Proof-Relevant Logical Relations for Name Generation

Nick Benton[1], Martin Hofmann[2], and Vivek Nigam[3]

Microsoft Research, Cambridge[1], LMU, Munich[2], and UFPB, João Pessoa[3]
nick@microsoft.com[1], hofmann@ifi.lmu.de[2], vivek.nigam@gmail.com[3]

Abstract. Pitts and Stark's ν-calculus is a paradigmatic total language for studying the problem of contextual equivalence in higher-order languages with name generation. Models for the ν-calculus that validate basic equivalences concerning names may be constructed using functor categories or nominal sets, with a dynamic allocation monad used to model computations that may allocate fresh names. If recursion is added to the language and one attempts to adapt the models from (nominal) sets to (nominal) domains, however, the direct-style construction of the allocation monad no longer works. This issue has previously been addressed by using a monad that combines dynamic allocation with continuations, at some cost to abstraction.

This paper presents a direct-style model of a ν-calculus-like language with recursion using the novel framework of *proof-relevant logical relations*, in which logical relations also contain objects (or proofs) demonstrating the equivalence of (the semantic counterparts of) programs. Apart from providing a fresh solution to an old problem, this work provides an accessible setting in which to introduce the use of proof-relevant logical relations, free of the additional complexities associated with their use for more sophisticated languages.

1 Introduction

Reasoning about contextual equivalence in higher-order languages that feature dynamic allocation of names, references, objects or keys is challenging. Pitts and Stark's ν-calculus boils the problem down to its purest form, being a total, simply-typed lambda calculus with just names and booleans as base types, an operation `new` that generates fresh names, and equality testing on names. The full equational theory of the ν-calculus is surprisingly complex and has been studied both operationally and denotationally, using logical relations [16,11], environmental bisimulations [6] and nominal game semantics [1,17].

Even before one considers 'exotic' equivalences, there are two basic equivalences that hold for essentially all forms of generativity:

$$(\texttt{let}\ x \Leftarrow \texttt{new}\ \texttt{in}\ e) = e,\ \text{provided } x \text{ is not free in } e. \qquad \text{(Drop)}$$
$$(\texttt{let}\ x \Leftarrow \texttt{new}\ \texttt{in}\ \texttt{let}\ y \Leftarrow \texttt{new}\ \texttt{in}\ e) = (\texttt{let}\ y \Leftarrow \texttt{new}\ \texttt{in}\ \texttt{let}\ x \Leftarrow \texttt{new}\ \texttt{in}\ e) \ \text{(Swap)}.$$

The (Drop) equivalence says that removing the generation of unused names preserves behaviour; this is sometimes called the 'garbage collection' rule. The (Swap) equivalence says that the order in which names are generated is immaterial. These two equations also appear as structural congruences for name restriction in the π-calculus.

Denotational models for the ν-calculus validating (Drop) and (Swap) may be constructed using (pullback-preserving) functors in $\mathit{Set}^{\mathbf{W}}$, where $\mathbf{W}$ is the category of sets and injections [16], or in FM-sets [10]. These models use a dynamic allocation monad to interpret possibly-allocating computations. One might expect that moving to $\mathit{Cpo}^{\mathbf{W}}$ or FM-cpos would allow such models to adapt straightforwardly to a language with recursion, and indeed Shinwell, Pitts and Gabbay originally proposed [15] a dynamic allocation monad over FM-cpos. However, it turned out that the underlying FM-cppo of such monad does not have least upper bounds for all finitely-supported chains. A counter-example is given in Shinwell's thesis [13, page 86]. To avoid the problem, Shinwell and Pitts subsequently [14] moved to an *indirect-style* model, using a *continuation monad* [11]: $(-)^{\top\top} \stackrel{def}{=} (- \to 1_\perp) \to 1_\perp$ to interpret computations. In particular, one shows that two programs are equivalent by proving that they co-terminate in any context. The CPS approach was also adopted by Benton and Leperchey [7] for modelling a language with references.

In the context of our on-going research on the semantics of effect-based program transformations [5], we have been developing *proof-relevant* logical relations [3]. These interpret types not merely as partial equivalence relations, as is commonly done, but as a proof-relevant generalization thereof: *setoids*. A setoid is like a category all of whose morphisms are isomorphisms (a groupoid) with the difference that no equations between these morphisms are imposed. The objects of a setoid establish that values inhabit semantic types, whilst its morphisms are understood as explicit proofs of semantic equivalence. This paper shows how we can use proof-relevant logical relations to give a direct-style model of a language with name generation and recursion, validating (Drop) and (Swap). Apart from providing a fresh approach to an old problem, our aim in doing this is to provide a comparatively accessible presentation of proof-relevant logical relations in a simple setting, free of the extra complexities associated with specialising them to abstract regions and effects [3].

Section 2 sketches the language with which we will be working, and a naive 'raw' domain-theoretic semantics for it. This semantics does not validate interesting equivalences, but is adequate. By constructing a realizability relation between it and the more abstract semantics we subsequently introduce, we will be able to show adequacy of the more abstract semantics. In Section 3 we introduce our category of setoids; these are predomains where there is a (possibly-empty) set of 'proofs' witnessing the equality of each pair of elements. We then describe functors from the category of worlds $\mathbf{W}$ into the category of setoids. Such functors will interpret types of our language in the more abstract semantics, with morphisms between them interpreting terms. The interesting construction here is that of a dynamic allocation monad over the category of functors. Section 4 shows how the abstract semantics is defined and related to the more concrete one. Section 5 then shows how the semantics may be used to establish equivalences involving name generation.

2 Syntax and Semantics

We work with an entirely conventional CBV language, featuring recursive functions and base types that include names, equipped with equality testing and fresh name generation (here + is just a representative operation on integers):

$$\begin{aligned}
\tau &:= \texttt{int} \mid \texttt{bool} \mid \texttt{name} \mid \tau \to \tau' \\
v &:= x \mid b \mid i \mid \texttt{rec}\ f\ x = e \\
e &:= v \mid v + v' \mid v = v' \mid \texttt{new} \ \mid \texttt{let}\ x \Leftarrow e\ \texttt{in}\ e' \mid v\,v' \\
&\quad\ \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e' \\
\Gamma &:= x_1 : \tau_1, \ldots, x_n : \tau_n
\end{aligned}$$

