A Foundation for the Model Theory of Abstract Data Types with Mutation and Aliasing (preliminary version)

Gary T. Leavens and Krishna Kishore Dhara

TR #92-35

November 1992

Keywords: abstract data type, mutation, aliasing, model theory, simulation relation.

1992 CR Categories: D.3.3 [*Programming Languages*] Language Constructs — Abstract data types; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — algebraic approaches to semantics, denotational sematics.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011-1040, USA

A Foundation for the Model Theory of Abstract Data Types with Mutation and Aliasing (preliminary version)

Gary T. Leavens*and Krishna Kishore Dhara
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu and dhara@cs.iastate.edu

November 23, 1992

Abstract

To aid in understanding object-oriented programming languages, we present some fundamentals of model theory for languages with mutable abstract data types and aliasing. Our semantics for such languages is parameterized by an algebraic model of all the abstract data types involved in the program, including types with mutable objects. We give an algebraic characterization of simulation between states of such algebraic models. We present a definition of aliasing that is based on the observable behavior of objects.

1 Introduction

Despite its importance for object-oriented programming, less is known about the model theory of mutation and aliasing than about the model theory of abstract data types with immutable objects. Variables that can change their state by assignment, arrays, records, and other primitive domains are treated in work on denotational semantics. But the primitive semantic domains rarely include arbitrary abstract data types, and we know of no such work that defines aliasing behaviorally. The literature on models of abstract data types, on the other hand, rarely treats issues such as mutation and aliasing. Yet for a full understanding of the semantics of object-oriented programming, both abstract data types and mutable state must be treated.

The work reported here describes some fundamentals of a model theory for abstract data types whose objects have mutable state. (We call such types *mutable types*.) In particular, we give a behavioral description of aliasing, and describe a notion of simulation relation

*This work was supported in part by the National Science Foundation under Grant CCR-9108654.

appropriate for such models. The notion of a simulation relation is similar to the notion of a congruence relation, except that it is not symmetric; it is also similar to the base case of the logical relations used in the study of typed lambda calculi [1] [2]. We also show that our notion of simulation relation incorporates the notion of aliasing.

Our definition of aliasing is unique in that it defines aliasing based on object behavior. That is, aliasing is only affected by changes in specification, not by changes in representation details. A treatment of aliasing that only concerned itself with representations would not qualify as a treatment of aliasing for abstract data types.

2 Algebraic Model

Our models of abstract types with mutable objects are somewhat nonstandard from the standpoint of denotational semantics, because they do not describe objects in terms of a few basic types, such as products, sums, and functions. Instead, they more closely resemble the models of equational algebraic specifications, having carrier sets and operations that are only constrained to satisfy a specification. Our particular models are most strongly inspired by the work of Wing [3] and Chen [4].

In a typed language with abstract data types and mutation, such as CLU [5], objects can be thought of as typed memory cells containing values. The values may contain the locations of other cells. This resembles Scheifler's denotational semantics of CLU [6]. It is also much like LISP, except that cells come in many different types. Abstract values can be thought of as mathematical abstractions of the concrete representations used in programs [7]. Locations can be thought of as the names of objects; these are sometimes called object identifiers. Locations are typed in the same sense that identifiers are typed; a location $l : T$ can only store an abstract value of type T.

The types in a signature are used in programs. Identifiers are typed, and contain denotable values, which can be either locations or abstract values. Following Wing, names of the corresponding abstract values are called sorts. The mapping $TtoS$ gives the corresponding sort for a given type.

Definition 2.1 *A* signature*, Σ, is a tuple,*

$$(TYPES, VIS, SORTS, IDS, TtoS, OPS),$$

where

- *TYPES is a non-empty set of type symbols, such that* `Void` $\in$ *TYPES.*
- *VIS* $\subseteq$ *TYPES is a set of visible type symbols, such that* `Bool` $\in$ *VIS*
- *SORTS is a non-empty set of sort symbols,*
- *IDS is a set of identifier symbols, indexed by TYPES,*
- *an injective mapping, TtoS* : *TYPES* $\rightarrow$ *SORTS,*
- *a set OPS, of operation symbols, indexed by* $TYPES^* \times TYPES$.

As usual, we write $g : \vec{S} \rightarrow T$ for $g \in OPS_{\langle \vec{S}, T \rangle}$. (We use angle brackets ($\langle, \rangle$) to surround tuples of types, to avoid confusion with tuples of values.)

Figure 1 gives an example signature, Σ_E. Its mutable types are `Point` and `Rect`. That the objects of these types are mutable can be seen from the signature of operations such as `addX` and `horizMove`, which have no results.

In what follows, let $\Sigma = (TYPES, VIS, SORTS, IDS, TtoS, OPS)$ stand for an arbitrary signature.

Unlike the usual work in algebraic models, our models not only have a carrier set for the abstract values (sorts), but also have a carrier set for typed locations.

Definition 2.2 *A* Σ*-algebra is a tuple,* $(LOCS^A, |A|, OPS^A)$*, where*

- $LOCS^A = \bigcup_{T \in TYPES} LOCS^A_T$ *is a family of sets, representing typed locations, and,*
- $VALS^A = |A| = \bigcup_{S \in SORTS} A_S$ *is a family of sets, representing the abstract values of each type, such that* $* \in A_{TtoS[\texttt{Void}]}$ *and* $\{true, false\} = A_{TtoS[\texttt{Bool}]}$*, and*
- OPS^A *is a family of functions, indexed by* $TYPES^* \times TYPES$*, such that for each operation symbol* $g : \langle S_1, \ldots, S_n \rangle \rightarrow T$*, the functionality of* g^A *is*

$$g^A : (DVAL^A_{S_1} \times \cdots \times DVAL^A_{S_n}) \times STORE[A] \rightarrow DVAL^A_T \times STORE[A],$$

where $STORE[A]$ *is the set of all finite functions from* $LOCS^A$ *to* $|A|$*:*

$$STORE[A] \quad = \quad LOCS^A \xrightarrow{\text{fin}} |A|. \tag{1}$$

and $DVAL^A$ *is the set of denotable values:*

$$DVAL^A \quad = \quad LOCS^A + VALS^A \tag{2}$$

We write $DVAL^A_T$ *for the set of denotable values from either* $LOCS^A_T$ *or* $A_{TtoS[T]}$*.*

$$
\begin{array}{lcl}
VIS_E & \stackrel{\text{def}}{=} & \{\texttt{Bool}, \texttt{Int}\} \\
TYPES_E & \stackrel{\text{def}}{=} & \{\texttt{Void}, \texttt{Bool}, \texttt{Int}, \texttt{Point}, \texttt{Rect}\} \\
SORTS_E & \stackrel{\text{def}}{=} & \{\mathit{Unit}, \mathit{Boolean}, \mathit{Integer}, \mathit{PointSort}, \mathit{RectSort}\}
\end{array}
$$

$$
IDS \stackrel{\text{def}}{=} \biguplus_{T \in TYPES_E} \{s \mid s \text{ is a non-empty string of alphanumeric characters}\}
$$

Type to Sort Mapping ($TtoS$)

$$
\begin{array}{lcl}
\texttt{Void} & \mapsto & \mathit{Unit} \\
\texttt{Bool} & \mapsto & \mathit{Boolean} \\
\texttt{Int} & \mapsto & \mathit{Integer} \\
\texttt{Point} & \mapsto & \mathit{PointSort} \\
\texttt{Rect} & \mapsto & \mathit{RectSort}
\end{array}
$$

Operation symbols (OPS)

$$
\begin{array}{lcl}
\texttt{()} & : & \langle\rangle \to \texttt{Void} \\
\texttt{true} & : & \langle\rangle \to \texttt{Bool} \\
\texttt{false} & : & \langle\rangle \to \texttt{Bool} \\
\texttt{and} & : & \langle \texttt{Bool}, \texttt{Bool} \rangle \to \texttt{Bool} \\
\texttt{or} & : & \langle \texttt{Bool}, \texttt{Bool} \rangle \to \texttt{Bool} \\
\texttt{not} & : & \langle \texttt{Bool} \rangle \to \texttt{Bool} \\
\texttt{0} & : & \langle\rangle \to \texttt{Int} \\
\texttt{1} & : & \langle\rangle \to \texttt{Int} \\
\texttt{add} & : & \langle \texttt{Int}, \texttt{Int} \rangle \to \texttt{Int} \\
\texttt{mult} & : & \langle \texttt{Int}, \texttt{Int} \rangle \to \texttt{Int} \\
\texttt{negate} & : & \langle \texttt{Int} \rangle \to \texttt{Int} \\
\texttt{equal} & : & \langle \texttt{Int}, \texttt{Int} \rangle \to \texttt{Bool} \\
\texttt{mkPoint} & : & \langle \texttt{Int}, \texttt{Int} \rangle \to \texttt{Point} \\
\texttt{abscissa} & : & \langle \texttt{Point} \rangle \to \texttt{Int} \\
\texttt{ordinate} & : & \langle \texttt{Point} \rangle \to \texttt{Int} \\
\texttt{addX} & : & \langle \texttt{Point}, \texttt{Int} \rangle \to \texttt{Void} \\
\texttt{addY} & : & \langle \texttt{Point}, \texttt{Int} \rangle \to \texttt{Void} \\
\texttt{mkRect} & : & \langle \texttt{Point}, \texttt{Point} \rangle \to \texttt{Rect} \\
\texttt{botLeft} & : & \langle \texttt{Rect} \rangle \to \texttt{Point} \\
\texttt{topRight} & : & \langle \texttt{Rect} \rangle \to \texttt{Point} \\
\texttt{horizMove} & : & \langle \texttt{Rect}, \texttt{Int} \rangle \to \texttt{Void} \\
\texttt{vertMove} & : & \langle \texttt{Rect}, \texttt{Int} \rangle \to \texttt{Void}
\end{array}
$$

Figure 1: The signature Σ_E.

$$
\begin{array}{lcl}
\mathit{Unit}^E & \stackrel{\text{def}}{=} & \{*\} \\
\mathit{Boolean}^E & \stackrel{\text{def}}{=} & \{\mathit{true}, \mathit{false}\} \\
\mathit{Integer}^E & \stackrel{\text{def}}{=} & \{0, 1, -1, 2, -2, \ldots\} \\
\mathit{PointSort}^E & \stackrel{\text{def}}{=} & \{(l_x, l_y) \mid l_x, l_y \in \mathit{LOCS}^E_{\texttt{Int}}\} \\
\mathit{RectSort}^E & \stackrel{\text{def}}{=} & \{(l_{bl}, l_{tr}) \mid l_{bl}, l_{tr} \in \mathit{LOCS}^E_{\texttt{Point}}\}
\end{array}
$$

Figure 2: Carrier sets for the example Σ_E-algebra, E.

The special abstract value $*$ of sort $\mathit{TtoS}[\texttt{Void}]$ is used for the result of an operation that would otherwise not have a value. An operation is modeled by a function from a sequence of argument objects and an initial store to a result object and a final store. (We thus do not model nondeterministic operations.)

Stores are finite functions from a subset of LOCS^A to abstract values; in other words, they are not defined for all locations.

We now describe an example Σ_E-algebra, E. In this algebra, all operations take and return locations. (We will later describe an algebra without this property.) As the locations have no interesting structure, we adopt the convention that for each type T:

$$\mathit{LOCS}^E_T \stackrel{\text{def}}{=} \{l^T_i \mid i \in \mathit{Nat}\}. \tag{3}$$

The carrier sets for this algebra are defined in Figure 2. Our use of pairs in the carrier sets of rectangles and points in E is simply a convenience; we could also have used functions or stacks or some other mathematically defined carrier set.

The operations of E are defined in Figure 3. We use $\mathit{nextFree}[T]$ to find the next free location of type T in a given store.

$$
\begin{array}{l}
\mathit{nextFree}[T] : \mathit{STORE}[E] \to \mathit{LOCS}^E_T \\
\mathit{nextFree}[T](\sigma) \stackrel{\text{def}}{=} l^T_{1+\text{lub}\{i \mid l^T_i \in \text{dom}(\sigma)\}}
\end{array}
$$

We use the function $\mathit{alloc}[T]$ to find a free location and initialize it with an abstract value of type T.

$$
\begin{array}{l}
\mathit{alloc}[T] : (\mathit{TtoS}[T]^E \times \mathit{STORE}[A]) \to \mathit{DVAL}^E_T \times \mathit{STORE}[A] \\
\mathit{alloc}[T](v, \sigma) \stackrel{\text{def}}{=} \textbf{let } l = \mathit{nextFree}[T](\sigma) \textbf{ in } (\mathit{inLOCS}(l), [l \mapsto v]\sigma)
\end{array}
$$

The notation $[l \mapsto v]\sigma$ is the function σ extended to bind l to v; that is $[l \mapsto v]\sigma \stackrel{\text{def}}{=} \lambda l_2 . (l_2 = l) \to v [\!] \sigma(l_2)$, where the notation $b \to e_1 [\!] e_2$ means if b is *true*, then e_1, else e_2. The notation $\mathit{inLOCS}(l)$ means the result of injecting the location l into the LOCS summand of DVAL.

We use this notation also to pattern matching when defining functions and picking apart values, as in the language Standard ML. For example, `mult` takes 2 denotable values that are locations as arguments.

In Figure 3, the following operations work on objects of type `Point`.

- `mkPoint`, which takes two integers and returns a new point.
- `abscissa` and `ordinate`, which return the x and y coordinates of a point.
- `addX` and `addY`, which mutate a point by adding an offset to either the x or the y coordinate.

The following operations work on `Rect` objects.

- `mkRect`, which takes two points, that have abstract values (x_1, y_1) and (x_2, y_2) such that $x_1 \leq x_2$ and $y_1 \leq y_2$, and returns a new rectangle having the first point as its bottom left corner, and with the second as its top right corner. Note that the points are not copied, but put directly into the abstract value of the rectangle; this is a dangerous programming practice, but it is useful for our purposes in demonstrating aliasing.
- `botLeft` and `topRight`, which return the points at the bottom left and the top right corners of a rectangle. Note that these do not return copies of the points, but the points themselves.
- `horizMove` and `vertMove`, which mutates the rectangle, moving it horizontally or vertically by the amount given in the second argument.

