Correctness of Data Representations involving Heap Data Structures

Uday S. Reddy[1] and Hongseok Yang[2]

[1] School of Computer Science, University of Birmingham
[2] ROPAS, Korean Advanced Inst. of Science and Technology

Abstract. While the semantics of local variables in programming languages is by now well-understood, the semantics of pointer-addressed heap variables is still an outstanding issue. In particular, the commonly assumed relational reasoning principles for data representations have not been validated in a semantic model of heap variables. In this paper, we define a parametricity semantics for a Pascal-like language with pointers and heap variables which gives such reasoning principles. It is found that the correspondences between data representations are not simply relations between states, but more intricate correspondences that also need to keep track of visible locations whose pointers can be stored and leaked.

1 Introduction

Programming languages with dynamically allocated storage variables ("heap variables") date back to Algol W [27] and include the majority of languages in use today: imperative languages like C, Pascal and Ada, object-oriented languages ranging from Simula 67 to Java, and functional languages like Scheme, Standard ML, and variants of Haskell [6]. However, the semantic structure of these languages is not yet clear. In particular, the oft-used principles for data representation reasoning, involving invariants or simulation relations, have not been validated. While remarkable progress has been made in understanding local variables (cf. the collection [15]), none of this theory is directly applicable to heap variables because the shape of the heap storage dynamically varies.

A number of attacks have been made on the problem: Stark's thesis [25, 24], which deals with dynamic allocation but not pointers, and Ghica's and Levy's theses [4, 5, 7, 8], which address the general semantic structure but not data representation reasoning. The recent paper of Banerjee and Naumann [2] is the first to address data representation correctness with heap variables and pointers. While their work is remarkably successful in dealing with a Java-like language with dynamically allocated objects, their treatment falls short of explicating the semantic structure of the language relying instead on a strong notion of "confinement" to simplify the problem.

In this paper, we define a parametricity semantics for a Pascal-like language with dynamically allocated variables, pointers, and call-by-value procedures. The validity of simulation-based reasoning principles follows from the structure of the semantics (similar to Tennent's treatment in [26] for local variables). The type structure of the semantics makes explicit where information hiding is going on, while the formal parametricity conditions back up one's intuitions and allow one to produce formal proofs. We do not use any confinement conditions in our definitions. Where there is information leakage, our semantics explicates the breakdown of the data encapsulation, so that faulty conclusions are avoided.

Our treatment bears a close relationship with the ongoing work on separation logic for local reasoning about heap storage [22, 11, 28]. In particular, our relations are "local" in the same sense as the assertions of separation logic. We use the ideas of partial heaps and heap-splitting developed there to formulate the relations. We envisage that in future work, these connections with local reasoning will be further strengthened.

2 Motivation

Local variables get hidden in program contexts due to scope restrictions in the programming language. This gives rise to information hiding which is exploited in devising data representations. Since dynamically allocated heap variables can only be accessed through entry points given by local variables, the same scope restrictions also give rise to information hiding for heap data structures. In this section, we give an informal introduction to these information hiding aspects through a series of examples.

Example 1. Consider the following program block adapted from Meyer and Sieber [9]:

{ **local var** `int x; x := 0; p();` **if** `x = 0` **then** `diverge` }

Here, `p` is an arbitrary non-local procedure with no arguments, and `diverge` is a diverging command. The program block should be observationally equivalent to `diverge` for the following reason: The local variable `x` is not visible to the non-local procedure `p`. Hence, if `x` is 0 before the procedure call, it should be 0 after the procedure call too.

Next consider a similar program using pointer-addressed variables:[3]

{ `x :=` **new** `int; x↑ := 0; p();` **if** `x↑ = 0` **then** `diverge` }

Here, `x` is a non-local variable (of type `↑int`) that can store pointers to integer variables in the heap. The command `x :=` **new** `int` allocates a new integer variable on the heap and sets `x` to point to this variable. Unlike in the local variable case, we cannot expect this program block to be equivalent to `diverge`. The reason is that the heap variable is accessible to `p` via the non-local variable `x` and `p` has the ability to modify it. There is no information hiding for the heap variable.

On the other hand, the following variant does implement information hiding:

[3] The notation for pointers is borrowed from Pascal. For any data type δ, $\uparrow\delta$ is the type of pointers to δ-typed variables. If p is a pointer, $p\uparrow$ denotes the variable that p points to. (In the syntax of C, $\uparrow\delta$ would be written as $\delta*$ and $p\uparrow$ as $*p$.)

```
{ local var (↑int) x; x := new int; x↑ := 0; p(); if x↑ = 0 then diverge }
```

Here, the pointer variable `x` is local. Since it is the only access point to the heap variable, the procedure `p` has no access to the heap variable. If `x↑` is 0 before the procedure call, it should remain 0 after the procedure call. Hence, this block is equivalent to `diverge`. □

We give an indication of how this form of selective information hiding can be modelled in the semantics. Using a possible world form of semantics as in [21, 16, 13], we take worlds to be sets of typed locations (or equivalently record types) of the form $W = \{l_1 : \delta_1, \ldots, l_k : \delta_k\}$. We write $X <: W$ to mean that X is an extension of W with additional locations (or a "subtype" of W). All program terms are given meanings with reference to a possible world W denoting the set of locations available in a particular (dynamic) context of execution. Now, in a world W, the procedure `p` denotes a parametrically polymorphic function of type:

$$[\![\mathtt{p}]\!] : \forall_{X<:W} \mathrm{St}(X) \to \mathrm{St}(X)$$

where $\mathrm{St}(X)$ means the set of states for the location world X. Here, X refers to the set of locations available when `p` may be called, which will include all the locations of W plus any additional locations allocated before the call. However, since `p` has been defined before these new locations are allocated, it should have no direct access to these new locations. The parametric interpretation of $\forall_{X<:W}$ captures information hiding for *all parts of X that are not accessible from W*. This is defined via relation-preservation for appropriate kinds of relations. The definition of these relations is the main technical contribution of this paper.

Corresponding to the subtyping $X <: W$, there is a relation-subtyping $S <: R$ that says that a relation S between potential instantiations of X is an extension of a relation R between potential instantiations of W. Intuitively the relation-subtyping $S <: R$ says that the S relation expects the contents of all W-accessible locations to be related by R and imposes new constraints for the other new locations that are inaccessible from W. The parametric interpretation of $\forall_{X<:W}$ implies that $[\![\mathtt{p}]\!]$ must preserve all relations S that extend the identity relation I_W, i.e., preserve all additional conditions that can be stated for W-inaccessible locations. Using this intuition, we can explain how the three program blocks in Example 1 are treated. In each case, we choose W to be the set of all locations allocated before the entry of the program block:

- In the first program block with a local variable `x`, the extended relation S can impose the condition that the new location for `x` contains a specific value such as 0. Since the binding of `p` preserves all such relations, it follows that `p` cannot affect `x`.
- In the second program block, where the heap location is accessible via a *non-local* pointer variable `x`, recall that the extended relation S can impose additional conditions only for *W-inaccessible* locations. Since the new location is accessible from W before the procedure call to `p`, there is no requirement that `p` should preserve its value.

