Correctness of Data Representations involving Heap Data Structures

Uday S. Reddy

School of Computer Science, University of Birmingham, Birmingham B15 2TT, United Kingdom

Hongseok Yang

Micros Research Center, Korea Advanced Institute of Science and Technology, Daejeon, 305-701, South Korea

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 turns out that the correspondences between data representations cannot simply be relations between states, but more intricate correspondences that also need to keep track of visible locations whose pointers can be stored and leaked.

Key words: Denotational semantics, relational parametricity, separation logic, imperative programming languages.

1 Introduction

Programming languages with dynamically allocated storage variables ("heap variables") date back to Algol W [30] 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,

Email addresses: `U.Reddy@cs.bham.ac.uk` (Uday S. Reddy), `hyang@kaist.ac.kr` (Hongseok Yang).

Standard ML, and variants of Haskell [5]. 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 [16]), 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 [27,26], which deals with dynamic allocation but not pointers, and Ghica's and Levy's theses [4,6] (see also [7]) 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 [28] 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. Instead, we treat all programs in the language whether confined or not. 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 [24,11,31]. 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.

This paper is a revised version of [21] incorporating a more general framework of relation extensions which gives rise to a *subsumptive* reflexive graph structure and leads to a simpler presentation because naturality is subsumed by parametricity. (Cf. Definition 14 and Corollary 18.) It also allows more instances of program equivalences to be proved than with the original definitions. (Cf. Example 4 and Example 20.)

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 [8]:

```
{ 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: [1]

```
{ 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:

[1] The notation for pointers is borrowed from Pascal. For any data type δ, $\uparrow\delta$ is the type of pointers to δ-typed heap 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 modeled in the semantics. Using a possible world form of semantics as in [23,17,14], 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. 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). Now, in a world W, a procedure `p` denotes a parametrically polymorphic function of type:

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

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 $\uparrow\mathrm{int} \rightarrow \mathbf{com}$:

```
{ 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. [2] □

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 ρ between the location sets, 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

[2] 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.

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 [24,11,31], 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 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 [19,20], 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:

```
type listsig = { insert : int → com,
                 delete : int → com,
                 lookup : int × var bool → com}

List = class : listsig
         local var (↑node) head;
         init head := nil;
         meth { insert = λx. { head := new node(x,head) }
                delete = ...
                lookup = ... }
       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 for 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 { insert = λx. { s := ⟨x⟩·s }
                 delete = ...
                 lookup = ... }
        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: $W = \{l : \uparrow\texttt{node}\}$ 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:

$$s\ [R]\ s' \iff \mathrm{rep}(s, s(l), s'(l'))$$

$$\mathrm{rep}(s, p, \alpha) \iff (p = \mathrm{nil} \wedge \alpha = \langle\rangle) \vee \exists n, q, \beta. (s(p) = (n, q) \wedge \alpha = \langle n\rangle \cdot \beta \wedge \mathrm{rep}(s, q, \beta))$$

The relation requires that the linked list starting at location l in the state s stores the same sequence of values as in the (sequence-typed) location l' in the state s'. □

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.

A common method of ensuring correctness of data representations is by maintaining representation invariants. The preservation of invariants constitutes *unary* relational parametricity, whose theory can be presented along the same lines as that of binary relational parametricity. (In fact, relational parametricity works for relations of all arities.) However, unary relations can be reduced to binary relations as relations between a representation and itself. We use this reduction in the following example.

Example 4 Consider a class for *ordered* lists with *dummy headers* similar to that of lists above.

$$\texttt{type listsig} = \{\ \texttt{insert} : \texttt{int} \to \mathbf{com},\ \texttt{delete} : \texttt{int} \to \mathbf{com},\ \texttt{lookup} : \texttt{int} \times \mathbf{var}\ \texttt{bool} \to \mathbf{com}\}$$

The reader can easily envisage a class that implements the signature using linked lists. The fact that it maintains the "ordered" invariant can be represented as the following correspondence between a world $W = \{l : \uparrow\texttt{node}\}$ and itself. The partial bijection part of the correspondence is empty. The state relation part of the correspondence is a relation:

$$s[R]s' \iff s = s' \wedge \exists\alpha.\ (\mathrm{rep}(s, s(s(l)).2, \alpha) \wedge \mathrm{ordered}(\alpha))$$

$$\mathrm{ordered}(\alpha) \iff \forall\alpha_1, \alpha_2, x_1, x_2.\ \alpha = \alpha_1{\cdot}\langle x_1, x_2\rangle{\cdot}\alpha_2 \Longrightarrow x_1 \le x_2$$

Note that R relates a state to itself provided it contains an ordered list starting from the node following the one at $s(l)$, thus capturing the orderedness invariant.

Suppose we modify the `lookup` operation so that, instead of returning a boolean, it returns a pointer to the node where the element is stored. The modification is as follows.

```
lookup : int × var (↑node) → com
lookup = λ(x,p). { p := head;
                   while (p ≠ nil and p↑.1 < x) do
                       p := p↑.2;
                   if p↑.1 > x then p := nil
                 }
```

It is easy to fall into the trap of asserting that this modified implementation preserves the representation invariant. Clearly, there is nothing in the `lookup` procedure itself that destroys the orderedness invariant. However, since `lookup` returns a pointer to an internal node of the list, a client of the class would be able to modify the contents of the list cells, thereby breaking the invariant. This is the same problem of information leakage as in Example 2.

Our semantics blocks the conclusion that this implementation maintains the representation invariant. As mentioned in connection with Example 2, when representations are related by a correspondence (ρ, R), only ρ-related locations can be leaked. However, ρ-related locations cannot form part of the heap related by R (since the relation to be preserved is $EQ_\rho * R$ and the $*$ connective requires the two portions of the heap to be disjoint). As all the list cells are involved in satisfying the relation R, the ρ part of the correspondence is empty and, so, no pointers can be leaked.

As a minor twist on this example, consider modifying the `delete` operation to return a pointer to the deleted node:

```
delete : int × var (↑node) → com
delete = λ(x,p). { local var (↑node) pred := head;
                   p := head↑.2;
                   while (p ≠ nil and p↑.1 < x) do
                       { pred := p; p := p↑.2 }
                   if p↑.1 > x then p := nil
                   else { pred↑.2 := p↑.2; p↑.2 := nil }
                 }
```

If care is taken to reset the next pointer of the deleted node to be nil, it is permissible to return a pointer to the deleted node via the second argument. Even though the partial bijection part of the correspondence is initially empty (as explained in connection with `lookup`), the relational correspondence expected for the output state is some *extension* of the correspondence given for the input state, as indicated by the type of command meanings:

$$\forall_{X<:W}\ \mathrm{St}(X) \to \exists_{Y<:X}\ \mathrm{St}(Y).$$

Such an extension can add new locations to the partial bijection part of the correspondence as long as they are not used in the binary relation part. $\square$

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 name in Loc is often annotated with its type, as in l^δ, to indicate which Loc_δ it comes from. 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 $s \colon \prod_{l^\delta \in L} \mathrm{Val}(\delta)$ 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 $\mathrm{St}(W)$.

In our semantic model, we use a single form of state which we generally refer to as a "heap." It would have been possible to partition the state into a separate stack-state and a heap-state, but it would not have made any technical difference to the model. Using a single state (and thereby thinking of the stack storage as a subpart of the heap) does not seem to lose any accuracy.

Definition 5 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 $\sigma \supseteq \rho$. □

As mentioned in the context of Example 2, the purpose of renaming relations is to identify the visible locations. An extension of a renaming relation can make previously hidden locations to become visible, as well as making new locations visible. [3]

Since the pointers to visible 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

[3] In the predecessor of this paper [21], relation extension did not allow previously hidden locations to become visible. The present generalization allows more program equivalences. Cf. Examples 4 and 20.

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:

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

  (where $\rho \upharpoonright i$ denotes projection of i'th components).
- If H is a set of heaps, Δ_H denotes its diagonal relation:

$$s\ [\Delta_H]\ s' \iff s = s' \wedge s \in H$$

- 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:

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

  Note that $R * \mathrm{emp} = R$ for all heap relations R.

This $*$ connective above 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 6 A **relational correspondence** between location worlds is a pair $(\rho, R) : W \leftrightarrow W'$ (often written as $\rho R : W \leftrightarrow W'$ to avoid notational clutter) where

- ρ is a renaming relation between W and W' and
- R is a function mapping all extensions $\psi <: \rho$ to relations between heaps, such that, whenever $\psi_2 <: \psi_1 <: \rho$, $R(\psi_2) \supseteq R(\psi_1)$.

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

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 $R(\psi)$, 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 $R(\psi)$ that captures the data representation invariants. The relation $R(\psi)$ is parameterized by renamings ψ so that the information about visible locations mentioned in ψ can be incorporated in its formulation. The condition $R(\psi_1) \subseteq R(\psi_2)$ 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 [17,15] for local variables.

Definition 7 The *identity correspondence* for a world W is $I_W = (i_W, \overline{\mathrm{emp}}) : W \leftrightarrow W$, where i_W is the diagonal relation for W and $\overline{\mathrm{emp}}$ maps all extensions ψ of i_W to emp.

Fact 8 *Whenever $X <: W$, $I_X <: I_W$.*

PROOF. If $X <: W$, i.e., $X \supseteq W$, then $i_X \supseteq i_W$ which is the same thing as $i_X <: i_W$. Secondly, for any $\psi <: i_X$, $\overline{\mathrm{emp}}(\psi) = \mathrm{emp} = \mathrm{emp} * \mathrm{emp} = \overline{\mathrm{emp}}(\psi) * \mathrm{emp}$. Thus, $(i_X, \overline{\mathrm{emp}}) <: (i_W, \overline{\mathrm{emp}})$. $\square$

Whenever $X <: W$, there is an *embedding correspondence*

$$J_{X,W} = (j_{X,W}, \overline{\mathrm{emp}}) : X \leftrightarrow W$$

where $j_{X,W} = \{\, (l, l) \mid l \in W \,\}$ and $\overline{\mathrm{emp}}$ maps all extensions of $j_{X,W}$ to emp. Note that $J_{W,W}$ is the same as 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 9 $\mathrm{St}(I_W)$ *is the identity relation on* $\mathrm{St}(W)$.