The state of a program is given by two mappings: an environment and a store. An environment maps identifiers to denotable values. Denotable values are object identifiers or abstract values. A store maps object identifiers to their abstract values; the type of stores is given in Equation (1) above. Both stores and environments are necessary for a full treatment of aliasing and mutation. Both types of mappings are parameterized by an algebra, A.

$$ENV[A] = IDS \xrightarrow{\text{fin}} DVAL^A \tag{4}$$

$$STATE[A] = ENV[A] \times STORE[A] \tag{5}$$

Operations

$$
\begin{array}{lcl}
\texttt{()}^E((),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Void}](*,\sigma) \\
\texttt{true}^E((),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](true,\sigma) \\
\texttt{false}^E((),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](false,\sigma) \\
\texttt{and}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](\sigma(l_1) \wedge \sigma(l_2),\sigma) \\
\texttt{or}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](\sigma(l_1) \vee \sigma(l_2),\sigma) \\
\texttt{not}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](\neg\sigma(l),\sigma) \\
\texttt{0}^E((),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Int}](0,\sigma) \\
\texttt{1}^E((),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Int}](1,\sigma) \\
\texttt{add}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Int}](\sigma(l_1) + \sigma(l_2),\sigma) \\
\texttt{mult}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Int}](\sigma(l_1) \times \sigma(l_2),\sigma) \\
\texttt{negate}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Int}](-\sigma(l),\sigma) \\
\texttt{equal}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Bool}](\sigma(l_1) = \sigma(l_2),\sigma) \\
\texttt{mkPoint}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Point}]((l_1,l_2),\sigma) \\
\texttt{abscissa}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l) \textbf{ in } (inLOCS(l_1),\sigma) \\
\texttt{ordinate}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l) \textbf{ in } (inLOCS(l_2),\sigma) \\
\texttt{addX}^E((inLOCS(l^{\texttt{Point}}), inLOCS(l^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l_x),\sigma') = \\
 & & \quad \texttt{add}^E((inLOCS(l_1), inLOCS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & alloc[\texttt{Void}](*,[l^{\texttt{Point}} \mapsto (l_x,l_2)]\sigma') \\
\texttt{addY}^E(inLOCS(l^{\texttt{Point}}), inLOCS(l^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l_y),\sigma') = \\
 & & \quad \texttt{add}^E((inLOCS(l_2), inLOCS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & alloc[\texttt{Void}](*,[l^{\texttt{Point}} \mapsto (l_1,l_y)]\sigma') \\
\texttt{mkRect}^E((inLOCS(l_1), inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Rect}]((l_1,l_2),\sigma) \\
\texttt{botLeft}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l) \textbf{ in } (inLOCS(l_1),\sigma) \\
\texttt{topRight}^E(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = \sigma(l) \textbf{ in } (inLOCS(l_2),\sigma) \\
\texttt{horizMove}^E((inLOCS(l^{\texttt{Rect}}), inLOCS(l^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr}) = \sigma(l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l)',\sigma') = \\
 & & \quad \texttt{addX}^E((inLOCS(l_{bl}), inLOCS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l''),\sigma'') = \\
 & & \quad \texttt{addX}^E((inLOCS(l_{tr}), inLOCS(l^{\texttt{Int}})),\sigma') \textbf{ in} \\
 & & alloc[\texttt{Void}](*,\sigma'') \\
\texttt{vertMove}^E((inLOCS(l^{\texttt{Rect}}), inLOCS(l^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr}) = \sigma(l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l'),\sigma') = \\
 & & \quad \texttt{addY}^E((inLOCS(l_{bl}), inLOCS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & \textbf{let } (inLOCS(l''),\sigma'') = \\
 & & \quad \texttt{addY}^E((inLOCS(l_{tr}), inLOCS(l^{\texttt{Int}})),\sigma') \textbf{ in} \\
 & & alloc[\texttt{Void}](*,\sigma'')
\end{array}
$$

Figure 3: Operations for the example Σ_E-algebra, E.

Since locations are typed, we use $l : T$ to stand for a location $l \in LOCS^A_T$ when the algebra A is clear from context. For this we also write "$l : T$ is a location." Similarly we use $d : T$ to stand for a denotable value $d \in DVAL^A_T$. Similarly, we denote identifiers by $x : T$, and write "$x : T$ is an identifier" to mean $x \in IDS_T$, when the set IDS is clear from context.

Just as stores are not defined on all locations, environments are not defined on the whole of IDS. We denote the extension of an environment η with the binding of x to v as $[x \mapsto v]\eta$.

We impose the usual restrictions on environments and stores. That is, if $\eta : ENV[A]$ is an environment over a Σ-algebra A, and if $x : T$ is an identifier in its domain, then $\eta(x) \in DVAL^A_T$. If $\sigma : STORE[A]$ is a store over A, and if $l : T$ is a location in its domain, then $\sigma(l) \in A_{TtoS[T]}$.

We sometimes use a function $absVal$ to get an abstract value from a denotable value and a store. It is defined as follows.

$$absVal : DVAL^A \times STORE[A] \to VALS^A$$
$$absVal(d, \sigma) \stackrel{\text{def}}{=} \textbf{cases } d \textbf{ of } isVALS(v) \to v [\!] isLOCS(l) \to \sigma(l) \textbf{end}$$

An example state over the Σ_E-algebra E is pictured in Figure 4. In the formal model of this picture, the environment η_E would be defined as follows.

$$\begin{aligned}
\eta_E(\mathtt{w}) &= inLOCS(l^{\mathtt{Rect}}_{10}) \\
\eta_E(\mathtt{x}) &= inLOCS(l^{\mathtt{Rect}}_{9}) \\
\eta_E(\mathtt{y}) &= inLOCS(l^{\mathtt{Rect}}_{9}) \\
\eta_E(\mathtt{z}) &= inLOCS(l^{\mathtt{Point}}_{8})
\end{aligned}$$

In Figure 4, we have arranged the picture so that each location has a unique index across all types. For this example, the store mapping, σ_E would be defined as follows.

$$\begin{aligned}
\sigma_E(l^{\mathtt{Rect}}_{10}) &= (l^{\mathtt{Point}}_{6}, l^{\mathtt{Point}}_{7}) \\
\sigma_E(l^{\mathtt{Rect}}_{9}) &= (l^{\mathtt{Point}}_{6}, l^{\mathtt{Point}}_{8}) \\
\sigma_E(l^{\mathtt{Point}}_{8}) &= (l^{\mathtt{Int}}_{2}, l^{\mathtt{Int}}_{5}) \\
\sigma_E(l^{\mathtt{Point}}_{7}) &= (l^{\mathtt{Int}}_{3}, l^{\mathtt{Int}}_{4}) \\
\sigma_E(l^{\mathtt{Point}}_{6}) &= (l^{\mathtt{Int}}_{1}, l^{\mathtt{Int}}_{1}) \\
\sigma_E(l^{\mathtt{Int}}_{5}) &= 5 \\
\sigma_E(l^{\mathtt{Int}}_{4}) &= 4 \\
\sigma_E(l^{\mathtt{Int}}_{3}) &= 3 \\
\sigma_E(l^{\mathtt{Int}}_{2}) &= 2 \\
\sigma_E(l^{\mathtt{Int}}_{1}) &= 1
\end{aligned}$$

Note that the abstract values may "contain" locations, as described for the carrier sets of E. A program, however, does not have direct access to locations that may be contained in an abstract value, but can only access locations and abstract values in ways permitted by the operations of the types.

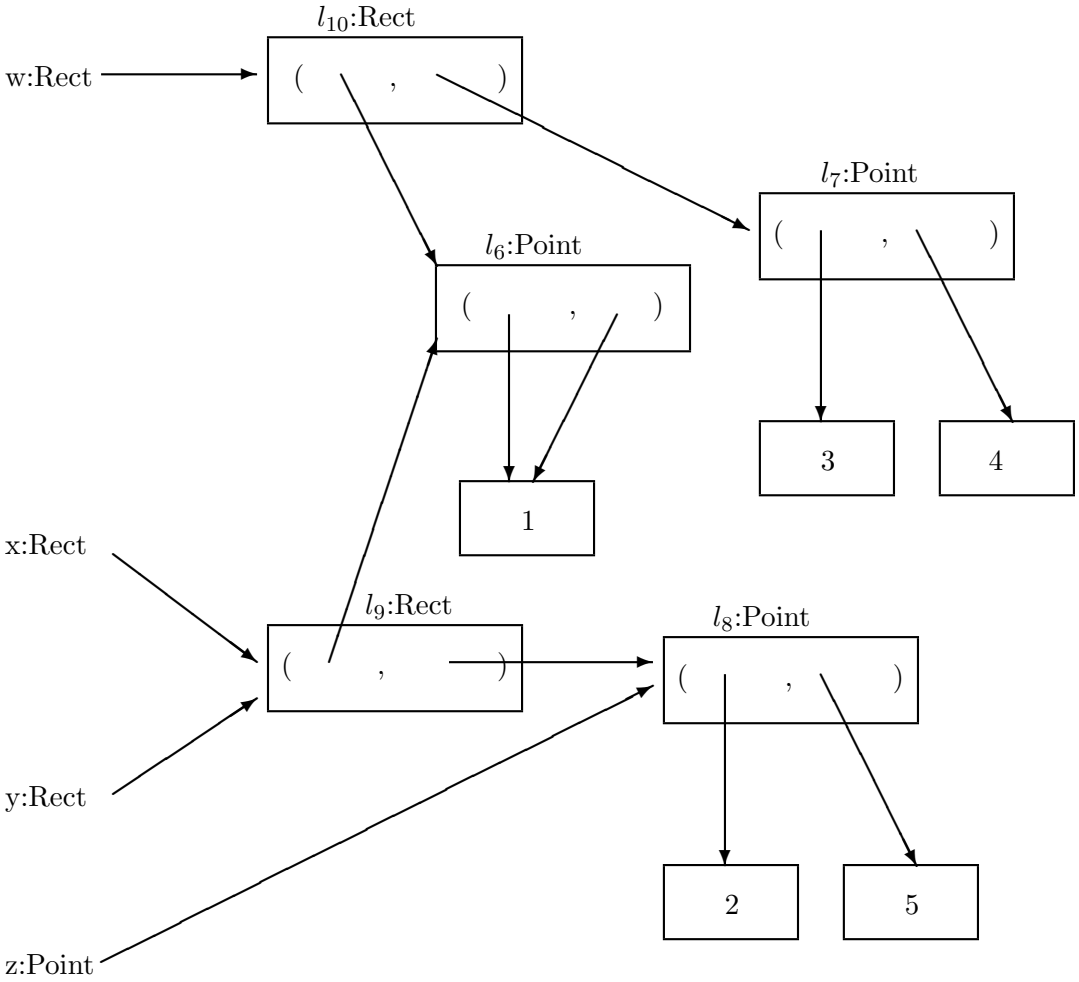

Figure 4: Picture of the state over the algebra E.

$$
\begin{array}{lcl}
\mathit{Unit}^D & \stackrel{\text{def}}{=} & \{*\} \\
\mathit{Boolean}^D & \stackrel{\text{def}}{=} & \{\mathit{true}, \mathit{false}\} \\
\mathit{Integer}^D & \stackrel{\text{def}}{=} & \{0, 1, -1, 2, -2, \ldots\} \\
\mathit{PointSort}^D & \stackrel{\text{def}}{=} & \{(v_x, v_y) \mid v_x, v_y \in \mathit{Integer}^E\} \\
\mathit{RectSort}^D & \stackrel{\text{def}}{=} & \{(l_{bl}, l_{tr}) \mid l_{bl}, l_{tr} \in \mathit{LOCS}^E_{\texttt{Point}}\}
\end{array}
$$

Figure 5: Carrier sets for the example Σ_E-algebra, D.

In the Σ_E-algebra E described above everything is modelled as an object; that is all values are stored in locations. We now describe another Σ_E-algebra, D, where the visible types are modelled as abstract values directly instead of objects. This is closer to the semantics of hybrid object-oriented languages, such as C++, CLOS, and Eiffel.

The carrier sets for D is given in Figure 5. We use *nextFree* and *alloc* as defined for E, but with D substituted for E everywhere in their definitions. The operations of D are shown in Figure 6.

A state like the one in Figure 4, but for the algebra D, is shown in Figure 7. Since the visible types are modelled as values in D, the components of $l_6 : \texttt{Point}$ in D are not shared locations. However, the aliasing a program can observe, while respecting data abstraction is the same. We will describe the observable aliasing relationships that are in these figures below. But first we give an algebraic characterization of how the states in the two figures are similar.

3 Simulation Relations

Simulation relations on algebras help us to decide when one algebra behaves like another [8] [9] [10]. If the simulating algebra corresponds to a faster implementation, that implementation may be used to replace the implementation that corresponds to the simulated algebra. (See [11] for more intuition and an application to subtyping in object-oriented programming languages.)

To define simulation relations one might think that the appropriate notion would simply relate elements in the carrier set of one algebra to those in another. However, that would not take the locations in our algebras into account. Neither can one simply relate locations, since one must take the abstract values stored in the locations into account if the relationships are to preserve observable behavior. Finally, it is not enough to relate locations together with

Operations

$$
\begin{array}{lcl}
\texttt{()}^D((),\sigma) & \stackrel{\text{def}}{=} & (inVALS(*),\sigma) \\
\texttt{true}^D((),\sigma) & \stackrel{\text{def}}{=} & (inVALS(true),\sigma) \\
\texttt{false}^D((),\sigma) & \stackrel{\text{def}}{=} & (inVALS(false),\sigma) \\
\texttt{and}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & (inVALS(v_1 \wedge v_2),\sigma) \\
\texttt{or}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & (inVALS(v_1 \vee v_2),\sigma) \\
\texttt{not}^D(inVALS(v),\sigma) & \stackrel{\text{def}}{=} & (inVALS(\neg(v)),\sigma) \\
\texttt{0}^D((),\sigma) & \stackrel{\text{def}}{=} & (inVALS(0),\sigma) \\
\texttt{1}^D((),\sigma) & \stackrel{\text{def}}{=} & (inVALS(1),\sigma) \\
\texttt{add}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & (inVALS(v_1+v_2),\sigma) \\
\texttt{mult}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & (inVALS(v_1\times v_2),\sigma) \\
\texttt{negate}^D(inVALS(l),\sigma) & \stackrel{\text{def}}{=} & (inVALS(-v_1),\sigma) \\
\texttt{equal}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & (inVALS(v_1=v_2),\sigma) \\
\texttt{mkPoint}^D((inVALS(v_1),inVALS(v_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Point}]((v_1,v_2),\sigma) \\
\texttt{abscissa}^D(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1,v_2)=\sigma(l) \textbf{ in } (inVALS(v_1),\sigma) \\
\texttt{ordinate}^D(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1,v_2)=\sigma(l) \textbf{ in } (inVALS(v_2),\sigma) \\
\texttt{addX}^D((inLOCS(l^{\texttt{Point}}),inVALS(v^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1,v_2)=\sigma(l^{\texttt{Point}}) \textbf{ in} \\
 & & (inVALS(*),[l^{\texttt{Point}}\mapsto(v_1+v^{\texttt{Int}},v_2)]\sigma) \\
\texttt{addY}^D(inLOCS(l^{\texttt{Point}}),inVALS(v^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1,v_2)=\sigma(l^{\texttt{Point}}) \textbf{ in} \\
 & & (inVALS(*),[l^{\texttt{Point}}\mapsto(v_1,v_2+v^{\texttt{Int}})]\sigma) \\
\texttt{mkRect}^D((inLOCS(l_1),inLOCS(l_2)),\sigma) & \stackrel{\text{def}}{=} & alloc[\texttt{Rect}]((l_1,l_2),\sigma) \\
\texttt{botLeft}^D(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2)=\sigma(l) \textbf{ in } (inLOCS(l_1),\sigma) \\
\texttt{topRight}^D(inLOCS(l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2)=\sigma(l) \textbf{ in } (inLOCS(l_2),\sigma) \\
\texttt{horizMove}^D((inLOCS(l^{\texttt{Rect}}),inVALS(v^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr})=\sigma(l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (inVALS(v'),\sigma')= \\
 & & \quad \texttt{addX}^D((inLOCS(l_{bl}),inVALS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & \textbf{let } (inVALS(v''),\sigma'')= \\
 & & \quad \texttt{addX}^D((inLOCS(l_{tr}),inLOCS(l^{\texttt{Int}})),\sigma') \textbf{ in} \\
 & & (inVALS(*),\sigma'') \\
\texttt{vertMove}^D((inLOCS(l^{\texttt{Rect}}),inVALS(l^{\texttt{Int}})),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr})=\sigma(l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (inVALS(v'),\sigma')= \\
 & & \quad \texttt{addY}^D((inLOCS(l_{bl}),inVALS(l^{\texttt{Int}})),\sigma) \textbf{ in} \\
 & & \textbf{let } (inVALS(v''),\sigma'')= \\
 & & \quad \texttt{addY}^D((inLOCS(l_{tr}),inLOCS(l^{\texttt{Int}})),\sigma') \textbf{ in} \\
 & & (inVALS(*),\sigma'')
\end{array}
$$

Figure 6: Operations for the example Σ_E-algebra, D.