– In the third program block where the heap location is accessible via a *local* pointer variable `x`, both `x` and the heap location are inaccessible from W. Hence the extended relation S can impose the additional condition `x↑ = 0` and `p` must preserve it.

The second example, due to Peter O'Hearn, illustrates information leakage:

Example 2. Consider the program block that calls a non-local procedure `h` of type $\uparrow\text{int} \to \textbf{com}$:

```
{ local var (↑int) x; x := new int;
  h(x); x↑ := 0; p(); if x↑ = 0 then diverge }
```

As in the previous example, `x` and `x↑` are not directly visible to the non-local procedures. However, `h` is given as argument the pointer value of `x`. It has the ability to dereference `x` and modify `x↑`. It can also store the pointer `x` in a non-local variable. In other words, the access to the local data structure `x↑` has been *leaked* and encapsulation is lost. It is not guaranteed that the later call to `p` will not affect `x↑` because `p` can receive access to `x↑` from `h` via a shared variable. This block is not equivalent to `diverge` in general.

If, however, `h` were to be passed `x↑` as an argument, instead of the pointer value `x`, it would not have the ability to store `x` and information leakage would be avoided.[4] □

To model information leakage, we split the relations mentioned previously into two parts: one part that relates *visible* heap locations, given by a partial bijection between the location sets $\rho : W \leftrightarrow W'$, and a second part that relates the contents of *hidden* locations, given by a relation R between partial states. A pair consisting of the two parts $(\rho, R) : W \leftrightarrow W'$ will be referred to as a "relational correspondence." Such a correspondence determines a relation between state sets expressed as $EQ_\rho * R$, where EQ_ρ means that the ρ-related locations have equal values (modulo ρ) and the $*$ connective, adapted from separation logic [22, 11, 28], means that the two parts of the relation access disjoint sets of locations. Now, a state transformation that preserves $EQ_\rho * R$ is allowed to look up and update ρ-related locations. It is also allowed to store pointers to ρ-related locations in other locations. However, it cannot store pointers to locations not related by ρ. The parametricity constraints imply that only the ρ-related locations can be leaked.

The information leakage in Example 2 is then explained as follows: The procedure call to `h` must preserve all relational correspondences $(\sigma, S) <: I_W$ that allow its argument `x` to be interpreted. Since the argument is a pointer to a heap location, the extended partial bijection σ must contain a pair (l, l), where l is the heap location that `x` points to. Hence `h(x)` can store pointers to l in

[4] Because of the subtle distinction between pointer values `x` and the pointed variables `x↑`, we prefer to work with an explicit pointer language like Pascal. Languages like Java, where pointers are treated implicitly, do not make this distinction and consequently lack the facility to control access. Surreptitious leakage is pervasive in the programs of such languages.

W-accessible locations with the result that l itself becomes W-accessible. This has an effect for the later procedure call `p()`, which can modify any W-accessible location including l.

Both of our previous examples have to do with data abstraction, albeit in a veiled form. (The program blocks create local data structures which they attempt to hide from the client procedures in varying ways.) Our programming language also contains a class construct, previously studied in [18, 19], providing a more direct form of data abstraction. The next example uses this to illustrate relational reasoning:

Example 3. Consider a list class implemented using linked lists in heap:

```
List = class : listsig
           local var (↑node) head; init head := nil; meth ...
       end
```

Here, `listsig` is the interface type of the `List` class and `node` is a recursively-defined storable data type: `node = int × ↑node`. We omit the details of the methods which include the usual operations for insertion, deletion and look-up.

To verify the correctness of such a class, one can prove its equivalence with another class that uses mathematical sequences as the internal representation:

```
List' = class : listsig
            local var (int*) s; init s := ⟨ ⟩; meth ...
        end
```

Here int* represents the set of integer sequences regarded as a data type, and the methods update the variable `s` to achieve the same effect as the methods in the concrete class. Intuitively, one reasons about the equivalence of the two classes by considering a relation between their states to the effect that the variable `s` in `List'` holds exactly the sequence of elements stored in the linked list starting at `head`, and showing that all the methods preserve this relation. Such a relation is formalized in our setting as follows.

The two representation worlds contain one location each, for the local variables of the classes: $W = \{l : \uparrow node\}$ and $W' = \{l' : \mathrm{int}^*\}$. The partial bijection part of the correspondence is the empty relation $\emptyset : W \leftrightarrow W'$ because only visible locations need be included in the partial bijection but l and l' are not visible to the clients of the classes. The state relation part of the correspondence is a relation R defined as follows:

$$s\,[R]\,s' \iff \mathrm{rep}(s, l, s'(l'))$$
$$\mathrm{rep}(s, l, \alpha) \iff (s(l) = \mathrm{nil} \wedge \alpha = \langle\rangle) \vee \exists n, k, \beta.(s(l) = (n, k) \wedge \alpha = \langle n\rangle{\cdot}\beta \wedge \mathrm{rep}(s, k, \beta))$$

□

The important point to notice is that R is not simply a relation between $\mathrm{St}(W)$ and $\mathrm{St}(W')$. In fact, the world W does not contain any locations that can be used for the nodes of the linked list. Rather R should be viewed as a relation that applies not only to the states for W and W' but also to *all their future extensions* with additional locations. This is one of the key technical issues that is addressed in the definitions to follow.

3 Definitions

Let δ range over a collection of data types. In particular, we assume that $\uparrow\delta$ is a data type for any data type δ.

Let $\mathrm{Loc} = \uplus_\delta \mathrm{Loc}_\delta$ be a countable set, countable for each δ, whose elements are regarded as names of "typed locations." A *location world* is a finite subset $W \subseteq_{\mathrm{fin}} \mathrm{Loc}$. It is also intuitive to think of a location world as a record type $[l_1 \colon \delta_1, \ldots, l_n \colon \delta_n]$. A *subtype* $X <: W$ is a superset $X \supseteq W$ of locations. In terms of records, X is a longer record type than W.

Fix a set of values $\mathrm{Val}(\delta)$ for each data type δ such that $\mathrm{Val}(\uparrow\delta) = \mathrm{Loc}_\delta \uplus \{\mathrm{nil}\}$.

We use the following technical notion of a "heap" (or a partial state with pointers) from the work on separation logic [11]. A *heap* is a pair $\langle L, s\rangle$ where $L \subseteq_{\mathrm{fin}} \mathrm{Loc}$ and $s : \prod_{l^\delta \in L} \mathrm{Val}(\delta)$ is a mapping of locations to values. We simply denote a heap $\langle L, s\rangle$ by s, and denote L by $\mathrm{dom}(s)$. If $s(l)$ is a data value involving another location l', l' may or may not be in $\mathrm{dom}(s)$. If $l' \notin \mathrm{dom}(s)$ then its occurrence in $s(l)$ is called a "dangling pointer." A heap with no dangling pointers is said to be *total*.