There are typing judgements for values, $\Gamma \vdash v : \tau$, and computations, $\Gamma \vdash e : \tau$, defined as usual. In particular, $\Gamma \vdash \texttt{new} : \texttt{name}$. We define a simple-minded concrete denotational semantics $\llbracket\cdot\rrbracket$ for this language using predomains and continuous maps. For types we take

$$\begin{array}{l}
\llbracket\texttt{int}\rrbracket = \mathbb{Z} \qquad \llbracket\texttt{bool}\rrbracket = \mathbb{B} \qquad \llbracket\texttt{name}\rrbracket = \mathbb{N} \\
\llbracket\tau \to \tau'\rrbracket = \llbracket\tau\rrbracket \to (\mathbb{N} \to \mathbb{N} \times \llbracket\tau'\rrbracket)_\bot \\
\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n\rrbracket = \llbracket\tau_1\rrbracket \times \cdots \times \llbracket\tau_n\rrbracket
\end{array}$$

and there are then conventional clauses defining

$$\begin{array}{l}
\llbracket\Gamma \vdash v : \tau\rrbracket : \llbracket\Gamma\rrbracket \to \llbracket\tau\rrbracket \qquad \text{and} \\
\llbracket\Gamma \vdash e : \tau\rrbracket : \llbracket\Gamma\rrbracket \to (\mathbb{N} \to \mathbb{N} \times \llbracket\tau\rrbracket)_\bot
\end{array}$$

Note that this semantics just uses naturals to interpret names, and a state monad over names to interpret possibly-allocating computations. For allocation we take

$$\llbracket\Gamma \vdash \texttt{new} : \texttt{name}\rrbracket(\eta) = [\lambda n.(n+1, n)]$$

returning the next free name and incrementing the name supply. This semantics validates no interesting equivalences involving names, but is adequate for the obvious operational semantics. Our more abstract semantics, $[\![\cdot]\!]$, will be related to $\llbracket\cdot\rrbracket$ in order to establish *its* adequacy.

3 Proof-Relevant Logical Relations

We define the *category of setoids* as the exact completion of the category of predomains, see [9,8]. We give here an elementary description using the language of dependent types. A *setoid* A consists of a predomain $|A|$ and for any two $x, y \in |A|$ a set $A(x, y)$ of "proofs" (that x and y are equal). The set of triples $\{(x, y, p) \mid p \in A(x, y)\}$ must itself be a predomain and the first and second projections must be continuous. Furthermore, there are continuous functions $r_A : \Pi x \in |A|.A(x, x)$ and $s_A : \Pi x, y \in |A|.A(x, y) \to A(y, x)$ and $t_A : \Pi x, y, z.A(x, y) \times A(y, z) \to A(x, z)$, witnessing reflexivity, symmetry and transitivity; note that no equations between these are imposed.

We should explain what continuity of a dependent function like $t(-, -)$ is: if $(x_i)_i$ and $(y_i)_i$ and $(z_i)_i$ are ascending chains in A with suprema x, y, z and $p_i \in A(x_i, y_i)$ and $q_i \in A(y_i, z_i)$ are proofs such that $(x_i, y_i, p_i)_i$ and $(y_i, z_i, q_i)_i$ are ascending chains, too, with suprema (x, y, p) and (y, z, q) then $(x_i, z_i, t(p_i, q_i))$ is an ascending chain of proofs (by monotonicity of $t(-,-)$) and its supremum is $(x, z, t(p, q))$. Formally, such dependent functions can be reduced to non-dependent ones using pullbacks, that is t would be a function defined on the pullback of the second and first projections from $\{(x, y, p) \mid p \in A(x, y)\}$ to $|A|$, but we find the dependent notation to be much more readable. If $p \in A(x, y)$ we may write $p : x \sim y$ or simply $x \sim y$. We also omit $|-|$ wherever appropriate. We remark that "setoids" also appear in constructive mathematics and formal proof, see *e.g.*, [2], but the proof-relevant nature of equality proofs is not exploited there and everything is based on sets (types) rather than predomains. A morphism from setoid A to setoid B is an equivalence class of pairs $f = (f_0, f_1)$ of continuous functions where $f_0 : |A| \to |B|$ and $f_1 : \Pi x, y \in |A|.A(x, y) \to B(f_0(x), f_0(y))$. Two such pairs $f, g : A \to B$ are *identified* if there exists a continuous function $\mu : \Pi a \in |A|.B(f(a), g(a))$.

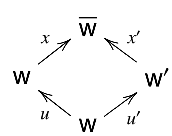
Proposition 1. *The category of setoids is cartesian closed; moreover, if D is a setoid such that $|D|$ has a least element $\bot$ and there is also a least proof $\bot \in D(\bot, \bot)$ then there is a morphism of setoids $Y : [D \to D] \to D$ satisfying the usual fixpoint equations.*

Definition 1. *A setoid D is discrete if for all $x, y \in D$ we have $|D(x, y)| \leq 1$ and $|D(x, y)| = 1 \iff x = y$.*

Thus, in a discrete setoid proof-relevant equality and actual equality coincide and moreover any two equality proofs are actually equal (proof irrelevance).

3.1 Pullback squares

Pullback squares are a central notion in our framework. As it will become clear later, they are the "proof-relevant" component of logical relations. Recall that a morphism u in a category is a monomorphism if $ux = ux'$ implies $x = x'$ for all morphisms x, x'. A commuting square $xu = x'u'$ of morphisms is a *pullback* if whenever $xv = x'v'$ there is unique t such that $v = ut$ and $v' = u't$. This can be visualized as follows:



We write ${}^{x}_{u}\Diamond^{x'}_{u'}$ or $\mathsf{w}\,{}^{x}_{u}\Diamond^{x'}_{u'}\,\mathsf{w}'$ (when $\mathsf{w}^{(\prime)} = \mathrm{dom}(x^{(\prime)})$) for such a pullback square. We call the common codomain of x and x' the *apex* of the pullback, written $\overline{\mathsf{w}}$, while the common domain of u, u' is the *low point* of the square, written $\underline{\mathsf{w}}$. A pullback square $xu = x'u'$ is *minimal* if whenever $fx = gx$ and $fx' = gx'$ then $f = g$, in other words, x and x' are *jointly epic*. A pair of morphisms u, u' with common domain is a span, a pair of morphisms x, x' with common codomain is a co-span. A category has pullbacks if every co-span can be completed to a pullback square.

In our more general treatment of proof-relevant logical relations for reasoning about stateful computation [3], we treat worlds axiomatically, defining a category of worlds to be a category with pullbacks in which every span can be completed to a minimal pullback square, and all morphisms are monomorphisms. That report gives various useful examples, including ones built from PERs on heaps. For the simpler setting of this paper, however, we fix on one particular instance:

Definition 2 (Category of worlds). *The* category of worlds $\mathbf{W}$ *has finite sets of natural numbers as objects and injective functions for morphisms.*

An object w of $\mathbf{W}$ is a set of generated/allocated names, with injective maps corresponding to renamings and extensions with newly generated names.