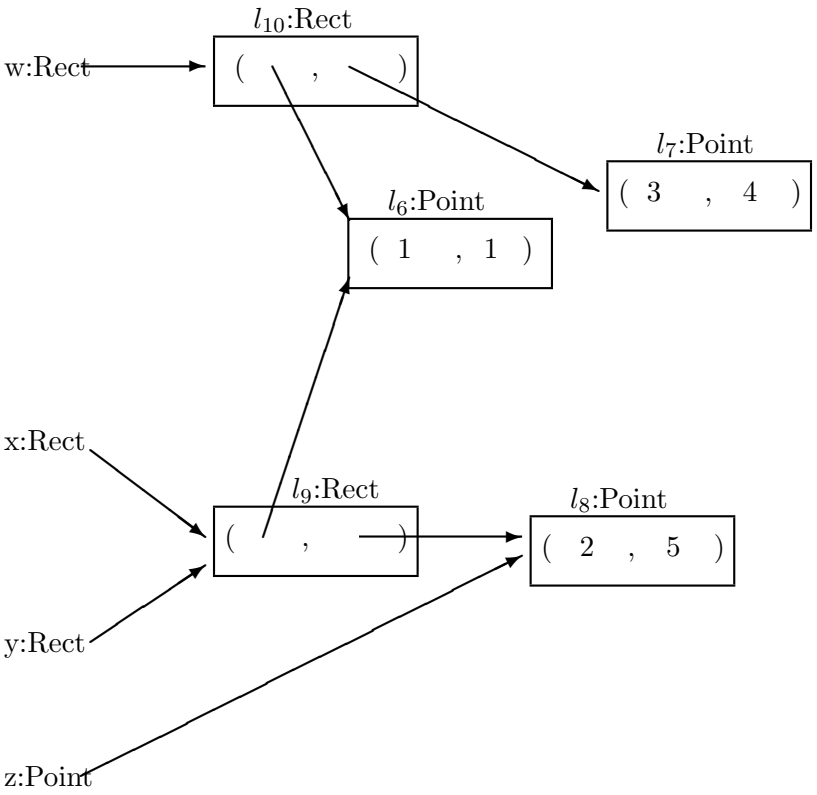

Figure 7: Picture of the state over the algebra D.

stores over the algebras, since that would not take aliasing in the environment into account. So the formulation of simulation relations we present relates states over one algebra to states over another algebra.

Definition 3.1 (simulation relation) *Let C and A be Σ-algebras. A* Σ-simulation relation $\mathcal{R}$ from C to A *is a binary relation on states*

$$\mathcal{R} \quad \subseteq \quad STATE[C] \times STATE[A]$$

such that for each $(\eta_C, \sigma_C) \in STATE[C]$ and for each $(\eta_A, \sigma_A) \in STATE[A]$, the following properties hold:

well-formed: $(\eta_C, \sigma_C)\, \mathcal{R}\, (\eta_A, \sigma_A) \Rightarrow \mathrm{dom}(\eta_C) \subseteq \mathrm{dom}(\eta_A)$,

bindable: *for each type T, for each identifier $x : T$, and for each identifier $y : T \in$* $\mathrm{dom}(\eta_C)$,

$$(\eta_C, \sigma_C)\, \mathcal{R}\, (\eta_A, \sigma_A) \Rightarrow ([x \mapsto \eta_C(y)]\eta_C, \sigma_C)\, \mathcal{R}\, ([x \mapsto \eta_A(y)]\eta_A, \sigma_A), \qquad (6)$$

substitution: *for each tuple of types* $\vec{S}$*, for each type* T*, for each operation symbol* $g : \vec{S} \to T$*, for each tuple of identifiers* $\vec{x} : \vec{S} \in \mathrm{dom}(\eta_C)$*, and for each identifier* $y : T$*, if* $(r_C, \sigma'_C) = g^C(\eta_C(\vec{x}), \sigma_C)$ *and* $(r_A, \sigma'_A) = g^A(\eta_A(\vec{x}), \sigma_A)$*, then*

$$(\eta_C, \sigma_C)\,\mathcal{R}\,(\eta_A, \sigma_A) \Rightarrow ([y \mapsto r_C]\eta_C, \sigma'_C)\,\mathcal{R}\,([y \mapsto r_A]\eta_A, \sigma'_A). \tag{7}$$

shrinkable: *if* $(\eta'_C, \sigma'_C) \subseteq (\eta_C, \sigma_C)$*, and* $(\eta'_A, \sigma'_A) \subseteq (\eta_A, \sigma_A)$*, and* $\mathrm{dom}(\eta'_C) \subseteq \mathrm{dom}(\eta'_A)$*, then* $(\eta_C, \sigma_C)\,\mathcal{R}\,(\eta_A, \sigma_A) \Rightarrow (\eta'_C, \sigma'_C)\,\mathcal{R}\,(\eta'_A, \sigma'_A)$*,*

VIS-**identical:** *for each type* $T \in VIS$*, for each identifier* $x : T \in \mathrm{dom}(\eta_C)$*,*

$$(\eta_C, \sigma_C)\,\mathcal{R}\,(\eta_A, \sigma_A) \Rightarrow absVal(\eta_C(x), \sigma_C) = absVal(\eta_A(x), \sigma_A) \tag{8}$$

In the substitution property, $\vec{x} : \vec{S} \in \mathrm{dom}(\eta_C)$, means that for each i, $x_i : S_i \in \mathrm{dom}(\eta_C)$, and $\eta_C(\vec{x})$ means the tuple of $\eta_C(x_i)$ for each i. The tuples $\vec{S}$ and $\vec{x} : \vec{S}$ can be empty.

The *VIS*-identical property ensures that a simulation is the identity on the carrier sets of visible types, such as `Bool`. Such an assumption about the visible types amounts to requiring that the carrier set of each sort associated with a visible type is the same in each algebra.

In the shrinkable property, $(\eta'_C, \sigma'_C) \subseteq (\eta_C, \sigma_C)$ means that for all types T, and for all identifiers $x : T$, $x : T \in \mathrm{dom}(\eta'_C) \Rightarrow \eta'_C(x) = \eta_C(x)$ and for all locations $l : T$, $l : T \in \mathrm{dom}(\sigma'_C) \Rightarrow \sigma'_C(l) = \sigma_C(l)$.

As a trivial example, the identity relation on $STATE[E] \times STATE[E]$ is a Σ_E-simulation relation from our example algebra E to itself.

To build a more interesting example, we consider first a formalization of the notion of similarity of abstract values of two locations and then augment that with some conditions that ensure well-formedness and that only states with the same aliasing (in a sense to be made precise in the next section) are related. Given two stores over E and D, the function

$$\mathcal{S}' : (STORE[E] \times STORE[D]) \to ((DVAL^E \times DVAL^D) \to Boolean)$$

returns a relation that tests two denotable values for having the same abstract value in the corresponding stores. It is defined inductively by requiring locations of the immutable types to have equal abstract values and by requiring `Point` and `Rect` locations to have related abstract values in each component:

basis: For each type $T \in \{\texttt{Void}, \texttt{Bool}, \texttt{Int}\}$, for each pair of stores (σ_E, σ_D), for each $d_E \in DVAL_T^E$ and $d_D \in DVAL_T^D$:

$$\mathcal{S}'_T(\sigma_E, \sigma_D)(d_E, d_D) \stackrel{\text{def}}{=} absVal(d_E, \sigma_E) = absVal(d_D, \sigma_D). \tag{9}$$

Point: For each pair of stores (σ_E, σ_D), for each $l_E : \texttt{Point} \in \text{dom}(\sigma_E)$ such that $\sigma_E(l_E) = (l_x, l_y)$, and for each $l_D : \texttt{Point} \in \text{dom}(\sigma_D)$ such that $\sigma_D(l_D) = (v'_x, v'_y)$:

$$\begin{aligned} &\mathcal{S}'_{\texttt{Point}}(\sigma_E, \sigma_D)(inLOCS(l_E), inLOCS(l_D)) \\ &\stackrel{\text{def}}{=} \mathcal{S}'_{\texttt{Int}}(\sigma_E, \sigma_D)(inLOCS(l_x), inVALS(v'_x)) \\ &\quad \wedge \mathcal{S}'_{\texttt{Int}}(\sigma_E, \sigma_D)(inLOCS(l_y), inVALS(v'_y)). \end{aligned} \tag{10}$$

rectangle: For each pair of stores (σ_E, σ_D), for each $l : \texttt{Rect} \in \text{dom}(\sigma_E)$ such that $\sigma_E(l) = (l_{bl}, l_{tr})$ and for each $l' : \texttt{Rect} \in \text{dom}(\sigma_D)$ such that $\sigma_D(l') = (l'_{bl}, l'_{tr})$:

$$\begin{aligned} &\mathcal{S}'_{\texttt{Rect}}(\sigma_E, \sigma_D)(inLOCS(l), inLOCS(l')) \\ &\stackrel{\text{def}}{=} \mathcal{S}'_{\texttt{Point}}(\sigma_E, \sigma_D)(inLOCS(l_{bl}), inLOCS(l'_{bl})) \\ &\quad \wedge \mathcal{S}'_{\texttt{Point}}(\sigma_E, \sigma_D)(inLOCS(l_{tr}), inLOCS(l'_{tr})). \end{aligned} \tag{11}$$

We can relate two states only if the aliasing present in the first is mimicked in the second. To this end we introduce the aliasing graph of a state (η, σ) over E. This directed graph has as its nodes: the identifiers in $\text{dom}(\eta)$, the locations in $\text{dom}(\sigma)$ that have type `Point` or `Rect`. It has directed edges as follows:

- From an identifier x of type `Point` or `Rect` to a location l if $\eta(x) = inLOCS(l)$.
- From a location $l : \texttt{Rect}$ to locations $l', l'' : \texttt{Point}$ if $(l', l'') = \sigma(l)$.

We write $AliasG(\eta, \sigma)$ for this graph.

For example, the picture in Figure 8 is the aliasing graph of the state (η_E, σ_E), which is itself pictured in Figure 4. Figure 8 is also the aliasing graph of the state (η_D, σ_D), pictured in Figure 7.

We consider one aliasing graph to be mimicked by another if there is an injective graph homomorphism from the first to the second. Recall that if (N_1, E_1) and (N_2, E_2) are graphs, then $f = (f_n, f_e)$ is a graph homomorphism if and only if

$$(n, n') \in E_1 \Rightarrow f_e((n, n')) = (f_n(n), f_n(n')). \tag{12}$$

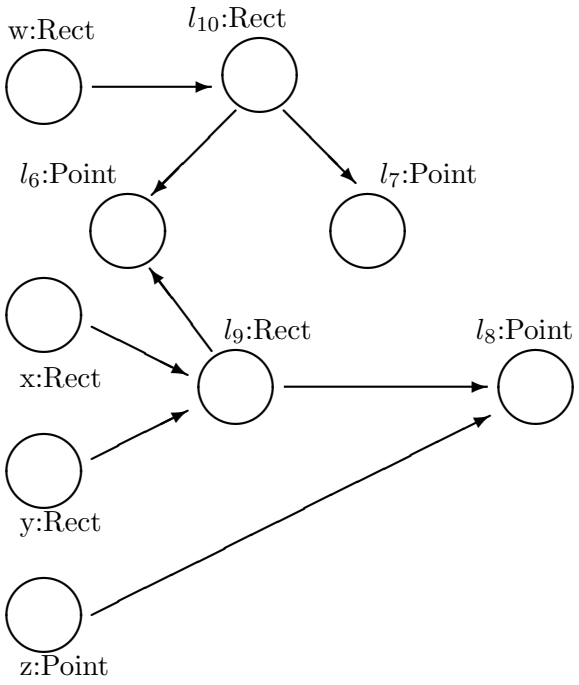

Figure 8: Aliasing structure for the state given in Figure 4

We now have enough machinery to define an interesting Σ_E-simulation relation from E to D, which we will call $\mathcal{R}'$. We let $(\eta_1, \sigma_1)\mathcal{R}'(\eta_2, \sigma_2)$ if and only if

- $\text{dom}(\eta_1) \subseteq \text{dom}(\eta_2)$,
- for each type T, for each $x : T \in \text{dom}(\eta_1)$, $\mathcal{S}'_T(\sigma_1, \sigma_2)(\eta_1(x), \eta_2(x))$ holds, so that the abstract values of x in both states are similar, and
- there is a injective[1] graph homomorphism from $AliasG(\eta_1, \sigma_1)$ to $AliasG(\eta_2, \sigma_2)$ that is the identity on $\text{dom}(\eta_1)$.

Requiring that there be an injective graph homomorphism ensures that aliasing for the mutable types `Point` and `Rect` is taken into account.

Lemma 3.2 *The relation $\mathcal{R}'$ is a Σ_E-simulation relation from E to D.*

Proof: Let $(\eta_E, \sigma_E) \in STATE[E]$, and $(\eta_D, \sigma_D) \in STATE[D]$ be given. We prove that $\mathcal{R}'$ has each of the defining properties of a simulation relation.

well-formed: By construction, $\mathcal{R}'$ is well-formed.

[1] A graph homomorphism, $f = (f_n, f_e)$ is *injective* if and only if f_n and f_e are injective functions on the sets of nodes and edges.

bindable: To show that $\mathcal{R}'$ satisfies the bindable property, let T be a type of Σ_E, let $x : T$ be an identifier, and let $y : T \in \mathrm{dom}(\eta_E)$ be given. Suppose that $(\eta_E, \sigma_E)\,\mathcal{R}'\,(\eta_D, \sigma_D)$. We show that $([x \mapsto \eta(y)]\eta_E, \sigma_E)\;\mathcal{R}'\;([x \mapsto \eta_D(y)]\eta_D, \sigma_D)$ by using the definition of $\mathcal{R}'$.

- By the construction of $\mathcal{R}'$, $\mathrm{dom}(\eta_E) \subseteq \mathrm{dom}(\eta_D)$, and thus $\mathrm{dom}([x \mapsto \eta_E(y)]\eta_E) \subseteq \mathrm{dom}([x \mapsto \eta_D(y)]\eta_D)$.
- By construction of $\mathcal{R}'$, the abstract values of each identifier z in both states are similar; in particular, the abstract values of y are similar:

  $$\mathcal{S}'_T(\sigma_E, \sigma_D)(\eta_E(y), \eta_D(y)). \tag{13}$$

  Since $([x \mapsto \eta_E(y)]\eta_E)(x) = \eta_E(y)$ and similarly for η_D, the abstract values of x in the extended environment are similar: the following holds:

  $$\mathcal{S}'_T(\sigma_E, \sigma_D)(([x \mapsto \eta_E(y)]\eta_E)(x), ([x \mapsto \eta_D(y)]\eta_D)(x)). \tag{14}$$

  Since the only difference between $[x \mapsto \eta_E(y)]\eta_E$ and η_E is the binding for x (and similarly for η_D), the following holds for all types S and for all $z : S$:

  $$\mathcal{S}'_T(\sigma_E, \sigma_D)(([x \mapsto \eta_E(y)]\eta_E)(z), ([x \mapsto \eta_D(y)]\eta_D)(z)). \tag{15}$$

- Also by construction of $\mathcal{R}'$, there is an injective graph homomorphism, $f = (f_n, f_e)$ from $\mathit{AliasG}(\eta_E, \sigma_E)$ to $\mathit{AliasG}(\eta_D, \sigma_D)$. We define an injective graph homomorphism $f' = (f'_n, f'_e)$ from $\mathit{AliasG}([x \mapsto \eta(y)]\eta, \sigma)$ to $\mathit{AliasG}([x \mapsto \eta'(y)]\eta', \sigma')$ as follows. Let $f'_n \stackrel{\text{def}}{=} [x \mapsto x]f_n$; that is, $f'_n(x) = x$. If $\eta_E(y) = \mathit{inLOCS}(l)$ and $\eta_D(y) = \mathit{inLOCS}(l')$, then let $f'_e \stackrel{\text{def}}{=} [(x, l) \mapsto (x, l')]f_e$. Since f' is defined by extending f, the following calculation suffices to show that f' is a graph homomorphism.

  $$\begin{array}{ll} & f'_e((x, l)) \\ = & \quad \langle \text{by definition of } f'_e \rangle \\ & (x, l') \\ = & \quad \langle \text{by definition of } f'_n \rangle \end{array}$$

$(f'_n(x), f'_n(l'))$

To show f' is injective we need to show f'_n, f'_e are injective. Since f_n is injective and since f_n is the identity on the identifiers in $\mathrm{dom}(\eta_E)$, f'_n is injective. The following calculation shows that f'_e is injective. Let u be any identifier, and l_u be any location in $LOCS^E$.

