Relational Reasoning in a Nominal Semantics for Storage

Nick Benton[1] and Benjamin Leperchey[2]

[1] Microsoft Research, Cambridge
[2] PPS, Université Denis Diderot, Paris

Abstract. We give a monadic semantics in the category of FM-cpos to a higher-order CBV language with recursion and dynamically allocated mutable references that may store both ground data and the addresses of other references, but not functions. This model is adequate, though far from fully abstract. We then develop a relational reasoning principle over the denotational model, and show how it may be used to establish various contextual equivalences involving allocation and encapsulation of store.

1 Introduction

The search for good models and reasoning principles for programming languages with mutable storage has a long history [21], and we are still some way from a fully satisfactory account of general dynamically-allocated storage in higher-order languages. Here we take a small step forward, building on the work of Pitts and Stark on an operational logical relation for an ML-like language with integer references [12], that of Reddy and Yang on a parametric model for a divergence-free, Pascal-like language with heap-allocated references [14] and that of Shinwell and Pitts on an FM-cpo model of FreshML [17].

Section 2 introduces MILler, a monadic, monomorphic, ML-like language with references storing integers or the addresses of other references. Section 3 defines a computationally adequate denotational semantics for MILler by using a continuation monad over FM-cpos (cpos with an action by permutations on location names). Working with FM-cpos is much like doing ordinary domain theory; although it is technically equivalent to using pullback preserving functors from the category of finite sets and injections into Cpo, it is significantly more concrete and convenient in practice. The basic model in FM-cpo gives us an elegant interpretation of dynamic allocation, but fails to validate most interesting equivalences involving the use of encapsulated state.

Section 4 defines a logical relation over our model, parameterized by relations on stores that specify explicitly both a part of the store accessible to an expression and a separated (non-interfering) invariant that is preserved by the context. Section 5 uses the relation to establish a number of equivalences involving the encapsulation of dynamically allocated references, and shows the incompleteness of our reasoning principle. A fuller account of this work, including proofs, more discussion and examples may be found in the companion technical report [2].

$$(rec)\frac{\Delta;\Gamma,x:\tau,f:\tau\to\mathbf{T}(\tau')\vdash M:\mathbf{T}(\tau')}{\Delta;\Gamma\vdash(\mathrm{rec}\,f(x{:}\tau){:}\tau'=M):\tau\to\mathbf{T}(\tau')}\qquad (loc)\frac{\ell:\sigma\in\Delta}{\Delta;\Gamma\vdash\underline{\ell}:\sigma\ \mathrm{ref}}$$

$$(app)\frac{\Delta;\Gamma\vdash V_1:\tau\to\mathbf{T}\tau'\quad\Delta;\Gamma\vdash V_2:\tau}{\Delta;\Gamma\vdash V_1\,V_2:\mathbf{T}\tau'}$$

$$(let)\frac{\Delta;\Gamma\vdash M_1:\mathbf{T}(\tau_1)\quad\Delta;\Gamma,x:\tau_1\vdash M_2:\mathbf{T}(\tau_2)}{\Delta;\Gamma\vdash\mathrm{let}\,x\Leftarrow M_1\,\mathrm{in}\,M_2:\mathbf{T}(\tau_2)}\qquad (val)\frac{\Delta;\Gamma\vdash V:\tau}{\Delta;\Gamma\vdash\mathrm{val}\,V:\mathbf{T}(\tau)}$$

$$(eq)\frac{\Delta;\Gamma\vdash V_1:\sigma\ \mathrm{ref}\quad\Delta;\Gamma\vdash V_2:\sigma\ \mathrm{ref}}{\Delta;\Gamma\vdash V_1=V_2:\mathbf{T}(\mathrm{unit}+\mathrm{unit})}\qquad (deref)\frac{\Delta;\Gamma\vdash V:\sigma\ \mathrm{ref}}{\Delta;\Gamma\vdash!V:\mathbf{T}\sigma}$$

$$(alloc)\frac{\Delta;\Gamma\vdash V:\sigma}{\Delta;\Gamma\vdash\mathrm{ref}\,V:\mathbf{T}(\sigma\ \mathrm{ref})}\qquad (assign)\frac{\Delta;\Gamma\vdash V_1:\sigma\ \mathrm{ref}\qquad\Delta;\Gamma\vdash V_2:\sigma}{\Delta;\Gamma\vdash V_1:=V_2:\mathbf{T}(\mathrm{unit})}$$

Fig. 1. Type Rules for MILler (extract)

2 The Language

MILler (MIL-lite with extended references), is a CBV, monadically-typed λ-calculus with recursion and dynamically allocated references. It is a close relative of the MIL-lite fragment [1] of the intermediate language of MLj and SML.NET, and of ReFS [12]. MILler distinguishes *value types*, τ, from *computation types*, of the form $\mathbf{T}\tau$. The *storable types*, σ, are a subset of the value types:

$$\begin{array}{l}\tau ::= \mathrm{unit} \mid \mathrm{int} \mid \sigma\ \mathrm{ref} \mid \tau\times\tau \mid \tau+\tau \mid \tau\to\mathbf{T}\tau\\ \sigma ::= \mathrm{int} \mid \sigma\ \mathrm{ref}\\ \gamma ::= \tau \mid \mathbf{T}\tau\end{array}$$

Typing contexts, Γ, are finite maps from variable names to value types. We assume an infinite set of locations $\mathbb{L}$, ranged over by ℓ. Store types Δ are finite maps from locations to storable types. Terms, G, are subdivided into values, V, and computations, M:

$$\begin{array}{l}V ::= x \mid \underline{n} \mid \underline{\ell} \mid () \mid (V,V') \mid \mathrm{in}_i^{\tau}V \mid \mathrm{rec}\,f(x{:}\tau){:}\tau'=M\\ M ::= V\,V' \mid \mathrm{let}\,x\Leftarrow M\,\mathrm{in}\,M' \mid \mathrm{val}\,V \mid \pi_i V \mid \mathrm{ref}\,V \mid !V \mid V:=V'\\ \quad \mid \mathrm{case}\,V\,\mathrm{of}\,\mathrm{in}_1 x\Rightarrow M\ ;\ \mathrm{in}_2 x\Rightarrow M'\\ \quad \mid V=V' \mid V+V' \mid \mathrm{iszero}\,V\end{array}$$

Some of the typing rules for MILler are shown in Figure 1. One can define syntactic sugar for booleans, conditionals, non-recursive λ-abstractions, sequencing of commands, etc. in the obvious way.