Whenever s_1 and s_2 are heaps with disjoint domains, $s_1 * s_2$ denotes their *join* with $\mathrm{dom}(s_1 * s_2) = \mathrm{dom}(s_1) \uplus \mathrm{dom}(s_2)$. Much use is made of this operation in the separation logic [11] and the Banerjee-Naumann work [2]. It will play a central role in our work as well.

A *state* for a world W is a heap s such that $\mathrm{dom}(s) = W$ and there are *no dangling pointers* in s. The set of states for a world W is denoted $\mathrm{St}(W)$.

Definition 1. A **renaming relation** is a triple $\rho = \langle W, W', \rho\rangle$ where

- W and W' are location worlds, and
- $\rho \subseteq W \times W'$ is a type-respecting relation that is single-valued and injective.

(That is, ρ is a type-respecting bijection between some subsets $L \subseteq W$ and $L' \subseteq W'$.) We refer to W as $\mathrm{dom}(\rho)$, W' as $\mathrm{cod}(\rho)$ and the relation as the "graph" of ρ.

If $X <: W$ and $X' <: W'$ are extended worlds and $\sigma = \langle X, X', \sigma\rangle$ and $\rho = \langle W, W', \rho\rangle$ are renaming relations, we say that σ is an *extension* of ρ and write $\sigma <: \rho$ if $\sigma \cap (W \times W') = \rho$. $\square$

As mentioned in the context of Example 2, the purpose of renaming relations is to identify the visible locations. Since the pointers to such locations can be stored in other visible locations, we define the following notation. For $d, d' \in \mathrm{Val}(\delta)$, we say that d and d' are equivalent modulo ρ, and write $d \equiv_\rho d'$, if d and d' denote the same data value assuming that all ρ-related locations are deemed to be equal.

In the following definitions, we make crucial use of relations between *partial* heaps. Even though we are, in the end, interested in relations between total states, these relations will be defined using those on heaps.

- If ρ is a renaming relation, EQ_ρ relates heaps that have equal values in ρ-related locations (where $\rho \restriction i$ denotes projection of i'th components):

  $$s\ [EQ_\rho]\ s' \iff \mathrm{dom}(s) = \rho \restriction 1 \wedge \mathrm{dom}(s') = \rho \restriction 2 \wedge \forall (l, l') \in \rho.\, s(l) \equiv_\rho s'(l').$$

- The relation emp relates empty heaps: $s\ [\mathrm{emp}]\ s' \iff \mathrm{dom}(s) = \emptyset = \mathrm{dom}(s')$
- The relation $R * S$ puts together two relations R and S side by side:

  $$s\ [R * S]\ s' \iff \exists s_1, s_2, s_1', s_2'.\ s = s_1 * s_2 \wedge s' = s_1' * s_2' \wedge s_1\ [R]\ s_1' \wedge s_2\ [S]\ s_2'$$

  This is the binary version of the $*$ connective in separation logic [11] and is extremely powerful. Its power owes to the fact that we do not have to specify in advance which parts of the heaps R and S run between. In a manner of speaking, R and S are "untyped" relations even if $R * S$ may be a "typed" relation.

Definition 2. A **relational correspondence** between location worlds is a pair $(\rho, R) : W \leftrightarrow W'$ where

- ρ is a renaming relation between W and W' and
- R is a function mapping all extensions $\pi <: \rho$ to relations between heaps,

such that, whenever $\pi_2 <: \pi_1 <: \rho$, $R(\pi_1) \subseteq R(\pi_2)$.

The extension relation for correspondences is defined by $(\sigma, S) <: (\rho, R)$ if and only if (i) $\sigma <: \rho$, and (ii) for any $\pi <: \sigma$, there is a relation P such that $S(\pi) = R(\pi) * P$. $\square$

This is the key definition of this paper. We explain it in detail. The intuition is that the state consists of

- *visible locations*, identified by ρ, which must allow look-up, update and storage, and
- *hidden locations*, related by $R(\pi)$, which contain representations for abstract data and, so, can only be modified by invariant-preserving operations.

The visible locations and the hidden locations are disjoint. The visible locations must have equal values in related states. The hidden locations, on the other hand, are related by some relation $R(\pi)$ that captures the data representation invariants. The relation $R(\pi)$ is parameterized by renamings π so that information about visible locations mentioned in π can be incorporated in its formulation. The condition $R(\pi_1) \subseteq R(\pi_2)$ means that related states continue to be related if the states are extended with additional visible locations. The intuition for the definition of $(\sigma, S) <: (\rho, R)$ is that S extends R by imposing additional conditions for new locations but does not alter R for the part of the heap that R deals with. This is the same intuition as that in [16, 14] for local variables.

The identity correspondence for a world W is $I_W = (\iota_W, \mathrm{emp}_W) : W \leftrightarrow W$, where ι_W is the diagonal relation for W and emp_W maps every $\pi <: \iota_W$ to emp.

Fact 1. *Whenever $X <: W$, $I_X <: I_W$.*

Having defined relational correspondences, we must specify how these are used to relate states. Note that the relation $EQ_\rho * R(\rho)$ relates *heaps* (or partial states with arbitrary domains). The corresponding relation for states is obtained by restricting the heap relation to states:

$$\begin{aligned} &\mathrm{St}(\rho, R) : \mathrm{St}(W) \leftrightarrow \mathrm{St}(W') \\ &\mathrm{St}(\rho, R) = (EQ_\rho * R(\rho)) \cap (\mathrm{St}(W) \times \mathrm{St}(W')) \end{aligned}$$

The idea is that in order to define a typed relation between states, we transit to the untyped world of partial heaps where we have the powerful $*$ connective available and coerce the results back to the typed world. Defining the required relations without the $*$ connective would be extremely awkward.

Fact 2. $\mathrm{St}(I_W)$ *is the identity relation on* $\mathrm{St}(W)$.

To make these definitions concrete, we give an example:

Example 4. Consider the list data structure from Example 3 but now adapted to contain pointers to integer cells instead of just integers. The type of nodes is given by $node = \uparrow\mathrm{int} \times \uparrow node$. For the worlds $W = \{l : \uparrow node\}$ and $W' = \{l' : (\uparrow\mathrm{int})^*\}$, we define a correspondence $(\emptyset, R)$ where the relation function $R(\pi)$ is defined by:

$$
\begin{aligned}
s\ [R(\pi)]\ s' &\iff \mathrm{rep}_\pi(s, l, s'(l')) \\
\mathrm{rep}_\pi(s, l, \alpha) &\iff (s(l){=}\mathrm{nil} \wedge \alpha{=}\langle\rangle) \vee \\
&\qquad \exists n, n', k, \beta.(s(l){=}(n,k) \wedge \alpha{=}\langle n'\rangle{\cdot}\beta \wedge (n,n') \in \pi \wedge \mathrm{rep}_\pi(s,k,\beta))
\end{aligned}
$$

Notice the use of π argument in relating the contents of the list cells. The corresponding definition for Example 3 would use a constant function $R(\pi)$ because no pointers are to be related. □

Categorical matters

We use the setting of reflexive graph categories [14, 23, 3] to explicate the categorical structure that we use.