$f'_e((x,l)) = f'_e((u,l_u))$

$\Rightarrow$ $\langle$by the homomorphism property$\rangle$

$(f'_n(x), f'_n(l)) = (f'_n(u), f'_n(l_u))$

$\Rightarrow$ $\langle$by definition of equality for edges$\rangle$

$(f'_n(x) = f'_n(u)) \wedge (f'_n(l) = f'_n(l_u))$

$\Rightarrow$ $\langle$by injectivity of $f'_n\rangle$

$(x = u) \wedge (l = l_u)$

substitution: To show that $\mathcal{R}'$ satisfies substitution property we must show it for all operations. Since presenting the proof for all operations would require a considerable amount of space, we only give an example to show how the proof goes. As take as our example the operation `addX`. Let `p` : `Point` and `i` : `Int` be identifiers in $\mathrm{dom}(\eta_E)$. The substitution property can be shown in a similar way for the other operations. The only significant difference is for the mutator operations of `Rect`,where one must construct a new graph homomorphism and show that it is injective.

We assume without loss of generality that the following hold.

$$\begin{array}{rcl} \eta_E(\mathtt{p}) & = & inLOCS(l_E^{\mathtt{Point}}) \\ \eta_D(\mathtt{p}) & = & inLOCS(l_D^{\mathtt{Point}}) \\ \eta_E(\mathtt{i}) & = & inLOCS(l_E^{\mathtt{Int}}) \\ \eta_D(\mathtt{i}) & = & inVALS(v_D^{\mathtt{Int}}) \end{array}$$

Let a type T and an identifier $y : T$ be given.

Finally, assuming that

$$(\eta_E, \sigma_E)\, \mathcal{R}'\, (\eta_D, \sigma_D), \tag{16}$$

we must show that the following states, are related by $\mathcal{R}'$.

$$\begin{array}{lcl}
\texttt{addX}^E((\eta_E(\texttt{p}), \eta_E(\texttt{i})), \sigma_E) & = & \texttt{addX}^E((inLOCS(l_E^{\texttt{Point}}), inLOCS(l_E^{\texttt{Int}})), \sigma_E) \\
& \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = \sigma_E(l_E^{\texttt{Point}}) \textbf{ in} \\
& & \textbf{let } (inLOCS(l_x), \sigma'_E) = \\
& & \quad \texttt{add}^E((inLOCS(l_1), inLOCS(l_E^{\texttt{Int}})), \sigma_E) \textbf{ in} \\
& & alloc[\texttt{Void}](*, [l_E^{\texttt{Point}} \mapsto (l_x, l_2)]\sigma'_E) \\
\texttt{addX}^D((\eta_D(\texttt{p}), \eta_D(\texttt{i})), \sigma_D) & = & \texttt{addX}^D((inLOCS(l_D^{\texttt{Point}}), inVALS(v_D^{\texttt{Int}})), \sigma_D) \\
& \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = \sigma_D(l_D^{\texttt{Point}}) \textbf{ in} \\
& & (inVALS(*), [l_D^{\texttt{Point}} \mapsto (v_1 + v_D^{\texttt{Int}}, v_2)]\sigma_D)
\end{array}$$

We use σ_{D_r} and σ_{E_r} to refer to the stores in these final (or result) states. We will also refer to the store σ'_E to refer to the intermediate store defined in the course of evaluating $\texttt{addX}^E((\eta_E(\texttt{p}), \eta_E(\texttt{i})), \sigma_E)$.

With the above notation, what we have to show for the substitution property is as follows.

$$([y \mapsto *]\eta_E, \sigma_{E_r}) \; \mathcal{R}' \; ([y \mapsto *]\eta_D, \sigma_{D_r}) \tag{17}$$

We show that this holds by showing that it satisfies each of the properties in the definition of $\mathcal{R}'$.

- By hypothesis, $\text{dom}(\eta_E) \subseteq \text{dom}(\eta_D)$, and thus $\text{dom}([y \mapsto *]\eta_E) \subseteq \text{dom}([y \mapsto *]\eta_D)$.
- To show that the abstract values each identifier are similar in the final states, let T be a type and let $z : T$ be a identifier in $\text{dom}([y \mapsto *]\eta_E)$. We do this by cases.
  - Suppose that there is no path from z to $l_E^{\texttt{Point}}$ in the graph $AliasG(\eta_E, \sigma_E)$. Then the abstract value of z is unchanged. Hence, by the hypothesis:

$$\mathcal{S}'_T(\sigma_{E_r}, \sigma_{D_r})(\eta_E(z), \eta_D(z)). \tag{18}$$

  - If z is the identifier $\texttt{p} : \texttt{Point}$, we need to show $\mathcal{S}'_{\texttt{Point}}(\sigma_{E_r}, \sigma_{D_r})(\eta_E(\texttt{p}), \eta_D(\texttt{p}))$. It suffices to show the following two conditions, where the names of the locations and values are as in the defining expressions for the result states.

$$\mathcal{S}'_{\texttt{Int}}(\sigma'_E, \sigma_D)(inLOCS(l_x), inVALS(v_1 + v_D^{\texttt{Int}})) \tag{19}$$

$$\mathcal{S}'_{\texttt{Int}}(\sigma'_E, \sigma_D)(inLOCS(l_2), inVALS(v_2)) \tag{20}$$

    The second condition (20) holds by hypothesis, which implies that the abstract values of $\texttt{p}$ are related by $\mathcal{S}'_{\texttt{Int}}$ in the original states. The first condition

(19) follows from the definition of $\texttt{add}^E$, and the relationships of the abstract values in the original states.

– Otherwise z is a identifier that is not `p`, but from which `p` is reachable along some path of $AliasG(\eta_E, \sigma_E)$. Note that $AliasG([y \mapsto *]\eta_E, \sigma_{E_r}) = AliasG(\eta_E, \sigma_E)$. because the environments in the result state only differ by the binding of y, which does not create any edges, and because of the definition of `addX` in both algebras.

∗ If T, the type of z, is `Point` then z and `p` must denote the same location in $[y \mapsto *]\eta_E$. The same situation must hold in $[y \mapsto *]\eta_E$, because there is a homomorphism from $AliasG([y \mapsto *]\eta_E, \sigma_{E_r})$ to $AliasG([y \mapsto *]\eta_D, \sigma_{D_r})$. Since z denotes the same location as `p` in both final states, the above case shows that the abstract values of z are similar.

∗ If T is `Rect`, then one (or both!) of the corners of z is the same location as denoted by `p`. Again, the graph homomorphism on the aliasing graphs ensures that the same situation holds in both algebras, and thus the abstract values are similar.

- To show that there is an injective graph homomorphism on the result states, we observe that by definition of $\mathcal{R}'$ and the hypothesis, there is an injective homomorphism $f = (f_n, f_e)$ from $AliasG(\eta_E, \sigma_E)$ to $AliasG(\eta_D, \sigma_D)$. We construct a new injective homomorphism $f' = (f'_n, f'_e)$ from $AliasG([y \mapsto *]\eta_E, \sigma_{E_r})$ to $AliasG([y \mapsto *]\eta_D, \sigma_{D_r})$ by simply letting $f'_n = f_n$ and $f'_e = f_e$. This suffices because the only new identifiers does not have a type that matters, and because no new locations of type `Point` or `Rect` are introduced.

shrinkable: To show that $\mathcal{R}'$ satisfies shrinkable property, suppose $(\eta'_E, \sigma'_E) \subseteq (\eta_E, \sigma_E)$, $(\eta'_D, \sigma'_D) \subseteq (\eta_D, \sigma_D)$, and $\mathrm{dom}(\eta'_E) \subseteq \mathrm{dom}(\eta'_D)$. Suppose further that

$$(\eta_E, \sigma_E)\, \mathcal{R}'\, (\eta_D, \sigma_D). \tag{21}$$

We must show that

$$(\eta'_E, \sigma'_E)\, \mathcal{R}'\, (\eta'_D, \sigma'_D). \tag{22}$$

We do this by checking the defining properties of $\mathcal{R}'$.

- By hypothesis $dom(\eta'_E) \subseteq dom(\eta'_D)$.

- Let T be a type and let $x : T \in \mathrm{dom}(\eta_E)$ be given. Then by hypothesis, $\mathcal{S}'_T(\sigma_E, \sigma_D)(\eta_E(x), \eta_D(x))$ holds. If T is a visible type, then $absVal(\eta_E(x), \sigma_E) = absVal(\eta_D(x), \sigma_D)$, and so by definition of $\mathcal{S}'$, $\mathcal{S}'_T(\sigma'_E, \sigma'_D)(\eta'_E(x), \eta'_D(x))$. If T is `Point`, then the components of the abstract value are of visible type, and so are related as above. Similarly, if T is `Rect` the components are related points and thus are also related.
- By hypothesis, there is an injective graph homomorphism from $AliasG(\eta_E, \sigma_E)$ to $AliasG(\eta_D, \sigma_D)$ that is the identity on the identifiers of $\mathrm{dom}(\eta_E)$. The restriction of this homomorphism to the identifiers and locations in the smaller states is thus an injective homomorphism from $AliasG(\eta'_E, \sigma'_E)$ to $AliasG(\eta'_D, \sigma'_D)$.

VIS-**identical:** That $\mathcal{R}'$ satisfies the *VIS*-identical property follows from basis of the definition of $\mathcal{S}'$.

∎

The requirement that there be a graph homomorphism on the aliasing graphs in the definition of $\mathcal{R}'$ cannot be dropped. In section 5 we show that if two states are related then the aliasing between any two identifiers must be the same in both the states. (We give a precise definition of this below.) The injective homomorphism in the third condition ensures this necessary condition for simulation relations.

3.1 A Simple Term Language

The fundamental property of simulation relations is that simulation relationships are preserved by expressions and commands. To state and prove this, we first define an extremely simple term language, which we will call π. Because of our wish to deal with aliasing and mutation, the usual terms of nested operations are not adequate for observing our algebraic models.

In the definition of the semantics of π, we use notations from [12].

The syntax of π is given in Figure 9. For convenience in examples, we consider nullary operation symbols used in expressions to be syntactic sugar for their invocations. For example, we consider the expression `f(true, 1)` to be sugar for the technically correct but strange looking `f(true(), 1())`.

The input and output identifiers of a program are defined as follows.

inIds : Program → Declaration-List

Abstract Syntax:

P ∈ Program
D* ∈ Declaration-List
D ∈ Declaration
C ∈ Command
E* ∈ Expression-List
E ∈ Expression
I ∈ Identifier
g ∈ Operation
T ∈ Type-Symbol

P ::= `program (`D^*_1`) observes` D^*_2 `;` C
D* ::= | D D*
D ::= `const` V `:` T
C ::=E | `let const` I `:=` E `in` C `end` | C_1`;` C_2 | `if` E_1 `then` C_1 `else` C_2 `fi`
E ::= I | g`(`E*`)`
E* ::= | E* E

Figure 9: Syntax of π. Both D* and E* can be empty.

$inIds[\![$`program (`D^*_1`) returns` D^*_2 `;` C$]\!] = D^*_1$

$outIds$: Program → Declaration-List

$outIds[\![$`program (`D^*_1`) returns` D^*_2 `;` C$]\!] = D^*_2$

Each identifer $x : T$ in $outIds[\![\text{P}]\!]$ must have a visible type; that is, $x : T \in outIds[\![\text{P}]\!]$ implies $T \in VIS$.

The meaning of a program text, P, is a function that takes a Σ-algebra, A, and returns an A-observation with free identifiers from $inIds[\![\text{P}]\!]$. Observations are defined by the following domain equation.

$$OBSERVATION[A] = STATE[A] \to ANSWERS[A]_\perp \tag{23}$$

The domain of answers is a simply a mapping from each of the program's output identifiers to its abstract value. One can think of the program as printing these values (labeled by each identifier name).

$$ANSWERS[A] = IDS \xrightarrow{\text{fin}} VALS^A{}_\perp \tag{24}$$

The answer a particular program, P, returns has as its domain the output identifiers, $outIds[\![\mathrm{P}]\!]$, of that program. This is accomplished with an auxiliary function

output : Declaration-List $\to A : Alg(\Sigma) \to STATE[A] \to ANSWERS[A]$,

which is defined by the following properties. An identifier $x : T \in \mathrm{dom}(\mathrm{output}[\![\mathrm{D}*]\!](A)(\eta, \sigma))$ if and only if $x : T \in \mathrm{toSet}(\mathrm{D}*)$, where the auxiliary function "toSet" extracts the set of identifier-type pairs from the list D*. Furthermore, for all output identifiers declaration lists D*, for all Σ-algebras A, for all visible types T, for all identifiers $x : T$ in D*, and for all states (η, σ):

$$x : T \in \mathrm{dom}(\eta) \quad \Rightarrow \quad \mathrm{output}[\![\mathrm{D}*]\!](A)(\eta, \sigma)(x) = absVal(\eta(x), \sigma) \tag{25}$$

$$x : T \notin \mathrm{dom}(\eta) \quad \Rightarrow \quad \mathrm{output}[\![\mathrm{D}*]\!](A)(\eta, \sigma)(x) = \bot. \tag{26}$$

An observation can bind one of the output identifiers by an assignment command. So the denotation of a program is defined as follows.

$\mathcal{P}$: Program $\to A : Alg(\Sigma) \to OBSERVATION[A]$
$\mathcal{P}[\![\texttt{program (}\mathrm{D}*_1\texttt{) observes }\mathrm{D}*_2\texttt{ ; }\mathrm{C}]\!](A)(s) = \mathrm{output}[\![\mathrm{D}*_2]\!](A)(\mathcal{C}[\![\mathrm{C}]\!](A)(s))$

The meaning of a command text is a function that takes a Σ-algebra, A, and returns an A-command denotation with free identifiers from $\mathrm{D}*_1$, where $\mathrm{D}*_1$ is as above.

$$COMMAND[A] \quad = \quad ENV[A] \to STORE[A] \to STORE[A] \tag{27}$$

In a command "`let const` I `:=` E `in` C `end`" the type of I must be the same as E. The identifier "I" in a block command is bound by that command and can only be used within C. The expression in an `if`-command must have type `Bool`. For an algebra A, we also use l for an element of $LOCS^A$ of unspecified type and d for an element of $DVAL^A$ of unspecified type.