Given $f : X \to Z$ and $g : Y \to Z$ forming a co-span in $\mathbf{W}$, we form their pullback as $X \xleftarrow{f^{-1}} fX \cap gY \xrightarrow{g^{-1}} Y$. This is minimal when $fX \cup gY = Z$. Conversely, given a span $Y \xleftarrow{f} X \xrightarrow{g} Z$, we can complete to a minimal pullback by

$$(Y \setminus fX) \uplus fX \xrightarrow{[in_1, in_3 \circ f^{-1}]} (Y \setminus fX) + (Z \setminus gX) + X \xleftarrow{[in_2, in_3 \circ g^{-1}]} (Z \setminus gX) \uplus gX$$

where $[-,-]$ is case analysis on the disjoint union $Y = (Y \setminus fX) \uplus fX$. Thus a minimal pullback square in $\mathbf{W}$ is of the form:

$$\begin{array}{ccccc} & & X'_1 \cup X'_2 & & \\ & \overset{x}{\nearrow} & & \overset{x'}{\nwarrow} & \\ X_1 \cong X'_1 & & & & X_2 \cong X'_2 \\ & \underset{u}{\nwarrow} & & \underset{u'}{\nearrow} & \\ & & X'_1 \cap X'_2 & & \end{array}$$

Such a minimal pullback corresponds to a *partial bijection* between X_1 and X_2, as used in other work on logical relations for generativity [12,4]. We write $u : x \hookrightarrow y$ to mean that u is a subset inclusion and note that if we have a span u, u' then we can choose x, x' so that ${}^{x}_{u}\diamond^{x'}_{u'}$ is a minimal pullback and x' is an inclusion, too. To do that, we simply replace the apex of any minimal pullback completion with an isomorphic one. The analogous property holds for completion of co-spans to pullbacks.

Definition 3. *Two pullbacks* $\mathsf{w}{}^{x}_{u}\diamond^{x'}_{u'}\mathsf{w}'$ *and* $\mathsf{w}{}^{y}_{v}\diamond^{y'}_{v'}\mathsf{w}'$ *are isomorphic if there is an isomorphism f between the two low points of the squares so that $vf = u$ and $v'f = u'$, thus also $uf^{-1} = v$ and $u'f^{-1} = v'$.*

Lemma 1. *If* $\mathsf{w}, \mathsf{w}', \mathsf{w}'' \in \mathbf{W}$, *if* $\mathsf{w}{}^{x}_{u}\diamond^{x'}_{u'}\mathsf{w}'$ *and* $\mathsf{w}'{}^{y}_{v}\diamond^{y'}_{v'}\mathsf{w}''$ *are pullback squares as indicated then there exist z, z', t, t' such that* $\mathsf{w}{}^{zx}_{ut}\diamond^{z'y'}_{v't'}\mathsf{w}''$ *is also a pullback.*

Proof. Choose z, z', t, t' in such a way that ${}^{z}_{x'}\diamond^{z'}_{y}$ and ${}^{u'}_{t}\diamond^{v}_{t'}$ are pullbacks. The verifications are then an easy diagram chase.

We write $r(\mathsf{w})$ for $\mathsf{w}{}^{1}_{1}\diamond^{1}_{1}\mathsf{w}$ and $s({}^{x}_{u}\diamond^{x'}_{u'}) = {}^{x'}_{u'}\diamond^{x}_{u}$ and $t({}^{x}_{u}\diamond^{x'}_{u'}, {}^{y}_{v}\diamond^{y'}_{v'}) = {}^{zx}_{z'y'}\diamond^{ut}_{v't'}$ where z, z', t, t' are given by Lemma 1 (which requires choice).

Lemma 2. *A pullback square* ${}^{x}_{u}\diamond^{x'}_{u'}$ *in* $\mathbf{W}$ *is isomorphic to* $t({}^{x}_{1}\diamond^{1}_{x}, {}^{1}_{x'}\diamond^{x'}_{1})$.

3.2 Setoid-valued functors

A functor A (actually a pseudo functor) from the category of worlds $\mathbf{W}$ to the category of setoids comprises as usual for each $\mathsf{w} \in \mathbf{W}$ a setoid $A\mathsf{w}$ and for each $u : \mathsf{w} \to \mathsf{w}'$ a morphism of setoids $Au : A\mathsf{w} \to A\mathsf{w}'$ preserving identities and composition; for an identity morphism id, a continuous function of type $\Pi a.A\mathsf{w}(a,(Aid)\,a)$; and for two morphisms $u:\mathsf{w}\to\mathsf{w}_1$ and $v:\mathsf{w}_1\to\mathsf{w}_2$ a continuous function of type $\Pi a.A\mathsf{w}_2(Av(Au\,a),A(vu)\,a)$.

If $u:\mathsf{w}\to\mathsf{w}'$ and $a\in A\mathsf{w}$ we may write $u.a$ or even ua for $Au(a)$ and likewise for proofs in $A\mathsf{w}$. Note that there is a proof of equality of $(uv).a$ and $u.(v.a)$.

In the sequel, we shall abbreviate these setoid-valued (pseudo-)functors as s.v.f.

Intuitively, s.v.f. will become the denotations of value types and computations. Thus, an element of $A\mathsf{w}$ represents values involving the names in w. If $u:\mathsf{w}\to\mathsf{w}_1$ then $A\mathsf{w}\ni a\mapsto u.a\in A\mathsf{w}_1$ represents renaming and possible weakening by names not "actually" occurring in a. Note that due to the restriction to injective functions identification of names ("contraction") is precluded. This is in line with Stark's use of set-valued functors on the category **W** to model fresh names.

Definition 4. *We call a functor A* pullback-preserving *(s.v.f.) if for every pullback square $\mathsf{w}{}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ with apex $\overline{\mathsf{w}}$ and low point $\underline{\mathsf{w}}$ the diagram $A\mathsf{w}{}^{Ax}_{Au}\Diamond^{Ax'}_{Au'}A\mathsf{w}'$ is a pullback in Std. This means that there is a continuous function of type*

$$\Pi a\in A\mathsf{w}.\Pi a'\in A\mathsf{w}'.A\overline{\mathsf{w}}(x.a,x'.a')\to\Sigma\underline{a}\in A\underline{\mathsf{w}}.A\mathsf{w}(u.\underline{a},a)\times A\mathsf{w}'(u'.\underline{a},a')$$

Thus, if two values $a\in A\mathsf{w}$ and $a'\in A\mathsf{w}'$ are equal in a common world $\overline{\mathsf{w}}$ then this can only be the case because there is a value in the "intersection world" $\underline{\mathsf{w}}$ from which both a,a' arise.

All the s.v.f. that we define in this paper will turn out to be pullback-preserving. However, for the results described in this paper pullback preservation is not needed. Thus, we will not use it any further, but note that there is always the option to require that property should the need arise subsequently.