States, Σ, are finite maps from locations to $\mathbb{Z}+\mathbb{L}$. We write $\mathrm{in}_{\mathbb{Z}}$,$\mathrm{in}_{\mathbb{L}}$ for the injections and $\Sigma[\ell\mapsto\mathrm{in}_{\mathbb{Z}}n]$ (resp.$\mathrm{in}_{\mathbb{L}}$) for updating.

$$\frac{}{\Sigma, \text{let } x \Leftarrow \text{val } V \text{ in val } x \ \downarrow} \qquad \frac{\Sigma, \text{let } y \Leftarrow M[V/x] \text{ in } K \ \downarrow}{\Sigma, \text{let } x \Leftarrow \text{val } V \text{ in (let } y \Leftarrow M \text{ in } K) \ \downarrow}$$

$$\frac{\Sigma, \text{let } x_2 \Leftarrow M_1 \text{ in (let } x_1 \Leftarrow M_2 \text{ in } K) \ \downarrow}{\Sigma, \text{let } x_1 \Leftarrow (\text{let } x_2 \Leftarrow M_1 \text{ in } M_2) \text{ in } K \ \downarrow}$$

$$\frac{\Sigma, \text{let } x_1 \Leftarrow M[V/x_2, (\text{rec } f(x_2 : \tau_1) : \tau_2 = M)/f] \text{ in } K \ \downarrow}{\Sigma, \text{let } x_1 \Leftarrow (\text{rec } f(x_2 : \tau_1) : \tau_2 = M)\ V \text{ in } K \ \downarrow}$$

$$\frac{\Sigma, \text{let } x \Leftarrow \text{val } \mathit{false} \text{ in } K \ \downarrow}{\Sigma, \text{let } x \Leftarrow \underline{\ell} = \underline{\ell'} \text{ in } K \ \downarrow} \ \ell \neq \ell' \qquad \frac{\Sigma[\ell \mapsto \mathit{in}_{\mathbb{L}} \ell'], \text{let } x \Leftarrow \text{val } () \text{ in } K \ \downarrow}{\Sigma, \text{let } x \Leftarrow \underline{\ell} := \underline{\ell'} \text{ in } K \ \downarrow}$$

$$\frac{\Sigma(\ell) = \mathit{in}_{\mathbb{L}} \ell' \quad \Sigma, \text{let } x \Leftarrow \text{val } \underline{\ell'} \text{ in } K \ \downarrow}{\Sigma, \text{let } x \Leftarrow !\underline{\ell} \text{ in } K \ \downarrow}$$

$$\frac{\Sigma[\ell \mapsto \mathit{in}_{\mathbb{L}} \ell'], \text{let } x \Leftarrow \text{val } \underline{\ell} \text{ in } K \ \downarrow}{\Sigma, \text{let } x \Leftarrow \text{ref } \underline{\ell'} \text{ in } K \ \downarrow} \ \ell \notin \mathit{locs}(\Sigma) \cup \mathit{locs}(K) \cup \{\ell'\}$$

Fig. 2. Operational Semantics of MILler (extract)

Definition 1 (Typed states and equivalence). If Σ,Σ' are states, and Δ is a store type, we write $\Sigma \sim \Sigma' : \Delta$ to mean $\forall \ell \in \text{dom}\, \Delta, \Sigma \sim \Sigma' : (\ell : \Delta(\ell))$, where $\Sigma \sim \Sigma' : (\ell : \text{int})$ means $\exists n \in \mathbb{Z}. \Sigma\ell = \text{in}_{\mathbb{Z}} n = \Sigma'\ell$ and $\Sigma \sim \Sigma' : (\ell : \sigma' \text{ ref})$ means $\exists \ell' \in \mathbb{L}. \Sigma\ell = \text{in}_{\mathbb{L}} \ell' = \Sigma'\ell \ \wedge \ \Sigma \sim \Sigma' : (\ell' : \sigma')$. We say that a state Σ has type Δ, and we write $\Sigma : \Delta$, when $\Sigma \sim \Sigma : \Delta$.

The restricted grammar of storable types means that if $\Sigma : \Delta$ then the part of Σ accessible from $\mathit{dom}(\Delta)$ will be acyclic.

The operational semantics is defined using a termination judgement [12] $\Sigma, \text{let } x \Leftarrow M \text{ in } K \ \downarrow$ where M is closed and K is a *continuation term in* x. Typed continuation terms are defined by

$$\frac{}{\Delta; \vdash \text{val } x : (x : \tau)^{\top}} \qquad \frac{\Delta; x : \tau \vdash M : \mathbf{T}\tau' \quad \Delta; \vdash K : (y : \tau')^{\top}}{\Delta; \vdash \text{let } y \Leftarrow M \text{ in } K : (x : \tau)^{\top}}$$

and the rules for defining untyped ones are the same with types (though not variables) erased. Some of the rules defining the termination predicate are shown in Figure 2.

Definition 2 (Contextual equivalence). Contexts, $C[\cdot]$, are 'computation terms with holes in' and we write $C[\cdot] : (\Delta; \Gamma \vdash \gamma) \Rightarrow (\Delta; - \vdash \mathbf{T}\tau)$ to mean that whenever $\Delta; \Gamma \vdash G : \gamma$ then $\Delta; \vdash C[G] : \mathbf{T}\tau$. If $\Delta; \Gamma \vdash G_i : \gamma$ for $i = 1, 2$ then we write $\Delta; \Gamma \vdash G_1 =_{\text{ctx}} G_2 : \gamma$ to mean

$$\forall \tau. \forall C[\cdot] : (\Delta; \Gamma \vdash \gamma) \Rightarrow (\Delta; - \vdash \mathbf{T}\tau). \forall \Sigma : \Delta.$$
$$\Sigma, \text{let } x \Leftarrow C[G_1] \text{ in val } x \ \downarrow \iff \Sigma, \text{let } x \Leftarrow C[G_2] \text{ in val } x \ \downarrow$$

3 Denotational Semantics

We first summarize basic facts about FM-cpos. A more comprehensive account appears in Shinwell's thesis [16].

Fix a countable set of *atoms*, which in our case will be the locations, $\mathbb{L}$. Then an *FM-set* X is a set equipped with a *permutation action*: an operation $\pi \bullet - : perms(\mathbb{L}) \times X \to X$ that preserves composition and identity, and such that each element $x \in X$ is *finitely supported*: there is a finite set $L \subseteq \mathbb{L}$ such that whenever π fixes each element of L, the action of π fixes x: $\pi \bullet x = x$. There is a smallest such set, which we write $supp(x)$. A morphism of FM-sets is a function $f : X \to Y$ between the underlying sets that is *equivariant*: $\forall x. \forall \pi.\, \pi \bullet (f\, x) = f\, (\pi \bullet x)$.

An *FM-cpo* is an FM-set with an equivariant partial order relation $\sqsubseteq$ and least upper bounds of all finitely-supported ω-chains. A morphism of FM-cpos is a morphism of their underlying FM-sets that is monotone and preserves lubs of finitely-supported chains. We only require the existence and preservation of finitely-supported chains, so an FM-cpo may not be a cpo in the usual sense. The sets $\mathbb{Z}$, $\mathbb{N}$, etc. are discrete FM-cpos with the trivial action. The set of locations, $\mathbb{L}$, is a discrete FM-cpo with the action $\pi \bullet \ell = \pi(\ell)$.

The category of FM-cpos is bicartesian closed: we write 1 and $\times$ for the finite products, $\Rightarrow$ for the internal hom and 0,+ for the coproducts. The action on products is pointwise, and on functions is given by conjugation: $\pi \bullet f \stackrel{def}{=} \lambda x. \pi \bullet (f\, (\pi^{-1} \bullet x))$. The category is not well-pointed: morphisms $1 \to D$ correspond to elements of $1 \Rightarrow D$ with empty support.