$\mathcal{C}$: Command $\to A : Alg(\Sigma) \to COMMAND[A]$
$\mathcal{C}[\![\mathrm{E}]\!](A)(\eta)(\sigma) = \mathbf{let}\ (d, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}\ \ \sigma'$
$\mathcal{C}[\![\texttt{let const I: T := E in C end}]\!](A)(\eta)(\sigma) =$
$\quad \mathbf{let}\ (d, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta)(\sigma)\ \mathbf{in}\ \ (\mathcal{C}[\![\mathrm{C}]\!](A)([\mathrm{I} \mapsto d]\eta)(\sigma'))$
$\mathcal{C}[\![\mathrm{C}_1\texttt{ ; }\mathrm{C}_2]\!](A)(\eta)(\sigma) = \mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta, \mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta)(\sigma))$
$\mathcal{C}[\![\texttt{if }\mathrm{E}_1\texttt{ then }\mathrm{C}_1\texttt{ else }\mathrm{C}_2\texttt{ fi}]\!](A)(\eta)(\sigma) =$
$\quad \mathbf{let}\ (d, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta)(\sigma)\ \mathbf{in}$

$$absVal(d, \sigma') \rightarrow (\mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta)(\sigma'))[\![(\mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta)(\sigma'))$$

An identifier in the above semantics is a constant in the sense that the value it denotes does not change. However, if that value is a location, then the contents of that location might change by an operation call. Thus, although the above semantics does not explicitly treat identifiers and assignment, identifiers can be modeled as mutable objects containing a value, as in the ref type of Standard ML. In such a model, getting the value of an identifier would be modeled by calling operations on such objects.

The meaning of an expression is found by either looking up the identifier in the environment, or by using the algebra to evaluate the operation with the given arguments. The auxiliary function "productize" converts a list of n locations into an n-tuple of locations, preserving the ordering.

$$\mathcal{E} : \text{Expression} \rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow DVAL^A \times STORE[A]$$
$$\mathcal{E}[\![\mathrm{I}]\!](A)(\eta, \sigma) = (\eta[\![\mathrm{I}]\!], \sigma)$$
$$\mathcal{E}[\![\mathrm{g(E^*)}]\!](A)(\eta, \sigma) = \textbf{let } (\hat{d}, \sigma') = \mathcal{E}*[\![\mathrm{E^*}]\!](A)(\eta, \sigma) \textbf{ in } \ \mathrm{g}^A(\text{productize}(\hat{d}), \sigma')$$

The meaning of a list of expressions is a list of locations together with the store that results from their evaluation. The expressions in the list are evaluated from left to right.

$$\mathcal{E}* : \text{Expression-List} \rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow (List((DVAL^A)) \times STORE[A])$$
$$\mathcal{E}*[\![]\!](A)(\eta, \sigma) = (nil, \sigma)$$
$$\mathcal{E}*[\![\mathrm{E^*\ E}]\!](A)(\eta, \sigma) =$$
$$\quad \textbf{let } (\hat{d}, \sigma') = \mathcal{E}*[\![\mathrm{E^*}]\!](A)(\eta, \sigma) \textbf{ in}$$
$$\quad\quad \textbf{let } (d_n, \sigma'') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma') \textbf{ in}$$
$$\quad\quad\quad (\text{addToEnd}(\hat{d}, d_n), \sigma'')$$

The auxiliary function "addToEnd" adds a location to the end of a list of locations; It satisfies the following property for all $1 \leq i \leq \text{length}(\hat{d})$.

$$\begin{aligned} &\text{productize}(\text{addToEnd}(\hat{d}, d')) \downarrow i \\ &\quad = \quad (i = \text{length}(\hat{d})) \rightarrow d'[\![(\text{productize}(\hat{d}) \downarrow i) \end{aligned} \tag{28}$$

Recall that our definition of command denotation and observation mentions a notion of "free identifiers." We can define this notion for π as follows. For a non-empty list of expressions, E* E, $FV[\![\mathrm{E^*\ E}]\!] = FV[\![\mathrm{E^*}]\!] \cup FV[\![\mathrm{E}]\!]$; there are no free identifiers in the empty

list of expressions. For expressions, $FV[\![\mathrm{V}]\!] = \{\mathrm{I} : \mathrm{T}\}$ (assuming I is declared to have type T), and $FV[\![\mathrm{g(E^*)}]\!] = FV[\![\mathrm{E^*}]\!]$. The free identifiers of a command are the identifiers of its subexpressions, except that

$$FV[\![\texttt{let const I: T := E in C end}]\!] = (FV[\![\mathrm{E}]\!] \cup FV[\![\mathrm{C}]\!]) \setminus \{\mathrm{I} : \mathrm{T}\}. \tag{29}$$

The free identifiers of a program are declared

$$FV[\![\texttt{program (D*}_1\texttt{) returns D*}_2\texttt{ ; C}]\!] = \mathrm{toSet}(\mathrm{D^*}_1). \tag{30}$$

We require that the free identifiers of the command C in a program be a subset of the free identifiers of the set of free identifiers of the entire program.

3.2 The Fundamental Theorem

The following lemma says that simulation relations are preserved by expression evaluation in related states.

Lemma 3.3 *Let C and A be Σ-algebras. Let $\mathcal{R}$ be a Σ-simulation relation from C to A.*

For all $(\eta_C, \sigma_C) \in STATE[C]$ and for all $(\eta_A, \sigma_A) \in STATE[A]$ such that $\mathrm{dom}(\eta_C) \subseteq \mathrm{dom}(\eta_A)$, for each type T, for each expression $\mathrm{E} : T$ such that $FV[\![\mathrm{E}]\!] \subseteq \mathrm{dom}(\eta_C)$, for each variable $y : T$, if $(d_C, \sigma'_C) = \mathcal{E}[\![\mathrm{E}]\!](C)(\eta_C, \sigma_C)$ and $(d_A, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A)$, then

$$(\eta_C, \sigma_C)\, \mathcal{R}\, (\eta_A, \sigma_A) \Rightarrow ([y \mapsto d_C]\eta_C, \sigma'_C)\, \mathcal{R}\, ([y \mapsto d_A]\eta_A, \sigma'_A)$$

Proof: (by induction on the structure of E).

Let $(\eta_C, \sigma_C) \in STATE[C]$ and $(\eta_A, \sigma_A) \in STATE[A]$ be given such that $\mathrm{dom}(\eta_C) \subseteq \mathrm{dom}(\eta_A)$. Let T and $\mathrm{E} : T$ be given such that $FV[\![\mathrm{E}]\!] \subseteq \mathrm{dom}(\eta_C)$. Let $y : T$ be given. Let $(d_C, \sigma'_C) = \mathcal{E}[\![\mathrm{E}]\!](C)(\eta_C, \sigma_C)$ and $(d_A, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A)$. Suppose that $(\eta_C, \sigma_C)\, \mathcal{R}\, (\eta_A, \sigma_A)$.

(basis) Suppose that E is a identifier I of type T, then the result follows from the assumption and the bindable property of simulation relations.

(inductive step) Suppose that E has the form $\mathrm{g(E^*)}$. Since $\mathrm{g(E^*)}$ has type T, it must be that $\mathrm{g} : \vec{S} \to T$, for some $\vec{S}$. The inductive hypothesis is that the lemma is true for each subexpression, E_i of type S_i in the list E*. That is, for all $(\eta_{C,i-1}, \sigma_{C,i-1}) \in STATE[C]$ and for all $(\eta_{A,i-1}, \sigma_{A,i-1}) \in STATE[A]$ such that $\mathrm{dom}(\eta_{C,i-1}) \subseteq \mathrm{dom}(\eta_{A,i-1})$, for each type S_i and for each expression E_i of type S_i such that $FV[\![\mathrm{E}_i]\!] \subseteq \mathrm{dom}(\eta_{C,i-1})$, for each identifier

$z_i : S_i$, if $(d_i, \sigma_{C,i}) = \mathcal{E}[\![\mathrm{E}_i]\!](C)(\eta_{C,i-1}, \sigma_{C,i-1})$ and $(e_i, \sigma_{A,i}) = \mathcal{E}[\![\mathrm{E}_i]\!](A)(\eta_{A,i-1}, \sigma_{A,i-1})$, then

$$\begin{array}{l}(\eta_{C,i-1}, \sigma_{C,i-1})\ \mathcal{R}\ (\eta_{A,i-1}, \sigma_{A,i-1}) \\ \quad\Rightarrow\quad ([z_i \mapsto d_i]\eta_{C,i-1}, \sigma_{C,i})\ \mathcal{R}\ ([z_i \mapsto e_i]\eta_{A,i-1}, \sigma_{A,i})\end{array}$$

The plan is to apply the inductive hypothesis for each expression in E*; for each i, binding the result of E_i to a distinct fresh identifier z_i. The resulting states are related by the inductive hypothesis, and then the substitution property gives the desired result. More precisely, we construct new states $(\eta_{C,n}, \sigma_{C,n})$ and $(\eta_{A,n}, \sigma_{A,n})$ such that the following hold for each $1 \le i \le n$:

$$(\hat{d}_n, \sigma_{C,n}) = \mathcal{E}*[\![\mathrm{E}^*]\!](C)(\eta_C, \sigma_C) \tag{31}$$

$$\eta_{C,n}(z_i) = \mathrm{productize}(\hat{d}_n) \downarrow i \tag{32}$$

$$(\hat{e}, \sigma_{A,n}) = \mathcal{E}*[\![\mathrm{E}^*]\!](A)(\eta_A, \sigma_A) \tag{33}$$

$$\eta_{A,n}(z_i) = \mathrm{productize}(\hat{e}_n) \downarrow i \tag{34}$$

$$(\eta_{C,n}, \sigma_{C,n})\ \mathcal{R}\ (\eta_{A,n}, \sigma_{A,n}) \tag{35}$$

Once this is done, the following calculation proves the inductive step.

$\mathcal{E}[\![\mathrm{g}(\mathrm{E}^*)]\!](C)(\eta_C, \sigma_C)$

$=$ $\langle$by definition of $\mathcal{E}$ and $\sigma_{C,n}\rangle$

let $(\hat{d}_n, \sigma_{C,n}) = \mathcal{E}*[\![\mathrm{E}^*]\!](C)(\eta_C, \sigma_C)$ **in** $\mathrm{g}^C(\mathrm{productize}(\hat{d}_n), \sigma_{C,n})$

$=$ $\langle$by definition of $\eta_{C,n}\rangle$

let $(\hat{d}_n, \sigma_{C,n}) = \mathcal{E}*[\![\vec{z}]\!](C)(\eta_{C,n}, \sigma_{C,n})$ **in** $\mathrm{g}^C(\mathrm{productize}(\hat{d}_n), \sigma_{C,n})$

$=$ $\langle$by definition of $\mathcal{E}\rangle$

$\mathcal{E}[\![\mathrm{g}(\vec{z})]\!](C)(\eta_{C,n}, \sigma_{C,n})$

$\mathcal{R}$ $\langle$by the substitution property and Equation (35)$\rangle$

$\mathcal{E}[\![\mathrm{g}(\vec{z})]\!](A)(\eta_{A,n}, \sigma_{A,n})$

$=$ $\langle$by the same reasoning as above$\rangle$

$\mathcal{E}[\![\mathrm{g}(\mathrm{E}^*)]\!](A)(\eta_A, \sigma_A)$

So it remains to construct the states $(\eta_{C,n}, \sigma_{C,n})$ and $(\eta_{A,n}, \sigma_{A,n})$ and the lists $\hat{d}_n$ and $\hat{e}_n$ with the required properties. These are constructed by induction on n.

For the basis, if $n = 0$, then E* is empty and we let $(\eta_{C,0}, \sigma_{C,0}) = (\eta_C, \sigma_C)$, $(\eta_{A,0}, \sigma_{A,0}) = (\eta_A, \sigma_A)$, $\hat{d}_0 = nil$, $\hat{e}_0 = nil$. The required properties hold trivially.

For the inductive step, suppose that E* is $\mathrm{E}_1 \cdots \mathrm{E}_{j-1}\ \mathrm{E}_j$ and further suppose that $(\eta_{C,j-1}, \sigma_{C,j-1})$, $(\eta_{A,j-1}, \sigma_{A,j-1})$, $\hat{d}_{j-1}$, and $\hat{e}_{j-1}$ satisfy the required properties. The re-

quired stores, along with locations that will be used shortly, are constructed as follows.

$$(d_j, \sigma_{C,j}) \stackrel{\text{def}}{=} \mathcal{E}[\![\mathrm{E}_j]\!](C)(\eta_{C,j-1}, \sigma_{C,j-1}) \tag{36}$$

$$(e_j, \sigma_{A,j}) \stackrel{\text{def}}{=} \mathcal{E}[\![\mathrm{E}_j]\!](A)(\eta_{A,j-1}, \sigma_{A,j-1}). \tag{37}$$

We define the required environments and lists as follows.

$$\hat{d}_j \stackrel{\text{def}}{=} \mathrm{addToEnd}(\hat{d}_{j-1}, d_j) \tag{38}$$

$$\hat{e}_j \stackrel{\text{def}}{=} \mathrm{addToEnd}(\hat{e}_{j-1}, e_j) \tag{39}$$

$$\eta_{C,j} \stackrel{\text{def}}{=} [z_j \mapsto d_j]\eta_{C,j-1} \tag{40}$$

$$\eta_{A,j} \stackrel{\text{def}}{=} [z_j \mapsto e_j]\eta_{A,j-1}. \tag{41}$$

The properties required of the constructed states and lists are verified as follows. To show that $\hat{d}_j$ and $\sigma_{C,j}$ have the required properties we calculate as follows.

$\mathcal{E}*[\![\mathrm{E}_1 \cdots \mathrm{E}_{j-1}\ \mathrm{E}_j]\!](C)(\eta_C, \sigma_C)$

$=$ $\quad\langle$by definition of $\mathcal{E}*$ and inductive hypothesis$\rangle$

let $(d_j, \sigma_{C,j}) = \mathcal{E}[\![\mathrm{E}_j]\!](C)(\eta_C, \sigma_{C,j-1})$ **in** $(\mathrm{addToEnd}(\hat{d}_{j-1}, d_j), \sigma_{C,j})$

$=$ $\quad\langle$by by definition of $(\hat{d}_j, \sigma_{C,j})\rangle$

$(\hat{d}_j, \sigma_{C,j})$

Similarly, $\hat{e}_j$ and $\sigma_{A,j}$ have the required properties.

Let $1 \leq i \leq j$ be given. Then we can verify the required property of $\eta_{C,j}$ as follows.

$\eta_{C,j}(z_i)$

$=$ $\quad\langle$by definition of $\eta_{C,j}$ and environment extension$\rangle$

$(z_i = z_j) \to d_j[\![\eta_{C,j-1}(z_i)$

$=$ $\quad\langle$by inductive hypothesis for $\eta_{C,j-1}$ and $\hat{d}_{j-1}\rangle$

$(z_i = z_j) \to d_j[\![\mathrm{productize}(\hat{d}_{j-1}) \downarrow i$

$=$ $\quad\langle$by distinctness of the $z_k\rangle$

$(i = j) \to d_j[\![\mathrm{productize}(\hat{d}_{j-1}) \downarrow i$

$=$ $\quad\langle$by Equation (28)$\rangle$

$\mathrm{productize}(\mathrm{addToEnd}(\hat{d}_{j-1}, d_j)) \downarrow j$

$=$ $\quad\langle$by construction of $\hat{d}_j\rangle$

$\mathrm{productize}(\hat{d}_j) \downarrow j$

Similarly, $\eta_{A,j}$ has the required property.

Equation (35) thus follows directly from the main inductive hypothesis, because of the inductive assumption that $(\eta_{C,j-1}, \sigma_{C,j-1})\mathcal{R}(\eta_{A,j-1}, \sigma_{A,j-1})$. ∎

We do not know whether the converse of the above lemma holds.