PROOF. Let s, $s' \in \mathrm{St}(W)$. By definition, $\mathrm{dom}(s) = \mathrm{dom}(s') = W$.

$$\begin{aligned} s\,[\mathrm{St}(I_W)]\,s' &\iff s\,[EQ_{i_W} * \mathrm{emp}]\,s' \\ &\iff \forall l \in W.\, s(l) = s'(l) \\ &\iff s = s' \end{aligned}$$

□

To make these definitions concrete, we give a few examples:

Example 10 Consider the first program block from Example 1. Let W be some set of locations (the storage context for the block). The storage context for the body of the block is $X = W \uplus \{l_x\}$ where l_x is the location allocated for `x`.

Define $\sigma S : X \leftrightarrow X$ by $\sigma = i_W$ and $S(\psi) = \Delta_H$ where $H = \{\, s \mid \mathrm{dom}(s) = \{l_x\} \wedge s(l_x) = 0 \,\}$. Clearly $\sigma S <: I_W$. The preservation of this correspondence by the call `p()` implies that `x` continues to remain 0 after the call.

Similarly, for the third program block of Example 1, use $X = W \uplus \{l_x, l_0\}$, $\sigma = i_W$, $S(\psi) = \Delta_H$ where $H = \{\, s \mid \mathrm{dom}(s) = \{l_x, l_0\} \wedge s(l_x) = l_0 \wedge s(l_0) = 0 \,\}$. □

Example 11 Consider the list data structure from Example 3 but adapted now to contain pointers to integer cells instead of just integers. The type of nodes is given by $\mathtt{node} = {\uparrow}\mathrm{int} \times {\uparrow}\mathtt{node}$. For the worlds $W = \{l\colon {\uparrow}\mathtt{node}\}$ and $W' = \{l'\colon ({\uparrow}\mathrm{int})^*\}$, we define a correspondence $(\emptyset, R)$ where the relation function $R(\psi)$ is defined by:

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

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

We use the setting of reflexive graphs of categories [15,25,3] to explicate the categorical structure that we use. The key definitions are recalled in Appendix A.

Proposition 12 *There is a subsumptive reflexive graph* **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. The subsumption map sends each vertex morphism $X <: W$ to the embedding correspondence $J_{X,W}$.*

Let **Set** denote the subsumptive reflexive graph with sets and functions forming the vertex category and binary relations and relation-preserving squares forming the edge category. The subsumption map sends each function $f \colon A \to B$ to its graph $\langle f \rangle : A \leftrightarrow B$.

We will be working with the functor category $\mathbf{Set}^{\mathbf{World}^{\mathrm{op}}}$ whose objects are subsumptive reflexive graph-functors $F : \mathbf{World}^{\mathrm{op}} \to \mathbf{Set}$, and morphisms are parametric transformations [3]. (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 [15].)

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).

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 $T(W)$ from **World** to **Set**. The notation $\exists_{X<:W} F(X)$ similarly refers to the parametric colimit of $F \circ J^\circ$ (covariantly 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 $(F \Rightarrow G)(W) = \forall_{X<:W} F(X) \to G(X)$ [15,3].

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" [9,20].
- $\exists_{X<:W} F(X)$ 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 13 *Let* $\mathrm{GC}_W : \exists_{X<:W}\,\mathrm{St}(X) \to \exists_{X<:W}\,\mathrm{St}(X)$ *be defined by*

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

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* $\exists_{X<:W}\,\mathrm{St}(X)$*.*

PROOF. Denote $\mathrm{reach}_X(W, s)$ by L. Define a renaming relation $\rho\colon X \leftrightarrow L$ as $\{\,(l, l) \mid l \in L\,\}$. Let $R(\psi)$ relate any heap to the empty heap. Then it is easy to see that $s\;[\mathrm{St}(\rho R)]\;(s \upharpoonright L)$. Consequently, $\langle X, s\rangle \approx_W \langle L,\, s \upharpoonright L\rangle$. $\square$

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

A type operator F is a contravariant subsumptive functor if, whenever $V <: W$ is an extension of worlds, for every $d \in F(W)$ there is a unique value $d' \in F(V)$ such that $d'\;[F(J_{V,W})]\;d$ satisfying certain conditions. (See below.) We denote this unique value d' as $d{\uparrow}^V_W$. All the types of the imperative programming language are interpreted as contravariant functors of this form.

Definition 14 A type operator F is a *contravariant subsumptive functor* if it has an associated action on world extensions $V <: W$, i.e., for every $d \in F(W)$, there is a unique $d{\uparrow}^V_W \in F(V)$ such that $d{\uparrow}^V_W\;[F(J_{V,W})]\;d$, satisfying the following conditions:

- composition is preserved, i.e., whenever $U <: V <: W$, $(d{\uparrow}^V_W){\uparrow}^U_V = d{\uparrow}^U_W$, and
- the relation action is preserved, i.e., whenever $\pi P : V \leftrightarrow V'$ is an extension of $\rho R : W \leftrightarrow W'$,

$$d\;[F(\rho R)]\;d' \Longrightarrow d{\uparrow}^V_W\;[F(\pi P)]\;d'{\uparrow}^{V'}_{W'}$$

Similarly, there is a notion of *covariant subsumptive functors*, which have an associated covariant action $d{\uparrow}^W_V$ for world extensions $V <: W$.

If F and G are contravariant subsumptive functors then any parametric family of functions $\{p_X : F(X) \to G(X)\}_X$ of type $\forall_X F(X) \to G(X)$ is automatically a natural transformation, i.e., for all world extensions $V <: W$ and $d \in F(W)$,

$$(p_W(d)){\uparrow}^V_W = p_V(d{\uparrow}^V_W)$$

We call such a uniform family a *parametric transformation.*

Similarly, every family $\{p_X \in F(X) \to G(X)\}_{X<:W}$ in $\forall_{X<:W}\,F(X) \to G(X)$

satisfies the naturality condition with respect to W:

$$(p_X(d))\!\uparrow^Y_X = p_Y(d\!\uparrow^Y_X) \text{ for all extensions } Y <: X \text{ such that } X <: W.$$

These results follow from [18,3], where it is shown that naturality is subsumed under parametricity if we are only considering subsumptive reflexive graphs and subsumptive reflexive graph-functors.

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

- The type expression $T(W) = \forall_{X<:W} F(X)$ has an associated relation action $T(\rho R) : T(W) \leftrightarrow T(W')$ defined by

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

  We write this relation as $\forall_{\sigma S<:\rho R} F(\sigma S)$.
  The type operator $T(W)$ forms a contravariant subsumptive functor in W, independent of whether F is functorial. The action for world extensions $V <: W$ is given by $(\{p_X\}_{X<:W})\!\uparrow^V_W = \{p_X\}_{X<:V}$.
- The type expression $T(W) = \exists_{X<:W} F(X)$ has an associated relation action $T(\rho R) : T(W) \leftrightarrow T(W')$ defined by

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

  We write this relation as $\exists_{\sigma S<:\rho R} F(\sigma S)$.
  This type operator is a covariant subsumptive functor in W. If $V <: W$, we have the covariant action $(\langle\!|X, a|\!\rangle_V)\!\uparrow^W_V = \langle\!|X, a|\!\rangle_W$. Since any simulation relation with respect to V is also a simulation relation with respect to W, this action is well-defined.

Lemma 15 *The above subsumptive functors are well-defined.*

PROOF. Consider the type operator $T(W) = \forall_{X<:W} F(X)$. We first need to show that, whenever $V <: W$, $\{p_X\}_{X<:V}\ [T(J_{V,W})]\ \{p_X\}_{X<:W}$. This is is equivalent to showing $p_X\ [F(\sigma S)]\ p_{X'}$ for all $\sigma S : X \leftrightarrow X'$ such that $\sigma S <: J_{V,W}$. Since $J_{V,W} <: I_W$, we have $\sigma S <: I_W$. By definition of parametric families (with respect to W), we have that $p_X\ [F(\sigma S)]\ p_{X'}$.

Secondly, we show that $\{p_X\}_{X<:V}$ is the unique such family. Suppose q is another family satisfying $\{q_X\}_{X<:V}\ [T(J_{V,W})]\ \{p_X\}_{X<:W}$. Then, $q_X\ [F(\sigma S)]$

$p_{X'}$ for all $\sigma S : X \leftrightarrow X'$ extending $J_{V,W}$. For any $X <: V$, it is easy to see that $I_X <: J_{V,W}$. Hence, $q_X\ [F(I_X)]\ p_X$. Since $F(I_X) = \Delta_{F(X)}$, we have that $q_X = p_X$.

It is clear that composition is preserved. As for the preservation of relation action, suppose $\{p_X\}_{X<:W}$ and $\{p'_{X'}\}_{X'<:W'}$ are related by $\forall_{\tau T<:\rho R} F(\tau T)$. Then their restrictions $\{p_X\}_{X<:V}$ and $\{p'_{X'}\}_{X'<:V'}$ are related by $\forall_{\tau T<:\sigma S} F(\tau T)$ for any $\sigma S <: \rho R$.

