorner m \urcorner : \text{nat}$ . So by the induction hypothesis (A.2) applied to this, (A.3), (A.5) and (A.7) we get  $\langle \vec{a}, S', \ulcorner m \urcorner \rangle \downarrow$ . Since  $m = \llbracket \text{obs} \rrbracket_{\vec{a}}(a_1, \dots, a_k)$ , this implies that  $\langle \vec{a}, S', \text{obs } a_1 \dots a_k \rangle \downarrow$ ; and hence from (A.26) we have that  $\langle \vec{a}, S', e' \rangle \downarrow$  holds, as required.

This completes the proof of Lemma A.5.  $\square$

**Lemma A.6.** *Let  $(\cong^*)^+$  denote the transitive closure of  $\cong^*$ . Then*

$$\Gamma \vdash_w e \cong^* e' : \tau \Rightarrow \Gamma \vdash_w e' (\cong^*)^+ e : \tau .$$

*Proof.* This can be proved by induction on the derivation of  $\Gamma \vdash_w e \cong^* e' : \tau$  from the rule in (A.1) and the rules for compatible refinement in Figure 8, using the fact that  $\cong^\circ$  is symmetric and using Lemmas A.3 and A.1(iii).  $\square$

We can now complete the proof of Theorem 4.4 by showing that  $\cong^\circ$  is compatible and substitutive (Definition 4.2). Since  $\cong^*$  has those properties by Lemma A.1(ii), it suffices to show that  $\cong^\circ$  coincides with  $\cong^*$ . We already noted in Lemma A.3 that  $\cong^\circ$  is contained in  $\cong^*$ . For the reverse inclusion, since  $\cong^*$  is substitutive and reflexive (Lemma A.1), it is closed under substituting values for variables; so by Definition 4.3, it suffices to show that

$$\emptyset \vdash_w e \cong^* e' : \tau \Rightarrow \vdash_w e \cong e' : \tau . \quad (\text{A.27})$$

To see this, note that by Lemma A.5 (together with Lemmas A.1(iv) and A.2(ii)) we have:

$$\begin{aligned} \emptyset \vdash_w e \cong^* e' : \tau &\Rightarrow \forall \vec{a}, S, \tau'. w \cup \text{atom}(S) \subseteq \text{atom}(\vec{a}) \wedge \emptyset \vdash S : \tau \rightarrow \tau' \wedge \langle \vec{a}, S, e \rangle \downarrow \\ &\Rightarrow \langle \vec{a}, S, e' \rangle \downarrow . \end{aligned} \quad (\text{A.28})$$

Since the right-hand side of the implication in (A.28) is a transitive relation between expressions  $e, e'$ , we also have

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

and therefore Lemma A.6 gives

$$\begin{aligned} \{\} \vdash_w e \cong^* e' : \tau &\Rightarrow \forall \vec{a}, S, \tau'. (w \cup \text{atom}(S) \subseteq \text{atom}(\vec{a}) \wedge \emptyset \vdash S : \tau \rightarrow \tau' \wedge \langle \vec{a}, S, e \rangle \downarrow \\ &\Rightarrow \langle \vec{a}, S, e' \rangle \downarrow . \end{aligned} \quad (\text{A.29})$$

Combining (A.28) and (A.29) gives (A.27).  $\square$

## APPENDIX B. PROOF OF PROPOSITION 5.7

Let  $\mathcal{E}$  be the closure under compatible refinement (Figure 8) 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.

$$\frac{a'' \notin w \subseteq \text{atom}(a, v, a', v') \quad \vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a'\} : \tau}{\emptyset \vdash_w \langle a \rangle v \mathcal{E} \langle a' \rangle v' : \tau \text{ bnd}} \quad \frac{\Gamma \vdash_w e \widehat{\mathcal{E}} e' : \tau}{\Gamma \vdash_w e \mathcal{E} e' : \tau} \quad (\text{B.1})$$

**Lemma B.1.**

- (i)  $\mathcal{E}$  is compatible and substitutive.
- (ii)  $\text{atom}(e) \subseteq w \wedge \Gamma \vdash e : \tau \Rightarrow \Gamma \vdash_w e \mathcal{E} e : \tau$ .
- (iii)  $\text{atom}(S) \subseteq w \wedge \Gamma \vdash S : \tau \rightarrow \tau' \Rightarrow \Gamma \vdash_w S \widehat{\mathcal{E}} S : \tau \rightarrow \tau'$ .
- (iv)  $\Gamma \vdash_w v \mathcal{E} e' : \tau \Rightarrow e' \in \text{Val}$ .