The lift monad, $(\cdot)_\perp$, is defined as usual with the obvious action. The Kleisli category is the category of pointed FM-cpos (FM-cppos) and strict continuous maps, which is symmetric monoidal closed, with smash product $\otimes$ and strict function space $\multimap$. We use the same notation for partial constructions on the category of FM-cpos, defined on the range of the forgetful functor from FM-cppo. If D is a pointed FM-cpo then $fix : (D \Rightarrow D) \multimap D$ is defined by the lub of an ascending chain in the usual way.

We now turn to the denotational semantics of MILler. Define the FM-cpo of states, $\mathbb{S}$, to be $\mathbb{L} \Rightarrow (\mathbb{Z} + \mathbb{L})$, the finitely-supported functions mapping locations to either locations or integers, and write $[\![\Sigma]\!]$ for $\{S \in \mathbb{S} \mid \forall \ell \in dom\Sigma. S(\ell) = \Sigma(\ell)\}$. The update operation $\cdot[\cdot \mapsto \cdot]$ of type $\mathbb{S} \times \mathbb{L} \times (\mathbb{Z} + \mathbb{L}) \to \mathbb{S}$ is equivariant and continuous. The equivalence of operational states at a type of Definition 1 extends naturally to denotational states.

We write $\mathbb{O}$ for the flat two-element FM-cpo $\{\perp \sqsubseteq \top\}$, with the trivial action and then define the FM-cpo $[\![\gamma]\!]$, interpreting the type γ, inductively:

$$\begin{array}{rlrl} [\![\text{unit}]\!] &= 1 & [\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\ [\![\text{int}]\!] &= \mathbb{Z} & [\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] + [\![\tau_2]\!] \\ [\![\sigma\ \text{ref}]\!] &= \mathbb{L} & [\![\tau_1 \to \mathbf{T}\tau_2]\!] &= [\![\tau_1]\!] \Rightarrow \mathbf{T}[\![\tau_2]\!] \\ \mathbf{T}D &= (\mathbb{S} \Rightarrow D \Rightarrow \mathbb{O}) \multimap (\mathbb{S} \Rightarrow \mathbb{O}) & & \end{array}$$

For terms in context, we define $[\![\Delta; \Gamma \vdash G : \gamma]\!] \in [\![\Gamma]\!] \Rightarrow [\![\gamma]\!]$, where $[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]$ is the record $\{x_1 : [\![\tau_1]\!], \cdots, x_n : [\![\tau_n]\!]\}$, inductively. Some of

$$[\![\Delta; \Gamma \vdash \underline{\ell} : \sigma \text{ ref}]\!]\ \rho \ = \ \ell$$

$$[\![\Delta; \Gamma \vdash \text{let } x \Leftarrow M_1 \text{ in } M_2 : \mathbf{T}\tau_2]\!]\ \rho\ k\ S \ =$$
$$[\![\Delta; \Gamma \vdash M_1 : \mathbf{T}\tau_1]\!]\ \rho\ (\lambda S' : \mathbb{S}.\lambda d : [\![\tau_1]\!].[\![\Delta; \Gamma, x : \tau_1 \vdash M_2 : \mathbf{T}\tau_2]\!]\ \rho[x \mapsto d]\ k\ S')\ S$$

$$[\![\Delta; \Gamma \vdash \text{val}\, V : \mathbf{T}\tau]\!]\ \rho\ k\ S \ = \ k\ S\ ([\![\Delta; \Gamma \vdash V : \tau]\!]\ \rho)$$

$$[\![\Delta; \Gamma \vdash !V : \mathbf{T}\sigma]\!]\ \rho\ k\ S \ = \begin{cases} k\ S\ v \text{ if } S([\![\Delta; \Gamma \vdash V : \sigma \text{ ref}]\!]\ \rho) = in_{[\![\sigma]\!]} v \\ \bot \qquad \text{otherwise} \end{cases}$$

$$[\![\Delta; \Gamma \vdash V_1 := V_2 : \mathbf{T}\text{unit}]\!]\ \rho\ k\ S \ =$$
$$k\ S[([\![\Delta; \Gamma \vdash V_1 : \sigma \text{ ref}]\!]\ \rho) \ \mapsto \ in_{[\![\sigma]\!]}([\![\Delta; \Gamma \vdash V_2 : \sigma]\!]\ \rho)] *$$

$$[\![\Delta; \Gamma \vdash \text{ref}\, V : \mathbf{T}\sigma \text{ ref}]\!]\ \rho\ k\ S \ = \ k\, S[\ell \mapsto in_{[\![\sigma]\!]}([\![\Delta; \Gamma \vdash V : \sigma]\!]\rho)]\, \ell$$
$$\text{for some/any } \ell \notin \mathit{supp}(\lambda \ell'.k\, S[\ell' \mapsto in_\sigma([\![\Delta; \Gamma \vdash V : \sigma]\!]\rho)]\, \ell').$$

$$[\![\Delta; \Gamma \vdash (\text{rec } f\ x = M) : \tau \to \mathbf{T}\tau']\!]\ \rho \ =$$
$$fix(\lambda f' : [\![\tau \to \mathbf{T}\tau']\!].\lambda x' : [\![\tau]\!].[\![\Delta; \Gamma, f : \tau \to \mathbf{T}\tau', x : \tau \vdash M : \mathbf{T}\tau']\!]\ \rho[f \mapsto f', x \mapsto x'])$$

Fig. 3. Denotational Semantics of MILler (extract)

the cases are shown in Figure 3. The most interesting case is the definition of the semantics of allocation. Note that the monad T combines state with continuations, even though there are no control operators in MILler. However, explicit continuations give us a handle on just what the new location has to be fresh *for*, whilst the ambient use of FM-cpos is exactly what we need to ensure that this really is a good definition, i.e. that one *can* pick a sufficiently fresh ℓ and, moreover, one gets the *same* result in $\mathbb{O}$ for any such choice. An equivalent, perhaps simpler, definition uses the quantification

$$\text{`}\ldots \text{for some/any } \ell \notin \mathit{supp}(k) \cup \mathit{supp}(S) \cup \mathit{supp}([\![\Delta; \Gamma \vdash V : \sigma]\!]\rho)\text{'}$$

Our formulation emphasizes that the notion of support is semantic, not syntactic. The quantification can be seen as ranging not merely over locations that have not previously been allocated (as in the operational semantics), but over all locations that the specific continuation does not distinguish from any of the unallocated ones, i.e. including those which are 'extensionally garbage'.