For the type operator $T(W) = \exists_{X<:W} F(X)$, we first need to show that, whenever $V <: W$, $\langle\!| X, a |\!\rangle_V\ [T(J_{V,W})]\ \langle\!| X, a |\!\rangle_W$. But, this is immediate since $I_X <: J_{V,W}$ and $a\ [F(I_X)]\ a$. Next, we show that $\langle\!| X, a |\!\rangle_W$ is the unique abstract type satisfying $\langle\!| X, a |\!\rangle_V\ [T(J_{V,W})]\ \langle\!| X, a |\!\rangle_W$. Suppose $\langle\!| Y, b |\!\rangle_W$ satisfies $\langle\!| X, a |\!\rangle_V\ [T(J_{V,W})]\ \langle\!| Y, b |\!\rangle_W$, i.e., there exist $\langle X', a' \rangle \approx_V \langle X, a \rangle$ and $\langle Y', b' \rangle \approx_W \langle Y, b \rangle$ and a correspondence $\sigma S : X' \leftrightarrow Y'$ such that $\sigma S <: J_{V,W}$ and $a'\ [F(\sigma S)]\ b'$. Since $J_{V,W} <: I_W$, we have $\sigma S <: I_W$, which implies $\langle X', a' \rangle \approx_W \langle Y', b' \rangle$. Hence, $\langle\!| X, a |\!\rangle_W = \langle\!| Y, b |\!\rangle_W$. It is straightforward to show that composition and the relation action are preserved. $\square$

Notation We use a convenient notation for polymorphic families borrowed from the polymorphic lambda calculus [22]. 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:

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

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. We use two phrase types: "expressions" read the state to produce values whereas "commands" carry out state changes. A term has a typing of the form $x_1 : \nu_1, \ldots, x_n : \nu_n \vdash M : \theta$, with value types on the left and phrase type on the right. This asymmetry between value and phrase types is typical of call-by-value programming languages [6]. Unlike in call-by-name Algol-like languages [23], expressions in our language can yield values of *all* value types, not only those of data types. This represents additional expressive power.

$$\frac{}{\Gamma, x : \nu \vdash x : \mathbf{exp}\ \nu}$$

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{com}} \qquad \frac{\Gamma \vdash C_1 : \mathbf{com} \quad \Gamma \vdash C_2 : \mathbf{com}}{\Gamma \vdash C_1; C_2 : \mathbf{com}}$$

$$\frac{\Gamma, x : \mathbf{var}\ \delta \vdash C : \mathbf{com}}{\Gamma \vdash \{\mathbf{local\ var}\ \delta\ x; C\} : \mathbf{com}}$$

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

$$\frac{\Gamma \vdash E_i : \mathbf{exp}\ \nu_i,\ i = 1, \ldots, n}{\Gamma \vdash (E_1, \ldots, E_n) : \mathbf{exp}\ (\nu_1 \times \cdots \times \nu_n)} \qquad \frac{\Gamma \vdash E : \mathbf{exp}(\nu_1 \times \cdots \times \nu_n)}{\Gamma \vdash E.i : \mathbf{exp}\ \nu_i}$$

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

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

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

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

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

Fig. 1. Type Syntax of Terms

The term syntax for our language is given in Figure 1. We use a sample of command forms. Other forms can be accommodated in a similar fashion. The notation for classes is borrowed from [19,20]. Objects instantiated from classes are bound to *local identifiers* via the **local** $K\ x$ declaration. We do not consider pointer-addressed class instances in this paper.

The types are interpreted as contravariant subsumptive functors from **World** to **Set**. The interpretation comes in two parts: The set part $[\![\tau]\!]$ maps worlds to sets and the relation part $\langle\!\langle\tau\rangle\!\rangle$ maps correspondences $\rho R : W \leftrightarrow W'$ to relations $[\![\tau]\!](W) \leftrightarrow [\![\tau]\!](W')$. The interpretation uniquely determines a contravariant action on world extensions $V <: W$ as functions mapping values

$d \in [\![\tau]\!](W)$ to $d{\uparrow}_W^V \in [\![\tau]\!](V)$.

Data and Value Types:

$$
\begin{aligned}
[\![\mathrm{int}]\!](W) &= \mathit{Int} \\
[\![{\uparrow}\delta]\!](W) &= (\mathrm{Loc}_\delta \cap W) + \{\mathrm{nil}\} \\
[\![\delta_1 \times \cdots \times \delta_n]\!](W) &= [\![\delta_1]\!](W) \times \cdots \times [\![\delta_n]\!](W) \\
[\![\mathbf{var}\,\delta]\!](W) &= [\![\delta \to \mathbf{com}]\!](W) \times [\![\mathbf{exp}\,\delta]\!](W) \\
[\![\nu_1 \times \cdots \times \nu_n]\!](W) &= [\![\nu_1]\!](W) \times \cdots \times [\![\nu_n]\!](W) \\
[\![\nu \to \theta]\!](W) &= \forall_{V<:W}\, [\![\nu]\!](V) \to [\![\theta]\!](V) \\
[\![\mathbf{cls}\,\nu]\!](W) &= \forall_{V<:W}\, \exists_{Z<:V}\, [\![\mathbf{exp}\,\nu]\!](Z) \times \\
&\qquad\qquad [\mathrm{St}(V) \to (\exists_{Y<:Z}\, \mathrm{St}(Y)) + \{\varepsilon\}]
\end{aligned}
$$

Phrase Types:

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

(We use the convention that the scope of a quantifier extends as far to the right as possible.) The position of the type quantifications $\forall$ and $\exists$ in the type interpretations has been recognized in earlier work [27,4,7]. 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 world W represents the static context of the command (similar to the "static chain" locations in a typical implementation), X represents the initial dynamic context (the "dynamic chain" locations as well as heap locations) and Y represents the final dynamic context. The parametricity interpretation of the type quantifiers means that the command does not have direct access to the extra locations in its dynamic context and the successor commands will not have direct access to the locations allocated by the present command. (However, access may be available via the pointers stored in the static context.) The symbol ε is used to denote a special error value that results from dereferencing a nil pointer.

Variables are interpreted in the object-oriented style as pairs of "put" and "get" methods [23]. Indeed, if $l \in W$ is a δ-typed location, we can map it to a pair of methods $\mathit{var}^\delta_W(l) = (\mathit{put}^\delta_W(l),\, \mathit{get}^\delta_W(l))$ defined as follows:

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

Classes for a signature type ν specify an abstract type with a hidden representation Z for the objects of the class, a method suite of type $\mathbf{exp}\,\nu$ and an initialization operation that initializes the local representation Z as well as any heap variables created during the initialization.

The relation part of the interpretation is given with respect to a relational correspondence $\rho R : W \leftrightarrow W'$, as follows:

Data and Value Types:

$$
\begin{aligned}
\langle\!\langle \mathrm{int} \rangle\!\rangle(\rho R) &= \Delta_{Int} \\
\langle\!\langle \uparrow\delta \rangle\!\rangle(\rho R) &= \rho + \Delta_{\{\mathrm{nil}\}} \\
\langle\!\langle \delta_1 \times \cdots \times \delta_n \rangle\!\rangle(\rho R) &= \langle\!\langle \delta_1 \rangle\!\rangle(\rho R) \times \cdots \times \langle\!\langle \delta_n \rangle\!\rangle(\rho R) \\
\langle\!\langle \mathbf{var}\,\delta \rangle\!\rangle(\rho R) &= \langle\!\langle \delta \to \mathbf{com} \rangle\!\rangle(\rho R) \times \langle\!\langle \mathbf{exp}\,\delta \rangle\!\rangle(\rho R) \\
\langle\!\langle \nu_1 \times \cdots \times \nu_n \rangle\!\rangle(\rho R) &= \langle\!\langle \nu_1 \rangle\!\rangle(\rho R) \times \cdots \times \langle\!\langle \nu_n \rangle\!\rangle(\rho R) \\
\langle\!\langle \nu \to \theta \rangle\!\rangle(\rho R) &= \forall_{\pi P <: \rho R}\, \langle\!\langle \nu \rangle\!\rangle(\pi P) \to \langle\!\langle \theta \rangle\!\rangle(\pi P) \\
\langle\!\langle \mathbf{cls}\,\nu \rangle\!\rangle(\rho R) &= \forall_{\pi P <: \rho R}\, \exists_{\sigma S <: \pi P} \\
&\qquad\quad \langle\!\langle \mathbf{exp}\,\nu \rangle\!\rangle(\sigma S) \times \\
&\qquad\quad [\mathrm{St}(\pi P) \to (\exists_{\tau T <: \sigma S}\, \mathrm{St}(\tau T)) + \Delta_{\{\varepsilon\}}]
\end{aligned}
$$

Phrase Types:

$$
\begin{aligned}
\langle\!\langle \mathbf{exp}\,\nu \rangle\!\rangle(\rho R) &= \forall_{\sigma S <: \rho R}\, \mathrm{St}(\sigma S) \to \langle\!\langle \nu \rangle\!\rangle(\sigma S) + \Delta_{\{\varepsilon\}} \\
\langle\!\langle \mathbf{com} \rangle\!\rangle(\rho R) &= \forall_{\sigma S <: \rho R}\, \mathrm{St}(\sigma S) \to (\exists_{\tau T <: \sigma S}\, \mathrm{St}(\tau T)) + \Delta_{\{\varepsilon\}}
\end{aligned}
$$

Note that $d\;[\langle\!\langle \delta \rangle\!\rangle \rho R]\; d'$ is equivalent to $d \equiv_\rho d'$.

Theorem 16 *All the type interpretations are contravariant subsumptive functors.*

PROOF. First we need to verify that the interpretation preserves identity relations, i.e., $\langle\!\langle \tau \rangle\!\rangle(I_W) = \Delta_{[\![\tau]\!](W)}$. This is easily done by induction on the structure of τ. We show a few sample cases.

- For $\uparrow\delta$, $\langle\!\langle \uparrow\delta \rangle\!\rangle(I_W) = i_W + \Delta_{\{\mathrm{nil}\}} = \Delta_{[\![\uparrow\delta]\!](W)}$.
- For $\nu \to \theta$, $\langle\!\langle \nu \to \theta \rangle\!\rangle(I_W) = \forall_{\pi P <: I_W}\, \langle\!\langle \nu \rangle\!\rangle(\pi P) \to \langle\!\langle \theta \rangle\!\rangle(\pi P)$. Suppose p and p' are two families related by this relation. Then, since $I_V <: I_W$ (for every $V <: W$), we must have $p_V\; [\langle\!\langle \nu \rangle\!\rangle(I_V) \to \langle\!\langle \theta \rangle\!\rangle(I_V)]\; p'_V$. Appealing to the inductive hypothesis for ν and θ, we infer that $p_V = p'_V$. Thus, $\langle\!\langle \nu \to \theta \rangle\!\rangle(I_W) \subseteq \Delta_{[\![\nu \to \theta]\!](W)}$. In the reverse direction, let $p \in [\![\nu \to \theta]\!](W)$, which is related to itself by the diagonal relation. The definition of parametric families means that it is related to itself by the relation $\forall_{\pi P <: I_W}\, \langle\!\langle \nu \rangle\!\rangle(\pi P) \to \langle\!\langle \theta \rangle\!\rangle(\pi P)$.