*Proof.* These properties of  $\mathcal{E}$  are simple consequences of its definition in (B.1), the definition of compatible refinement in Figure 8, and the definition of its extension to a relation between frame stacks given by the last two rules in that figure.  $\square$

**Lemma B.2.**

- (i)  $\mathcal{E}$  is equivariant.
- (ii)  $\Gamma \vdash_w e \mathcal{E} e' : \tau \wedge w \subseteq w' \Rightarrow \Gamma \vdash_{w'} e \mathcal{E} e' : \tau$ .
- (iii)  $\Gamma \vdash_w S \widehat{\mathcal{E}} S' : \tau \rightarrow \tau' \wedge w \subseteq w' \Rightarrow \Gamma \vdash_{w'} S \mathcal{E} S' : \tau \rightarrow \tau'$ .

*Proof.* This is the analogue of Lemma A.2 for  $\mathcal{E}$ , and is proved in the same way as that lemma.  $\square$

**Lemma B.3.** For all  $n \geq 0$  and all  $w, S, S', \tau, \tau', e, e', \vec{a}$ 

$$\emptyset \vdash_w S \widehat{\mathcal{E}} S' : \tau \rightarrow \tau' \wedge \emptyset \vdash_w e \mathcal{E} e' : \tau \wedge \text{atom}(\vec{a}) = w \wedge \langle \vec{a}, S, e \rangle \downarrow_n \Rightarrow \langle \vec{a}, S', e' \rangle \downarrow. \quad (\text{B.2})$$

*Proof.* The lemma is proved by induction on  $n$ . The base case  $n = 0$  follows directly from Lemma B.1(iii) and the definition of  $\widehat{\mathcal{E}}$  (which implies that  $\{\} \vdash_w \text{Id} \widehat{\mathcal{E}} S' : \tau \rightarrow \tau'$  can only hold when  $S' = \text{Id}$ ). For the induction step, assume (B.2) holds and that

$$\emptyset \vdash_w S \widehat{\mathcal{E}} S' : \tau \rightarrow \tau' \quad (\text{B.3})$$

$$\emptyset \vdash_w e \mathcal{E} e' : \tau \quad (\text{B.4})$$

$$\text{atom}(\vec{a}) = w \quad (\text{B.5})$$

$$\langle \vec{a}, S, e \rangle \longrightarrow \langle \vec{a}_1, S_1, e_1 \rangle \quad (\text{B.6})$$

$$\langle \vec{a}_1, S_1, e_1 \rangle \downarrow^n \quad (\text{B.7})$$

We have to prove  $\langle \vec{a}, S', e' \rangle \downarrow$  and do so by an analysis of (B.6) against the possible cases 1–9 in the definition of the transition relation in Figure 5. Cases 1, 3 and 6 follow from the definition of  $\mathcal{E}$  and its substitutivity property; we give the details for the first one and omit the other two. Cases 4, 5 and 9 also follow easily from the definition of  $\mathcal{E}$  (using Lemma B.1(ii) in the third case). So we give the proofs just for cases 1, 2, 7 and 8.

**Case 1.** In this case  $S = S_1 \circ (x.e_2)$ ,  $e = v \in \text{Val}$ ,  $\vec{a}_1 = \vec{a}$ , and  $e_1 = e_2[v/x]$ , for some  $e_2$  and  $v$ . For (B.3) to hold, by definition of  $\widehat{\mathcal{E}}$  it must be the case that  $S' = S'_1 \circ (x.e'_2)$  for some  $S'_1$  and  $e'_2$  with

$$\{x : \tau\} \vdash_w e_2 \mathcal{E} e'_2 : \tau_2 \quad (\text{B.8})$$

$$\emptyset \vdash_w S_1 \widehat{\mathcal{E}} S'_1 : \tau_2 \rightarrow \tau' \quad (\text{B.9})$$

for some type  $\tau_2$ . Since  $e = v$  is a value, applying Lemma B.1(iv) to (B.4) we get  $e' = v'$  for some  $v' \in \text{Val}$ . So since  $\widehat{\mathcal{E}}$  is substitutive (Lemma B.1(i)), from (B.4) and (B.8) we get

$$\emptyset \vdash_w e_2[v/x] \mathcal{E} e'_2[v'/x] : \tau_2. \quad (\text{B.10})$$