Lemma 3. *If A is a s.v.f., $u:\mathsf{w}\to\mathsf{w}'$ and $a,a'\in A\mathsf{w}$, there is a continuous function $A\mathsf{w}'(u.a,u.a')\to A\mathsf{w}(a,a')$. Moreover, the "common ancestor" $\underline{a}$ of a and a' is unique up to $\sim$.*

Note that the ordering on worlds and world morphisms is discrete so that continuity only refers to the $A\mathsf{w}'(u.a,u.a')$ argument.

Definition 5 (Morphism of functors). *If A,B are s.v.f., a morphism from A to B is a pair $e=(e_0,e_1)$ of continuous functions where $e_0:\Pi\mathsf{w}.A\mathsf{w}\to B\mathsf{w}$ and $e_1:\Pi\mathsf{w}.\Pi\mathsf{w}'.\Pi x:\mathsf{w}\to\mathsf{w}'.\Pi a\in A\mathsf{w}.\Pi a'\in A\mathsf{w}'.A\mathsf{w}'(x.a,a')\to B\mathsf{w}'(x.e_0(a),e_0(a'))$. A proof that morphisms e,e' are equal is given by a continuous function $\mu:\Pi\mathsf{w}.\Pi a\in A\mathsf{w}.B\mathsf{w}(e(a),e'(a))$.*

These morphisms compose in the obvious way and so the s.v.f. and morphisms between them form a category.

3.3 Instances of setoid-valued functors

We now describe some concrete functors that will allow us to interpret types of the ν-calculus as s.v.f. The simplest one endows any predomain with the structure of a s.v.f. where the equality is proof-irrelevant and coincides with standard equality. The second one generalises the function space of setoids and is used to interpret function types. The third one is used to model dynamic allocation and is the only one that introduces proper proof-relevance.

Constant functor Let D be a predomain. Then the s.v.f. over this domain, written also as D, has D itself as underlying set (irrespective of w), i.e., $D\mathsf{w} = D$ and $D\mathsf{w}(d, d')$ is given by a singleton set, say, $\{\star\}$ if $d = d'$ and is empty otherwise.

Names The s.v.f. N of names is given by $N\mathsf{w} = \mathsf{w}$ where w on the right hand side stands for the discrete setoid over the discrete cpo of locations in w. Thus, e.g. $N\{1, 2, 3\} = \{1, 2, 3\}$.

Product Let A and B be s.v.f. The product $A \times B$ is the s.v.f. given as follows. We have $(A \times B)\mathsf{w} = A\mathsf{w} \times B\mathsf{w}$ (product predomain) and $(A \times B)\mathsf{w}((a, b), (a', b')) = A\mathsf{w}(a, a') \times B\mathsf{w}(b, b')$. This defines a cartesian product on the category of s.v.f. More generally, we can define indexed products $\prod_{i \in I} A_i$ of a family $(A_i)_i$ of s.v.f.

Function Space Let A and B be s.v.f. The function space $A \Rightarrow B$ is the s.v.f. given as follows. We have $(\mathsf{f}_0, \mathsf{f}_1) \in (A \Rightarrow B)\mathsf{w}$ when f_0 has type $\Pi\mathsf{w}_1\Pi u : \mathsf{w} \to \mathsf{w}_1.A\mathsf{w}_1 \to B\mathsf{w}_1$, that is, it takes a morphism $u : \mathsf{w} \to \mathsf{w}_1$ and an object in $A\mathsf{w}_1$ and returns an object in $B\mathsf{w}_1$. The second component, f_1, which takes care of proofs is a bit more complicated, having type:

$$\begin{aligned}&\Pi\mathsf{w}_1.\Pi\mathsf{w}_2.\Pi u : \mathsf{w} \to \mathsf{w}_1.\Pi v : \mathsf{w}_1 \to \mathsf{w}_2.\Pi a \in A\mathsf{w}_1.\Pi a' \in A\mathsf{w}_2.\\ &\quad A\mathsf{w}_2(v.a, a') \to B\mathsf{w}_2(v.\mathsf{f}_0(u, a), \mathsf{f}_0(vu, a'))\end{aligned}$$

Intuitively, the definition above encompasses two desired properties. The first one is when v is instantiated as the identity yielding a function of mapping proofs in $A\mathsf{w}_1$ to proofs in $B\mathsf{w}_1$:

$$\Pi\mathsf{w}_1.\Pi u : \mathsf{w} \to \mathsf{w}_1.\Pi a \in A\mathsf{w}_1.\Pi a' \in A\mathsf{w}_1.A\mathsf{w}_1(a, a') \to B\mathsf{w}_1(\mathsf{f}_0(u, a), \mathsf{f}_0(u, a'))$$

We note that, since A is only a pseudo functor we must compose with a proof that $id.f_0(u, a)$ equals $f_0(u, a)$.

The second desired property is that the proof in $B\mathsf{w}_2$ can be achieve either by obtaining an object, $\mathsf{f}_0(vu, a')$, the directly from $A\mathsf{w}_2$, or by first obtaining an object $\mathsf{f}_0(u, a)$ in $B\mathsf{w}_1$ and then taking it to $B\mathsf{w}_2$ by using v.

Definition 6. *A s.v.f. A is discrete if* $A\mathsf{w}$ *is a discrete setoid for every world* w.

The constructions presented so far only yield discrete s.v.f., *i.e.*, proof relevance is merely propagated but never actually created. This is not so for the next operator on s.v.f. which is to model dynamic allocation.

Dynamic Allocation Monad Finally, the third instantiation is the dynamic allocation monad T. For natural number n let us write $[n]$ for the set $\{1, \ldots, n\}$.

Let A be a s.v.f., then the elements of $(TA)\mathsf{w}$ are again pairs $(\mathsf{c}_0, \mathsf{c}_1)$ where c_0 is of type

$$\Pi n \in \{n \mid [n] \supseteq \mathsf{w}\}.(\Sigma\mathsf{w}_1.I(\mathsf{w}, \mathsf{w}_1) \times A\mathsf{w}_1 \times \{n_1 \mid [n_1] \supseteq \mathsf{w}_1\})_\perp$$

where $I(\mathsf{w}, \mathsf{w}_1)$ is the set of inclusions $u : \mathsf{w} \hookrightarrow \mathsf{w}_1$ and such that either $\mathsf{c}_0(n) = \perp$ for all n such that $[n] \supseteq \mathsf{w}$ or else $\mathsf{c}_0(n) \neq \perp$ for all such n. The naturals n and n_1 represent concrete allocator states, whilst w and w_1 are smaller sets of names on which values actually depend.