Secondly we need to verify that there is a proper contravariant action on world extensions which is included in $\langle\langle\tau\rangle\rangle(J_{V,W})$. This proceeds by induction on the structure of τ.

- For `int`, the action is clearly the identity: $d{\uparrow}^V_W = d$.
- For $\uparrow\delta$, the action is injection: $d{\uparrow}^V_W = d$, which is the only function included in $\langle\langle\uparrow\delta\rangle\rangle(J_{V,W}) = j_{V,W} + \Delta_{\{\mathrm{nil}\}}$.
- For $\delta_1 \times \cdots \times \delta_n$, $\mathbf{var}\ \delta$ and $\nu_1 \times \cdots \times \nu_n$, we note that $\times$ preserves subsumptiveness. The associated action is $(d_1, \ldots, d_n){\uparrow}^V_W = (d_1{\uparrow}^V_W, \ldots, d_n{\uparrow}^V_W)$.
- For all other type constructors, the relation interpretation has $\forall_{\sigma S <: \rho R}$ at the outermost level, and such type operators are subsumptive by Lemma 15. $\square$

The semantics of a term with typing $x_1\colon \nu_1, \ldots, x_n\colon \nu_n \vdash M : \theta$ is a parametric transformation of type $\forall_W\ [\![\nu_1]\!](W) \times \cdots \times [\![\nu_n]\!](W) \to [\![\theta]\!](W)$. (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 Figure 2. The combinators *unit* and *bind* are similar to monad combinators [10,29]: $\mathit{unit}_W\ d$ is an expression that simply returns the value d in every state; $\mathit{bind}_W\,(e, f)$ evaluates the expression e, feeds the resulting value to the function f and evaluates the result of application. The function f may either yield an expression, in which case $\mathit{bind}_W\ (e, f)$ is an expression, or it may yield a command, in which case $\mathit{bind}_W\ (e, f)$ is a command. We also use a similar combinator for binary functions f. The *seq* combinator represents the sequential composition of commands.

We also assume that there is a family of functions $\mathit{newloc}_\delta(X)$ that give, for each world X, a δ-typed location that is not in X. A constant init_δ specifies the initial value for each δ-typed location.

The interpretation of the various term forms is given in Fig. 3. These definitions are expressed using the operations:

$$\mathit{alloc}^\delta_W : [\![\mathbf{var}(\uparrow\delta) \to \mathbf{com}]\!](W)$$

$$\mathit{alloc}^\delta_W\ [V]\ (p, g)\ [X]\ s = \mathit{hide}_{X^+ <: X}(p\ [X^+]\ l\ [X^+]\ (s * [l {\to} \mathrm{init}_\delta]))$$
$$\text{where } l = \mathit{newloc}^\delta(X) \text{ and } X^+ = X \uplus \{l\}$$

$$\mathit{deref}^\delta_W : [\![\uparrow\delta \to \mathbf{exp}\ (\mathbf{var}\ \delta)]\!](W)$$

$$\mathit{deref}^\delta_W\ [V]\ l\ [X]\ s = \text{if } l \neq \mathrm{nil} \text{ then } \mathit{var}^\delta_X(l) \text{ else } \varepsilon$$

We explain the general form of the semantic definition by looking at the local variable declaration (**local var** $\delta\ x$; C) and dynamic allocation ($V := \mathbf{new}\,\delta$).

$$unit_W^\nu : [\![\nu]\!](W) \to [\![\mathbf{exp}\,\nu]\!](W)$$
$$unit_W\; d = \Lambda X{<:}W.\, \lambda s.\, d \uparrow_W^X$$

$$bind_W^{\nu,\theta} : [\![\mathbf{exp}\,\nu]\!](W) \times [\![\nu \to \theta]\!](W) \to [\![\theta]\!](W)$$
$$bind_W\,(e, f) = \Lambda X{<:}W.\, \lambda s.\, \text{let } d = e\,[X]\,s \;\text{ in if } d = \varepsilon \text{ then } \varepsilon \text{ else } f\,[X]\,d\,[X]\,s$$

$$bind_W^{\nu_1,\nu_2,\theta} : [\![\mathbf{exp}\,\nu_1]\!](W) \times [\![\mathbf{exp}\,\nu_2]\!](W) \times [\![\nu_1 \times \nu_2 \to \theta]\!](W) \to [\![\theta]\!](W)$$
$$\begin{aligned} bind_W\,(e_1, e_2, f) = \Lambda X{<:}W.\, \lambda s.\, &\text{let } d_1 = e_1\,[X]\,s,\; d_2 = e_2\,[X]\,s \\ &\text{in if } d_1 = \varepsilon \vee d_2 = \varepsilon \text{ then } \varepsilon \\ &\text{else } f\,[X]\,(d_1, d_2)\,[X]\,s \end{aligned}$$

$$seq_W : [\![\mathbf{com}]\!](W) \times [\![\mathbf{com}]\!](W) \to [\![\mathbf{com}]\!](W)$$
$$seq_W\,(c, c') = \Lambda X{<:}W.\, \lambda s. \begin{cases} \varepsilon & \text{if } c\,[X]\,s = \varepsilon \\ hide_{Y<:X}(c'\,[Y]\,s') & \text{if } c\,[X]\,s = \langle\!|Y, s'|\!\rangle_X \end{cases}$$

$$hide_{Y<:X} : [(\exists_{Z<:Y}\,\mathrm{St}(Z)) + \{\varepsilon\}] \to [(\exists_{Z<:X}\,\mathrm{St}(Z)) + \{\varepsilon\}]$$
$$hide_{Y<:X}\; r = \text{case } r \text{ of } \langle\!|Z, s|\!\rangle_Y \Rightarrow \langle\!|Z, s|\!\rangle_X \mid \varepsilon \Rightarrow \varepsilon$$

Fig. 2. Semantic Combinators

For the local variable construct, we are given a dynamic context $X <: W$ and a state s (in this context). The interpretation finds a new location l for the local variable, builds the extended context $X^+ = X \uplus \{l\}$ and extended state $s * [l{\to}\mathrm{init}_\delta]$ where the new location is initialized. The body of the block, C, is interpreted in the extended context starting with the extended state. The resulting state is then cut back to X by hiding the location l (representing the deallocation of the local variable). Any pointer value stored in the location l will thus be lost turning its heap variable into a potential garbage location.

For the dynamic allocation construct, the interpretation is via the $alloc_W$ operation. The context W is somewhat redundant since *alloc* is parametric in W. It is appropriate to think of V as the static context and X as the dynamic context. The interpretation is then quite similar to that of the local variable construct, the only difference being that the identity of the newly

$$[\![x]\!]_W\eta = unit_W(\eta(x))$$

$$[\![\mathbf{skip}]\!]_W\eta = \Lambda X{<:}W.\,\lambda s.\,\langle\!| X, s |\!\rangle_X$$

$$[\![C_1; C_2]\!]_W\eta = seq_W([\![C_1]\!]_W\eta, [\![C_2]\!]_W\eta)$$

$$\begin{aligned}
&[\![\{\mathbf{local\ var}\ \delta\ x; C\}]\!]_W\eta = \\
&\quad \Lambda X{<:}W.\,\lambda s.\,hide_{X^+<:X}\,([\![C]\!]_{X^+}\,(\eta{\uparrow}_W^{X^+}\,[x{\to}var^\delta_{X^+}(l)])\,[X^+]\,s^+) \\
&\qquad\qquad\qquad \text{where } l = newloc_\delta(X),\ X^+ = X \uplus \{l\} \text{ and} \\
&\qquad\qquad\qquad s^+ = s * [l{\to}\mathrm{init}_\delta]
\end{aligned}$$

$$[\![\mathbf{read}\ V]\!]_W\eta = bind_W\ ([\![V]\!]_W\eta,\ \Lambda X{<:}W.\,\lambda(p,g).\,g)$$

$$[\![V := E]\!]_W\eta = bind_W\ ([\![V]\!]_W\eta,\ [\![E]\!]_W\eta,\ \Lambda X{<:}W.\,\lambda((p,g),k).\,p[X]k)$$

$$[\![E{\uparrow}]\!]_W\eta = bind_W\ ([\![E]\!]_W\eta,\ deref^\delta_W)$$

$$[\![V := \mathbf{new}\ \delta]\!]_W\eta = bind_W\ ([\![V]\!]_W\eta,\ alloc^\delta_W)$$

$$[\![\lambda x.\,M]\!]_W\eta = unit_W(\Lambda V{<:}W.\,\lambda d.\,[\![M]\!]_V(\eta{\uparrow}_W^V[x{\to}d]))$$

$$[\![M(N)]\!]_W\eta = bind_W\ ([\![M]\!]_W\eta,\ [\![N]\!]_W\eta,\ \Lambda V{<:}W.\,\lambda(f,d).\ f[V](d))$$

$$\begin{aligned}
&[\![\mathbf{class} : \nu\ \mathbf{local\ var}\ \delta\ x\ \mathbf{init}\ A\ \mathbf{meth}\ M\ \mathbf{end}]\!]_W\eta = \\
&\quad unit_W(\Lambda V{<:}W.\,\langle\!| V^+, [\![M]\!]_{V^+}\eta^+,\ \lambda s.\,[\![A]\!]_{V^+}(\eta^+)(s * [l{\to}\mathrm{init}_\delta]) |\!\rangle_V) \\
&\qquad \text{where } l = newloc_\delta(V),\ V^+ = V \uplus \{l\}, \text{ and} \\
&\qquad \eta^+ = \eta{\uparrow}_W^{V^+}[x{\to}var^\delta_{V^+}(l)]
\end{aligned}$$

$$\begin{aligned}
&[\![\{\mathbf{local}\ K\ x;\ C\}]\!]_W\eta = \\
&\quad bind_W\ ([\![K]\!]_W\eta,\ \Lambda X{<:}W.\,\lambda k.\,\Lambda Y{<:}X.\,\lambda s. \\
&\qquad\qquad\qquad \text{let } \langle\!| Z, m, i |\!\rangle_Y = k[Y] \\
&\qquad\qquad\qquad \text{in if } \langle\!| Z', s' |\!\rangle_Z = i(s) \wedge m[Z']s' \neq \varepsilon \\
&\qquad\qquad\qquad\qquad \text{then } hide_{Z'<:Y}([\![C]\!]_{Z'}(\eta{\uparrow}_W^{Z'}[x{\to}m[Z']s'])\,s') \\
&\qquad\qquad\qquad\qquad \text{else } \varepsilon)
\end{aligned}$$

Fig. 3. Semantics of Terms

allocated location l is stored in a variable. Hiding the new location l does *not* signify deallocation in this case, since a pointer to it has been stored within the context X.

