9-1994

Blended Algebraic and Denotational Semantics for ADT Languages with Mutable Objects

Gary T. Leavens
Iowa State University

Krishna Kishore Dhara
Iowa State University

Blended Algebraic and Denotational Semantics for ADT Languages with Mutable Objects

Abstract
This paper presents a semantics for a simple language that is a blend of algebraic models and traditional denotational semantics. In this semantics, implementations of user-defined abstract data types are "compiled" into an algebraic structure, which is used by the denotational part of the semantics whenever an operation of the data type is invoked. To show the utility of such a semantics, an algebraic characterization of simulation between states over such algebras is given, and it is shown that simulation is preserved by expressions and commands in the language. (Note: versions TR93-21 and TR93-21a were titled: "A Model Theory for Abstract Data Types with Mutable Objects (extended abstract)".)

Keywords
algebraic semantics, algebraic models, denotational semantics, abstract data type, objects, mutation, model theory, simulation relation

Disciplines
Programming Languages and Compilers | Theory and Algorithms

Blended Algebraic and Denotational Semantics for ADT Languages with Mutable Objects

Gary T. Leavens and Krishna Kishore Dhara

TR #93-21b
September, 1993 (Revised March, September 1994)

The original versions (TR93-21 and TR93-21a) of this report were titled "A Model Theory for Abstract Data Types with Mutable Objects (extended abstract)".

Keywords: algebraic semantics, algebraic models, denotational semantics, abstract data type, objects, mutation, 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 semantics.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Blended Algebraic and Denotational Semantics for ADT Languages with Mutable Objects

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

September 6, 1994

Abstract

This paper presents a semantics for a simple language that is a blend of algebraic models and traditional denotational semantics. In this semantics, implementations of user-defined abstract data types are "compiled" into an algebraic structure, which is used by the denotational part of the semantics whenever an operation of the data type is invoked. To show the utility of such a semantics, an algebraic characterization of simulation between states over such algebras is given, and it is shown that simulation is preserved by expressions and commands in the language.

1 Introduction

Abstract data types (ADTs) are important in programming, particularly in object-oriented (OO) programming. The recent interest in OO programming has focused renewed attention on problems of specifying and verifying such programs [2] [40] [32] [3] [28] [68] [67].

A key idea in the verification of OO programs is that of behavioral subtyping. Because a model of an ADT can be thought of as a mathematical abstraction of an ADT specification (or the code that implements it) [49] [35] [15] [14] [74] [5], several authors have used the variations on multi-sorted algebras as a formal framework for studying behavioral subtyping [7] [32] [30] [51] [33]. However, when put to such uses, multi-sorted algebras have two distinct problems:

- the theory is not directly applicable to the study of ADTs with *mutable* objects — objects with time-varying state,
- the theory is not directly connected to the study of programming languages, because no standard semantic description technique uses multi-sorted algebras.

The first problem arises because multi-sorted algebras are designed to be used as models of equations, for which referential transparency is fundamental. Hence, they do not have a notion of locations (object identities). For example, in the standard `Stack` ADT, the `push` method produces a new stack value, it does not mutate an object.

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

Denotational semantics [64] [57] [44] also is based on a referentially transparent model theory (the lambda calculus [9] [4]), but since the techniques of denotational semantics are designed to model programming languages, not equational specifications, it does not have any difficulty in modeling state, mutation, and aliasing. The essential technique used in denotational semantics to model state is to pass around a "store mapping" to simulate the computer's memory.

However, when one looks at the treatment of ADTs in the denotational semantics of programming languages such as CLU [56], or OO languages such as Smalltalk and CLOS [10], one searches in vain for something that is easily identifiable as an algebraic model of the ADTs. Despite the clearly identifiable modules that define ADTs in such languages, there is no algebraic structure that is identifiable as the denotation of the program modules that implement ADTs. Because there is nothing analogous to a multi-sorted algebra in the semantics of such programming languages, it is difficult to apply ideas from multi-sorted algebras (such as behavioral subtyping for immutable objects).