The following theorem extends the above lemma to show that simulation relations are preserved by commands in π.

Theorem 3.4 (fundamental theorem) *Let B and A be Σ-algebras. Let $\mathcal{R}$ be a Σ-simulation relation from B to A.*

For all $(\eta_B, \sigma_B) \in STATE[B]$, for all $(\eta_A, \sigma_A) \in STATE[A]$, for all commands C *that type check,*

$$(\eta_B, \sigma_B)\, \mathcal{R}\, (\eta_A, \sigma_A) \Rightarrow (\eta_B, \mathcal{C}[\![\mathrm{C}]\!](B)(\eta_B)(\sigma_B))\, \mathcal{R}\, (\eta_A, \mathcal{C}[\![\mathrm{C}]\!](A)(\eta_A)(\sigma_A)).$$

Proof: (by induction on the structure of C).

Let $(\eta_B, \sigma_B) \in STATE[B]$, let $(\eta_A, \sigma_A) \in STATE[A]$, and let C be given. Suppose that $(\eta_B, \sigma_B)\, \mathcal{R}\, (\eta_A, \sigma_A)$.

(basis) There are two cases, expressions and assignments.

Suppose that C is an expression, E. Let $(d_B, \sigma'_B) = \mathcal{E}[\![E]\!](B)(\eta_B, \sigma_B)$, and let $(d_A, \sigma'_A) = \mathcal{E}[\![E]\!](A)(\eta_A, \sigma_A)$. Then we can show the result as follows.

$(\eta_B, \sigma_B)\, \mathcal{R}\, (\eta_A, \sigma_A)$
$\Rightarrow$ $\langle$by previous lemma$\rangle$
$([y \mapsto d_B]\eta_B, \sigma'_B)\, \mathcal{R}\, ([y \mapsto d_A]\eta_A, \sigma'_A)$
$\Rightarrow$ $\langle$by shrinkable property of simulation relations$\rangle$
$(\eta_B, \sigma'_B)\, \mathcal{R}\, (\eta_A, \sigma'_A)$
$\Leftrightarrow$ $\langle$by definition of command denotation of an expression$\rangle$
$(\eta_B, \mathcal{C}[\![\mathrm{C}]\!](B)(\eta_B)(\sigma_B))\, \mathcal{R}\, (\eta_A, \mathcal{C}[\![\mathrm{C}]\!](A)(\eta_A)(\sigma_A))$

(inductive step) Assume, inductively, that the result holds for all subcommands of C. There are three cases.

1. Suppose C is "`let const` I:T `=` E `in` C_1 `end`". Then by the Lemma 3.3 the first formula in the following calculation holds.

   (**let** $(v_B, \sigma'_B) = \mathcal{E}[\![\mathrm{E}]\!](B)(\eta_B, \sigma_B)$ **in** $([I \mapsto v_B]\eta_B, \sigma'_B)$)
   $\mathcal{R}$ (**let** $(v_A, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A)$ **in** $([I \mapsto v_A]\eta_A, \sigma'_A)$)
   $\Rightarrow$ $\langle$by the inductive hypothesis$\rangle$

$$
\begin{array}{ll}
 & ([I \mapsto v_B]\eta_B, \textbf{let } (v, \sigma'_B) = \mathcal{E}[\![\mathrm{E}]\!](B)(\eta_B, \sigma_B) \textbf{ in } \mathcal{C}[\![\mathrm{C}_1]\!](B)([\mathrm{I} \mapsto v]\eta_B)(\sigma'_B)) \\
 & \mathcal{R} \quad ([I \mapsto v_A]\eta_A, \textbf{let } (v, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A) \textbf{ in } \mathcal{C}[\![\mathrm{C}_1]\!](A)([\mathrm{I} \mapsto v]\eta_A)(\sigma'_A)) \\
\Rightarrow & \quad \langle \text{by by the shrinkable property} \rangle \\
 & (\eta_B, \textbf{let } (v, \sigma'_B) = \mathcal{E}[\![\mathrm{E}]\!](B)(\eta_B, \sigma_B) \textbf{ in } \mathcal{C}[\![\mathrm{C}_1]\!](B)([\mathrm{I} \mapsto v]\eta_B)(\sigma'_B)) \\
 & \mathcal{R} \quad (\eta_A, \textbf{let } (v, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A) \textbf{ in } \mathcal{C}[\![\mathrm{C}_1]\!](A)([\mathrm{I} \mapsto v]\eta_A)(\sigma'_A)) \\
\Leftrightarrow & \quad \langle \text{by definition of } \mathcal{C}_\Sigma \rangle \\
 & (\eta_B, \mathcal{C}[\![\texttt{let const I: T = E in } \mathrm{C}_1]\!](B)(\eta_B)(\sigma_B)) \\
 & \mathcal{R} \quad (\eta_A, \mathcal{C}[\![\texttt{let const I: T = E in } \mathrm{C}_1]\!](A)(\eta_A)(\sigma_A))
\end{array}
$$

2. Suppose C is "C_1; C_2". Let $s_B = (\eta_B, \sigma_B)$ and $s_A = (\eta_A, \sigma_A)$. Then we can build up the result from the hypothesis as follows.

$$
\begin{array}{ll}
 & s_B \mathcal{R} s_A \\
\Rightarrow & \quad \langle \text{by the inductive hypothesis} \rangle \\
 & (\eta_B, \mathcal{C}[\![\mathrm{C}_1]\!](B)(\eta_B)(\sigma_B)) \mathcal{R} (\eta_A, \mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta_A)(\sigma_A)) \\
\Rightarrow & \quad \langle \text{by the inductive hypothesis} \rangle \\
 & (\eta_B, \mathcal{C}[\![\mathrm{C}_2]\!](B)(\eta_B)(\mathcal{C}[\![\mathrm{C}_1]\!](B)(\eta_B)(\sigma_B))) \mathcal{R} (\eta_A, \mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta_A)(\mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta_A)(\sigma_A))) \\
\Leftrightarrow & \quad \langle \text{by definition} \rangle \\
 & (\eta_B, \mathcal{C}[\![\mathrm{C}_1; \mathrm{C}_2]\!](B)(\eta_B)(\sigma_B)) \mathcal{R} (\eta_A, \mathcal{C}[\![\mathrm{C}_1; \mathrm{C}_2]\!](A)(\eta_A)(\sigma_A))
\end{array}
$$

3. Suppose C is "if E then C_1 else C_2". Let y : `Bool` be a fresh variable. Let $(d_B, \sigma'_B) = \mathcal{E}[\![\mathrm{E}]\!](B)(\eta_B, \sigma_B)$ and $(d_A, \sigma'_A) = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta_A, \sigma_A)$. Then by the Lemma 3.3

$$([y \mapsto d_B]\eta_B, \sigma'_B) \mathcal{R} ([y \mapsto d_A]\eta_A, \sigma'_A). \tag{42}$$

Since `Bool` is a visible type, and $\mathcal{R}$ is *VIS*-identical,

$$absVal(d_B, \sigma'_B) = absVal(d_A, \sigma'_A). \tag{43}$$

Hence the result of the test is the same in both B and in A. So starting with Equation (42), the result is shown as follows.

$$
\begin{array}{ll}
 & ([y \mapsto d_B]\eta_B, \sigma'_B) \mathcal{R} ([y \mapsto d_A]\eta_A, \sigma'_A) \\
\Rightarrow & \quad \langle \text{by the shrinkable property of simulation relations} \rangle \\
 & (\eta_B, \sigma'_B) \mathcal{R} (\eta_A, \sigma'_A) \\
\Rightarrow & \quad \langle \text{by the inductive hypothesis} \rangle \\
 & ((\eta_B, \mathcal{C}[\![\mathrm{C}_1]\!](B)(\eta_B)(\sigma'_B)) \mathcal{R} (\eta_A, \mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta_A)(\sigma'_A))) \\
 & \wedge \quad ((\eta_B, \mathcal{C}[\![\mathrm{C}_2]\!](B)(\eta_B)(\sigma'_B)) \mathcal{R} (\eta_A, \mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta_A)(\sigma'_A))) \\
\Leftrightarrow & \quad \langle \text{by Equation (43)} \rangle \\
 & (\eta_B, (absVal(d_B, \sigma'_B) \to \mathcal{C}[\![\mathrm{C}_1]\!](B)(\eta_B)(\sigma'_B) | \mathcal{C}[\![\mathrm{C}_2]\!](B)(\eta_B)(\sigma'_B))) \\
 & \mathcal{R} \quad (\eta_A, (absVal(d_A, \sigma'_A) \to \mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta_A)(\sigma'_A) | \mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta_A)(\sigma'_A)))
\end{array}
$$

$\Leftrightarrow$ $\langle$by definition of $\mathcal{C}\rangle$
$\mathcal{C}[\![(\eta_B, \text{if E then } C_1 \text{ else } C_2 \text{ fi}(B)(\eta_B)(\sigma_B))]\!]$
$\mathcal{R}$ $(\eta_A, \mathcal{C}[\![\text{if E then } C_1 \text{ else } C_2 \text{ fi}]\!](A)(\eta_A)(\sigma_A))$

∎

The following corollary to the above theorem follows directly, using the *VIS*-identical property of simulation relations, and the semantics of programs.

Corollary 3.5 *Let B and A be Σ-algebras. Let $\mathcal{R}$ be a Σ-simulation relation from B to A.*

For all $(\eta_B, \sigma_B) \in STATE[B]$, for all $(\eta_A, \sigma_A) \in STATE[A]$, for all programs P *that type check,*

$$(\eta_B, \sigma_B)\, \mathcal{R}\, (\eta_A, \sigma_A) \Rightarrow \mathcal{P}[\![\mathrm{P}]\!](B)(\eta_B, \sigma_B) = \mathcal{P}[\![\mathrm{P}]\!](A)(\eta_A, \sigma_A).$$

∎

4 Observable aliasing

Aliasing depends not only on objects and identifiers that share locations, but also on the behavior of the operations that manipulate the objects. For example, a CLU program cannot tell whether the integer 2 is represented by several copies of a bit pattern or by an immutable location containing that bit pattern. In the example program state depicted in Figure 4, the integer location $l_1^{\texttt{Int}}$ is shared by both components of the point $l_6^{\texttt{Point}}$, but there is no way to observe this sharing.

4.1 Observations, Commands, and Free Identifiers

We formalize aliasing in a way that is independent of any particular programming language. That is, although our definitions apply to aliasing and mutation as observed by the language π, they also would apply to subsets of π and to other languages. To make our definitions independent of the programming language, we parameterize our definitions by sets of observations and command denotations. The reader may then specialize these definitions to all of π, a subset of π, or to some other language.

Because the domain $STATE[A]$ is defined by reference to the algebra A, not the programming language, the domains $OBSERVATION[A]$ and $COMMAND[A]$, defined in Equations (23) and (27), are meaningful for both π, subsets of π, and other programming languages. We only need the following assumptions about the domain $ANSWERS[A]$,

which is specific to each programming language. We assume that we can compare elements of the domain $ANSWERS[A]$ for equality (=), and that for all Σ-algebras A and B, $ANSWERS[A] = ANSWERS[B]$. This last assumption is plausible, if one considers the answer domain to be fixed by the programming language. It amounts to saying that the carrier sets of the visible types are fixed.

A command or observation has a set of free identifiers that it can access. The set of free identifiers is important because to observe aliasing between two identifiers $x : T$ and $y : S$ one must be able to mutate x and observe the change through y, and vice versa. One must be careful not to use any other identifiers to do the mutation or observation. Otherwise if $z : U$ is aliased to y but x has nothing to do with y, then "mutating x" by a command that has z as a free variable could simply act on the object through z.

To speak precisely about such matters, we make the following definitions. If a command's set of free identifiers is X, the command needs a state that has those identifiers defined in its environment. So if $X \subseteq \mathrm{dom}(\eta)$ (i.e., for each type T, $X_T \subseteq \mathrm{dom}(\eta_T)$), we call a state (η, σ) an *X-state*. An observation is restricted to returning answers.

Definition 4.1 (command denotation, observation) *Let X be a $TYPES$-indexed set of variable symbols. Let A be a Σ-algebra. An* A-command denotation with free identifiers from X *is an element of $COMMAND[A]$ that takes a X-environment, a X-store and returns a X-store. An* A-observation with free identifiers from X *is an element of $OBSERVATION[A]$ that takes a X-state and returns an element of $ANSWERS[A]$.*

For example, in π, the denotation of a program P applied to an algebra A, $\mathcal{P}[\![P]\!](A)$ is an A-observation with free identifiers from $FV[\![\mathrm{P}]\!]$. Similarly, the denotation of a command C applied to an algebra A is an A-command denotation with free identifiers from $FV[\![\mathrm{C}]\!]$.

4.2 Mutation

As a first step towards such a definition of observable aliasing, we define when a command mutates an identifier. Our intuition is that a command mutates an identifier if there is an observation that shows a change in the identifier. Thus the "bits" may change without mutation taking place in our sense, if there is no way to observe it. Changes to the representation of an object that cannot be observed are *benevolent side effects* [13]. Even the abstract value may change without there being a mutation, if the abstract values before and after the command's execution are observably equivalent.

Definition 4.2 (mutates an identifier, mutable identifier) *Let A be a Σ-algebra. Let the state, $(\eta, \sigma) \in STATE[A]$, be given. Let $T \in TYPES$ be a type symbol. Let $x : T$ be a identifier in* $\mathrm{dom}(\eta)$.

Then c mutates $x : T$ in (η, σ) with respect to O *if and only if either*

- *O is a set of A-observations with free identifiers from $\{x : T\}$ and c is an A-command denotation with free identifiers from* $\mathrm{dom}(\eta)$, *or*
- *O is a set of A-observations with free identifiers from* $\mathrm{dom}(\eta)$ *and c is an A-command denotation with free identifiers from $\{x : T\}$,*

and for some $o \in O$, $o(\eta, c(\eta)(\sigma)) \neq o(\eta, \sigma)$.

Let C be a set of A-command denotations. Then $x : T$ is mutable in (η, σ) with respect to C and O *if and only if there is some $c \in C$ that mutates $x : T$ with respect to O.*

Notice that either the free identifiers of the command or the free identifiers of the observations are limited to the identifier in question, while the other is essentially unrestricted. It is not sensible to allow both to have access to all identifiers, for then the denotation of the command `z := 1` could be said to mutate the identifier `x`, because one could also observe the change in `z`. Also recall that an observation returns an answer, which is a mapping from identifiers of visible types to abstract values.

It might seem that every identifier is mutable, because it can be assigned to; but the point of the definition of when an identifier in mutable is to say when the given commands can mutate it in a way that can be observed. That is, the definition of a mutable identifier is more a description of the sets of commands and observations than it is a description of the identifier.

Example 4.3 *Consider the algebra E and the state (η_E, σ_E) given in Figure 4. In π, $\mathcal{C}[\![$`addX(z,1)`$]\!](E)$ is an E-command denotation with free identifiers from $\{$`z:Point`$\}$. Call this command denotation c_E. Consider the observation*

$$\mathcal{P}[\![\texttt{program(z:Point) returns y:Int; y := abscissa(z)}]\!](E),$$

which is an E-observation with free identifiers from $\{$`z:Point`$\}$. Call this observation o_E. Then c_E mutates `z:Point` *in (η_E, σ_E) with respect to $\{o_E\}$. It also follows that* `z` *is mutable in (η_E, σ_E) with respect to $\{c_E\}$ and $\{o_E\}$.*

Similarly, $\mathcal{C}[\![$`vertMove(y,1)`$]\!](E)$, which is an E-command denotation with free identifiers from $\{$`y:Rect`$\}$, mutates `z:Point` *in (η_E, σ_E) with respect to $\{o_E\}$.*

In the above example, the observations are limited to the identifier in question. This suffices for most normal types. However, one can imagine types for which another object is needed to observe the mutation. Allowing the observations to have access to other identifiers is then required. For example, imagine a "deposition" object, which anyone can write, but which only someone with holding a capability object can read. Let `dep: Depo` be a identifier that is bound to the deposition object, and let `key: Capa` be the capability identifier. Imagine that the object bound to `dep` contains the location of the object to which `key` is bound. Then the `read` operation on the type `Depo` would take a `Depo` object and a `Capa` object and compare the locations of the `Capa` object and the one it contains; if they are the same, the `read` operation returns the value, otherwise it returns a constant. In this case, one cannot observe the mutation of `dep` without an observation that has access to `key`.