Applying the induction hypothesis (B.2) to (B.9), (B.10), (B.5) and to (B.7), we get  $\langle \vec{a}, S'_1, e'_2[v'/x] \rangle \downarrow$ ; hence  $\langle \vec{a}, S'_1 \circ (x.e'_2), v' \rangle \downarrow$ , that is,  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 2.** In this case  $e = \text{let } x = e_1 \text{ in } e_2$ ,  $\vec{a}_1 = \vec{a}$  and  $S_1 = S \circ (x.e_2)$  for some  $e_2$ . For (B.4) to hold, by definition of  $\widehat{\mathcal{E}}$  it must be the case that  $e' = \text{let } x = e'_1 \text{ in } e'_2$  for some  $e'_1, e'_2$  and  $\tau_1$  with

$$\{\} \vdash_w e_1 \mathcal{E} e'_1 : \tau_1 \quad (\text{B.11})$$

$$\{x : \tau_1\} \vdash_w e_2 \mathcal{E} e'_2 : \tau. \quad (\text{B.12})$$

From (B.3) and (B.12) we get  $\emptyset \vdash_w S \circ (x.e_2) \widehat{\mathcal{E}} S' \circ (x.e'_2) : \tau \rightarrow \tau'$ ; and the induction hypothesis (B.2) applied to this, (B.11), (B.5) and (B.7) gives  $\langle \vec{a}, S' \circ (x.e'_2), e'_1 \rangle \downarrow$ . Hence  $\langle \vec{a}, S', \text{let } x = e'_1 \text{ in } e'_2 \rangle \downarrow$ , that is,  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 7.** In this case  $\tau = \text{atm}$ ,  $e = \text{fresh}()$ ,  $\vec{a}_1 = \vec{a} \otimes a$ ,  $S_1 = S$  and  $e_1 = a$ , for some atom  $a \notin w$ . For (B.4) to hold, by definition of  $\mathcal{E}$  it must be the case that  $e' = \text{fresh}()$ . Now Lemma B.2(iii) applied to (B.3) gives  $\emptyset \vdash_{w \cup \{a\}} S \widehat{\mathcal{E}} S' : \tau \rightarrow \tau'$ ; and Lemma B.1(ii) gives  $\emptyset \vdash_{w \cup \{a\}} a \mathcal{E} a : \text{atm}$ . Applying the induction hypothesis (B.2) to these two facts,  $\text{atom}(\vec{a} \otimes a) = w \cup \{a\}$  and (B.7) gives  $\langle \vec{a} \otimes a, S', a \rangle \downarrow$ . Hence  $\langle \vec{a}, S', \text{fresh}() \rangle \downarrow$ , that is,  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

**Case 8.** In this case  $\tau = \text{atm} * \tau_1$ ,  $e = \text{unbind} \langle a \rangle v$ ,  $\vec{a}_1 = \vec{a} \otimes a_1$ ,  $S_1 = S$ , and  $e_1 = (a_1, v\{a_1/a\})$ , for some  $\tau_1, a, v$  and  $a_1$  with  $a_1 \notin w$ . For (B.4) to hold, by definition of  $\mathcal{E}$  it must be the case that  $e' = \text{unbind} \langle a' \rangle v'$  with

$$\begin{aligned} \text{either (a): } & a = a' \wedge \emptyset \vdash_w v \mathcal{E} v' : \tau_1 \\ \text{or (b): } & \exists a'' \notin w. \vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a\} : \tau_1 \end{aligned} \quad (\text{B.13})$$