Writing $[\![\Delta; \vdash K : (x : \tau)^\top]\!]^{\mathcal{K}}$ for $\lambda S d.[\![\Delta; x : \tau \vdash K : \mathbf{T}\tau']\!]\{x \mapsto d\}(\lambda S d.\top)\, S$, the following is proved via a logical 'formal approximation' relation:

Theorem 3 (Soundness and Adequacy). *If* $\Delta; \vdash M : \mathbf{T}\tau$, $\Delta; \vdash K : (x : \tau)^\tau$, $\Sigma : \Delta$ *and* $S \in [\![\Sigma]\!]$ *then*

$$\Sigma, \mathit{let}\, x \Leftarrow M\ \mathit{in}\, K \ \downarrow \iff \ [\![\Delta; \vdash M : \mathbf{T}\tau]\!]\{\}[\![\Delta; \vdash K : (x : \tau)^\top]\!]^{\mathcal{K}}\, S = \top.$$

Corollary 4. $[\![\Delta; \Gamma \vdash G_1 : \gamma]\!] = [\![\Delta; \Gamma \vdash G_2 : \gamma]\!]$ *implies* $\Delta; \Gamma \vdash G_1 =_{ctx} G_2 : \gamma$.

The denotational semantics validates as contextual equivalences the basic equalities of the computational metalanguage and simple properties of assignment and dereferencing. It also proves the 'swap' equivalence:

$$\frac{\Delta;\Gamma \vdash V_1 : \sigma_1 \quad \Delta;\Gamma \vdash V_2 : \sigma_2 \quad \Delta;\Gamma, x : \sigma_1\,\mathrm{ref}, y : \sigma_2\,\mathrm{ref} \vdash N : \mathbf{T}\tau}{\Delta;\Gamma \vdash \begin{array}{c}\mathrm{let}\, x \Leftarrow \mathrm{ref}\, V_1 \,\mathrm{in}\, (\mathrm{let}\, y \Leftarrow \mathrm{ref}\, V_2 \,\mathrm{in}\, N) \\ =_{\mathrm{ctx}} \mathrm{let}\, y \Leftarrow \mathrm{ref}\, V_2 \,\mathrm{in}\, (\mathrm{let}\, x \Leftarrow \mathrm{ref}\, V_1 \,\mathrm{in}\, N) : \mathbf{T}\tau\end{array}}$$

But many interesting valid equivalences, including the garbage collection rule

$$\frac{\Delta;\Gamma \vdash V : \sigma \quad \Delta;\Gamma \vdash N : \mathbf{T}\tau}{\Delta;\Gamma \vdash \mathrm{let}\, x \Leftarrow \mathrm{ref}\, V \,\mathrm{in}\, N \;=_{\mathrm{ctx}}\; N : \mathbf{T}\tau}\, x \notin fvN$$

are, unfortunately, not equalities in the model. The above fails because the model contains undefinable elements that test for properties like $\exists \ell \in \mathbb{L}.S(\ell) = in_{\mathbb{Z}}(3)$ (note that this has empty support) and so make the effect of the initialization visible. The garbage collection and swap equations correspond to two of the structural congruences for restriction in the π-calculus. One might regard them as rather minimal requirements for a useful model, but they also fail in other models in the literature: Levy's possible worlds model [3] fails to validate either and, like ours, a model due to Stark [19, Chapter 5] fails to validate the garbage collection rule.

4 A Parametric Logical Relation

We now embark on refining our model using parameterized logical relations. Many authors have used forms of parametricity to reason about storage; our approach is particularly influenced by the work of Pitts and Stark [12] and of Reddy and Yang [14]. We will define a partially ordered set of *parameters*, p, and a parameter-indexed collection of binary relations on the FM-cpos interpreting states and types: $\forall p.\, \mathcal{R}_{\mathbb{S}}(p) \subseteq \mathbb{S} \times \mathbb{S}$ and $\forall p.\, \forall \gamma.\, \mathcal{R}_{\gamma}(p) \subseteq [\![\gamma]\!] \times [\![\gamma]\!]$. We then show that the denotation of each term is related to itself and, as a corollary, that typed terms with related denotations are contextually equivalent.

An important feature of the state relations we choose will be that they depend on only part of the state: this will allow us to reason that related states are still related if we update them in parts on which the relation does not depend.

One might expect that the notion of support, which is already built into our denotational model, would help here; for example by taking relations to be finitely-supported functions in $\mathbb{S} \times \mathbb{S} \Rightarrow 1 + 1$. Unfortunately, the support is not the right notion for defining separation of relations. For example, the relation

$$\{S_1, S_2 \mid \exists \ell_1, \ell_2 . S_1(\ell_1) = S_2(\ell_2) = in_{\mathbb{Z}} 0 \wedge S_1(\ell) = in_{\mathbb{L}} \ell_1 \wedge S_2(\ell) = in_{\mathbb{L}} \ell_2\}$$

has only ℓ in its support, but writing to the existentially quantified locations 'in the middle' can make related states unrelated. Even with only integers in the store, a relation like $\{(S_1, S_2) \mid \exists \ell, S_1 \ell = 0 = S_2 \ell\}$ can be perturbed by writes outside of its (empty) support.

Separation logic [6] leaves the part of the store which is 'relevant' to a predicate implicit and enforces separation by existential quantification over partial stores in the definition of the (partial) separating conjunction $*$. We instead make the finite part of the state on which our relations depend explicit, using what we call *accessibility maps*. Because, as above, relations can 'follow pointers', the set of locations on which a relation depends can itself be a function of the states. We make no explicit use of support in this section, though working with FM-cpos allows the use of equality of locations, rather than partial bijections, in our definitions.

Definition 5 (Accessibility map). An accessibility map A is a function from $\mathbb{S}$ to finite subsets of $\mathbb{L}$, such that:

$$\forall S, S' \in \mathbb{S}, (\forall \ell \in AS, S\ell = S'\ell) \Longrightarrow A(S) = A(S')$$

The subtyping ordering $<:$ is defined as:

$$A <: A' \iff \forall S, A(S) \supseteq A'(S)$$

The subtype relation is a partial order, and the function $\lambda S.\emptyset$, abbreviated $\emptyset$, is the greatest accessibility map with respect to $<:$. One source of concrete accessibility maps is our existing notion of state type:

Definition 6 (Accessible part of a state at a type). If Δ is a state type, then $\mathrm{Acc}_\Delta : \mathbb{S} \to \mathbb{P}_{fin}(\mathbb{L})$ is defined by $\mathrm{Acc}_\Delta(S) = \bigcup_{(\ell:\sigma)\in\Delta} \mathrm{Acc}(\ell, \sigma, S)$ where $\mathrm{Acc}(\ell, \mathrm{int}, S) \stackrel{def}{=} \{\ell\}$ and

$$\mathrm{Acc}(\ell, \sigma\ \mathrm{ref}, S) \stackrel{def}{=} \{\ell\} \cup \begin{cases} \mathrm{Acc}(\ell', \sigma, S) & \text{if } S\,\ell = \mathrm{in}_{\mathbb{L}}\ell' \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 7. *Acc_Δ is an accessibility map, and if $\Delta \subseteq \Delta'$ then $Acc_{\Delta'} <: Acc_\Delta$.*

Definition 8 (Accessible equality of states). If A is an accessibility map, we define $S \sim S' : A$ to mean $\forall \ell \in A(S), S\ell = S'\ell$.

Definition 9 (Finitary State Relation). A finitary state relation r is a pair $\langle |r|, A_r \rangle$ where $|r| \subseteq \mathbb{S} \times \mathbb{S}$ and A_r is an accessibility map, subject to the following saturation condition: if $S_1 \sim S_1' : A_r$ and $S_2 \sim S_2' : A_r$ then $(S_1, S_2) \in |r| \iff (S_1', S_2') \in |r|$.