How do heap variables get deallocated? Indeed, a cursory reading of the interpretation of the **com** type might suggest that contexts only get bigger, never smaller. However, this is not actually the case. The result type of a command is of the form $(\exists_{Y<:X}\mathrm{St}(Y)) + \{\varepsilon\}$, which signifies that all the new locations allocated by a command get hidden. Unless pointers to them are stored within the initial context X, the new locations are free to be deallocated. (Cf. Lemma 13.) For example, the following equivalence holds in the semantics:

$$x := \mathbf{new}\,\delta;\; x := \mathtt{nil} \quad \cong \quad x := \mathtt{nil}$$

Evaluating the left hand side in a world $W = \{l_0\}$, environment $\eta = [x \to var_W^{\uparrow\delta}(l_0)]$, dynamic context $X <: W$ and state s, the resulting state is:

$$\langle\!| X \uplus \{l\},\ s[l_0 \to \mathtt{nil}] * [l \to \mathrm{init}_\delta] |\!\rangle_X$$

which is equal to $\langle\!| X,\ s[l_0 \to \mathtt{nil}] |\!\rangle_X$. Note that this is also the resulting state of the right hand side with the given parameters.

The class construct similarly allocates a location for the local variable in the dynamic context. However, it packages its result as a structure $\langle\!| V^+, m, i |\!\rangle_V$ where V^+ is the extended context, m the interpretation of the method suite in the context V^+ and i the initialization operation. Such a structure is unpacked when a class is instantiated using the **local** $K\ x$ declaration.

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

(1) for all worlds W *and all environments* $\eta \in [\![\Gamma]\!](W)$*,* $[\![M]\!]_W \eta \in [\![\theta]\!](W)$*, and*
(2) for all $\rho R : W \leftrightarrow W'$*, and all related environments* $\eta [\langle\!\langle \Gamma \rangle\!\rangle(\rho R)] \eta'$*,*
$[\![M]\!]_W \eta\ [\langle\!\langle \theta \rangle\!\rangle(\rho R)]\ [\![M]\!]_{W'} \eta'$*.*

PROOF. [Sketch] The proof is by induction on the structure of M. All cases follow easily from the definition once all operators used in the semantics, such as *bind*, *deref* and *alloc*, are proved to be parametric transformations. We show the parametricity of *alloc*, which illustrates the subtleties of the definition of correspondences. Let ρR be a correspondence between W and W'. We need to show that for all extensions $\pi P : V \leftrightarrow V'$ of ρR, the functions $alloc^{\delta}_{W}[V]$ and $alloc^{\delta}_{W'}[V']$ are related by

$$[\langle\!\langle \mathbf{var}(\uparrow\delta)\rangle\!\rangle(\pi P) \to \langle\!\langle \mathbf{com}\rangle\!\rangle(\pi P)].$$

Let (p, g) and (p', g') be variables related by $\langle\!\langle \mathbf{var}(\uparrow\delta)\rangle\!\rangle(\pi P)$, let $\sigma S : X \leftrightarrow X'$ be a relational correspondence extending πP, and let s and s' be states related by $\mathrm{St}(\sigma S)$. From the definition of $\langle\!\langle \mathbf{var}(\uparrow\delta)\rangle\!\rangle$, the put methods p and p' are related by $\langle\!\langle \uparrow\delta \to \mathbf{com}\rangle\!\rangle(\pi P)$. Let l and l' be the locations chosen by *alloc* in X and X'. We define a correspondence $\tau T : X \uplus \{l\} \leftrightarrow X' \uplus \{l'\}$ such that

$$\tau T <: \sigma S, \quad l\ [\tau]\ l', \quad \text{and} \quad s{*}[l{\to}\mathrm{init}_{\delta}]\ [\mathrm{St}(\tau T)]\ s{*}[l'{\to}\mathrm{init}_{\delta}].$$

Then, since $p[X{\uplus}\{l\}]\ [\langle\!\langle\uparrow\delta\rangle\!\rangle(\tau T) \to \langle\!\langle \mathbf{com}\rangle\!\rangle(\tau T)]\ p'[X'{\uplus}\{l'\}]$, the parametricity of *hide* shows that $alloc^{\delta}_{W}[V](p, g)[X]s$ and $alloc^{\delta}_{W'}[V'](p', g')[X']s'$ are related. Such a correspondence is given by

$$\tau = \sigma \uplus \{(l, l')\} \quad \text{and} \quad T(\psi) = S(\psi) \ \text{ for all } \psi <: \tau.$$

Note that the states $s * [l{\to}\mathrm{init}_{\delta}]$ and $s' * [l'{\to}\mathrm{init}_{\delta}]$ are related by $\mathrm{St}(\tau T)$: From definition, we see that they are related by $EQ_{\tau} * S(\sigma)$, and, since $S(\sigma) \subseteq S(\tau)$ by the condition for correspondences, they are related by $EQ_{\tau} * S(\tau)$ which is the same as $EQ_{\tau} * T(\tau)$. □

Corollary 18 *The meaning of every term $[\![\Gamma \vdash M{:}\,\theta]\!]$ is natural, i.e., for all extensions $V <: W$ and all $\eta \in [\![\Gamma]\!](W)$, $([\![M]\!]_W \eta){\uparrow}^V_W = [\![M]\!]_V(\eta{\uparrow}^V_W)$.*

PROOF. Since $[\![\Gamma]\!]$ and $[\![\theta]\!]$ are subsumptive functors, a parametric transformation $[\![\Gamma \vdash M{:}\,\theta]\!]$ is natural by Proposition 21. □

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

$$\begin{aligned} &hide_{Z<:V}([\![M]\!]_Z(\eta{\uparrow}^Z_W[x{\to}m[Z]s_1])[Z]s_1) = \\ &\qquad hide_{Z'<:V}([\![M]\!]_{Z'}(\eta{\uparrow}^{Z'}_W[x{\to}m'[Z']s'_1])[Z']s'_1) \end{aligned}$$

for all $\eta \in [\![\Gamma]\!]_W$ and $V <: W$, which follows from the abstraction theorem.

The separation logic for reasoning about heap data structures [24,11,31] 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 original state. This property is satisfied by our semantics. Say that a command $c \in [\![\mathbf{com}]\!](W)$ is *safe* for world $X <: W$ and state $s \in \mathrm{St}(X)$ if $c[X](s) \neq \varepsilon$.

Theorem 19 *Let $c \in [\![\mathbf{com}]\!](W)$ 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 $c\,[X \uplus Z]\,(s * t) = \langle\!|Y \uplus Z, s' * t|\!\rangle_{X \uplus Z}$.

PROOF. Let $jR : X \uplus Z \leftrightarrow X$ be a relational correspondence defined by:

$$l\,[j]\,l' \iff l = l' \in X \quad s_0\,[R(\psi)]\,s_0' \iff s_0 = t \,\wedge\, \mathrm{dom}(s_0') = \emptyset.$$

Since $jR <: I_W$, $s * t\,[\mathrm{St}(jR)]\,s$, and $c\,[X]\,s \neq \varepsilon$, we have that

$$c\,[X \uplus Z]\,(s * t)\,[\exists_{\sigma S <: jR} \mathrm{St}(\sigma S)]\,c\,[X]\,s.$$

Thus, there exist $\langle\!|V, s_0|\!\rangle_{X \uplus Z} = c\,[X \uplus Z]\,(s * t)$, $\langle\!|V', s_0'|\!\rangle_X = c\,[X]\,s$, and $\pi P : V \leftrightarrow V'$ such that $s_0\,[\mathrm{St}(\pi P)]\,s_0'$ and $\pi P <: jR$. Note that $c[X \uplus Z](s*t) \neq \varepsilon$; thus, c is safe for $X \uplus Z$ and $s * t$. Since $S(\sigma) = R(\sigma) * P$ for some P, there are heaps s_1, s_2, s_1', s_2' such that

$$s_0 = s_1 * t * s_2, \quad s_0' = s_1' * s_2', \quad \text{and} \quad s_2\,[EQ_\sigma]\,s_2'.$$

Pick a world Y and a renaming relation $\tau : Y \leftrightarrow (\mathrm{dom}(s_2) - X)$ such that τ is a type-respecting isomorphism between Y and $(\mathrm{dom}(s_2) - X)$, and Y is disjoint from $V \cup V'$. Let $\sigma_0 : X \uplus Y \leftrightarrow \mathrm{dom}(s_2)$ be a renaming relation defined by

$$l\,[\sigma_0]\,l' \iff l = l' \in X \,\vee\, l\,[\tau]\,l'$$

and let s_3 be a state in $\mathrm{St}(X \uplus Y)$ such that $s_3\,[EQ_{\sigma_1}]\,s_2$. We will show that

$$\langle V, s_1 * s_2 * t\rangle_{X \uplus Z} \approx \langle X \uplus Y \uplus Z, s_3 * t\rangle_{X \uplus Z} \quad \text{and}$$
$$\langle V', s_1' * s_2'\rangle_X \approx \langle X \uplus Y, s_3\rangle_X.$$