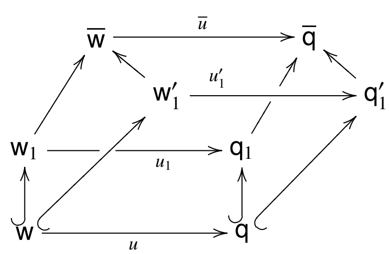
The second component c_1 assigns to any two n, n' with $[n] \supseteq \mathsf{w}, [n'] \supseteq \mathsf{w}$ where $\mathsf{c}_0(n) = (\mathsf{w}_1, u, \mathsf{v}, n_1)$ and $\mathsf{c}_0(n') = (\mathsf{w}'_1, u', \mathsf{v}', n'_1)$ a co-span x, x' such that $xu = x'u'$ and a proof $p \in A\overline{\mathsf{w}}(x.\mathsf{v}, x'.\mathsf{v}')$ with $\overline{\mathsf{w}}$ the apex of the co-span.

The ordering on $(TA)\mathsf{w}$ is given by $(\mathsf{c}_0, \mathsf{c}_1) \sqsubseteq (\mathsf{c}'_0, \mathsf{c}'_1)$ just when $\mathsf{c}_0 \sqsubseteq \mathsf{c}'_0$ in the natural componentwise fashion (the second components are ignored).

A proof in $TA\mathsf{w}((\mathsf{c}_0, \mathsf{c}_1), (\mathsf{c}'_0, \mathsf{c}'_1))$ is defined analogously. For any n such that $[n] \supseteq \mathsf{w}$ it must be that $\mathsf{c}_0(n) = \bot \iff \mathsf{c}'_0(n) = \bot$ (otherwise there is no proof) and if $\mathsf{c}_0(n) = (\mathsf{w}_1, u, \mathsf{v}, n_1)$ and $\mathsf{c}'_0(n) = (\mathsf{w}'_1, u', \mathsf{v}', n'_1)$ then the proof must assign a co-span x, x' such that $xu = x'u'$ and a proof $p \in A\overline{\mathsf{w}}(x.\mathsf{v}, x'.\mathsf{v}')$ with $\overline{\mathsf{w}}$ the apex of the co-span. If $\mathsf{c}_0(n) = \mathsf{c}'_0(n) = \bot$ then the proof is trivial (need not return anything).

To make TA a pseudo-functor, we also have to give its action on morphisms. Assume that $u : \mathsf{w} \to \mathsf{q}$ is a morphism in $\mathbf{W}$. We want to construct a morphism $(TA)u : (TA)\mathsf{w} \to (TA)\mathsf{q}$ in *Std*, so assume $(\mathsf{c}_0, \mathsf{c}_1) \in (TA)\mathsf{w}$ and $[m] \supseteq \mathsf{q}$. We let n be the largest element of w, an arbitrary choice ensuring $[n] \supseteq \mathsf{w}$. If $\mathsf{c}_0(n) = \bot$, then define $\mathsf{d}_0(m) = \bot$ too. Otherwise $\mathsf{c}_0(n) = (\mathsf{w}_1, i : \mathsf{w} \hookrightarrow \mathsf{w}_1, \mathsf{v}, n_1)$, and we define $\mathsf{d}_0(m) = (\mathsf{q}_1, i', u_1.\mathsf{v}, m_1)$ where q_1, $i' : \mathsf{q} \hookrightarrow \mathsf{q}_1$ and $u_1 : \mathsf{w}_1 \to \mathsf{q}_1$ are chosen to make $\mathsf{q}\,{}^{i'}_{u}\!\diamond^{u_1}_{i}\,\mathsf{w}_1$ a minimal pullback, and m_1 is (again arbitrarily) the largest element of q_1. We then take $(TA)(u)(\mathsf{c}_0, \mathsf{c}_1)$ to be $(\mathsf{d}_0, \mathsf{d}_1)$, where d_1 just has to return identity co-spans. This specifies how the functor TA transports objects from w to q using the morphism u.

The following diagram illustrates how an equality proof in $TA((\mathsf{c}_0, \mathsf{c}_1), (\mathsf{c}'_0, \mathsf{c}'_1))$ is transported to an equality proof in $TA(u(\mathsf{c}_0, \mathsf{c}_1), u(\mathsf{c}'_0, \mathsf{c}'_1))$.



Here $\mathsf{w}_1 \diamond \mathsf{w}'_1$ and $\mathsf{q}_1 \diamond \mathsf{q}'_1$ are pullback squares. It is easy to check how the morphisms u_1, u'_1 and $\bar{u}$ are constructed. Then we can take the values a and a' in $A\mathsf{w}_1$ and $A\mathsf{w}'_1$ and the proof p in $A\overline{\mathsf{w}}$ to the pullback square $\mathsf{q}_1 \diamond \mathsf{q}'_1$, by using u_1, u'_1 and $\bar{u}$, *i.e.*, $u_1.a \in A\mathsf{q}_1$, $u'_1.a' \in A\mathsf{q}'_1$ and $\bar{u}.p \in A\bar{q}$.

The following is direct from the definitions.

Proposition 2. *T is a monad on the category of s.v.f.; the unit sends $\mathsf{v} \in A\mathsf{w}$ to $(\mathsf{w}, id_{\mathsf{w}}, \mathsf{v}, n) \in (TA)\mathsf{w}$ and the multiplication sends $(\mathsf{w}_1, u, (\mathsf{w}_2, v, \mathsf{v}, n_2), n_1) \in (TTA)\mathsf{w}$ to $(\mathsf{w}_2, vu, \mathsf{v}, n_2) \in TA\mathsf{w}$. If $\mu : A \to B$ then $T\mu : TA \to TB$ at world w sends $(\mathsf{w}_1, u, \mathsf{v}, n_1) \in TA\mathsf{w}$ to $(\mathsf{w}_1, u, \mu u(\mathsf{v}), n_1) \in TB\mathsf{w}$.*

Comparison with FM domains It is well-known that Gabbay-Pitts FM-sets [10] are equivalent to pullback-preserving functors from our category of worlds $\mathbf{W}$ to the category of sets. Likewise, Pitts and Shinwell's FM-domains are equivalent to pullback preserving functors from **W** to the category of domains, thus corresponding exactly to the pullback-preserving discrete s.v.f.

As mentioned in the introduction, Mark Shinwell discusses a flawed attempt at defining a name allocation monad on the category of FM-domains which when transported along the equivalence between FM-domains and discrete s.v.f. would look as follows: Given a discrete s.v.f. A and world w define $SA\mathsf{w}$ as the set of triples $(\mathsf{w}_1, u, \mathsf{v})$ where $u : \mathsf{w} \hookrightarrow \mathsf{w}_1$ and $\mathsf{v} \in A\mathsf{w}_1$ modulo the equivalence relation generated by the identification of $(\mathsf{w}_1, u, \mathsf{v})$ with $(\mathsf{w}'_1, u', \mathsf{v}')$ if there exists a co-span v, v' such that $vu = v'u'$ and $v.\mathsf{v} = v'.\mathsf{v}'$.