Lemma 10.

1. *If A and A' are accessibility maps then so is $A \wedge A'$, where $\forall S.\, (A \wedge A')(S) = (A(S)) \cup (A'(S))$.*
2. *$\top \stackrel{def}{=} \langle \mathbb{S} \times \mathbb{S}, \emptyset \rangle$ is a finitary state relation.*
3. *If $\langle |r|, A \rangle$ is a finitary state relation and $A' <: A$ then $\langle |r|, A' \rangle$ is a finitary state relation.*
4. *$id_\Delta \stackrel{def}{=} \langle \sim_\Delta, Acc_\Delta \rangle$ is a finitary state relation.*

Definition 11 (Separating conjunction). Given two finitary state relations, $r_1 = \langle |r^1|, A^1 \rangle$ and $r_2 = \langle |r^2|, A^2 \rangle$, define $r^1 \otimes r^2 \stackrel{def}{=} \langle |r^1 \otimes r^2|, A^1 \wedge A^2 \rangle$ where

$$(S_1, S_2) \in |r^1 \otimes r^2| \iff \begin{cases} (S_1, S_2) \in |r^1| \cap |r^2| \\ \forall i \in \{1, 2\}, A^1(S_i) \cap A^2(S_i) = \emptyset \end{cases}$$

Lemma 12. *If r^1 and r^2 are finitary state relations, so is $r^1 \otimes r^2$. The conjunction is associative and commutative, with $\top$ as a unit.*

We now have all the ingredients needed to define the parameters of our relations. The intuition is that the parameters express that one part of the store is directly accessible, or *visible*, and that functions in the context also give access to other locations. Since we can do anything with visible locations, related states must be equal on that part. Moreover, we will preserve any invariant on hidden locations that is preserved by all the functions we can use, provided that invariant does not also depend on the contents of visible locations. Our parameters comprise these two components: the set of visible locations and a hidden invariant.[3]

Definition 13 (Parameters). A parameter is a pair (Δ, r), where Δ is a state type and r is a finitary relation; we will abbreviate this to Δr. If Δr is a parameter, we define the binary relation on states $\mathcal{R}_{\mathbb{S}}(\Delta r) \stackrel{def}{=} |id_\Delta \otimes r|$ and define the partial order $\rhd$ on parameters by

$$\Delta r \rhd \Delta' r' \iff (\Delta \supseteq \Delta') \wedge (\exists r'', r = r' \otimes r'')$$

Definition 14 (Logical Relation). We define the parameter- and typed- indexed family of relations $\mathcal{R}_\gamma(\Delta r)$ by induction over the types:

$$\begin{aligned}
\mathcal{R}_{\mathrm{unit}}(\Delta r) &= \{(*, *)\} \\
\mathcal{R}_{\mathrm{int}}(\Delta r) &= \{(n, n) \mid n \in N\} \\
\mathcal{R}_{\tau \times \tau'}(\Delta r) &= \{((d_1, d_1'), (d_2, d_2')) \mid (d_1, d_2) \in \mathcal{R}_\tau(\Delta r) \wedge (d_1', d_2') \in \mathcal{R}_{\tau'}(\Delta r)\} \\
\mathcal{R}_{\tau_1 + \tau_2}(\Delta r) &= \{(\mathrm{in}_1 d_1, \mathrm{in}_1 d_2) \mid (d_1, d_2) \in \mathcal{R}_{\tau_1}(\Delta r)\} \\
&\quad \cup \{(\mathrm{in}_2 d_1, \mathrm{in}_2 d_2) \mid (d_1, d_2) \in \mathcal{R}_{\tau_2}(\Delta r)\} \\
\mathcal{R}_{\sigma\ \mathrm{ref}}(\Delta r) &= \{(\ell, \ell) \mid (\ell : \sigma) \in \Delta\} \\
\mathcal{R}_{\tau \to \mathbf{T}\tau'}(\Delta r) &= \\
&\{(f_1, f_2) \mid \forall \Delta' r' \rhd \Delta r, (v_1, v_2) \in \mathcal{R}_\tau(\Delta' r'), (f_1 v_1, f_2, v_2) \in \mathcal{R}_{\mathbf{T}\tau'}(\Delta' r')\}
\end{aligned}$$

For continuations, we define $\mathcal{R}_{\tau^\top}(\Delta r)$ to be

$$\{(k_1, k_2) \mid \forall \Delta' r' \rhd \Delta r, (v_1, v_2) \in \mathcal{R}_\tau(\Delta' r'), (S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r'), k_1 S_1 v_1 = k_2 S_2 v_2\}$$

and for computations, $\mathcal{R}_{\mathbf{T}\tau}(\Delta r)$ is defined as $\{(f_1, f_2) \mid \forall \Delta' r' \rhd \Delta r, (k_1, k_2) \in \mathcal{R}_{\tau^\top}(\Delta' r'), (S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r'), f_1 k_1 S_1 = f_2 k_2 S_2\}$.

[3] This is in the style of Reddy and Yang [14]. Pitts and Stark [12] got away with simpler parameters, adding a new visible location ℓ to a parameter r by $r \otimes id_{\{\ell\}}$. In the presence of references to references, this doesn't work, as it prevents one visible location pointing to another.

Definition 15 (Relations in context). We define $\mathcal{R}_\Gamma(\Delta r)$ to be $\{(\rho_1, \rho_2) \mid \forall(x_i : \tau_i) \in \Gamma, (\rho_1 x_i, \rho_2 x_i) \in \mathcal{R}_{\tau_i}(\Delta r)\}$ and $\mathcal{R}_{\Gamma\vdash\gamma}(\Delta r)$ to be $\{(v_1, v_2) \mid \forall \Delta' r' \rhd \Delta r, \forall(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta' r'), (v_1\rho_1, v_2\rho_2) \in \mathcal{R}_\gamma(\Delta' r')\}$.

We now have to prove a number of non-trivial technical lemmas, which we omit here. These allow us to show that the interpretations of all the MILler typing rules preserve the logical relation, and hence deduce:

Theorem 16 (Fundamental lemma). *If* $\Delta; \Gamma \vdash G : \gamma$, *then*

$$\forall r. (\llbracket \Delta; \Gamma \vdash G : \gamma \rrbracket, \llbracket \Delta; \Gamma \vdash G : \gamma \rrbracket) \in \mathcal{R}_{\Gamma\vdash\gamma}(\Delta r).$$

Theorem 17 (Soundness of relational reasoning). *If* $\Delta; \Gamma \vdash G_i : \gamma$ *for* $i = 1, 2$ *and*

$$(\llbracket \Delta; \Gamma \vdash G_1 : \gamma \rrbracket, \llbracket \Delta; \Gamma \vdash G_2 : \gamma \rrbracket) \in \mathcal{R}_{\Gamma\vdash\mathbf{T}\tau}(\Delta\top)$$

then $\Delta; \Gamma \vdash G_1 =_{ctx} G_2 : \gamma$.