Proposition 3. *There is a reflexive graph of categories* **World** *with the following data: worlds as vertices, extensions* $X <: W$ *as vertex morphisms, correspondences* $(\rho, R) : W \leftrightarrow W'$ *as edges and extensions* $(\sigma, S) <: (\rho, R)$ *as edge morphisms. The identity edges are the identity correspondences.*

Let **Set** denote the reflexive graph with sets and functions forming the vertex category and binary relations and relation-preserving squares forming the edge category. We will be working with the functor category $\mathbf{Set}^{\mathbf{World}^{\mathrm{op}}}$ whose objects are reflexive graph-functors $\mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$ and morphisms are parametric natural transformations. (To deal with divergence and recursion, we must really use **Cpo** in place of **Set**. We omit the treatment of recursion in this version of the paper, but it can be treated the same way as in [14].)

Definitions of parametric limits $\forall_X F(X)$ and parametric colimits $\exists_X F(X)$ for arbitrary reflexive graph-functors F may be found in [3]. In our case, we will be using these with nonvariant functors $F : \mathbf{World}^\circ \to \mathbf{Set}$ (where $\mathbf{World}^\circ$ is the discrete reflexive graph corresponding to **World** with only identity morphisms). We will also use parametric ends $\int_X F(X, X)$ for functors F of type $\mathbf{World} \times \mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$. See below for explicit constructions for these limits, colimits and ends.

The notation $\forall_{X <: W} F(X)$ is used to denote the parametric limit of the functor $F \circ J^\circ : (\mathbf{World}_{<:W})^\circ \to \mathbf{Set}$ where $\mathbf{World}_{<:W}$ is the reflexive subgraph of **World** with vertices $X <: W$ and edges $(\sigma, S) <: I_W$, and J is its inclusion in **World**. It is to be noted that the type expression $\forall_{X<:W} F(X)$ forms a contravariant functor $T(W)$. The notation $\exists_{X<:W} F(X)$ similarly refers to the parametric colimit of $F \circ J^\circ$ (covariantly in W) and $\int_{X<:W} F(X, X)$ refers to the parametric end of $F \circ (J \times J^{\mathrm{op}})$ (contravariantly in W).

The functor category $\mathbf{Set}^{\mathbf{World}^{\mathrm{op}}}$ is cartesian closed with products given pointwise and exponents $F \Rightarrow G$ given by $(F \Rightarrow G)(W) = \int_{X<:W} F(X) \to G(X)$ [14, 3].

Explicit constructions

For the benefit of the reader unfamiliar with parametric limits, we give direct definitions of these constructions (which may be seen to be special cases of the definitions in [3]).

Let F be a type operator that associates, to every world W, a set $F(W)$ and, to every correspondence $(\rho, R) : W \leftrightarrow W'$, a relation $F(\rho, R) : F(W) \leftrightarrow F(W')$ such that $F(I_W) = \Delta_{F(W)}$. Then,

- $\prod_X F(X)$ is the set of families of the form $\{p_X \in F(X)\}_X$ indexed by all worlds X. $\prod_{X<:W} F(X)$ is similar except that the families are indexed only by subtypes of W.
- $\forall_X F(X)$ is a subset of $\prod_X F(X)$ consisting of families satisfying the *parametricity* condition: for all correspondences $(\rho, R) : X \leftrightarrow X'$ between different worlds, the components p_X and $p_{X'}$ are related by $F(\rho, R)$.
- $\forall_{X<:W} F(X)$ is a subset of $\prod_{X<:W} F(X)$ with a parametricity condition that applies only to correspondences $(\rho, R) <: I_W$. We say that the families are parametric with respect to W.
- $\sum_X F(X)$ is the set of pairs of the form $\langle X, a\rangle$ where X is a world and $a \in F(X)$. Such pairs should be viewed as "implementations" of abstract data types, where X denotes the representation type and a is the collection of operations. The set $\sum_{X<:W} F(X)$ is similar except that the worlds X are restricted to subtypes of W.
- $\exists_X F(X)$ is the quotient of $\sum_X F(X)$ under a *behavioral equivalence* relation. First, if $\langle X, a\rangle$ and $\langle X', a'\rangle$ are pairs in $\sum_X F(X)$, a *simulation* relation between them is a correspondence $(\rho, R) : X \leftrightarrow X'$ such that a and a' are related by $F(\rho, R)$. Two pairs $\langle X, a\rangle$ and $\langle X', a'\rangle$ are behaviorally equivalent, written $\langle X, a\rangle \approx \langle X', a'\rangle$, if there is a sequence of pairs $\langle X, a\rangle$, $\langle X_1, a_1\rangle, \ldots,$ $\langle X_{n-1}, a_{n-1}\rangle$, $\langle X', a'\rangle$ with simulation relations between successive pairs. The equivalence class of a pair $\langle X, a\rangle$ under the behavioral equivalence relation is denoted $\langle\!| X, a |\!\rangle$. These equivalence classes denote true "abstract data types" [10, 19].
- $\exists_{X<:W} F(X)$ is a quotient of $\sum_{X<:W} F(X)$ where the allowed simulations between pairs are restricted to correspondences $(\rho, R) <: I_W$. The induced behavioral equivalence relation with respect to W is denoted $\approx_W$ and the equivalence class of a pair $\langle X, a\rangle$ is denoted $\langle\!| X, a |\!\rangle_W$. These equivalence classes should be viewed as "partially abstract" types whose representations X are hidden except for the knowledge that they form subtypes of W.

The intuitive reading of $\exists_{X<:W}\mathrm{St}(X)$ is that all the locations in X that are not accessible from W are hidden. This intuition can be clearly seen in the following "garbage collection" lemma:

Lemma 4. *Let* $\mathrm{GC}_W : \exists_{X<:W}\mathrm{St}(X) \to \exists_{X<:W}\mathrm{St}(X)$ *be defined by*

$$\mathrm{GC}_W(\langle\!|X, s|\!\rangle) = \langle\!|\mathrm{reach}_X(W, s),\ s \restriction \mathrm{reach}(W, s)|\!\rangle$$

where $\mathrm{reach}_X(W, s)$ *is the subset of* X *consisting of all locations reachable from* W *in the heap* s*. Then* GC_W *is the identity function on* $\exists_{X<:W}\mathrm{St}(X)$*.*

This result signifies that reachability of locations has been properly captured by the relational correspondences.

A *reflexive graph-functor* $F : \mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$ is a type operator that also has an associated contravariant action on the morphisms of **World**. That means that, for all subtypings $X <: W$, there are functions $F(X{<:}W) : F(W) \to F(X)$ preserving identity and composition. Moreover, if $(\sigma, S) <: (\rho, R)$ then the functions $F(X{<:}W)$ and $F(X'{<:}W')$ map $F(\rho, R)$-related arguments to $F(\sigma, S)$-related results.