This implies the second condition, because $s_1 * s_2 * t = s_0$ and $s_1' * s_2' = s_0'$. Let $(\sigma_1, R_1) : V \leftrightarrow X \uplus Y \uplus Z$, and $(\sigma_1', R_1') : V' \leftrightarrow X \uplus Y$ be relational correspondences defined by:

$$\begin{aligned} l\ [\sigma_1]\ l' &\iff l'\ [\sigma_0]\ l \ \vee\ l = l' \in Z \\ t_0\ [R_1]\ t_0' &\iff \mathrm{dom}(t_0') = \emptyset \end{aligned}$$

$$\begin{aligned} l\ [\sigma_1']\ l' &\iff l = l' \in X \ \vee\ \exists l_0.\ l_0\ [\tau]\ l' \wedge l_0\ [\sigma]\ l \\ t_0\ [R_1']\ t_0' &\iff \mathrm{dom}(t_0') = \emptyset \end{aligned}$$

Then, $(\sigma_1, R_1) <: I_{X \uplus Z}$, and $(\sigma_1', R_1') <: I_X$. Moreover, $s_1 * s_2 * t$ and $s_3 * t$ are related by $\mathrm{St}(\sigma_1, R_1)$, and $s_1' * s_2'$ and s_3 are related by $\mathrm{St}(\sigma_1', R_1')$. Therefore, we have the required equivalence. $\Box$

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.

5.1 Examples of Reasoning

We illustrate the semantic definitions by returning to the examples from Section 2. Since we are not treating divergence in this paper, we interpret the command `diverge` as giving an error:

$$[\![\texttt{diverge}]\!]_W \eta = \Lambda X{<:}W.\ \lambda s.\ \epsilon$$

Consider the third program block from Example 1:

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

Let W be a world denoting the static context, l_0 a location that is not in W, $W^+ = W \uplus \{l_0\}$, η an environment for W, and $\eta^+ = \eta{\uparrow}_W^{W^+}[\mathtt{x} {\to} \mathit{var}_{W^+}(l_0)]$.

$$
\begin{aligned}
[\![\mathtt{x} := \textbf{new}\ \mathtt{int}]\!]_{W^+}\eta^+ &= \Lambda X{<:}W^+.\ \lambda s.\ \mathit{alloc}^{\mathtt{int}}_{W^+}[X](\mathit{var}_{W^+}(l_0){\uparrow}^X_{W^+})[X]s \\
&= \Lambda X{<:}W^+.\ \lambda s.\ \langle\!| X^+, s[l_0{\to}l_1] * [l_1{\to}\text{init}] |\!\rangle_X \\
&\qquad \text{where } l_1 = \mathit{newloc}(X) \text{ and } X^+ = X \uplus \{l_1\}
\end{aligned}
$$

$$
[\![\mathtt{x}{\uparrow} := \mathtt{0}]\!]_{W^+}\eta^+ = \Lambda X{<:}W^+.\ \lambda s.\ \begin{cases} \langle\!| X, s[s(l_0){\to}0] |\!\rangle_X, & \text{if } s(l_0) \neq \text{nil} \\ \varepsilon, & \text{otherwise} \end{cases}
$$

$$
[\![\mathtt{p}()]\!]_{W^+}\eta^+ = \Lambda X{<:}W^+.\ \lambda s.\ \eta^+(\mathtt{p})[X](*)[X](s)
$$

$$
[\![\textbf{if}\ (\mathtt{x}{\uparrow} = \mathtt{0})\ \textbf{then}\ \mathtt{diverge}]\!]_{W^+}\eta^+ =
$$
$$
\Lambda X{<:}W^+.\ \lambda s.\ \begin{cases} \varepsilon, & \text{if } (s(l_0) = \text{nil}) \vee (s(s(l_0)) = 0) \\ \langle\!| X, s |\!\rangle_X, & \text{otherwise} \end{cases}
$$

Hence,

$$
\begin{aligned}
&[\![\mathtt{x} := \textbf{new}\ \mathtt{int}; \mathtt{x}{\uparrow} := \mathtt{0}; \mathtt{p}(); \textbf{if}\ (\mathtt{x}{\uparrow} = \mathtt{0})\ \textbf{then}\ \mathtt{diverge}]\!]_{W^+}\eta^+ = \\
&\Lambda X{<:}W^+.\ \lambda s.\ \text{case}\ \Big(\eta^+(\mathtt{p})[X^+](*)[X^+]\,(s[l_0{\to}l_1] * [l_1{\to}0])\Big)\ \text{of} \\
&\qquad \varepsilon \Rightarrow \varepsilon \\
&\qquad |\ \langle\!| Y, t |\!\rangle_{X^+} \Rightarrow \text{if } (t(l_0) = \text{nil} \vee t(t(l_0)) = 0) \\
&\qquad\qquad \text{then } \varepsilon \\
&\qquad\qquad \text{else } \langle\!| Y, t |\!\rangle_X
\end{aligned}
$$

We would like to show that this function has the value ε for all arguments X and s. Note that $\eta^+(\mathtt{p}) = \eta(\mathtt{p}){\uparrow}_W^{W^+}$ and $\eta^+(\mathtt{p})[X^+] = \eta(\mathtt{p})[X^+]$.

The procedure $\mathtt{p}$ is of type **unit** $\to$ **com** where **unit** is the nullary product type with the interpretation $[\![\textbf{unit}]\!](W) = 1 = \{*\}$, a singleton set.

$$
\begin{aligned}
[\![\textbf{unit} \to \textbf{com}]\!](W) &= \forall_{V<:W}\, 1 \to [\![\textbf{com}]\!](V) \\
&\cong [\![\textbf{com}]\!](W)
\end{aligned}
$$

Any $p \in [\![\textbf{unit} \to \textbf{com}]\!](W)$ satisfies $p[V](*) = (p[W](*)){\uparrow}_W^V$ and, is uniquely determined by its W-component. Let $p_0 = \eta(\mathtt{p})[W](*)$ be the W-component of $\eta(\mathtt{p})$ with type $\forall_{X^+<:W}\, \text{St}(X^+) \to [\exists_{Z<:X^+} \text{St}(Z)] + \{\varepsilon\}$. So, for all correspondences $\rho R\colon X^+ \leftrightarrow X^+$ such that $\rho R <: I_W$, we have that

$$s_1[\mathrm{St}(\rho R)]s_2 \implies p_0[X^+]s_1 = p_0[X^+]s_2 = \varepsilon \\ \text{or } \; p_0[X^+]s_1 \left[\exists_{\sigma S <: \rho R} \mathrm{St}(\sigma S)\right] p_0[X^+]s_2$$

Let $\rho = i_W$, and $R(\psi)$ be the relation

$$s_1[R(\psi)]s_2 \iff s_1(l_0) = l_1 \wedge s_1(l_1) = 0$$

The state $s[l_0{\to}l_1] * [l_1{\to}0]$ is related to itself by $EQ_\rho * R(\rho)$. Then, either $p[X^+](s[l_0{\to}l_1] * [l_1{\to}0])$ is ε, or it is related to itself by $\exists_{\sigma S <: \rho R} \mathrm{St}(\sigma S)$. We will focus on the case that $p[X^+](s[l_0{\to}l_1] * [l_1{\to}0])$ is $\langle\!|Y, t|\!\rangle_{X^+}$. In this case, there are representatives, say $\langle Y_1, t_1\rangle$ and $\langle Y_2, t_2\rangle$, of $\langle\!|Y, t|\!\rangle_{X^+}$ and a correspondence $\sigma S\colon Y_1 \leftrightarrow Y_2$ such that $\sigma S <: \rho R$ and $t_1[EQ_\sigma * S(\sigma)]t_2$. Since $S(\sigma)$ is of the form $R(\sigma) * P$, we conclude that $t_1(l_0) = l_1$ and $t_1(l_1) = 0$. Thus, the boolean expression of the following conditional statement evaluates to false, so the whole command diverges.

For the second program block of Example 1, where `x` is a *non-local* variable, we can use the same calculation as above, but with world W and environment η (instead of W^+ and η^+). The difference this makes is that $\eta(\mathtt{p})$ denotes a procedure that has access to $\eta(\mathtt{x})$. Hence, the partial bijection ρ involves the locations l_0 and l_1 and it is not possible to choose a relation $R(\psi)$ that constrains the contents of these locations. So, we cannot conclude that the program block diverges.

The program block of Example 2 involves a procedure `h` of type

$$\mathtt{h} : {\uparrow}\mathtt{int} \to \mathbf{com}$$

and

$$[\![{\uparrow}\mathtt{int} \to \mathbf{com}]\!](W) = \forall_{V <: W}\, [\![{\uparrow}\mathtt{int}]\!](V) \to [\![\mathbf{com}]\!](V).$$

We have argued that there that the call `h(x)` amounts to leakage of the heap variable `x↑` and we cannot conclude that `x↑` is hidden from the subsequent procedure call to `p`. We now show how this leakage is handled in the semantics.