We can extend the above definition of mutation of identifiers to mutation of objects. Technically we define what it means for a location to be mutated. If there is no identifier bound to the location, then a fresh identifier is bound to the location for purposes of observation; in this case the command has no access to the identifier (and the observations have access only to this identifier). If there is a identifier already bound to the location, that can also be used to make observations, but then the definition reduces to mutation of that identifier, with the stipulation that the command leave the identifier bound to the location.

Definition 4.4 (mutates an object, mutable object) *Let A be a Σ-algebra. Let a state $(\eta, \sigma) \in STATE[A]$ be given. Let $T \in TYPES$ be a type symbol. Let $l : T$ be a location in* $\mathrm{dom}(\sigma)$*. Let $x : T$ be a identifier. Let c be an A-command denotation and let O be a set of A-observations.*

Then c mutates $l : T$ in (η, σ) with respect to O *if and only if*

- *the free identifiers of c are a subset of* $\mathrm{dom}(\eta)$*,*
- *c mutates $x : T$ in $([x \mapsto inLOCS(l)]\eta, \sigma)$ with respect to O, and*
- *if $(\eta', \sigma') = c([x \mapsto inLOCS(l)]\eta, \sigma)$, then $\eta'(x) = inLOCS(l)$.*

Let C be a set of A-command denotations. Then $l : T$ is mutable in (η, σ) with respect to C and O *if and only if there is some $c \in C$ that mutates $l : T$ with respect to O.*

The second condition in the definition of when a command mutates an object may seem a bit odd. But consider that in π the first assignment to an output identifier of a program extends the environment. So this condition is necessary to ensure that the observations can still access the location through the identifier x.

In Example 4.3, the E-command denotation, $\mathcal{C}[\![\texttt{addX(z,1)}]\!](E)$ mutates the location $l_8 : \texttt{Point}$ (of Figure 4) in (η_E, σ_E) with respect to $\{o_E\}$. To see this, for the identifier x of the definition, take `z: Point`, which works because it is the only free identifier in o_E. In the same example, the E-command denotation $\mathcal{C}[\![\texttt{vertMove(y,1)}]\!](E)$ mutates the locations $l_8 : \texttt{Point}$ and $l_9 : \texttt{Rect}$ in (η_E, σ_E) with respect to $\{o_E\}$.

There are two parameters in our definitions of "mutates." The state cannot be disregarded, as a identifier or object may not be able to make state changes in certain states, but can make changes in other states. Such behavior may seem strange, but as an example consider an object that can count up to 10, and thereafter, when the `increment` operation is applied, simply stays at 10. Error states of objects often have this character. We also consider the state to determine the algebra in the definition, as the range of the environment, and the domain and range of the store are both parts of the algebra. The set of observations also cannot be ignored. For example, no identifier or object can be mutated with respect to the empty set of observations.

Nevertheless, it is often convenient to fix a particular set of observations. So when the set of observations is clear from context it may safely be omitted. One way to fix a set of observations is by fixing a particular programming language, such as π. In such a case the set of observations is the set of all observations (appropriate for the definition) that are denotations of programs in the language; for example, if we are discussing the language π and say that c mutates $x : T$ in (η, σ), we mean with respect to the denotations of all programs in π that have the free identifiers from $\{x : T\}$. This kind of contextual abbreviation is particularly helpful when saying that a location is mutable in a given state, since the appropriate sets of command and observation denotations are given by all appropriate commands and programs in the language.

Example 4.5 *In this example, consider the language π. Then all the locations of types* `Point` *and* `Rect` *are mutable in the state (η_E, σ_E) of Figure 4.*

If an object $l : T$ is not mutable in a state, (η, σ), with respect to some set of observations, O, then $l : T$ is *immutable in (η, σ) with respect to O*.

We often describe a type as mutable or immutable if it has mutable or immutable objects. To be precise about this we have to bring states and observations into the definition. Let A be an algebra, and let C and O be sets of A-command denotations and A-observations. A *type T is mutable with respect to C and O* if and only if there is some state (η, σ) over A, and some location $l : T \in \mathrm{dom}(\sigma)$ such that $l : T$ is mutable in (η, σ) with respect to O; otherwise, T is *immutable with respect to C and O*.

Even more generally, if one has a fixed set of algebras, $SPEC$, such as the semantics of some specification, and functions f_C and f_O from algebras to sets of command denotations and observations, such as the functions that map an algebra A to the sets of all denotations of appropriate commands and programs of π, then one says that a type T is mutable with respect to f_C and f_O if for some $A \in SPEC$, T is mutable with respect to $f_C(A)$ and $f_O(A)$; otherwise one says that T is immutable with respect to f_C and f_O. For example, in the context of the language π, the type `Point` is mutable, and the type `Int` is immutable.

4.3 Aliasing

The intuition behind aliasing is that two identifiers are aliased when the mutation of one can be observed as mutation of the other. Note that this intuition allows identifiers of different types to be aliased. For example, if x denotes a stack of arrays and y denotes a set of arrays that share a component array, then x and y can be aliased despite the difference in types. To see this, one mutates the shared component array through either identifier.

Our definition is a bit more general than the above intuitive statement. We only require that a command with access limited to x can be observed to change y and vice versa; the command with access to x need not be observed to change x itself.

Definition 4.6 (observably aliased) *Let A be a Σ-algebra. Let $T, S \in TYPES$ be type symbols. Let $x : T$ and $y : S$ be identifiers. Let $(\eta, \sigma) \in STATE[A]$ be a $\{x : T, y : S\}$-state.*

Then $x : T$ and *$y : S$* are observably aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y *if and only if*

- *C_x and C_y are sets of A-command denotations with free identifiers from $\{x : T\}$ and $\{y : S\}$ respectively,*
- *O_x and O_y be sets of A-observations with free identifiers from $\{x : T\}$ and $\{y : S\}$ respectively,*
- *$y : S$ is mutable in (η, σ) with respect to C_x and O_y, and*

- $x : T$ *is mutable in* (η, σ) *with respect to* C_y *and* O_x.

Example 4.7 *Consider the algebra* E *and the state* (η_E, σ_E) *given in Figure 4. In* π, $\mathcal{C}[\![\texttt{addX(z,1)}]\!](E)$ *is an* E*-command denotation with free identifiers from* $\{\texttt{z:Point}\}$*; call this* $c_{\texttt{z}}$*, and let* $C_{\texttt{z}} = \{c_{\texttt{z}}\}$. *Let* $c_{\texttt{y}}$ *be* $\mathcal{C}[\![\texttt{horizMove(y,2)}]\!](E)$, *which is an* E*-command denotation with free identifiers from* $\{\texttt{y:Rect}\}$*; let* $C_{\texttt{y}} = \{c_{\texttt{y}}\}$. *For observing* `z`*, let*

$$o_{\texttt{z}} = \mathcal{P}[\![\texttt{program(z:Point) returns q:Int; q := abscissa(z)}]\!](E),$$

which is an E*-observation with free identifiers from* $\{\texttt{z:Point}\}$*; let* $O_{\texttt{z}} = \{o_{\texttt{z}}\}$. *For observing* `y`*, let*

$$o_{\texttt{y}} = \mathcal{P}[\![\texttt{program(y:Rect) returns r:Int; r := abscissa(topRight(y))}]\!](E),$$

which is an E*-observation with free identifiers from* $\{\texttt{y:Rect}\}$*; let* $O_{\texttt{y}} = \{o_{\texttt{y}}\}$. *Then* $\texttt{y} : \texttt{Rect}$ *and* $\texttt{z} : \texttt{Point}$ *are observably aliased in* (η_E, σ_E) *with respect to* $C_{\texttt{y}}$, $C_{\texttt{z}}$, $O_{\texttt{y}}$, *and* $O_{\texttt{z}}$.

If one of the sets C_x, C_y, O_x, or O_y are empty in the above definition, then $x : T$ and $y : S$ are not considered observably aliased. If no mutation can be effected or observed, then there is no observable aliasing. Hence an immutable object cannot be observably aliased with another object.

More interesting is that in a language like π, where identifiers denote values or objects, but identifier cells are not objects, for x and y to be aliased the commands in C_x and C_y must be able to invoke operations that accomplish mutation. This is because in π a simple assignment to a identifier cannot produce an observable effect on another object (or value that contains objects). That would not be the case in a language like C, where identifiers are considered locations.

The following theorem states that observable aliasing is reflexive and symmetric.

Theorem 4.8 *Let* A *be a* Σ*-algebra. Let* $T, S, U \in TYPES$ *be type symbols. Let* $(\eta, \sigma) \in STATE[A]$. *Let* $x : T \in \mathrm{dom}(\eta)$ *and* $y : T \in \mathrm{dom}(\eta)$ *be identifiers. Then:*

- *If* C_x *is a set to* A*-command denotations with free identifiers from* $\{x : T\}$, *if* O_x *is a set to* A*-observations with free identifiers from* $\{x : T\}$, *and if* $x : T$ *is mutable in* (η, σ) *with respect to* C_x *and* O_x, *then* $x : T$ *and* $x : T$ *are observably aliased in* (η, σ) *with respect to* C_x, C_x, O_x, *and* O_x.

- *The identifiers $x : T$ and $y : S$ are observably aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y if and only if $y : S$ and $x : T$ are observably aliased in (η, σ) with respect to C_y, C_x, O_y, and O_x.* ∎

In general observable aliasing is not a transitive relation. To see this consider commands and observations defined by π and the identifiers `x : Rect`, `w : Rect`, and `z : Point` in Figure 4. Then in the pictured state, (η_E, σ_E), identifiers `z` and `x` are observably aliased and `x` and `w` are observably aliased, but identifiers `z` and `w` are not observably aliased.

4.4 Contained and Component Objects

Traditionally, one says that an object l contains an object l' when the representation of l contains a pointer to l' [13]. In terms of our model, we could say that l contains l' when the abstract value of l contains the location l'. However, that definition would not be satisfactory, because it does not take into account the behavior of the objects. For example, it would say that the point l_8 in Figure 4 contains the integer 2, but it would not say the same thing about Figure 7. So to get a notion of object containment that is independent of particular representations, our definition is based solely on observations.

Our definition is an abstraction of examples such as records and arrays. An object contained in a record, for example, has none of its interface hidden, since the observations on the contained object can be composed with extraction of the object from the record. Since every mutation of the contained object is observable as a mutation of the record, we take this as the defining characteristic of contained objects. That is, our definition says that an object l_x contains an object l_y if all mutations of l_y can be observed as changes in l_x. This implies that immutable objects are never observable contained in other objects, since immutable objects have no observable connection with locations, but act as pure values.

Definition 4.9 (object containment) *Let A be a Σ-algebra. Let $T, S \in TYPES$ be type symbols. Let $(\eta, \sigma) \in STATE[A]$ be a state. Let $l_x : T \in \mathrm{dom}(\sigma)$ and $l_y : S \in \mathrm{dom}(\sigma)$ be locations.*

Then $l_x : T$ contains *$l_y : S$* in (η, σ) with respect to C_y, O_x, and O_y *if and only if for each c_y in C_y that mutates $l_y : S$ in (η, σ) with respect to O_y, c_y mutates $l_x : T$ with respect to O_x.*

For example, if one considers all applicable commands and observations written in π and the algebra E given in Figure 4, then the location l_9 : `Rect` contains l_8 : `Point` in (η_E, σ_E).

However, l_8 : `Point` does not contain l_9 : `Rect` in (η_E, σ_E). The E-command denotation, $\mathcal{C}[\![$`addX(botLeft(y),1)`$]\!](E)$ mutates l_9 but this mutation cannot be observed by programs that have only `z` : `Point` as a free identifier.

According to the above definition, every object contains itself. To distinguish the interesting components of an object from the object itself, we say that $l_x : T$ *properly contains* $l_y : S$ *in* (η, σ) *with respect to* C_y, O_x, *and* O_y if $l_x : T$ contains $l_y : S$ in (η, σ) with respect to C_y, O_x, and O_y, but $l_x \neq l_y$. We sometimes call properly contained objects component objects. The above examples of object containment are also examples of proper object containment.

4.5 Direct and Indirect Aliasing

We can make a gross distinction between two kinds of aliasing: direct and indirect. For example, in Figure 4 identifiers `x` and `y` are directly aliased, but the aliasing between `w` and `x` is indirect. Intuitively, direct aliasing of identifiers means that both denote the same location. However, it will not do to say that "identifiers x and y are directly aliased in (η, σ) if and only if they denote the same location in η," because the location might be immutable (e.g., an integer). So instead we say that two identifiers x and y directly aliased if all mutations of x can be observed through y and vice versa.

Definition 4.10 (directly aliased) *Let A be a Σ-algebra. Let $T, S \in TYPES$ be type symbols. Let $(\eta, \sigma) \in STATE[A]$ be a state. Let $l_x : T \in \mathrm{dom}(\sigma)$ and $l_y : S \in \mathrm{dom}(\sigma)$ be locations.*

Then $l_x : T$ and *$l_y : S$* are directly aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y *if and only if $l_x : T$ contains $l_y : S$ with respect to C_y, O_x, and O_y, and $l_y : S$ contains $l_x : T$ with respect to C_x, O_y, and O_x.*

Let $x : T$ and $y : S$ be identifiers such that $\eta(x) = inLOCS(l_x : T)$ and $\eta(y) = inLOCS(l_y : S)$. Then $x : T$ and *$y : S$* are directly aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y *if and only if $l_x : T$ and $l_y : S$ are directly aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y.*

Note that locations can be directly aliased without being identical. One way this can happen is if they both are "front ends" for some object that is the only object they contain.

Locations or identifiers that are observably aliased but not directly aliased are said to be *indirectly aliased*.

For example, if one considers all applicable commands and observations written in π and the algebra E given in Figure 4, then the identifiers $\texttt{x} : \texttt{Rect}$ and $\texttt{y} : \texttt{Rect}$ are directly aliased in (η_E, σ_E). In the same state, the identifiers $\texttt{x} : \texttt{Rect}$ and $\texttt{z} : \texttt{Point}$ are indirectly aliased, and the identifiers $\texttt{w} : \texttt{Rect}$ and $\texttt{x} : \texttt{Rect}$ are indirectly aliased. Also $l_9 : \texttt{Rect}$ and $l_8 : \texttt{Point}$ are indirectly aliased in (η_E, σ_E), because l_8 is a component of l_9. The locations $l_9 : \texttt{Rect}$ and $l_{10} : \texttt{Rect}$ are indirectly aliased in (η_E, σ_E), although neither contains the other. The location $l_8 : \texttt{Point}$ is directly aliased to itself.