We note two general cases of functors arising in our setting:

- The type expression $T(W) = \forall_{X<:W}F(X)$ forms a contravariant functor in W, independent of whether F is functorial. The morphism part $T(V{<:}W)$ sends $\{p_X\}_{X<:W}$ to $\{p_X\}_{X<:V}$. The relation action $T(\rho, R) : T(W) \leftrightarrow T(W')$ is given by

  $$\begin{array}{l}\{p_X\}_{X<:W}\ [T(\rho, R)]\ \{p'_{X'}\}_{X'<:W'} \iff \\ \qquad \text{for all } (\sigma, S) : X \leftrightarrow X' \text{ such that } (\sigma, S) <: (\rho, R),\ p_X\ [F(\sigma, S)]\ p'_{X'}\end{array}$$

  We write this relation as $\forall_{(\sigma,S)<:(\rho,R)}F(\sigma, S)$.
- The type expression $T(W) = \exists_{X<:W}F(X)$ determines a covariant functor in W. If $V <: W$, we have the morphism part $T(V{<:}W) : T(V) \to T(W)$ which sends $\langle\!|X, a|\!\rangle_V$ to $\langle\!|X, a|\!\rangle_W$. Since any simulation relation with respect to V is also a simulation relation with respect to W, this function is well-defined. We use the notation $\mathit{hide}_{V<:W}$ to denote it. The relation action $T(\rho, R) : T(W) \leftrightarrow T(W')$ is given by

  $$\begin{array}{l}\langle\!|X, a|\!\rangle_W\ [T(\rho, R)]\ \langle\!|X', a'|\!\rangle_{W'} \iff \\ \qquad \text{there exist } \langle Y, b\rangle \approx_W \langle X, a\rangle,\ \langle Y', b'\rangle \approx_{W'} \langle X', a'\rangle \\ \qquad \text{and } (\sigma, S) : Y \leftrightarrow Y' \text{ such that } (\sigma, S) <: (\rho, R) \text{ and } b\ [F(\sigma, S)]\ b'\end{array}$$

  We write this relation as $\exists_{(\sigma,S)<:(\rho,R)}F(\sigma, S)$.

The $\forall$ quantifier uses relational parametricity to capture uniformity and information hiding. The categorical condition of natural transformation is an alternative condition for uniformity. In [17, 3], it is argued that ideally naturality should be subsumed under parametricity. However, our relational correspondences for heap worlds are not rich enough to subsume naturality. So, for the present paper, we treat naturality separately. If $F, G : \mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$ are functors, we use the notation $\forall_X F(X) \to G(X)$ to mean families of functions that are *parametric as well as natural* (which would be written more formally as $\int_X F(X) \to G(X)$ in the notation of [3].) Similarly, $\forall_{X<:W}F(X) \to G(X)$ denotes families that are parametric as well as natural in X with respect to W.

$$\frac{}{\Gamma, x:\nu \vdash x : \mathbf{exp}\,\nu} \quad \frac{\Gamma \vdash C_1 : \mathbf{com} \quad \Gamma \vdash C_2 : \mathbf{com}}{\Gamma \vdash C_1; C_2 : \mathbf{com}} \quad \frac{\Gamma, x : \mathbf{var}\,\delta \vdash C : \mathbf{com}}{\Gamma \vdash \{\mathbf{local\ var}\ \delta\ x; C\} : \mathbf{com}}$$

$$\frac{\Gamma \vdash V : \mathbf{exp}(\mathbf{var}\,\delta)}{\Gamma \vdash \mathbf{read}\ V : \mathbf{exp}\,\delta} \quad \frac{\Gamma \vdash V : \mathbf{exp}(\mathbf{var}\,\delta) \quad \Gamma \vdash E : \mathbf{exp}\,\delta}{\Gamma \vdash V := E : \mathbf{com}} \quad \frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{com}}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}(\delta_1 \times \cdots \delta_n)}{\Gamma \vdash E.i : \mathbf{exp}\ \delta_i} \quad \frac{\Gamma \vdash V : \mathbf{exp}(\mathbf{var}(\delta_1 \times \cdots \times \delta_n)) \quad \Gamma \vdash E : \mathbf{exp}\ \delta_i}{\Gamma \vdash V.i := E : \mathbf{com}}$$

$$\frac{}{\Gamma \vdash \mathrm{nil} : \mathbf{exp}(\uparrow\delta)} \quad \frac{\Gamma \vdash E : \mathbf{exp}(\uparrow\delta)}{\Gamma \vdash E\!\uparrow : \mathbf{exp}(\mathbf{var}\ \delta)} \quad \frac{\Gamma \vdash V : \mathbf{exp}(\mathbf{var}(\uparrow\delta))}{\Gamma \vdash V := \mathbf{new}\ \delta : \mathbf{com}}$$

$$\frac{\Gamma, x:\nu \vdash M : \pi}{\Gamma \vdash \lambda x.\, M : \mathbf{exp}\ (\nu \to \pi)} \quad \frac{\Gamma \vdash M : \mathbf{exp}(\nu \to \pi) \quad \Gamma \vdash N : \mathbf{exp}\ \nu}{\Gamma \vdash M(N) : \pi}$$

$$\frac{\Gamma, x : \mathbf{var}\,\delta \vdash A : \mathbf{com} \quad \Gamma, x : \mathbf{var}\,\delta \vdash M : \mathbf{exp}\,\nu}{\Gamma \vdash \mathbf{class} : \nu\ \mathbf{local\ var}\ \delta\ x\ \mathbf{init}\ A\ \mathbf{meth}\ M\ \mathbf{end} : \mathbf{exp}\ (\mathbf{cls}\ \nu)}$$

$$\frac{\Gamma \vdash K : \mathbf{exp}\ (\mathbf{cls}\ \nu) \quad \Gamma, x:\nu \vdash C : \mathbf{com}}{\Gamma \vdash \{\mathbf{local}\ K\ x;\ C\} : \mathbf{com}}$$

Table 1. Type syntax of terms

Notation We use convenient notation borrowed from the polymorphic lambda calculus [20] to denote polymorphic families. A family $\{P(X)\}_{X<:W}$ is written as $\Lambda X{<:}W.\,P(X)$ and, if ϕ is such a family, then component selection ϕ_X is written as $\phi[X]$.

4 Semantics

We consider a Pascal-like language with types given by the following syntax:

$$\begin{array}{ll} \text{(data types)} & \delta ::= \mathrm{int} \mid \uparrow\delta \mid \delta_1 \times \cdots \times \delta_n \\ \text{(value types)} & \nu ::= \delta \mid \mathbf{var}\ \delta \mid \nu_1 \times \cdots \times \nu_n \mid \nu \to \pi \mid \mathbf{cls}\,\nu \\ \text{(phrase types)} & \pi ::= \mathbf{exp}\,\nu \mid \mathbf{com} \end{array}$$

Data types identify storable values, and value types identify bindable values (or values that can be passed to procedures). Phrase types are the types of terms.