Accessibility maps are convenient to prove generic results, but working with specific ones can be a little awkward. In most examples we do not need their full generality (for example, allowing the accessible locations to depend on the integer contents of a particular location). We find it useful to generalize the notion of state type a little, introducing a top type and a simple form of subtyping. This allows a corresponding generalization of the accessibility map associated with a state type that suffices to specify the accessiblity maps we need even in tricky cases in which there are pointers between the visible and hidden parts of the state, but the invariant does not follow them far enough to be affected.

Definition 18 (Extended storable types). Extended storable types are given by the grammar $\alpha ::= \mathbb{T} \mid \text{int} \mid \alpha \text{ ref}$. We define an order $<:$ between extended storable types by: $\text{int} <: \mathbb{T}$, $\alpha \text{ ref} <: \mathbb{T}$, and if $\alpha <: \alpha'$ then $\alpha \text{ ref} <: \alpha' \text{ ref}$.

Definition 19 (Extended state type). An extended state type θ is a map from $\mathbb{L}$ to location types which is $\mathbb{T}$ for all but a finite number of locations. Subtyping is defined pointwise.

The types α say how much of the value stored in a location is relevant: $\mathbb{T}$ means that we do not care about the value of the location. For instance, a location has type $\mathbb{T}$ ref if the value it carries is always a location, but we do not specify the type of this location: it might be an integer or a location.

Definition 20 (Accessibility map for extended state type). The map Acc_θ is defined as in Definition 6, with the extra clause $\text{Acc}(\ell, \mathbb{T}, S) \stackrel{def}{=} \emptyset$.

Lemma 21. *Acc_θ is an accessibility map. If $\theta <: \theta'$, $Acc_{\theta'} <: Acc_\theta$.*

5 Examples

Garbage Collection If x is not free in M, and $\Delta; \Gamma \vdash M : \mathbf{T}\tau$, then

$$\Gamma \vdash \text{let } x \Leftarrow \text{ref } V \text{ in } M =_{\text{ctx}} M : \mathbf{T}\tau$$

We prove that $[\![\text{let } x \Leftarrow \text{ref } V \text{ in } M]\!]$ and $[\![M]\!]$ are related by $\mathcal{R}_{\Gamma \vdash \mathbf{T}\tau}(\Delta\top)$, and we conclude using Theorem 17. Let $\Delta' r' \rhd \Delta\top$ be a parameter and $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta' r')$. We need to prove that $([\![\text{let } x \Leftarrow \text{ref } V \text{ in } M]\!]\rho_1, [\![M]\!]\rho_2) \in \mathcal{R}_{\mathbf{T}\tau}(\Delta' r')$. Let $\Delta'' r'' \rhd \Delta' r'$, $(k_1, k_2) \in \mathcal{R}_{\tau^\top}(\Delta'' r'')$ and $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$. We have to prove that

$$[\![\text{let } x \Leftarrow \text{ref } V \text{ in } M]\!]\rho_1 k_1 S_1 = [\![M]\!]\rho_2 k_2 S_2$$

For $\ell \notin \mathit{supp}(\lambda \ell'. k_1 S_1[\ell' \to [\![V]\!]\rho]\ell')$

$$[\![\text{let } x \Leftarrow \text{ref } V \text{ in } M]\!]\rho_1 k_1 S_1 = [\![M]\!]\rho_1 k_1 S_1[\ell \to [\![V]\!]\rho_1]$$

because x is not free in M. Since we can pick *any* such ℓ, we actually choose one also out of $\text{Acc}_{\Delta''}(S_i) \cup A_{r''}(S_i)$ for $i = 1, 2$. By the fundamental lemma, $[\![M]\!]$ is related to itself by $\mathcal{R}_{\Gamma \vdash \mathbf{T}\tau}(\Delta\top)$, so if we prove that $(S_1[\ell \to [\![V]\!]\rho_1], S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$ we are done.

First, since $\ell \notin \text{Acc}_{\Delta''}(S_i)$, $(S_1[\ell \to [\![V]\!]\rho_1], S_2) \in \mathit{id}_{\Delta''}$, and since $\ell \notin A_{r''}(S_i)$, $(S_1[\ell \to [\![V]\!]\rho_1], S_2) \in r''$. By definition of accessibility maps, $\text{Acc}_{\Delta''}$ and $A_{r''}$ are unchanged, so they still do not overlap, which concludes the proof.

Meyer-Sieber 6 We can validate all the Meyer-Sieber examples [4]. We explain here example 6, which can be translated in MILler as the program M:

$$\begin{array}{l}
\text{let } x \Leftarrow \text{ref } \underline{0} \text{ in} \\
\quad \text{let } \mathit{almost_add}2 \Leftarrow \lambda z.\text{if } z = x \\
\qquad\qquad\qquad\qquad \text{then } x := 1 \\
\qquad\qquad\qquad\qquad \text{else let } y \Leftarrow !x \text{ in let } y' \Leftarrow y + 2 \text{ in } x := y' \text{in} \\
\qquad p(\mathit{almost_add}2); \\
\qquad \text{let } y \Leftarrow !x \text{ in} \\
\qquad\quad \text{if } !x \bmod 2 = 0 \text{ then diverge}_{\text{unit}} \text{ else val } ()
\end{array}$$

This program always diverges: we prove that $([\![; \Gamma \vdash M : \mathbf{T}\text{unit}]\!], \lambda\rho k s.\bot) \in \mathcal{R}_{\Gamma \vdash \mathbf{T}\text{unit}}(\emptyset\top)$. We have, for some fresh ℓ:

$$[\![M]\!]\rho k S = \rho(p) f \left(\lambda S' v. \begin{cases} k S'* & \text{if } S\ell = \text{in}_{\mathbb{Z}} n \text{ for some odd } n \\ \bot & \text{otherwise} \end{cases} \right) S[\ell \to 0]$$

where

$$f z k s = \begin{cases} k s[\ell \to 1]* & \text{if } z = \ell \\ k s[\ell \to n+2]* & \text{if } z \neq \ell \text{ and } s\ell = \text{in}_{\mathbb{Z}} n \\ \bot & \text{otherwise} \end{cases}$$

Let $\Delta' r' \rhd \Delta r$ be two parameters, $(\rho_1, \rho_2) \in \mathcal{R}_\Gamma(\Delta r)$, $(k_1, k_2) \in \mathcal{R}_{\text{unit}^\top}(\Delta' r')$ and $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$. We let $r'' = r' \otimes \langle \{(S_1, S_2) \mid S_1\ell \text{ and } S_2\ell \text{ hold even integers}\}, \text{Acc}_{\{\ell:\text{int}\}} \rangle$ and prove that

$$(f, f) \in \mathcal{R}_{\text{int ref} \to \mathbf{T}\text{unit}}(\Delta' r'')$$

the proof is then straightforward.