As for the ordering, the only reasonable choice is to decree that on representatives $(\mathsf{w}_1, u, \mathsf{v}) \leq (\mathsf{w}'_1, u', \mathsf{v}')$ if $v.\mathsf{v} \leq v'.\mathsf{v}'$ for some co-span v, v' with $vu = v'u'$. However, while this defines a partial order it is not clear why it should have suprema of ascending chains and indeed, Shinwell's thesis [13] contains a concrete counterexample.

We also remark that this construction *does* work if we work with sets rather than predomains and thus do not need orderings or suprema. However, the exact completion of the category sets being equivalent to the category of sets itself is not very surprising.

The previous solution to this conundrum was to move to a continuation-passing style semantics or, equivalently, to use $\top\top$-closure. Intuitively, rather than quantifying existentially over sets of freshly allocated names, one quantifies universally over continuations, which has better order-theoretic properties. Using continuations, however, makes the derivation of concrete equivalences much more difficult and in some cases we still do not know whether it is possible at all.

4 Observational Equivalence and Fundamental Lemma

We now construct the machinery that connects the concrete language with the denotational machinery introduced in Section 2. In particular, we define the semantics of types, written using $[\![\cdot]\!]$, as s.v.f. inductively as follows:

- For basic types $[\![\tau]\!]$ is the corresponding discrete s.v.f..
- $[\![\tau \to \tau']\!]$ is defined as the function space $[\![\tau]\!] \to T[\![\tau]\!]$, where T is the dynamic allocation monad.
- For typing context Γ we define $[\![\Gamma]\!]$ as the indexed product of s.v.f. $\prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]$.

To each term in context $\Gamma \vdash e : \tau$ we can associate a morphism $[\![e]\!]$ from $[\![\Gamma]\!]$ to $T[\![\tau]\!]$ by interpreting the syntax in the category of s.v.f. using cartesian closure and the fact that T is a monad. We omit the straightforward but perhaps slightly tedious definition and only give the clause for "new" here:

$$[\![\mathtt{new}\ ]\!]\mathsf{w}(n) = (\mathsf{w} \cup \{n+1\}, u, n+1, n+1)$$

Here $u : \mathsf{w} \hookrightarrow \mathsf{w} \cup \{n+1\}$ is the inclusion. Note that since $[n] \supseteq \mathsf{w}$ we have $n+1 \notin \mathsf{w}$.

Our aim is now to relate these morphisms to the computational interpretation $\lceil\!\lceil e \rceil\!\rceil$.

Definition 7. *For each type τ and world* w *we define a relation* $\Vdash^{\tau}_{\mathsf{w}} \subseteq \llceil \tau \rrceil \times [\![\tau]\!]\mathsf{w}$*:*

$$b \Vdash^{\texttt{bool}}_{\mathsf{w}} \mathsf{b} \iff b = \mathsf{b}$$
$$i \Vdash^{\texttt{int}}_{\mathsf{w}} \mathsf{i} \iff i = \mathsf{i}$$
$$l \Vdash^{\texttt{name}}_{\mathsf{w}} k \iff l = k$$
$$f \Vdash^{\tau \to \tau'}_{\mathsf{w}} g \iff \forall \mathsf{w}_1. \forall u : \mathsf{w} \hookrightarrow \mathsf{w}_1. \forall v\, \mathsf{v}. v \Vdash^{\tau}_{\mathsf{w}_1} \mathsf{v} \Rightarrow f(v) \Vdash^{T\tau'}_{\mathsf{w}_1} g_0(u, \mathsf{v})$$
$$c \Vdash^{T\tau}_{\mathsf{w}} \mathsf{c} \iff \begin{array}{l} \forall n. \mathsf{w} \subseteq [n] \Rightarrow (\mathsf{c}(n) = \bot \Leftrightarrow c(n) = \bot) \wedge \\ \quad (\mathsf{c}(n) = (\mathsf{w}_1, u : \mathsf{w} \hookrightarrow \mathsf{w}_1, \mathsf{v}, n_1) \wedge c(n) = (n_1', v) \Rightarrow n_1 = n_1' \wedge v \Vdash^{\tau}_{\mathsf{w}_1} \mathsf{v})). \end{array}$$

The realizability relation for the allocation monad thus specifies that the abstract computation c is related to the concrete computation c at world w if they co-terminate, and if they do terminate then the resulting values are also related.

The following is a direct induction on types.

Lemma 4. *If $u : \mathsf{w} \hookrightarrow \mathsf{w}_1$ is an inclusion as indicated and $v \Vdash^{\tau}_{\mathsf{w}} \mathsf{v}$ then $v \Vdash^{\tau}_{\mathsf{w}_1} u.\mathsf{v}$, too.*

We extend $\Vdash$ to typing contexts by putting

$$\eta \Vdash^{\Gamma}_{\mathsf{w}} \gamma \iff \forall x \in \mathrm{dom}(\Gamma). \eta(x) \Vdash^{\Gamma(x)}_{\mathsf{w}} \gamma(x)$$

for $\eta \in \llceil \Gamma \rrceil$ and $\gamma \in [\![\Gamma]\!]$.

Theorem 1 (Fundamental lemma). *If $\Gamma \vdash e : \tau$ then whenever $\eta \Vdash^{\Gamma}_{\mathsf{w}} \gamma$ then $\llceil e \rrceil \eta \Vdash^{T\tau}_{\mathsf{w}}$ $[\![e]\!](\gamma)$.*

Proof. By induction on typing rules.

The most interesting case is for the `let` case: Assume that $\Gamma \vdash \texttt{let}\, x \Leftarrow e_1 \,\texttt{in}\, e_2 : \tau_2$, where $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. Moreover, assume that $\eta \Vdash^{\Gamma}_{\mathsf{w}} \gamma$, where w is an initial world and that (H1) $\llceil e_1 \rrceil \eta \Vdash^{T\tau_1}_{\mathsf{w}} [\![e_1]\!](\gamma)$ and (H2) $\llceil e_2 \rrceil(\eta, x) \Vdash^{T\tau_2}_{\mathsf{w}_1} [\![e_2]\!](\gamma, [\![x]\!])$ for all $x \Vdash^{\tau_1}_{\mathsf{w}} [\![x]\!]$ and world extension w_1, that is, a world for which there is an inclusion $u : \mathsf{w} \hookrightarrow \mathsf{w}_1$. We define $[\![\texttt{let}\, x \Leftarrow e_1 \,\texttt{in}\, e_2]\!](\mathsf{w})(\gamma)(n)$ for some n where $\mathsf{w} \subseteq [n]$ as follows: If $[\![e_1]\!]\mathsf{w}(\gamma)(n) = (\mathsf{w}_1, u_1 : \mathsf{w} \hookrightarrow \mathsf{w}_1, \mathsf{v}_1, n_1)$ and that $[\![e_2]\!]\mathsf{w}_1(\gamma, \mathsf{v}_1)(n_1) = (\mathsf{w}_2, u_2 : \mathsf{w}_1 \hookrightarrow \mathsf{w}_2, \mathsf{v}_2, n_2)$. Then

$$[\![\texttt{let}\, x \Leftarrow e_1 \,\texttt{in}\, e_2]\!](\mathsf{w})(\gamma)(n) = (\mathsf{w}_2, u_2 u_1 : \mathsf{w} \hookrightarrow \mathsf{w}_2, \mathsf{v}_2, n_2).$$