The term syntax for our language is given in Table 1. We use a sample of command forms. Other forms can be accommodated in a similar fashion. The notation for classes is borrowed from [18, 19].

The types are interpreted as reflexive graph-functors $\mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$. The interpretation comes in three parts: the set part $[\![\tau]\!]$ maps worlds to sets, the relation part $\langle\!\langle\tau\rangle\!\rangle$ maps correspondences $(\rho, R) : W \leftrightarrow W'$ to relations $[\![\tau]\!](W) \leftrightarrow [\![\tau]\!](W')$ and the morphism part gives, for every subtyping $X <: W$, a function $[\![\tau]\!](X{<:}W) : [\![\tau]\!](W) \to [\![\tau]\!](X)$ such that correspondences are preserved.

$$\begin{aligned} [\![\mathrm{int}]\!](W) &= \mathit{Int} \\ [\![\uparrow\delta]\!](W) &= (\mathrm{Loc}_\delta \cap W) + \{\mathrm{nil}\} \\ [\![\mathbf{var}\ \delta]\!](W) &= [\![\delta \to \mathbf{com}]\!](W) \times [\![\mathbf{exp}\ \delta]\!](W) \\ [\![\nu_1 \times \cdots \times \nu_n]\!](W) &= [\![\nu_1]\!](W) \times \cdots \times [\![\nu_n]\!](W) \end{aligned}$$

$$
\begin{array}{l}
unit^{\nu}_{W} : [\![\nu]\!](W) \to [\![\mathbf{exp}\,\nu]\!](W) \\
unit_W\; d = \Lambda X{<:}W.\, \lambda s.\, d \uparrow^{X}_{W} \\
bind^{\nu,\pi}_{W} : [\![\mathbf{exp}\,\nu]\!](W) \times [\![\nu \to \pi]\!](W) \to [\![\pi]\!](W) \\
bind_W\; (e, f) = \Lambda X{<:}W.\, \lambda s.\, \text{let } d = e\,[X]\, s \\
\qquad\qquad\qquad\qquad\qquad \text{in if } d = \text{fault then fault else } f\,[X]\,d\,[X]\,s \\
bind^{\nu_1,\nu_2,\pi}_{W} : [\![\mathbf{exp}\,\nu_1]\!](W) \times [\![\mathbf{exp}\,\nu_2]\!](W) \times [\![\nu_1 \times \nu_2 \to \pi]\!](W) \to [\![\pi]\!](W) \\
bind_W\; (e_1, e_2, f) = \Lambda X{<:}W.\, \lambda s.\, \text{let } d_1 = e_1\,[X]\,s,\; d_2 = e_2\,[X]\,s \\
\qquad\qquad\qquad\qquad\qquad \text{in if } d_1 = \text{fault} \vee d_2 = \text{fault then fault} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } f\,[X]\,(d_1, d_2)\,[X]\,s \\
hide_{Y<:X} : [(\exists_{Z<:Y} \mathrm{St}(Z)) + \{\text{fault}\}] \to [(\exists_{Z<:X} \mathrm{St}(Z)) + \{\text{fault}\}] \\
hide_{Y<:X}\; r = \text{case } r \text{ of } \langle\!| Z, s |\!\rangle_Y \Rightarrow \langle\!| Z, s |\!\rangle_X \mid \text{fault} \Rightarrow \text{fault} \\
seq_W : [\![\mathbf{com}]\!](W) \times [\![\mathbf{com}]\!](W) \to [\![\mathbf{com}]\!](W) \\
seq_W\; (c, c') = \Lambda X{<:}W.\, \lambda s. \begin{cases} \text{fault} & \text{if } c\,[X]\,s = \text{fault} \\ hide_{Y<:X}\left(c'\,[Y]\,s'\right) & \text{if } c\,[X]\,s = \langle\!| Y, s' |\!\rangle_X \end{cases}
\end{array}
$$

Table 2. Semantic combinators

$$
\begin{aligned}
[\![\nu \to \pi]\!](W) &= \forall_{X<:W} [\![\nu]\!](X) \to [\![\pi]\!](X) \\
[\![\mathbf{cls}\,\nu]\!](W) &= \forall_{X<:W} \exists_{Z<:X} [\![\mathbf{exp}\,\nu]\!](Z) \times [\mathrm{St}(X) \to \exists_{Y<:Z} \mathrm{St}(Y) + \{\text{fault}\}] \\
[\![\mathbf{exp}\,\nu]\!](W) &= \forall_{X<:W} \mathrm{St}(X) \to [\![\nu]\!](X) + \{\text{fault}\} \\
[\![\mathbf{com}]\!](W) &= \forall_{X<:W} \mathrm{St}(X) \to \exists_{Y<:X} \mathrm{St}(Y) + \{\text{fault}\}
\end{aligned}
$$

The position of the type quantifications $\forall$ and $\exists$ in the type interpretations has been recognized in earlier work [25, 4, 8]. Intuitively, a command defined for a world W should be prepared to accept additional locations (represented by X) in its input state, and it might itself allocate new locations during the execution (represented by Y). The parametricity interpretation of the type quantifiers means that the command does not have direct access to the extra locations in its input state and the successor commands will not have direct access to the locations allocated by the present command.

Variables are interpreted as pairs of "put" and "get" methods, as in Reynolds [21]. Indeed, if $l \in W$ is a δ-typed location, we can map it to a pair of methods $var^{\delta}_{W}(l) = (put^{\delta}_{W}(l),\; get^{\delta}_{W}(l))$ defined as follows:

$$put^{\delta}_{W}(l)\,[Y]\,k\,[Z]\,s = \langle\!| Z, s[l \to k] |\!\rangle_Z, \text{ and } get^{\delta}_{W}(l)\,[Y]\,s = s(l).$$

The relation interpretation of types $\langle\!\langle \tau \rangle\!\rangle$ is straightforward: $\langle\!\langle \text{int} \rangle\!\rangle(\rho, R) = \Delta_{Int}$, $\langle\!\langle \uparrow\!\delta \rangle\!\rangle(\rho, R) = \rho + \Delta_{\{\text{nil}\}}$ and, for all other cases, it follows from the structure of $[\![\tau]\!]$. For the morphism part, $[\![\text{int}]\!](X{<:}W)$ is id_{Int}, $[\![\uparrow\!\delta]\!](X{<:}W)$ is the evident inclusion, and for all other cases, it follows from the structure of the types. We use the shorthand notation $a{\uparrow}^{X}_{W}$ for $[\![\tau]\!](X{<:}W)(a)$ when $a \in [\![\tau]\!](W)$.