Suppose $\Delta^4 r^4 \triangleright \Delta^3 r^3 \triangleright \Delta' r''$, $(v_1^3, v_2^3) \in \mathcal{R}_{\text{int ref}}(\Delta^3 r^3)$, $(k_1^4, k_2^4) \in \mathcal{R}_{\text{unit}}(\Delta^4 r^4)$, $(S_1^4, S_2^4) \in \mathcal{R}_{\mathbb{S}}(\Delta^4 r^4)$. As $\Delta^4 r^4 \triangleright \Delta' r''$, $(S_1, S_2) \in r^4 \subseteq r''$, so S_1 and S_2 hold even integers n_1 and n_2. $v_1^3 = v_2^3$ is a visible location. As $A_{r^4} <: A_{r''}$, $\ell \in A_{r^4}(S_i)$, which entails that $v_i^3 \neq \ell$. We get

$$f v_i^3 k_i^4 S_i^4 = k_i^4 S_i^4[\ell \to n_i + 2]$$

$(S_1[\ell \to n_i + 2], S_2[\ell \to n_i + 2]) \in \mathcal{R}_{\mathbb{S}}(\Delta^4 r^4)$, because our invariant is preserved, and, by non interference, the other parts of the invariant are preserved too, so

$$f v_1 k_1 S_1 = k_1 S_1[\ell \to n_1 + 2] = k_2 S_2[\ell \to n_2 + 2] = f v_2 k_2 S_1$$

Extended types To illustrate the extended types, we give an example involving a pointer from the invariant to the visible locations. We show that the following program M diverges:

$$\begin{array}{l}
\text{let } x \Leftarrow \text{ref } 0 \text{ in} \\
\text{let } y \Leftarrow \text{ref } x \text{ in} \\
p\ x; \\
\text{let } z \Leftarrow !y \text{ in} \\
\text{if } z = x \text{ then diverge}_{\text{unit}} \text{ else val } ()
\end{array}$$

As before, we prove that $(\llbracket M \rrbracket, \bot) \in \mathcal{R}_{\Gamma \vdash \mathbf{T}\text{unit}}(\emptyset \top)$. For any fresh ℓ_x, ℓ_y:

$$\llbracket M \rrbracket \rho k S = \rho(p)\ell \left(\lambda S'' v. \begin{cases} \bot & \text{if } S''\ell_y = \text{in}_{\mathbb{L}} \ell_x \\ k_1 S'' * & \text{otherwise} \end{cases} \right) S_1'$$

where $S_1' = S_1[\ell_y \to \ell_x, \ell_x \to 0]$. We want to prove that when we reach the continuation, the value held in ℓ_y is ℓ_x. ℓ_x is given to p, so it must be visible, but the invariant "ℓ_y holds ℓ_x" is about the location held by ℓ_y, but not about what this location holds. We have to define an accessibility map that does not contain ℓ_x: the invariant is

$$dr = \langle \{(S_1, S_2) \mid S_1 \ell_y = \text{in}_{\mathbb{L}} \ell_x\}, \text{Acc}_{\{\ell_y : \mathbb{T}\ \text{ref}\}} \rangle$$

Secrecy We can prove some examples (very) loosely inspired by the work of Sumii and Pierce [20] on logical relations and encryption. The idea, though we certainly do not claim this is a particularly good model of cryptography, is that encrypted messages are sent through hidden locations. Encrytion is encoded as writing in a hidden location, and decryption as reading the same hidden location. The visible locations are the keys the context knows, and the locations in the invariant the private keys. We can represent public keys by providing a function that wraps the assignment.

For instance, in the following dummy protocol, A (the first function) sends a message to B (the second function) containing a fresh key, and B reads this key

and sends a message i using this key. The program M_i is

$$\begin{array}{l}\text{let } x \Leftarrow \text{ref } 0 \text{ in}\\ \text{let } k_b \Leftarrow \text{ref } x \text{ in}\\ \text{let cipher} \Leftarrow \text{val } \lambda\langle k, n\rangle.k := n \text{ in}\\ \text{let decipher} \Leftarrow \text{val } \lambda k.!k \text{ in}\\ \text{val } \langle \lambda().\text{let } k_a \Leftarrow \text{ref } 0 \text{ in cipher } \langle k_b, k_a\rangle,\\ \qquad \lambda().\text{let } k \Leftarrow \text{decipher } k_b \text{ in cipher } \langle k, i\rangle\rangle\end{array}$$

We prove that M_1 is equivalent to M_2, which shows that the information written in k is not accessible from the context, thus kept secret.[4] An easy calculation gives, for fresh ℓ_x and ℓ_b:

$$[\![M_i]\!]kS = kS[\ell_b \to \ell_x \to 0]\langle \phi^A, \phi_i^B\rangle$$

where

$$\begin{array}{l}\phi^A = \lambda * kS.kS[\ell_b \to \ell_a \to 0] * \quad \text{for a fresh } \ell_a\\ \phi_i^B = \lambda * kS. \begin{cases} kS[\ell \to i] * & \text{if } S\ell_b = \text{in}_{\mathbb{L}}\ell \\ \bot & \text{otherwise}\end{cases}\end{array}$$

Let Δr be a parameter $(k_1, k_2) \in \mathcal{R}_{(\tau\times\tau)^\top}(\Delta r)$ and $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta r)$, where $\tau = \text{unit} \to \mathbf{T}\text{unit}$ is the type of the processes A and B. The invariant dr says that ℓ_b holds a reference to an integer, which makes ℓ_b and, later, ℓ_a secret. Let $r' = r \otimes dr$, and $S_i' = S_i[\ell_b \to \ell_x \to 0]$. Of course $(S_1', S_2') \in \mathcal{R}_{\mathbb{S}}(\Delta r')$. If we show that (ϕ^A, ϕ^A) and (ϕ_1^B, ϕ_2^B) are in $\mathcal{R}_\tau(\Delta r')$, then we are done.

Let $\Delta'' r'' \rhd \Delta r'$, $(k_1'', k_2'') \in \mathcal{R}_{\text{unit}^\top}(\Delta'' r'')$ and $(S_1'', S_2'') \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$. We write $r'' = r \otimes dr \otimes dr'$. We can pick a fresh ℓ_a such that, for both $i = 1, 2$:

$$\phi^A * k_i'' S_i'' = k_i'' S_i''[\ell_b \to \ell_a \to 0] *$$

It is easy to check that the new states are still related by $\mathcal{R}_{\mathbb{S}}(\Delta'' r'')$: they are in dr, they also are in the other parts of the relation $id_{\Delta''}$, r and dr' thanks to the separating condition, because the only locations that were changed are ℓ_b (which is in $A_{dr}S_i$) and ℓ_a (which was fresh), and there is no new cross pointer between these parts. This gives $(\phi^A, \phi^A) \in \mathcal{R}_\tau(\Delta r')$.

The same holds for ϕ_1^B and ϕ_2^B because the relation dr only states that ℓ_x holds an integer, not which integer. We get the expected contextual equivalence of M_1 and M_2.

On the other hand, if the key is public, *i.e.* we give an encryption function to the context, then the secrecy is broken: the opponent can write a message to B, giving it a channel it can read, so that the value i written by B can be deciphered afterwards. We give to the context (as a third element in the tuple) the function

$$\text{public\_cipher} = \lambda n.\text{cipher } \langle k_b, n\rangle$$

[4] The equivalence of two pairs of functions of type unit $\to$ **T**unit with no free locations is not *completely* trivial: each pair might have differing intertwined termination behaviour, depending on hidden state.

public_cipher does not respect the relation $\mathcal{R}_{\mathbb{S}}(\Delta r')$:

$$\text{public\_cipher}\ \ell k S = k S[\ell_b \to \ell]*$$

ℓ_b certainly points to a reference to an integer after it is called, so it is in dr as before, but now this reference is ℓ, which is visible (since it is given by the context), so the separating condition between id_Δ and dr no longer holds.