Otherwise $[\![\texttt{let}\, x \Leftarrow e_1 \,\texttt{in}\, e_2]\!](\mathsf{w})(\gamma)(n) = \bot$ if $[\![e_1]\!]\mathsf{w}(\gamma)(n) = \bot$ or if $[\![e_1]\!]\mathsf{w}(\gamma)(n) = (\mathsf{w}_1, u_1 : \mathsf{w} \hookrightarrow \mathsf{w}_1, \mathsf{v}_1, n_1)$, but $[\![e_2]\!]\mathsf{w}_1(\gamma, \mathsf{v}_1)(n_1) = \bot$.

We only show the case where $[\![\texttt{let}\, x \Leftarrow e_1 \,\texttt{in}\, e_2]\!](\mathsf{w})(\gamma)(n)$ is different from $\bot$. The other cases are straighforward. Assume that $\llceil e_1 \rrceil \eta(n) = (v_1, n_1')$. From (H1), we have $n_1' = n_1$ and that $v_1 \Vdash^{\tau_1}_{\mathsf{w}_1} \mathsf{v}_1$. Thus from Lemma 4, we have $\eta, v_1 \Vdash^{\tau_1}_{\mathsf{w}_1} \gamma, \mathsf{v}_1$. Now, assume that $\llceil e_2 \rrceil(\eta, v_1)(n_1) = (v_2, n_2')$. Thus from (H2), we have that $n_2 = n_2'$ and that $v_2 \Vdash^{\tau_2}_{\mathsf{w}_2}$ v_2. This finishes the proof, as it is enough to conclude that $\llceil \texttt{let}\, e_1 \Leftarrow e_2 \,\texttt{in} \rrceil \eta \Vdash^{T\tau_2}_{\mathsf{w}}$ $[\![\texttt{let}\, e_1 \Leftarrow e_2 \,\texttt{in}]\!]\gamma$.

It is now possible to validate a number of equational rules on the level of the setoid semantics $[\![-]\!]$ including transitivity, $\beta\eta$, fixpoint unrolling, and congruence rules. We omit the definition of such an equational theory here and refer to [3] for details on how this could be set up. As we now show equality on the level of the setoid semantics entails observational equivalence on the level of the raw denotational semantics.

4.1 Observational Equivalence

Definition 8. *Let τ be a type. We define an* observation of type τ *as a closed term* $\vdash o : \tau \to \mathtt{bool}$. *Two values $v, v' \in \lceil\!\lceil \tau \rceil\!\rceil$ are* observationally equivalent at type τ *if for all observations o of type τ one has that $\lceil\!\lceil o \rceil\!\rceil(v)(0)$ is defined iff $\lceil\!\lceil o \rceil\!\rceil(v')(0)$ is defined and when $\lceil\!\lceil o \rceil\!\rceil(v)(0) = (n_1, v_1)$ and $\lceil\!\lceil o \rceil\!\rceil(v')(0) = (n_1', v_1')$ then $v_1 = v_1'$.*

We now show how the proof-relevant semantics can be used to deduce observational equivalences.

Theorem 2 (Observational equivalence). *If τ is a type and $v \Vdash^{\tau}_{\emptyset} e$ and $v' \Vdash^{\tau}_{\emptyset} e'$ with $e \sim e'$ in $[\![\tau]\!]\emptyset$ then v and v' are observationally equivalent at type τ.*

Proof. Let o be an observation at type τ. By the Fundamental Lemma (Theorem 1) we have $\lceil\!\lceil o \rceil\!\rceil \Vdash^{\tau \to \mathtt{bool}}_{\emptyset} [\![o]\!]$.

Now, since $e \sim e'$ we also have $[\![o]\!](e) \sim [\![o]\!](e')$ and, of course, $\lceil\!\lceil o \rceil\!\rceil(v) \Vdash^{T\mathtt{bool}}_{\emptyset} [\![o]\!](e)$ and $\lceil\!\lceil o \rceil\!\rceil(v') \Vdash^{T\mathtt{bool}}_{\emptyset} [\![o]\!](e')$.

From $[\![o]\!](e) \sim [\![o]\!](e')$ we conclude that either $[\![o]\!](e)(0)$ and $[\![o]\!](e')(0)$ both diverge in which case the same is true for $\lceil\!\lceil o \rceil\!\rceil(v)(0)$ and $\lceil\!\lceil o \rceil\!\rceil(v')(0)$ by definition of $\Vdash^{T\tau}$. Secondly, if $[\![o]\!](e)(0)) = (\_, \_, b, \_)$ and $[\![o]\!](e')(0)) = (\_, \_, b', \_)$ for booleans b, b' then, by definition of $\sim$ at $[\![T\tau]\!]$ we get $b = b'$ and, again by definition of $\Vdash^{T\tau}$ this then implies that $\lceil\!\lceil o \rceil\!\rceil(v)(0) = (\_, b)$ and $\lceil\!\lceil o \rceil\!\rceil(v)(0) = (\_, b')$ with $b = b'$, hence the claim.

5 Direct-Style Proofs

We now have enough machinery to provide a direct-style proofs for equivalences involving name generation.