The semantics of a term with typing $x_1{:}\,\nu_1, \ldots, x_n{:}\,\nu_n \vdash M : \pi$ is a parametric natural transformation of type $\forall_W\, [\![\nu_1]\!](W) \times \cdots \times [\![\nu_n]\!](W) \to [\![\pi]\!](W)$. (As usual values of the type $[\![\nu_1]\!](W) \times \cdots \times [\![\nu_n]\!](W)$ will be regarded as "environments" ranged over by the symbol η.)

We use the semantic combinators from Table 2. We also assume that there is a family of functions $newloc_{\delta}(X)$ that give, for each world X, a δ-typed location that is not in X.

$$
\begin{aligned}
&[\![x]\!]_W \eta = unit_W(\eta(x))\\
&[\![\mathbf{skip}]\!]_W \eta = \Lambda X{<:}W.\, \lambda s.\, \langle\!| X, s |\!\rangle_X\\
&[\![C_1; C_2]\!]_W \eta = seq_W([\![C_1]\!]_W \eta, [\![C_2]\!]_W \eta)\\
&[\![\{\mathbf{local\ var}\ \delta\ x; C\}]\!]_W \eta =\\
&\qquad \Lambda X{<:}W.\, \lambda s.\, hide_{X^+ <: X}\left([\![C]\!]_{X^+}\ (\eta{\uparrow}_W^{X^+}\ [x \to var^\delta_{X^+}(l)])\ [X^+]\ (s * [l \to \mathrm{init}_\delta])\right)\\
&\qquad\qquad\qquad\qquad \text{where } l = newloc_\delta(X) \text{ and } X^+ = X \uplus \{l\}\\
&[\![\mathbf{read}\ V]\!]_W \eta = bind_W\ ([\![V]\!]_W \eta,\ \Lambda X{<:}W.\, \lambda(p,g).\, g)\\
&[\![V := E]\!]_W \eta = bind_W\ ([\![V]\!]_W \eta,\ [\![E]\!]_W \eta,\ \Lambda X{<:}W.\, \lambda((p,g),k).\, p[X]k)\\
&[\![E{\uparrow}]\!]_W \eta = bind_W\ ([\![E]\!]_W \eta,\ deref^\delta_W)\\
&[\![V := \mathbf{new}\ \delta]\!]_W \eta = bind_W\ ([\![V]\!]_W \eta,\ alloc^\delta_W)\\
&[\![\lambda x.\, M]\!]_W \eta = unit_W(\Lambda X{<:}W.\, \lambda d.\, [\![M]\!]_X (\eta{\uparrow}_W^X [x \to d]))\\
&[\![M(N)]\!]_W \eta = bind_W\ ([\![M]\!]_W \eta,\ [\![N]\!]_W \eta,\ \Lambda X{<:}W.\, \lambda(f,d).\ f[X](d))\\
&[\![\mathbf{class} : \nu\ \mathbf{local\ var}\ \delta\ x\ \mathbf{init}\ A\ \mathbf{meth}\ M\ \mathbf{end}]\!]_W \eta =\\
&\qquad unit_W(\Lambda X{<:}W.\, \langle\!| X^+, [\![M]\!]_{X^+}\eta^+,\ \lambda s.\, [\![A]\!]_{X^+}(\eta^+)(s * [l \to \mathrm{init}_\delta]) |\!\rangle_X)\\
&\qquad\qquad \text{where } l = newloc_\delta(X),\ X^+ = X \uplus \{l\}, \text{ and } \eta^+ = \eta{\uparrow}_W^{X^+}[x \to var^\delta_{X^+}(l)]\\
&[\![\{\mathbf{local}\ K\ x; C\}]\!]_W \eta =\\
&\qquad bind_W \left([\![K]\!]_W \eta,\quad \begin{array}{l}
\Lambda X{<:}W.\, \lambda k.\, \Lambda Y{<:}X.\, \lambda s.\\
\quad \text{let } \langle\!| Z, m, i |\!\rangle_Y = k[Y]\\
\quad \text{in if } \langle\!| Z', s' |\!\rangle_Z = i(s) \wedge m[Z']s' \neq \text{fault}\\
\qquad \text{then } hide_{Z' <: Y}([\![C]\!]_{Z'}(\eta{\uparrow}_W^{Z'}[x \to m[Z']s'])\, s')\\
\qquad \text{else fault}
\end{array}\right)
\end{aligned}
$$

These definitions are expressed using operations $alloc^\delta_W : [\![\mathbf{var}(\uparrow\delta) \to \mathbf{com}]\!](W)$ and $deref^\delta_W : [\![\uparrow\delta \to \mathbf{exp}\ (\mathbf{var}\ \delta)]\!](W)$:

$$
\begin{aligned}
alloc^\delta_W\ [X]\ (p,g)\ [Z]\ s &= hide_{Z^+ <: Z}(p\ [Z^+]\ l\ [Z^+]\ (s * [l \to \mathrm{init}_\delta]))\\
&\qquad \text{where } l = newloc^\delta(Z) \text{ and } Z^+ = Z \uplus \{l\}\\
deref^\delta_W\ [X]\ l\ [Z]\ s &= \text{if } l \neq \text{nil then } var^\delta_Z(l) \text{ else fault}
\end{aligned}
$$

5 Results

The most basic result to be proved about our semantics is that it satisfies an abstraction theorem. (Really, this is not a separate result from the semantic definition, but rather an integral part of checking that the semantics is well-defined.)

Theorem 5. *The meaning of every term $[\![\Gamma \vdash M{:}\theta]\!]$ is a parametric natural transformation of type $[\![\Gamma]\!] \to [\![\theta]\!]$. That is,*

1. *for all worlds W and all environments $\eta \in [\![\Gamma]\!](W)$, $[\![M]\!]_W \eta \in [\![\theta]\!](W)$;*
2. *for all $(\rho, R) : W \leftrightarrow W'$, and all related environments $\eta[\langle\!\langle\Gamma\rangle\!\rangle(\rho,R)]\eta'$,*
   $[\![M]\!]_W \eta\ [\langle\!\langle\theta\rangle\!\rangle(\rho,R)]\ [\![M]\!]_{W'}\eta'$; and
3. *for all extensions $X <: W$ and all $\eta \in [\![\Gamma]\!](W)$, $([\![M]\!]_W \eta){\uparrow}_W^X = [\![M]\!]_X(\eta{\uparrow}_W^X)$.*