If (B.13)(a) holds, then as in the proof of Lemma A.5 we now appeal to the easily verified fact that since  $a_1 \notin w \supseteq \text{atom}(v, v')$ , the renamed values  $v\{a_1/a\}$  and  $v'\{a_1/a\}$  are respectively equal to the permuted values  $(a \ a_1) \cdot v$  and  $(a \ a_1) \cdot v'$  (where  $(a \ a_1)$  denotes the permutation swapping  $a$  and  $a_1$ ). Therefore from the fact that  $\emptyset \vdash_w v \mathcal{E} v' : \tau_1$  holds, from parts (i) and (ii) of Lemma B.2 we get  $\emptyset \vdash_{w \cup \{a_1\}} v\{a_1/a\} \mathcal{E} v'\{a_1/a\} : \tau_1$ . Then since  $a = a'$ , by Lemma B.1(ii) we have  $\emptyset \vdash_{w \cup \{a_1\}} (a_1, v\{a_1/a\}) \mathcal{E} (a_1, v'\{a_1/a\}) : \text{atm} * \tau_1$ . Applying the induction hypothesis (B.2) to this, (B.3) (weakened using Lemma B.2(iii)),  $\text{atom}(\vec{a} \otimes a_1) = w \cup \{a_1\}$  and (B.7) yields  $\langle \vec{a} \otimes a_1, S', (a_1, v'\{a_1/a\}) \rangle \downarrow$  with  $a_1 \notin \text{atom}(\vec{a})$ . Therefore by definition of  $\downarrow$ , we also have  $\langle \vec{a}, S', \text{unbind} \langle a' \rangle v' \rangle \downarrow$ .

If (B.13)(b) holds, then by Theorem 4.4, so does

$$\vdash_{w \cup \{a''\}} (a'', v\{a''/a\}) \cong (a'', v'\{a''/a\}) : \text{atm} * \tau_1 \quad (\text{B.14})$$

Lemma 3.2 applied to (B.7) with  $\pi = (a_1 a'')$  gives  $\langle \vec{a} \otimes a'', S, (a'', v\{a''/a\}) \rangle \downarrow_n$ . Combining this with (B.3) (weakened using Lemma B.2(iii)),  $\emptyset \vdash_{w \cup \{a''\}} (a'', v\{a''/a\}) \mathcal{E} (a'', v'\{a''/a\}) : \text{atm} * \tau_1$  (by Lemma B.1(ii)),  $\text{atom}(\vec{a} \otimes a'') = w \cup \{a''\}$  and the induction hypothesis (B.2), we get  $\langle \vec{a} \otimes a'', S', (a'', v\{a''/a\}) \rangle \downarrow$ . Then by definition of  $\cong$ , from this and (B.14) we get  $\langle \vec{a} \otimes a'', S', (a'', v'\{a''/a\}) \rangle \downarrow$  with  $a'' \notin \vec{a}$ . Therefore as before, by definition of  $\downarrow$ , we also have  $\langle \vec{a}, S', \text{unbind} \langle a' \rangle v' \rangle \downarrow$ .

So in either case in (B.13), since  $e' = \text{unbind} \langle a' \rangle v'$ , we get  $\langle \vec{a}, S', e' \rangle \downarrow$ , as required.

This completes the proof of Lemma B.3.  $\square$

We can now complete the proof of Proposition 5.7. For any type  $\tau \in \text{Typ}$ , suppose we are given closed, well-typed atom binding values  $\emptyset \vdash \langle a \rangle v : \tau \text{ bnd}$  and  $\emptyset \vdash \langle a' \rangle v' : \tau \text{ bnd}$  with  $\text{atom}(a, v, a', v') \subseteq w$  and satisfying

$$\vdash_{w \cup \{a''\}} v\{a''/a\} \cong v'\{a''/a'\} : \tau \quad (\text{B.15})$$

for some atom  $a'' \notin w$ . By definition of  $\mathcal{E}$  this implies

$$\emptyset \vdash_w \langle a \rangle v \mathcal{E} \langle a' \rangle v' : \tau \text{ bnd} . \quad (\text{B.16})$$

For any  $w', \vec{a}, S$ , and  $\tau'$  with  $\text{atom}(\vec{a}) = w' \supseteq w \cup \text{atom}(S)$  and  $\emptyset \vdash S : \tau \rightarrow \tau'$ , we have

$$\emptyset \vdash_{w'} S \widehat{\mathcal{E}} S : \tau \rightarrow \tau' \quad (\text{B.17})$$

by Lemma B.1(iii) and

$$\emptyset \vdash_{w'} \langle a \rangle v \mathcal{E} \langle a' \rangle v' : \tau \text{ bnd} \quad (\text{B.18})$$

by Lemma B.2(ii) applied to (B.16). So Lemma B.3 applied to (B.17), (B.18) and  $\text{atom}(\vec{a}) = w'$ , we have

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

Since  $\cong$  is symmetric, the same argument shows that (B.15) implies

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

Thus (B.15) implies that  $\langle a \rangle v$  and  $\langle a' \rangle v'$  are operationally equivalent, as required.  $\square$