Let W be a world denoting the static context, l_0 a location that is not in W, $W^+ = W \uplus \{l_0\}$, η an environment for W and $\eta^+ = \eta{\uparrow}_W^{W^+}[\mathtt{x}{\to}\mathit{var}_{W^+}(l_0)]$. Let $X <: W^+$ be a world denoting the dynamic context, $l_1 = \mathit{newloc}(X)$ and $X^+ = X \uplus \{l_1\}$. We can calculate:

$$[\![\mathtt{x} := \mathbf{new}\ \mathtt{int}; \mathtt{h}(\mathtt{x})]\!]_{W^+}\eta^+[X](s) = \\ \mathit{hide}_{X^+ <: X}\Big(\eta(\mathtt{h})[X^+](l_1)[X^+]\,(s[l_0{\to}l_1] * [l_1{\to}\mathrm{init}])\Big).$$

Let this be denoted $\langle\!|Y, t|\!\rangle_X$. The type of $\eta(h)$ implies that, for all $\rho R <: I_W$,

$$l_1 \, [\langle\!\langle \uparrow\mathtt{int} \rangle\!\rangle(\rho R)] \, l_1 \Longrightarrow \eta(h)[X^+](l_1) \left[\langle\!\langle \mathbf{com} \rangle\!\rangle(\rho R)\right] \eta(h)[X^+](l_1).$$

For the hypothesis to hold, we must have $\rho <: i_{W \uplus \{l_1\}}$. The result of the above command, $\langle\!|Y, t|\!\rangle_X$, is related to itself by $\exists_{\sigma S <: \rho R} \, \mathrm{St}(\sigma S)$. So, the best we can infer is that t is related to itself by $\mathrm{St}(\sigma S)$ where $\sigma <: i_{W \uplus \{l_1\}}$. We cannot duplicate the previous reasoning for the subsequent procedure call $\mathtt{p}()$, because we cannot show that the state $t[l_1 \to 0]$ is related to itself by a relation $\mathrm{St}(\tau T)$ where τ does not relate l_1. The effect of saying $\rho <: i_{W \uplus \{l_1\}}$ is that the procedure call $\mathtt{h}(\mathtt{x})$ is at liberty to store a pointer to l_1 somewhere in the visible part of the state. So, it is not protected from access by another procedure $\mathtt{p}$.

As an example of reasoning about classes, we consider a toy memory allocator object which keeps a store of list nodes and dispenses them one at a time. This represents half of a memory manager object. (The other half would include a routine for returning nodes to the memory manager [12]. We cannot handle such a routine using the techniques of this paper because it would leave "dangling" references to the returned nodes in client programs.) In addition to classes, this example also illustrates the transfer of heap cells from the hidden part of the object's storage to the visible part. This kind of transfer was mentioned in Example 4 which can also be handled in a similar fashion. The handling of such transfer crucially depends upon the more general conditions used in Definitions 5 and 6. The original definitions of [21] do not allow such transfer.

Example 20 We define a class for a toy memory allocator as follows:

```
C1 = class : var(↑node) → com
       local var (↑node) head;
       init head := nil;
       meth λx. { if head = nil then {
                      local var (↑node) t;
                      local var int i;
                      for i := 1 to 10 do {
                        t := head;
                        head := new node;
                        head↑.2 := t
                      }
                    }
                    x := head;
                    head := head↑.2;
                    x↑.2 := nil }
     end
```

The meaning of the procedure serving as the sole method of the class can be calculated for a world W and environment η with $\eta(\texttt{head}) = var_W(l)$.

$$
\begin{array}{l}
\Lambda V{<:}W.\,\lambda(p,g).\,\Lambda X{<:}V.\,\lambda s.\\
\qquad\qquad \text{if } (s(l) = \text{nil})\\
\qquad\qquad\quad \text{then } hide_{X^{++}<:X}(p\,[X^{++}](l_1)[X^{++}](s[l{\to}l_2] * s_f))\\
\qquad\qquad\qquad\qquad \text{where } l_1 = newloc(X),\ l_2 = newloc(X \uplus \{l_1\}), \ldots\\
\qquad\qquad\qquad\qquad\qquad s_f = [l_1{\to}(\text{init},\text{nil}),\ l_2{\to}(\text{init}, l_3),\ \ldots l_{10}{\to}(\text{init},\text{nil})]\\
\qquad\qquad\qquad\qquad\qquad X^{++} = X \uplus \{l_1,\ldots,l_{10}\}\\
\qquad\qquad\quad \text{else } p\,[X](s(l))[X](s[l{\to}s(s(l)).2,\ s(l){\to}(s(s(l)).1,\text{nil})])
\end{array}
$$

Since this value is constant for all η such that $\eta(\texttt{head}) = var_W(l)$, we denote it by $a_W(l)$.

The meaning of the class is given by

$$
\begin{array}{rcl}
[\![\texttt{C}_1]\!]_W\eta &=& \Lambda V{<:}W.\ \langle\!|V^+,\ \text{unit}_{V^+}(a_{V^+}(l_0)),\ \lambda s.\ \langle\!|V^+,\ s * [l_0{\to}\text{nil}]|\!\rangle_{V^+}|\!\rangle_V\\
&& \quad \text{where } l_0 = newloc(V),\ V^+ = V \uplus \{l_0\}
\end{array}
$$

We would like to show that this class is equivalent to a naive allocator that always creates a new node.

```
C2 = class : var(↑node) → com
       init skip;
       meth λx. { x := new node; x↑.2 := nil }
     end
```

The meaning of the method can be calculated for a world W and environment η as follows:

$$
\begin{array}{l}
\Lambda V{<:}W.\,\lambda(p,g).\,\Lambda X{<:}V.\,\lambda s.\\
\qquad hide_{X^+<:X}\Big(p\,[X^+](k_1)[X^+]\,(s * [k_1{\to}(\text{init},\text{nil})])\Big)\\
\qquad \text{where } k_1 = newloc(X),\ X^+ = X \uplus \{k_1\}
\end{array}
$$

Since this value is independent of η, we simply denote it as b_W. The meaning of the class is given by

$$[\![\texttt{C}_2]\!]_W\eta \;=\; \Lambda V{<:}W.\ \langle\!|V,\ \text{unit}_V(b_V),\ \lambda s.\ \langle\!|V, s|\!\rangle_V|\!\rangle_V$$

To prove that the two classes are equivalent, we need to show that

$$\langle V^+,\ \text{unit}_{V^+}(a_{V^+}(l_0)),\ \lambda s.\ \langle\!|V^+,\ s{*}[l_0{\to}\text{nil}]|\!\rangle_V\rangle \approx_V \langle V,\ \text{unit}_V(b_V),\ \lambda s.\ \langle\!|V, s|\!\rangle_V\rangle$$

We define a correspondence $\rho R\colon V^+ \leftrightarrow V$ such that $\rho R <: I_V$.

$$\begin{aligned} \rho \;&=\; j_{V^+,V} \;=\; \{(l,l) \mid l \in V\} \\ R(\psi) \;&=\; \{\, ([l_0{\to}l_1,\, l_1{\to}(\mathrm{init}, l_2), \ldots, l_n{\to}(\mathrm{init}, \mathrm{nil})],\, [\,]) \mid \\ &\qquad\qquad\qquad\qquad l_1, \ldots, l_n \in \mathrm{Loc}_{\mathtt{node}} \wedge n \geq 0 \,\} \end{aligned}$$

This signifies that the empty heap in the representation of $\mathtt{C}_2$ corresponds to an arbitrary linked list stored at `head` in the representation of $\mathtt{C}_1$. Note that the initial states of the classes are related by this correspondence:

$$(s * [l_0{\to}\mathrm{nil}])\ [\mathrm{St}(\rho) * R(\rho)]\ s.$$

To show that the two methods are related by the correspondence, we need:

$$\mathrm{unit}_{V^+}(a_{V^+}(l_0))\ \Big[\langle\!\langle \mathbf{exp}(\mathbf{var}\ (\uparrow\mathtt{node} \to \mathbf{com})) \rangle\!\rangle(\rho R)\Big]\ \mathrm{unit}_V(b_V)$$

That is,

$$a_{V^+}(l_0)\,[\langle\!\langle \mathbf{var}\ (\uparrow\mathtt{node}) \to \mathbf{com} \rangle\!\rangle(\rho R)]\, b_V.$$

After examining the definitions of $a_{V^+}(l_0)$ and b_V, we note that the key verification concerns the state transformation up to the assignment of the pointer (l_1 and k_1, in the two cases) to the argument variable. Therefore, assume that $X_1 <: V^+$ and $X_2 <: V$ are two worlds with a correspondence $\sigma S\colon X_1 \leftrightarrow X_2$ such that $\sigma S <: \rho R$. Let $s_1 \in \mathrm{St}(X_1)$ and $s_2 \in \mathrm{St}(X_2)$ be states such that $s_1[\mathrm{St}(\sigma S)]s_2$. Then, we have two cases:

- **Case** $s_1(l_0) = \mathrm{nil}$:
  We need a correspondence $\tau T : X_1^{++} \leftrightarrow X_2^{+}$ such that the states

  $$\begin{aligned} s_1' &= s_1[l_0{\to}l_2] * [l_1{\to}(\mathrm{init}, \mathrm{nil}),\, l_2{\to}(\mathrm{init}, l_3),\, \ldots, l_{10}{\to}(\mathrm{init}, \mathrm{nil})] \\ s_2' &= s_2 * [k_1{\to}(\mathrm{init}, \mathrm{nil})] \end{aligned}$$

  are related by $\mathrm{St}(\tau T)$. Choose:

  $$\begin{aligned} \tau &= \sigma \cup \{(l_1, k_1)\} \\ T(\psi) &= S(\psi) \end{aligned}$$

  To show that s_1' and s_2' are related by $\mathrm{St}(\tau T)$, recall that s_1 $[\mathrm{St}(\sigma S)]$ s_2, i.e., s_1 $[EQ_\sigma * S(\sigma)]$ s_2. Moreover, since $\sigma S <: \rho R$, $S(\sigma)$ is of the form $R(\sigma) * P$ for some heap relation P. Therefore, we can decompose s_1 and s_2 as $s_1 = h_1' * h_1'' * h_1'''$ and $s_2 = h_2' * h_2'' * h_2'''$ such that h_1' $[EQ_\sigma]$ h_2', h_1'' $[R(\sigma)]$ h_2'', and h_1''' $[P]$ h_2'''. It is easy to see that $h_1'' = [l_0{\to}\mathrm{nil}]$ and $h_2'' = [\,]$. We can then restate s_1' and s_2' as:

$$s_1' = (h_1' * [l_1{\to}(\mathrm{init}, \mathrm{nil})]) * ([l_0{\to}l_2] * [l_2{\to}(\mathrm{init}, l_3), \ldots, l_{10}{\to}(\mathrm{init}, \mathrm{nil})]) * h_1'''$$
$$s_2' = (h_2' * [k_1{\to}(\mathrm{init}, \mathrm{nil})]) * [\,] * h_2'''$$

and it is immediate that $s_1'\ [EQ_\sigma * R(\sigma) * P]\ s_2'$. Recall that $R(\sigma) * P = S(\sigma)$ and $S(\sigma) \subseteq S(\tau)$ by virtue of $\tau <: \sigma$. Since $T(\tau) = S(\tau)$, we have $s_1'\ [EQ_\tau * T(\tau)]\ s_2'$.