Drop equation We start with the following equation, which allows to eliminate a dummy allocation:

$$c = (\mathtt{let}\ x \Leftarrow \mathtt{new}\ \mathtt{in}\ e) = e, \text{ provided } x \text{ is not free in } e = c\text{'}.$$

Assume an initial world w and suppose that $c' \Vdash^{TA}_{\mathsf{w}} \mathsf{c}'$, where c' is an abstract computation related to c' at world w. We provide a semantic computation c, such that $c \Vdash^{TA}_{\mathsf{w}} \mathsf{c}$, that is, it is related to the computation that performs a dummy allocation, and we also provide a proof $\mathsf{c} \sim \mathsf{c}'$. From Theorem 2, this means that the two computations are observationally equivalent. Let $\mathsf{c} = (\mathsf{w}, id : \mathsf{w} \hookrightarrow \mathsf{w}, c', n)$, which does not advance the world. We can show that it is related to the expression c, with the dummy allocation, *i.e.*, $c \Vdash^{\Gamma \vdash TA}_{\mathsf{w}} \mathsf{c}$ by opening its definition, stated in Definition 7:

$$\begin{array}{l}\forall n. \mathsf{w} \subseteq [n] \Rightarrow (\mathsf{c} = \bot \Leftrightarrow c(n) = \bot) \wedge \\ \quad (\mathsf{c} = (\mathsf{w}, id : \mathsf{w} \hookrightarrow \mathsf{w}, \mathsf{c}', n) \wedge c(n) = ([n_1], c') \Rightarrow (\mathsf{w} \subseteq [n_1] \wedge c' \Vdash^{A}_{\mathsf{w}} \mathsf{c}')).\end{array}$$

where the value c' resulting is exactly the function without the dummy allocation, thus $\mathsf{c} \sim \mathsf{c}'$ with the identity pullback square. The key observation is that heaps $[n]$ are allowed to contain more locations that those in w, containing the locations that one actually needs.

Notice as well that if we were to annotate monads with the corresponding effects of the function, such as read, write or allocation effects, as done in [4], from the proof above the first allocation in c with the dummy allocation would not need to flag an allocation effect. That is, that step could be considered pure.

Swap equation Let us now consider the following equivalence where the order in which the names are generated is switched:

$$c = (\texttt{let}\ x \Leftarrow \texttt{new}\ \texttt{in}\ \texttt{let}\ y \Leftarrow \texttt{new}\ \texttt{in}\ e) = (\texttt{let}\ y \Leftarrow \texttt{new}\ \texttt{in}\ \texttt{let}\ x \Leftarrow \texttt{new}\ \texttt{in}\ e) = c'.$$

For showing that these programs are equivalent, we will need to consider world advancements. Assume that we start from an initial world w. Assume the abstract computations $\mathsf{c}_1 = (\mathsf{w} \cup \{\mathsf{l}_1\}, u_1 : \mathsf{w} \hookrightarrow \mathsf{w} \cup \{\mathsf{l}_1\}, c_2, n_1)$ and $\mathsf{c}'_1 = (\mathsf{w} \cup \{\mathsf{l}'\}, u'_1 : \mathsf{w} \hookrightarrow \mathsf{w} \cup \{\mathsf{l}'\}, c'_2, n'_1)$, where l and l' are the first proper concrete locations allocated. Moreover, let $c_2 = (\mathsf{w} \cup \{\mathsf{l}_1, \mathsf{l}_2\}, u_2 : \mathsf{w} \cup \{\mathsf{l}\} \hookrightarrow \mathsf{w} \cup \{\mathsf{l}_1, \mathsf{l}_2\}, c, n_2)$ and $c'_2 = (\mathsf{w} \cup \{\mathsf{l}'_1, \mathsf{l}'_2\}, u'_2 : \mathsf{w} \cup \{\mathsf{l}'\} \hookrightarrow \mathsf{w} \cup \{\mathsf{l}'_1, \mathsf{l}'_2\}, c', n_2)$, where the second location is allocated. The proof is now the pullback square $\mathsf{w} \cup \{\mathsf{l}_1, \mathsf{l}_2\}\, {}_{u_2 u_1}\!\!\overset{id}{}\!\diamond^{x'}_{u'_2 u'_1}\, \mathsf{w} \cup \{\mathsf{l}'_1, \mathsf{l}'_2\}$, with $\overline{\mathsf{w}} = \mathsf{w} \cup \{\mathsf{l}_1, \mathsf{l}_2\}$ and where x' fixes everything except that it maps l'_2 to l_1 and l'_1 to l_2, *i.e.*, it permutes the allocation order. In this way we get that $id.\mathsf{c} \sim x'.\mathsf{c}'$.

6 Discussion

We have introduced proof-relevant logical relations and shown how they may be used to model and reason about simple equivalences in a higher-order language with recursion and name generation. A key innovation compared with previous functor category models is the use of functors valued in setoids (which are here also built on predomains), rather than plain sets. One payoff is that we can work with a direct style model rather than one based on continuations (which, in the absence of control operators in the language, is less abstract).

The technical machinery used here is not *entirely* trivial, and the reader might be forgiven for thinking it slightly excessive for such a simple language and rudimentary equations. However, our aim has not been to present impressive new equivalences, but rather to present an accessible account of how the idea of proof relevant logical relations works in a simple setting. The companion report [3] gives significantly more advanced examples of applying the construction to reason about equivalences justified by abstract semantic notions of effects and separation, but the way in which setoids are used is there somewhat obscured by the details of, for example, much more sophisticated categories of worlds and a generalization of s.v.f.s for modelling computation types. Our hope is that this account will bring the idea to a wider audience, make the more advanced applications more accessible, and inspire others to investigate the construction in their own work.

Thanks to Andrew Kennedy for numerous discussions, and to an anonymous referee for suggesting that we write up the details of how proof-relevance applies to pure name generation.

References

1. S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *Proc. 19th Annual IEEE Symposium on Logic in Computer Science (LICS '04)*. IEEE Computer Society, 2004.
2. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, 2003.
3. N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. *CoRR*, abs/1212.5692, 2012.
4. N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *Proc. Ninth International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '07)*. ACM, 2007.
5. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *Proc. Fourth Asian Symposium on Programming Languages and Systems (APLAS '06)*, volume 4279 of *LNCS*. Springer, 2006.
6. N. Benton and V. Koutavas. A mechanized bisimulation for the nu-calculus. *Higher-Order and Symbolic Computation*, 2013. To appear.
7. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. Seventh International Conference on Typed Lambda Calculi and Applications (TLCA '05)*, volume 3461 of *LNCS*. Springer, 2005.
8. L. Birkedal, A. Carboni, G. Rosolini, and D. S. Scott. Type theory via exact categories. In *Proc. 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98)*. IEEE Computer Society, 1998.
9. A. Carboni, P. J. Freyd, and A. Scedrov. A categorical approach to realizability and polymorphic types. In *Proc. Third Workshop on Mathematical Foundations of Programming Language Semantics (MFPS '87)*, volume 298 of *LNCS*. Springer, 1987.
10. M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.
11. A. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.
12. A. M. Pitts and I. D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or what's new? In *Proc. 18th International Symposium on Mathematical Foundations of Computer Science (MFCS '93)*, volume 711 of *LNCS*. Springer, 1993.
13. M. R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge, 2004.
14. M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theor. Comput. Sci.*, 342(1):28–55, 2005.
15. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. Eighth ACM SIGPLAN International Conference on Functional programming (ICFP '03)*. ACM, 2003.
16. I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Cambridge, UK, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.
17. N. Tzevelekos. Program equivalence in a simple language with state. *Computer Languages, Systems and Structures*, 38(2), 2012.