In our previous model-theoretic work on behavioral subtyping [34] [27] [33] [13], we have tried to use algebraic structures as the denotations of ADT specifications, while at the same time working with a denotational semantics. This combination was achieved by only using "half" of a programming language, the part that used objects to do things, and leaving out the part of the programming language that implemented ADTs. The denotational semantics would use an algebraic model of the ADTs desired for the program, and these models were obtained directly from ADT specifications. This trick bypassed the problem of how to give a denotational semantics of a programming language that used algebraic structures as the denotations of ADT code.

However, because the worlds of denotational semantics and algebraic model theory are so distant, it was not clear how our previous work could ever be connected with the reality of a "whole" programming language. Skeptical readers of our previous work have also wondered how the half of a programming language that was not shown would affect the half that computed over the algebraic structures.

In this paper we ignore subtyping to focus on a semantic technique that connects the worlds of algebraic and denotational semantics. Specifically, in Section 2 we give a programming language with ADTs, and a denotational semantics that has an algebraic structure as the denotation of its ADT implementations. The programming language studied is a simple one, with mutable objects and some OO features, but not subtyping or inheritance. The algebraic structures are modifications of multi-sorted algebras that allow for mutation and aliasing. To show how this style of denotational semantics can be used to study questions about ADTs, in Section 3 we adapt our notion of simulation relations for types with immutable objects to this setting. In Section 3 we also show that this notion of simulation is preserved by expressions and commands in this language. This kind of theorem is the key to model-theoretic study of subtyping (compare [30]). Following this we discuss related work in Section 4, future work and open problems in Section 5, and offer some conclusions in Section 6.

2 A Simple Language

This section presents the syntax and semantics of a simple language, called π. The semantics demonstrates the technique of giving a denotational semantics where the denotations of types and their associated methods are given as an algebraic structure. Following the

Abstract Syntax:

P ∈ Program	TD ∈ Type-Declaration
T, S, U ∈ Type-Name	MD ∈ Method-Declaration
F* ∈ Formal-List	F ∈ Formal
B ∈ Body	D ∈ Declaration
E ∈ Expression	g ∈ Method-Name
E* ∈ Expression-List	C ∈ Command
M ∈ Main	I ∈ Identifier
N ∈ Numeric-literal	

P ::= TD MD M
TD ::= | `type` I `fields (` F* `)` | TD_1 `;` TD_2
FD ::= | I : T | FD_1 `;` FD_2
T ::= I
MD ::= | `method` I `(` F* `)` : T `{` B `}` | MD_1 `;` MD_2
F* ::= | F F*
F ::= I : T
B ::= D C `;` `return` E
D ::= | `const` I : T = E | D_1 `;` D_2
E ::= I | N | `true` | `false` | `nothing` | g`(`E*`)` | `new` T`(`E*`)` | I_1 `.` I_2
g ::= I
E* ::= | E* E
C ::= E | I_1 `.` I_2 `:=` E | C_1`;` C_2 | `if` E_1 `then` C_1 `else` C_2 `fi`
M ::= `main` `{` `observe` D_1 C_1 `by` C_2 D_2 `}`

Figure 1: Abstract syntax of π. In concrete examples, we separate elements of formal lists and expression lists with commas.

semantics, we present an example of the semantics and discuss the technique.

2.1 Syntax and Overview of π

The abstract syntax of π is given in Figure 1. A program consists of type definitions, method definitions, and a main procedure. Informally the program runs by elaborating the declarations of types, and methods, and then running the main procedure.

The types the user can define are all represented by records; for simplicity, there is no built-in variant representation. The objects of such a type can be mutated if desired, because the fields can be assigned within a method by the field assignment command (of the form "I_1`.`I_2 `:=` E"). The field assignment command, and the object creation ("`new` T`(`E*`)`") and field access ("I_1 `.` I_2") expressions can only be used directly within methods; this provides a simple form of information hiding. For better information hiding, a module system could be added, but it would complicate the semantics.

The language is first-order: for simplicity there is no lambda abstraction or type polymorphism. Also for simplicity, there is no recursion. Methods return the result of the last expression in their bodies; if nothing is to be returned, the built-in constant `nothing` may be used to return the only value of the built-in type `Void`. (The constant `nothing` may also be used as a command, in which case it acts like `skip`.)