- **Case** $s_1(l_0) \neq \mathrm{nil}$:
  We need a correspondence $\tau T : X_1 \leftrightarrow X_2^+$ such that $\tau T <: \sigma S$ and the states

  $$s_1' = s_1[l_0{\to}l_2,\ l_1{\to}(s_1(l_1).1, \mathrm{nil})]$$

  and

  $$s_2' = s_2 * [k_1{\to}(\mathrm{init}, \mathrm{nil})]$$

  are related by $\mathrm{St}(\tau T)$. Choose:

  $$\tau = \sigma \cup \{(l_1, k_1)\}$$
  $$T(\psi) = S(\psi)$$

  The argument that s_1' and s_2' are related follows along the same lines as above.

□

6 Conclusion

We have presented a semantic model for languages with heap data structures which makes explicit the information hiding properties of programs. Simulation-based reasoning principles for data abstractions are directly captured in the model. The main adjustment that has been made, compared to the models of local variables [15], is that correspondences between data representations must keep track of pointers to visible locations (pointers that can be leaked).

Comparing this to the precursor of work by Banerjee and Naumann [2], we note that they use traditional simulation relations for data representations, and impose additional conditions of confinement to prevent all leakage of pointers. While notions of confinement for protecting data abstractions are definitely worthy of study, we believe that they should not obscure the intrinsic information hiding properties that the languages possess. Thus, our focus has been on the latter. At the same time, it would be useful to unravel the confinement notions implicit in our semantic model and to make more direct comparisons with other work on confinement.

Object-oriented programming was not treated seriously in the present paper, even though we have used classes to illustrate the information hiding aspects of the semantics. In particular, heap-allocated class instances are not treated. There are, indeed, certain new technical issues involved in doing so because the state becomes a higher-order entity with self-application features and recursion can be simulated by assignment. Banerjee and Naumann use a "class-based" approach where objects remain first-order entities but include references to class names and recursion is handled in mapping class names to classes. The application of this idea to our semantic framework needs to be explored.

A further question that is worth investigating is the full abstraction property. Previous results for this form of semantics include [27] for the case of dynamic allocation and [13] for the treatment of local variables. It would also be interesting to find relationships with game semantics for pointer programs [1] which has been proved fully abstract but as yet lacks support for reasoning principles.

Finally, a fruitful direction for future research would be to integrate the semantics of heap storage with programming logics for heap storage such as Separation Logic [11,12]. Even though we have used ideas from Separation Logic such as the $*$ connective in formulating relational correspondences, we have treated semantics of *plain* programs without any specifications attached. How the annotations of programs with specifications might impact their semantics is an intriguing question that we leave open for future work.

Acknowledgements

This work was partly supported by National Science Foundation grant NSF-INT-9813845 while Reddy and Yang were at University of Illinois. In addition, Yang was supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Reddy would like to thank Professor Kwangkeun Yi and the Creative Research Initiatives of the Korean Ministry of Science and Technology for his visit to KAIST where this work was carried out. We have benefited greatly from discussions with Peter O'Hearn and David Naumann.

A Category-theoretic Background

We recall some key definitions from [3,15,18]:

- A *reflexive graph category* $\mathbf{G}$ consists of two categories and three functors

$$
\begin{tikzcd}
\mathbf{G}_v \arrow[r, "I" description] & \mathbf{G}_e \arrow[l, shift right=2, "\delta_0"'] \arrow[l, shift left=2, "\delta_1"]
\end{tikzcd}
$$

  such that both $\delta_0 \circ I$ and $\delta_1 \circ I$ are identity functors on $\mathbf{G}_v$. We call $\mathbf{G}_v$ the *vertex category* of $\mathbf{G}$, and $\mathbf{G}_e$ the *edge category* of $\mathbf{G}$. Objects and morphisms of $\mathbf{G}_v$ are called *vertices* and *vertex morphisms*, and those of $\mathbf{G}_e$ are called *edges* and *edge morphisms*. We write $E\colon V_0 \leftrightarrow V_1$ to denote that E is an edge that is projected to V_i by δ_i.
- A reflexive graph is *subsumptive* if and only if there is a map from vertex morphisms $f\colon X \to Y$ to edges $\langle f \rangle : X \leftrightarrow Y$ such that

(1) $\langle id_X \rangle = I_X$ for all identity vertex morphisms $id_X : X \to X$, and

(2) there is an edge morphism φ from $\langle g \rangle$ to $\langle h \rangle$ such that $\delta_i(\varphi) = f_i$ if and only if the following diagram commutes:

$$
\begin{tikzcd}
V_0 \arrow[r, "f_0"] \arrow[d, "g"'] & V_0' \arrow[d, "h"] \\
V_1 \arrow[r, "f_1"'] & V_1'
\end{tikzcd}
$$

- A *reflexive graph-functor* F from $\mathbf{G}$ to $\mathbf{G}'$ consists of two functors $F_v\colon \mathbf{G}_v \to \mathbf{G}'_v$ and $F_e\colon \mathbf{G}_e \to \mathbf{G}'_e$ such that $I \circ F_v = F_e \circ I$ and $\delta_i \circ F_e = F_v \circ \delta_i$. We will omit subscripts from F_v and F_e, and write F in both cases.
- A reflexive graph-functor F is *subsumptive* if and only if it preserves the subsumption map: $F(\langle f \rangle) = \langle F(f) \rangle$ for all vertex morphisms f. The notion of subsumptive functors in Definition 14 is a little stronger than this as it requires $F(f)$ to be *uniquely* determined by the action of F on edges. Hence, the action on morphisms can be omitted from the definition of F as in the "type operators" of Section 3.
- A parametric transformation $\tau\colon F \to F'\colon \mathbf{G} \to \mathbf{G}'$ is a family $\{\tau_X\colon F(X) \to G(X)\}$ of morphisms indexed by vertices X in $\mathbf{G}$ such that for each edge $E\colon V_0 \leftrightarrow V_1$ in $\mathbf{G}$, there is an edge morphism $\varphi\colon F(E) \to G(E)$ such that $\delta_i(\varphi) = \tau_{V_i}$.

We will use the following result about parametric transformations from [18,3], which says that for subsumptive reflexive graph-functors, parametricity subsumes naturality.

Proposition 21 *If reflexive graph-functors F, F' are subsumptive, every parametric transformation $\tau\colon F \to F'$ is a natural transformation of type $F_v \to F'_v$.*

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.

[5] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *J. Lisp and Symbolic Comput. 8*, 4 (1995), 293–341.

[6] Levy, P. B. *Call-by-Push-Value.* PhD thesis, Queen Mary, University of London, March 2001.

[7] Levy, P. B. Possible world semantics for general storage in call-by-value. In *CSL 2002* (2002), pp. 232–246.

[8] 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 [16]).

[9] Mitchell, J. C., and Plotkin, G. D. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502.

[10] Moggi, E. Notions of computations and monads. *Information and Computation 93* (1991), 55–92.

[11] O'Hearn, P. W., 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., Reynolds, J., and Yang, H. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang.* (2004), ACM. (to appear).

[13] O'Hearn, P. W., and Reynolds, J. C. From Algol to polymorphic linear lambda-calculus. *J. ACM 47*, 1 (Jan 2000), 167–223.

[14] 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.

[15] O'Hearn, P. W., and Tennent, R. D. Parametricity and local variables. *J. ACM 42*, 3 (1995), 658–709. (Reprinted as Chapter 16 of [16]).

[16] O'Hearn, P. W., and Tennent, R. D. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

[17] Oles, F. J. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[18] Reddy, U. S. When parametricity implies naturality. Electronic manuscript, University of Birmingham, http://www.cs.bham.ac.uk/˜udr/, July 1997.

[19] Reddy, U. S. Objects and classes in Algol-like languages. In *FOOL 5: Fifth Intern. Workshop on Foundations of Object-oriented Languages* (Jan 1998), electronic proceedings at http://pauillac.inria.fr/˜remy/ fool/proceedings.html.

[20] Reddy, U. S. Objects and classes in Algol-like languages. *Information and Computation 172* (2002), 63–97.

[21] Reddy, U. S., and Yang, H. Correctness of data represenations involving heap data structures. In *Programming Languages and Systems: 12th European Symposium on Programming* (2003), vol. 2618 of *LNCS*, Springer-Verlag, pp. 223–237.

[22] Reynolds, J. C. Towards a theory of type structure. In *Coll. sur la Programmation*, vol. 19 of *LNCS*. Springer-Verlag, 1974, pp. 408–425.

[23] 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 [16]).

[24] Reynolds, J. C. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2000.

[25] 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.

[26] Stark, I. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge Computer Laboratory, 1995. Technical Report 363.

[27] Stark, I. Categorical models for local names. *Lisp and Symbolic Computation 9*, 1 (Feb. 1996), 77–107.

[28] 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.

[29] Wadler, P. The essence of functional programming. In *ACM Symp. on Princ. of Program. Lang.* (1992), pp. 1–14.

[30] Wirth, N., and Hoare, C. A. R. A contribution to the development of Algol. *Comm. ACM 9*, 6 (June 1966), 413–432.

[31] YANG, H. *Local Reasoning for Stateful Programs.* PhD thesis, The University of Illinois at Urbana-Champaign, 2001. Technical Report UIUCDCS-R-2001-2227.