Snapback. Our logical relation fails to capture the irreversibility of state changes, which is a source of incompleteness relative to contextual equivalence. Consider the element 'snapback' of $[\![(\text{unit} \to \mathbf{T}\text{int}) \to \mathbf{T}\text{unit}]\!]$ defined by

$$\text{snapback}\ f\ k\ S\ =\ f\ *\ (\lambda S'.\lambda n.k\ S\ n)\ S$$

Snapback calls its argument f, passing it the state S and a continuation that will set the state *back* to S, discarding any updates made by f. Snapback is not definable, but it *is* parametric: assume $\Delta' r' \rhd \Delta r$ are parameters, $(f_1, f_2) \in \mathcal{R}_{\text{unit}\to\mathbf{T}\text{int}}(\Delta r)$, $(k_1, k_2) \in \mathcal{R}_{\text{int}^\top}(\Delta' r')$ and $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$. We only have to show that the continuations $\lambda S' n.k_i S_i n$ are related by $\mathcal{R}_{\text{int}^\top}(\Delta' r')$. Let $\Delta'' r'' \rhd \Delta' r'$ and $(S_1'', S_2'') \in \mathcal{R}_{\mathbb{S}}(\Delta'' r'')$. The only values related by $\mathcal{R}_{\text{int}}(\Delta'' r'')$ are (m, m) for some $m \in \mathbb{N}$. We have $(\lambda S' n.k_i S_i n) S_i'' m = k_i S_i m$, and, since $(k_1, k_2) \in \mathcal{R}_{\text{int}^\top}(\Delta' r')$ and $(S_1, S_2) \in \mathcal{R}_{\mathbb{S}}(\Delta' r')$, $k_1 S_1 m = k_2 S_2 m$, so we're done.

The fact that our relation does not eliminate snapback prevents us from proving some useful and valid contextual equivalences. For example, $; p : (\text{unit} \to \mathbf{T}\text{int}) \to \mathbf{T}\text{unit} \vdash M =_{\text{ctx}} N : \mathbf{T}\text{unit}$ where

$$\begin{array}{l} M = \text{let}\ x \Leftarrow \text{ref}\ 0\ \text{in} \\ \qquad p(\lambda\_.\, x := 1;\ 0); \\ \qquad \text{let}\ y \Leftarrow !x\ \text{in} \\ \qquad \text{if iszero}\ y\ \text{then val}\ ()\ \text{else diverge}_{\text{unit}} \end{array} \qquad N = p\ (\lambda\_.\, \text{diverge}_{\text{int}})$$

but this is not provable with our relation, as if p is snapback, then M converges and N diverges. Intuitively, the problem here is one of linearity: note that snapback duplicates S and discards S'.

6 Discussion and Further Work

There is much related work on the semantics of Algol-like languages; we can only pick out a few relevant highlights. Meyer and Sieber [4] gave a model for an Algol-like language with local integer variables which was based on a notion of 'support' and a refined model based on the preservation of (unary) predicates on stores that depend only on a finite number of locations. Reynolds [15] and Oles [9] pioneered the functor-category, or *possible-worlds* approach to modelling storage, further developed by O'Hearn and Tennent [7] and Sieber [18] to incorporate relational parametricity. This gives a good semantics for the locality of local variables but, like our relation, still does not account for the irreversibility of state changes: essentially the same snapback example as we have given

above causes incompleteness. Reynolds and O'Hearn [5] gave models via translations from Algol-like languages into a predicatively polymorphic *linear* lambda calculus, ruling out snapback. Pitts [10] used that idea to give an operationally-based logical relation for an Algol-like language that is complete for contextual equivalence. Following [5, 10], we believe a relatively small modification to the logical relation (involving lifting of states) may rule out snapback in our model too (strict relations over pointed cpos give a rather weak model of the polymorphic linear lambda calculus, but as snapback relies on both contraction and weakening, this seems to suffice), but we have not yet worked through all the details.

General dynamic allocation is more complex than the stack-structured local variables of Algol. Pitts and Stark [11, 19] introduced the *ν-calculus*, a simply-typed (and recursion-free) CBV lambda calculus with dynamic generation of pure names. They present both operational logical relations and monadically-structured denotational models, based on parametric functor categories, for the ν-calculus. We have already mentioned their later work [12] and that of Reddy and Yang [14], from which we drew much inspiration. Levy [3] has given an adequate, but non-parametric, possible-worlds semantics for an ML-like language with storage of arbitrary values, including functions. Our FM-cpo semantics for MILler derives from a model for FreshML due to Shinwell and Pitts [17, 16].

Although most of the pieces of our model and logical relation are closely related to ones in the literature, the way they are combined here is novel (possibly the first domain-theoretic, parametric treatment of dynamic allocation), elegant and, above all, elementary. Our basic model is considerably easier to work with than a functor category, and the relational reasoning principle is easy to apply. We believe our formulation of separation is more powerful than that of [14].

We would like to be able to reason relationally about references to values of functional (and recursive) types. Appropriate technology seems to exist [13], but it remains to be seen whether we can apply it successfully in our setting.

We have experimented with a simple inference system for proving expressions are related. Developing this, perhaps using ideas from nominal logic, may open the way to automated support. We have also looked at methods to prove the soundness of the effect-based transformations we previously studied operationally (and rather unsatisfactorily) in [1]. Something can certainly be pushed through using our current definitions, but making it work smoothly seems to call for some interesting modifications to our logical relation. We also plan to investigate more seriously the applications to secure information flow and the correctness of cryptographic protocols.

References

1. N. Benton and A. Kennedy. Monads, effects and transformations. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), Paris*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

2. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. Technical report, Microsoft Research, February 2005.
3. P. B. Levy. Possible world semantics for general storage in call-by-value. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2471 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2002.
4. A. R. Meyer and K. Sieber. Towards a fully abstract semantics for local variables: Preliminary report. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1988.
5. P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, January 2000.
6. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, 2001.
7. P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.
8. P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages*. Progress in Theoretical Computer Science. Birkhäuser, 1997. Two volumes.
9. F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.
10. A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In [8], 1997.
11. A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.
12. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.
13. A.M. Pitts. Relational properties of domains. *Information and Computation*, 127(2), 1996.
14. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.
15. J. C. Reynolds. The essence of Algol. In *Proceedings of the International Symposium on Algorithmic Languages*, 1981. Reprinted in [8].
16. M. R. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, Computer Laboratory, University of Cambridge, December 2004.
17. M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 2005. To appear.
18. K. Sieber. New steps towards full abstraction for local variables. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, 1993.
19. I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, Computer Laboratory, University of Cambridge, December 1994. Available as Technical Report 363.
20. E. Sumii and B. C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4), 2003.
21. R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.