For simplicity, the only declaration form binds a name to the result of an expression (an object). Such bindings cannot be changed, and are therefore constant bindings. This is not a great restriction, however, as the programmer can define any desired type of variables by defining objects with one field. (This is similar to the way variables are treated in, for example, SML [41] [50].) For example, the following shows how one would write integer variables.

```
type IntVar fields (val: Int);
method mkIntVar(e:Int): Void { nothing; return new IntVar(e) };
method assign(v:IntVar, e:Int): Void { v.val := e; return nothing };
method read(v:IntVar): Int { nothing; return v.val };
```

The form of a main procedure is unusual, because it is split into two parts; the declaration and command between the keywords `observe` and `by` (D_1 and C_1) define a state which the following command and declaration (C_2 and D_2) observe. The final declaration, D_2, may only declare constants of types `Int` or `Bool`. For simplicity, the "output" of the main procedure (and hence the program) is defined to be the values of the constants in D_2.

The split of the main procedure into two parts is motivated by our studies of behavioral subtyping [32] [28] [30]. Informally, OO programmers often add new subtypes of existing types to a running program, with the expectation that once objects of the new subtypes are created the rest of the program will continue running as before. The behavior will be "enhanced" but nothing should break in the unchanged code. Thus the first declaration and command in the main procedure model the part of the program that is changed by the addition of new types (new type and method definitions would be added as well). The second command and declaration in the main procedure model the part of the program that is unchanged by the addition of the new types. In this paper, since we are ignoring subtyping, we will use this split to study representation independence: replacing one algebraic model of the types with another, and seeing how the observation (the second command and declaration) are affected. The split will also make clear how the denotations for command are parameterized by an algebraic structure. However, such a split main procedure is *not* necessary for our semantic techniques. As will be shown below, one can just run the first part of the main procedure, and then the second.

Figure 2 is a complete example program. It declares two types: `Point` and `Rect` (rectangle). Of the methods, `mkRect` is the most complex; it declares a constant, `r`, which is bound to a new rectangle object, with the field `bl` initialized to the object `p1` and `tr` initialized to `p2`. (The actual arguments in object creation expression, "`new` T `(`E*`)`", are bound to the fields in the order in which the fields are declared.) If the point `p2` is not towards top and right of `p1`, then `r` is mutated so that its field `bl` becomes `p2`, and `tr` becomes `p1`. Parameter passing does not make copies; hence any other names for the arguments to `mkRect` outside that method become aliases. (Similarly, field assignments do not make copies.) The methods `botLeft` and `topRight` also do not make copies when they return their results. This is poor programming practice, but it helps us illustrate how the language allows mutation and aliasing. For example, at the end of the first part of the `main` procedure, just before the keyword "`by`", `x` and `y` are aliases, as are `botLeft(x)`, `botLeft(y)`, and `botLeft(w)`.

Hence, the `horizMove(w, 1)` adds 1 to the abscissa of the point which is `botLeft(w)` and also `botLeft(y)` and `botLeft(x)`. Similarly, `vertMove(x)` adds 1 to the ordinate of this point, and also adds 1 to the ordinate of `z`. Thus the output values are appropriate for their names in Figure 2.

2.2 Semantics of π

The fundamental idea of our semantic technique is to split the semantics of a language into two parts: an algebraic structure, and a denotational semantics that computes using the algebraic structure. This is illustrated in Figure 3. Here one sees that the denotation of types and methods resides in the algebra. The denotational part handles the manipulation of the program state (environment and store), and gives the semantics to commands and expressions. The denotational part relies on the algebra for invoking methods. The signature (Σ) of the algebra thus documents the interface between the denotational part of the semantics and the algebraic part.

Neither the algebra nor the denotational parts of the semantics are completely unaffected by the other. One interaction comes from the treatment of the store. To handle mutation and aliasing, the algebra uses the store that the denotational part maintains for it; when a method is invoked, it receives a store, and returns a modified store. Therefore, in a sense, an algebra is not really an "algebra" at all, because it is not closed in the sense that its methods take and return a store, but the store is not itself part of the algebra. This is done in order to keep the meaning of the algebra fixed and the semantics of the algebra's methods purely functional. If the store were part of the algebra, then the algebra would change (evolve, as in [21] [22]), but then the denotational semantics would lose its characteristic referential transparency; that is, the denotational semantics would no longer be written over a purely functional base.