The abstraction theorem immediately implies the soundness of the simulation principle for data representation reasoning. Suppose $\{\langle\!| F(Y), m_Y, i_Y |\!\rangle\}_Y$ and $\{\langle\!| F'(Y), m'_Y, i'_Y |\!\rangle\}_Y$ are two *similar* implementations of a class, i.e., for any world Y, there is a simulation relation $(\sigma, S)\colon F(Y) \leftrightarrow F'(Y)$ such that $(\sigma, S) <: I_Y$ and $m_Y\ [\langle\!\langle \mathbf{exp}\ \nu \rangle\!\rangle(\sigma, S)]\ m'_Y$ and i_Y and i'_Y are related by $\mathrm{St}(I_Y) \to \exists_{(\tau,T)<:(\sigma,S)} \mathrm{St}(\tau,T) + \Delta_{\{\mathrm{fault}\}}$. Then in any command term of the form $\Gamma, C\colon \mathbf{cls}\nu \vdash \{\mathbf{local}\quad C\ x;\ M\} : \mathbf{com}$, we get the same results independent of which implementation is used for C. This is because when $i_Y\ s\ =\ \langle\!| Z, s_1 |\!\rangle_{F(Y)}$ and $i'_Y\ s = \langle\!| Z', s'_1 |\!\rangle_{F'(Y)}$,

$$\begin{aligned} &\mathit{hide}_{Z<:Y}\left(\llbracket M \rrbracket_Z (\eta{\uparrow}^Z_W[x{\to}m[Z]s_1])[Z]s_1\right) = \\ &\qquad \mathit{hide}_{Z'<:Y}\left(\llbracket M \rrbracket_{Z'} (\eta{\uparrow}^{Z'}_W[x{\to}m'[Z']s'_1])[Z']s'_1\right) \end{aligned}$$

for all $\eta \in \llbracket \Gamma \rrbracket_W$ and $Y <: W$, which follows from the abstraction theorem.

The separation logic for reasoning about heap data structures [22, 11, 28] contains an important rule called the "frame rule," which is central to the *local reasoning* methodology developed there. The frame rule is supported by the frame property of commands which says that if a command is safe in a given state, then the result of executing it in a larger state can be predicted based on an execution on the smaller state. This property is satisfied by our semantics. Say that a command $c \in \llbracket \mathbf{com} \rrbracket(W)$ is *safe* for world $X <: W$ and state $s \in \mathrm{St}(X)$ if $c[X](s) \neq \mathrm{fault}$.

Theorem 6. *Let $c \in \llbracket \mathbf{com} \rrbracket(W)$ be safe for world $X <: W$ and state s. Then for all extended worlds $X \uplus Z$ and states $s * t \in \mathrm{St}(X \uplus Z)$,*

1. *c is safe for $X \uplus Z$ and $s * t$, and*
2. *there exist world $Y <: X$ and state $s' \in \mathrm{St}(Y)$ such that $Y \cap Z = \emptyset$, $c[X]s = \langle\!| Y, s' |\!\rangle_X$, and $c[X \uplus Z](s * t) = \langle\!| Y \uplus Z, s' * t |\!\rangle_{X \uplus Z}$.*

We expect that this connection will pave the way for integrating the data representation reasoning studied here and the state-based reasoning developed with separation logic.

Acknowledgments We have benefited from discussions with Peter O'Hearn and David Naumann. Yang was supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

References

1. Abramsky, S., Honda, K., and McCusker, G. A fully abstract game semantics for general references. In *LICS 1998* (1998), pp. 334–344.
2. Banerjee, A., and Naumann, D. A. Representation independence, confinement and access control. In *POPL 2002* (2002), ACM.
3. Dunphy, B. P. *Parametricity as a Notion of Uniformity in Reflexive Graphs*. PhD thesis, University of Illinois, Dep. of Mathematics, 2002.
4. Ghica, D. R. Semantics of dynamic variables in Algol-like languages. Master's thesis, Queen's University, Kingston, Canada, Mar 1997. (available electronically from ftp://ftp.qucis.queensu.ca/pub/rdt).

5. Ghica, D. R. Parameters and linked structures in algol-like languages. In *Report of the Dagstuhl Seminar 98261: The Semantic Challenge of Object-oriented Programming* (1998), Schloss Dagstuhl.
6. Launchbury, J., and Peyton Jones, S. L. State in Haskell. *J. Lisp and Symbolic Comput. 8*, 4 (1995), 293–341.
7. Levy, P. B. *Call-by-Push-Value.* PhD thesis, Queen Mary, University of London, March 2001.
8. Levy, P. B. Possible world semantics for general storage in call-by-value. In *CSL 2002* (2002), pp. 232–246.
9. Meyer, A. R., and Sieber, K. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.* (1988), ACM, pp. 191–203. (Reprinted as Chapter 7 of [15]).
10. Mitchell, J. C., and Plotkin, G. D. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502.
11. O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *CSL 2001* (Berlin, 2001), L. Fribourg, Ed., vol. 2142 of *LNCS*, Springer-Verlag, pp. 1–19.
12. O'Hearn, P. W., and Reynolds, J. C. From Algol to polymorphic linear lambda-calculus. *J. ACM 47*, 1 (2000), 167–223.
13. O'Hearn, P. W., and Tennent, R. D. Semantics of local variables. In *Applications of Categories in Computer Science*, M. P. Fourman, P. T. Johnstone, and A. M. Pitts, Eds. Cambridge Univ. Press, 1992, pp. 217–238.
14. O'Hearn, P. W., and Tennent, R. D. Parametricity and local variables. *J. ACM 42*, 3 (1995), 658–709. (Reprinted as Chapter 16 of [15]).
15. O'Hearn, P. W., and Tennent, R. D. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.
16. Oles, F. J. *A Category-Theoretic Approach to the Semantics of Programming Languages.* PhD thesis, Syracuse University, 1982.
17. Reddy, U. S. When parametricity implies naturality. Electronic manuscript, July 1997. URL http://www.cs.bham.ac.uk/~udr.
18. Reddy, U. S. Objects and classes in Algol-like languages. In *Fifth Intern. Workshop on Foundations of Object-oriented Languages* (Jan 1998), electronic proceedings at http://pauillac.inria.fr/~remy/fool/proceedings.html.
19. Reddy, U. S. Objects and classes in Algol-like languages. *Information and Computation 172* (2002), 63–97.
20. Reynolds, J. C. Towards a theory of type structure. In *Coll. sur la Programmation*, vol. 19 of *LNCS*. Springer-Verlag, 1974, pp. 408–425.
21. Reynolds, J. C. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland, 1981, pp. 345–372. (Reprinted as Chapter 3 of [15]).
22. Reynolds, J. C. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2000.
23. Robinson, E., and Rosolini, G. Reflexive graphs and parametric polymorphism. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* (July 1994), IEEE Computer Society Press.
24. Stark, I. Names and higher-order functions. Technical Report 363, University of Cambridge Computer Laboratory, April 1995.
25. Stark, I. Categorical models for local names. *Lisp and Symbolic Computation 9*, 1 (Feb. 1996), 77–107.
26. Tennent, R. D. Correctness of data representations in Algol-like languages. In *A Classical Mind: Essays in Honor of C. A. R. Hoare*, A. W. Roscoe, Ed. Prentice-Hall International, 1994, pp. 405–417.
27. Wirth, N., and Hoare, C. A. R. A contribution to the development of Algol. *Comm. ACM 9*, 6 (June 1966), 413–432.
28. Yang, H. Local reasoning for stateful programs. Tech. Rep. UIUCDCS-R-2001-2227, University of Illinois, Dep. of Computer Science, July 2001.