These direct and indirect aliasing relationships change as one changes the set of observations and commands. For example, consider for observations programs in π that do not use the operation `topRight`. Let $O_{\texttt{w}}$ be the set of all such observations with free identifiers from $\texttt{w} : \texttt{Rect}$, let $O_{\texttt{y}}$ be the set of all such observations with free identifiers from $\texttt{y} : \texttt{Rect}$. Let $C_{\texttt{w}}$ be the set of all command denotations in π with free identifiers from $\texttt{w} : \texttt{Rect}$, and let $C_{\texttt{y}}$ be the set of all command denotations in π with free identifiers from $\texttt{y} : \texttt{Rect}$. Then $\texttt{w}$ and $\texttt{y}$ are directly aliased in (η_E, σ_E) with respect to $C_{\texttt{w}}$, $C_{\texttt{y}}$, $O_{\texttt{w}}$, and $O_{\texttt{y}}$. In this setup, locations $l_{10} : \texttt{Rect}$ and $l_9 : \texttt{Rect}$ are directly aliased in (η_E, σ_E) with respect to $C_{\texttt{w}}$, $C_{\texttt{y}}$, $O_{\texttt{w}}$, and $O_{\texttt{y}}$, even though they are not identical.

The definition of direct aliasing for identifiers builds on several layers of definition. A more direct definition is given in the following lemma.

Lemma 4.11 *Let A be a Σ-algebra. Let $T, S \in TYPES$ be type symbols. Let $(\eta, \sigma) \in STATE[A]$ be a state. Let $x : T \in \mathrm{dom}(\eta)$ and $y : S \in \mathrm{dom}(\eta)$ be identifiers.*

Then $x : T$ and $y : S$ are directly aliased in (η, σ) with respect to C_x, C_y, O_x, and O_y if and only if for each $c_x \in C_x$ that mutates $x : T$ with respect to O_x, c_x mutates $y : S$ with respect to O_y, and for each $c_y \in C_y$ that mutates $y : S$ with respect to O_y, c_y mutates $x : T$ with respect to O_x.

4.6 Aliasing Graph

We can summarize the containment and aliasing relationships in a given state by a graph, called the aliasing graph of the given state. It is convenient to do this in the context of a particular programming language, and so the rest of this section takes place in the context of π.

Definition 4.12 (aliasing graph) *Let A be a Σ algebra. Let $(\eta, \sigma) \in STATE[A]$ be a state. For each type T, for each identifier $x : T$, let $f_C(x : T)$ be a set of A-command*

denotations with free identifiers from $\{x : T\}$*, and let* $f_O(x : T)$ *be a set of A-command observations with free identifiers from* $\{x : T\}$*.*

The aliasing graph of (η, σ) with respect to f_C and f_O *is a directed graph with nodes in* $\mathrm{dom}(\eta) \cup \mathrm{dom}(\sigma)$ *and edges defined as follows.*

- *For each type* T*, there is an edge* (x, l)*, from each identifier* $x : T \in \mathrm{dom}(\eta)$ *to the location* $l : T \in \mathrm{dom}(\sigma)$*, if* $\eta(x) = inLOCS(l)$ *and* $x : T$ *is mutable in* (η, σ) *with respect to* $f_C(x : T)$ *and* $f_O(x : T)$*.*

- *For each pair of types* T *and* S *there is an edge from each location* $l : T$ *to each location* $l' : S$ *if there are identifiers* $x : T$ *and* $y : S$ *such that* $l : T$ *properly contains* $l' : S$ *in* (η, σ) *with respect to* $f_C(x : T)$*,* $f_O(y : S)$*, and* $f_O(x : T)$*.*

Figure 8 shows the aliasing graph of the state (η_E, σ_E), which is pictured in Figure 4, with respect to the functions that map identifiers to the set of all E-command denotations and E-observations of commands and programs in π with that free identifier.

5 Discussion

5.1 Simulation and Aliasing

If we have a simulation relation that relates two states, then those states must have the similar aliasing graphs with respect to commands and observations defined by π. By similar aliasing graphs we mean that if two identifiers are observably aliased in one state then they are observably aliased in the other state. The idea of the proof is that a difference in aliasing could be observed, which would contradict the assumption that they are related by a simulation.

In the following theorem, when C_x is a set of commands written in π, we use the notation $C_x[A]$ to mean $\{\mathcal{C}[\![\mathrm{C}]\!](A) \mid \mathrm{C} \in C_x\}$. Similarly for sets of programs, O_x, the notation $O_x[A] = \{\mathcal{P}[\![\mathrm{P}]\!](A) \mid \mathrm{P} \in O_x\}$.

Theorem 5.1 *Let* B *and* A *be* Σ*-algebras. Let* $\mathcal{R}$ *be a* Σ*-simulation relation from* B *to* A*. Let states* $(\eta_B, \sigma_B) \in STATE[B]$ *and* $(\eta_A, \sigma_A) \in STATE[A]$ *be given. Let* T *and* S *be types. Let* $x : T$ *and* $y : S$ *be identifiers in* $\mathrm{dom}(\eta_B)$*. Let* $C_x \in$ Command *and* $C_y \in$ Command *be sets of commands in* π*, and let* $O_x \in$ Program *and* $O_y \in$ Program *be sets of programs in* π*.*

If $(\eta_B, \sigma_B)\ \mathcal{R}\ (\eta_A, \sigma_A)$*, then* $x : T$ *and* $y : T$ *are observably aliased in* (η_B, σ_B) *with respect to* $C_x[B]$*,* $C_y[B]$*,* $O_x[B]$*,* $O_y[B]$ *if and only if then* $x : T$ *and* $y : T$ *are observably*

aliased in (η_A, σ_A) *with respect to* $C_x[A]$, $C_y[A]$, $O_x[A]$, $O_y[A]$.

Proof: Suppose $(\eta_B, \sigma_B)\,\mathcal{R}\,(\eta_A, \sigma_A)$.

Suppose We show that $x : T$ and $y : T$ are observably aliased in (η_A, σ_A) with respect to $C_x[A]$, $C_y[A]$, $O_x[A]$, $O_y[A]$.

By definition $x : T$ and $y : S$ are observably aliased in (η_B, σ_B) with respect to $C_x[B]$, $C_y[B]$, $O_x[B]$, $O_y[B]$ if and only if:

- $C_x[B]$ and $C_y[B]$ are sets of B-command denotations with free identifiers from $\{x : T\}$ and $\{y : S\}$, respectively,
- $O_x[B]$ and $O_y[B]$ are sets of B-observations with free identifiers from $\{x : T\}$ and $\{y : S\}$, respectively,
- $y : S$ is mutable in (η_B, σ_B) with respect to $C_x[B]$, and $O_y[B]$, and
- $x : T$ is mutable in (η_B, σ_B) with respect to $C_y[B]$, and $O_x[B]$.

By the semantics of π, the first two items are true if and only if:

- $C_x[A]$ and $C_y[A]$ are sets of A-command denotations with free identifiers from $\{x : T\}$ and $\{y : S\}$, respectively, and
- $O_x[A]$ and $O_y[A]$ are sets of A-observations with free identifiers from $\{x : T\}$ and $\{y : S\}$, respectively.

The following calculation shows that for all commands $c_x \in C_x$, and for all programs $o_y \in O_y$ the composition of the program with the command gives the same results in B as in A.

$$
\begin{array}{ll}
 & (\eta_B, \sigma_B)\,\mathcal{R}\,(\eta_A, \sigma_A) \\
\Rightarrow & \quad \langle \text{by the fundamental theorem} \rangle \\
 & (\mathcal{C}[\![c_x]\!](B)(\eta_B)(\sigma_B))\,\mathcal{R}\,(\mathcal{C}[\![c_x]\!](A)(\eta_A)(\sigma_A)) \\
\Rightarrow & \quad \langle \text{by Corollary 3.5} \rangle \\
 & \mathcal{P}[\![o_y]\!](B)(\eta_B, \mathcal{C}[\![c_x]\!](B)(\eta_B, \sigma_B)) = \mathcal{P}[\![o_y]\!](A)(\eta_A, \mathcal{C}[\![c_x]\!](A)(\eta_A, \sigma_A))
\end{array}
$$

Similarly, by the hypothesis and Corollary 3.5, the following holds for all programs $o_y \in O_y$.

$$\mathcal{P}[\![o_y]\!](B)(\eta_B, \sigma_B) = \mathcal{P}[\![o_y]\!](A)(\eta_A, \sigma_A) \tag{44}$$

By definition, $y : S$ is mutable in (η_B, σ_B) with respect to $C_x[B]$ and $O_y[B]$, if and only if there is some command $c_x \in C_x$ and some program $o_y \in O_y$ such that

$$\mathcal{P}[\![o_y]\!](B)(\eta_B, \mathcal{C}[\![c_x]\!](B)(\eta_B, \sigma_B)) \neq \mathcal{P}[\![o_y]\!](B)(\eta_B, \sigma_B) \tag{45}$$

So by the calculations above and Equation (44), the above equation is true if and only if $y : S$ is mutable in (η_A, σ_A) with respect to $C_x[A]$ and $O_y[A]$.

In exactly the same way, $x : T$ is mutable in (η_B, σ_B) with respect to $C_y[B]$, and $O_x[B]$ if and only if $x : T$ is mutable in (η_A, σ_A) with respect to $C_y[A]$, and $O_x[A]$. ∎

Though the above theorem says that the observable aliasing of the identifiers in the two related states is identical, it is an open question whether there is any other necessary relationship between aliasing graphs of related algebras. Although we once thought there might be an injective graph homomorphism between the aliasing graphs of related states, this seems not to be the case. The problem is that the implementations of the abstract types in each algebra might use mutable locations in completely different ways.

Nevertheless, the above theorem does vindicate the intuition that simulation relations take aliasing into account.

5.2 Immutable Types with Mutable Components

Our treatment of mutation allows us to make sense out of an old puzzle: whether an immutable collection of mutable elements in mutable or immutable. To explain, in CLU [13], there is a type generator **sequence**. Being a type generator means that **sequence[int]** is a type and **sequence[array[int]]** is another type. The type `array[int]` has objects that are mutable; for example, one can store integers in the elements of an array. A CLU sequence is like an array in that it is an indexed collection of objects, but unlike an array, once the elements of a sequence are fixed, they cannot be changed. Thus clearly the type **sequence[int]** is immutable, because the objects cannot be changed. So the literature on CLU refers to types **sequence**[T] as immutable, for all types T.

However the type **sequence[array[int]]** is clearly mutable with respect to CLU programs. For example, if one has access to a identifier `x` of this type, then one can extract one of the array elements, store into it, and observe the change through `x`. So according to our definition, `x` is mutable in CLU.

The puzzle is how one can think of **sequence** as a generator of immutable types, when some of its instances are clearly mutable.

The traditional answer to this puzzle is that while mutation of the component objects of sequence may be possible, the abstract value of the sequence does not change. The idea here is that the abstract value of a **sequence[array[int]]** object is an ordered list of locations (of arrays). Although one can model sequences with such abstract values in our theory, such an answer does not refute the observations — one can still see the object changing.

We can provide a more satisfactory answer to this puzzle that still builds on the same intuition. The answer is that there is a formal sense in which an object of **sequence[array[int]]** is immutable. It is immutable with respect to all CLU programs that do not mutate the elements of the sequence — that is, with respect to CLU programs that treat the elements as objects without operations. Since this point of view is appropriate when considering the specification of **sequence**[T] for arbitrary types T, it is sensible to think of the types **sequence**[T] as immutable. We can adopt such a point of view, instead of being forced to consider all CLU programs, because our theory allow us to discuss mutation with respect to varying sets of observations.

The ability to consider varying sets of commands and observations is also useful in treating subtyping and protection, but those topics are outside the scope of this paper. Some work on extending the theory to subtyping is reported in [14].

6 Conclusions

The work in this paper provides a foundation for the model theory of mutable abstract data types.

Our models of abstract data types blend aspects of denotational semantics (locations, environment and store mappings) with traditional algebraic models. We believe that this blend gives a satisfactory foundation for the model theory of abstract data types with mutation. In support of this we offer a notion of homomorphic relation (our simulation relations) and show that it is preserved by commands in a simple term language. The addition of more realistic features to the language, such as loops, does not seem to destroy this fundamental property [14].

Careful examination of our example algebras shows that models like ours could have been used to model mutable types long ago. All that is needed to equationally characterize mutable types is to adopt the old denotational semantics trick of explicitly passing a store around. This is another sense in which our models unify denotational and algebraic techniques.

We also believe that our approach to describing the semantics of languages with abstract data types — computing over a model of the types — is a fruitful way to do semantics for such languages. Traditional denotational semantics "compiles" the definitions of abstract data types into an undifferentiated mass of functions, cartesian products, etc. (For example, see [6].) The mess that results from this "compilation" process is difficult to compare to the specifications of the abstract types, and difficult to extract from the rest of a program. If it were possible to take a CLU program and give meaning to the clusters in as an algebra in a way similar to what we have done, many of the algebraic tools that are useful in the study of absract data types could be applied in the study of such programs. Such a separation would also allow the semantics of the user-defined types to be studied in isolation from the rest of the program.

Because of our focus on observable behavior, mutation and aliasing are defined in a way that is representation independent. While the terms "mutation" and "aliasing" are commonly defined in terms of bits, we believe that our definitions are more appropriate when studying languages with data abstraction. Two identifiers that both point to an immutable object are not observably aliased. This point of view can be exploited to perform various optimizations; for example, sharing large immutable objects to save space, or representing small integers as bit patterns instead of as locations containing those bit patterns.

Finally, we have shown that our model theory ties in well our observational view of mutation and aliasing. In particular we have shown that our simulation relations, because they are relations on states, can deal with aliasing and preserve aliasing. Thus our simulation relations are ideal tools for the study of abstract data types with mutation and aliasing.

References

[1] R. Statman, "Logical relations and the typed λ-calculus," *Information and Control*, vol. 65, pp. 85–97, May/June 1985.

[2] G. T. Leavens and D. Pigozzi, "Typed homomorphic relations extended with subtypes," Tech. Rep. 91-14, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, June 1991. Appears in the proceedings of *Mathematical Foundations of Programming Semantics '91*, Springer-Verlag, Lecture Notes in Computer Science, volume 598, pages 144-167, 1992.

[3] J. M. Wing, "A two-tiered approach to specifying programs," Tech. Rep. TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[4] J. Chen, "The Larch/Generic interface language," tech. rep., Massachusetts Institute of Technology, EECS department, May 1989. The author's Bachelor's thesis. Available from John Guttag at MIT (guttag@lcs.mit.edu).

[5] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, vol. 114 of *Lecture Notes in Computer Science*. New York, N.Y.: Springer-Verlag, 1981.

[6] Scheifler, "A denotational semantics of CLU," Tech. Rep. TR-201, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1978.

[7] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972.

[8] O. Schoett, "Behavioural correctness of data representations," *Science of Computer Programming*, vol. 14, pp. 43–57, June 1990.

[9] T. Nipkow, "Non-deterministic data types: Models and implementations," *Acta Informatica*, vol. 22, pp. 629–661, Mar. 1986.

[10] G. T. Leavens and D. Pigozzi, "Typed homomorphic relations extended with subtypes," in *Mathematical Foundations of Programming Semantics '91* (S. Brookes, ed.), vol. 598 of *Lecture Notes in Computer Science*, pp. 144–167, New York, N.Y.: Springer-Verlag, 1992.

[11] G. T. Leavens, "Modular specification and verification of object-oriented programs," *IEEE Software*, vol. 8, pp. 72–80, July 1991.

[12] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*. Boston, Mass.: Allyn and Bacon, Inc., 1986.

[13] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass.: The MIT Press, 1986.

[14] K. K. Dhara, "Subtyping among mutable types in object-oriented programming languages," Master's thesis, Iowa State University, Department of Computer Science, Ames, Iowa, May 1992.