The other interaction is that when elaborating the method declarations, the denotational semantics is used to give meaning to the body of a method. We will see, however, that the semantic clauses for declarations, commands, and expressions, are parametrized with respect to an algebra, and thus there is still great deal of separation.

Some terminological remarks may also be in order before going into the details of the semantics. As is usual in denotational semantics, mutable objects are modeled using locations (memory cells). In other words, our model of an object is a typed location. Locations are typed in the sense that a location $l : T$ can only store a value of type T. Each object (location) contains a value, sometimes called an "abstract value" [24], which can be extracted by a store mapping, as is usual in denotational semantics [57]. (This resembles Scheifler's denotational semantics of CLU [56], and would be similar to a semantics of LISP.) An object may "contain" other objects, if its value contains other locations. The main procedure in 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 algebra's operations.

For simplicity, we do not worry about garbage collection in the semantics.

2.2.1 Visible Types and External Values

To facilitate the study of visible behavior, we distinguish a subset of types as *visible* types; these are the types of values that can be "output" by a program [60] [46]. These are defined

```
type Point fields (x:Int, y:Int);
type Rect fields (bl:Point, tr:Int);

method mkPoint (i:Int, j:Int):Point
    { const p:Point = new Point(i, j); return p };
method abscissa (p:Point):Int { nothing; return p.x };
method ordinate (p:Point):Int { nothing; return p.y };
method addX (p:Point, i:Int):Void { p.x := add(p.x, i); return nothing };
method addY (p:Point, i:Int):Void { p.y := add(p.y, i); return nothing };
method upRightOf (p1:Point, p2:Point):Bool
    { nothing; return and(leq(p1.x, p2.x), leq(p1.y, p2.y)) };
method pointEqual (p1:Point, p2:Point): Bool
    { nothing; return and(equal(abscissa(p1), absicissa(p2))
                          equal(ordinate(p1), ordinate(p2))) };

method mkRect (p1:Point, p2:Point):Rect
    { const r:Rect = new Rect(p1 p2);
      if upRightOf(p1, p2) then nothing
      else r.bl := p2; r.tr := b1
      fi;
      return r };
method botLeft (r:Rect):Point { nothing; return r.bl };
method topRight (r:Rect):Point { nothing; return r.tr };
method horizMove (r:Rect delta:Int):Void
    { addX(r.bl, delta); addX(r.tr, delta); return nothing };
method vertMove (r:Rect, delta:Int):Void
    { addY(r.bl, delta); addY(r.tr, delta); return nothing };

main { observe
         const z:Point = mkPoint(2, 4);
         const w:Rect = mkRect(mkPoint(0,0), mkPoint(2,4));
         const y:Rect = mkRect(botLeft(w), z);
         const x:Rect = y;
         horizMove(w, 1);
         vertMove(x,1)
       by
         if pointEqual(topRight(y), mkPoint(2, 5))
          then addX(topRight(w), 1)
          else nothing fi
         const shouldBe1:Int = abscissa(botLeft(y));
         const shouldBe5:Int = ordinate(z);
         const shouldBe4:Int = abscissa(topRight(w) }
```

Figure 2: A sample program. When executed, it should terminate with the value 1 in `shouldBe1`, 5 in `shouldBe5`, and 4 in `shouldBe4`.

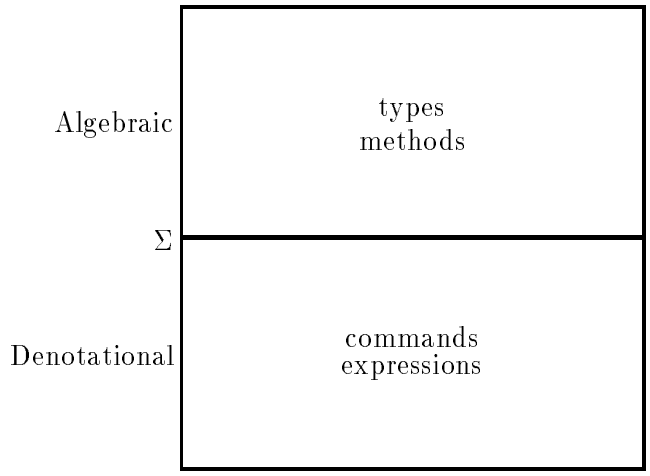

Figure 3: A picture that illustrates the idea of the split semantics.

as follows.

$$VIS \stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Int}\} \tag{1}$$

The sets of externally visible values of each of these types is also fixed, and are defined as follows.

$$EXTERNALS_{\texttt{Bool}} = \{true, false\} \tag{2}$$
$$EXTERNALS_{\texttt{Int}} = \{0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\} \tag{3}$$

We write $EXTERNALS$ for $\bigcup_{V \in VIS} EXTERNALS_V$.

2.2.2 Signatures

The interface between our programming language semantics and an ADT model is described by a signature. A signature contains, as usual, a set of type names, and method names. (Method names are historically called operation symbols in this context.) Overloading of operation symbols based on argument types is permitted, because this is needed to permit fields with the same names to be used in different representations. It will also be useful in the application of this work to object-oriented programming with message passing (which supports a kind of dynamic overloading [69]). The signatures are thus a simplified form of the signatures used in category-sorted algebras [53] [54].

Definition 2.1 (signature) *A* signature, Σ, *is a tuple,* $(TYPES, OPS, ResType)$, *where*

- $TYPES$ *is a non-empty set of type symbols, such that both* $\texttt{Void} \in TYPES$ *and* $VIS \subseteq TYPES$, *and*
- OPS *is a family of sets of operation symbols, indexed by the natural numbers, and*

$$TYPES^{\pi} \stackrel{\text{def}}{=} \{\texttt{Void}, \texttt{Bool}, \texttt{Int}\}$$

OPS^{π} and $ResType^{\pi}$

`nothing`	:	$() \to \texttt{Void}$
`true`	:	$() \to \texttt{Bool}$
`false`	:	$() \to \texttt{Bool}$
`and`	:	$(\texttt{Bool}, \texttt{Bool}) \to \texttt{Bool}$
`or`	:	$(\texttt{Bool}, \texttt{Bool}) \to \texttt{Bool}$
`not`	:	$(\texttt{Bool}) \to \texttt{Bool}$
`0`	:	$() \to \texttt{Int}$
`1`	:	$() \to \texttt{Int}$
`add`	:	$(\texttt{Int}, \texttt{Int}) \to \texttt{Int}$
`mult`	:	$(\texttt{Int}, \texttt{Int}) \to \texttt{Int}$
`negate`	:	$(\texttt{Int}) \to \texttt{Int}$
`equal`	:	$(\texttt{Int}, \texttt{Int}) \to \texttt{Bool}$
`less`	:	$(\texttt{Int}, \texttt{Int}) \to \texttt{Bool}$
`leq`	:	$(\texttt{Int}, \texttt{Int}) \to \texttt{Bool}$

Figure 4: The signature Σ^{π} of the built-in types in π.

- *ResType is a family of partial functions indexed by the natural numbers, such that for each natural number n, $ResType_n : OPS_n \times TYPES^n \to TYPES_{\perp}$.*

The type `Void` is used to indicate that a method has no results.

To simplify notation we usually write $g \in OPS$ as shorthand for $g \in \bigcup_{n \in Nat} OPS_n$; and we usually omit the subscript on *ResType*. An operation symbol g has *rank* n if $g \in OPS_n$. If g has rank n and $ResType(g, \vec{S}) = T$, then the pair $(\vec{S}, T)$ is called a *type* of g, and we write $g : \vec{S} \to T$. Because of overloading, an operation symbol may have many types but at most one result type for each tuple of argument types.

Example Signatures Figure 4 gives the signature, Σ^{π}, of the built-in types in π.

The denotational semantics for π produces a signature for the type and method declarations. This signature contains Σ^{π} as a subsignature. As an example, the signature $\Sigma^{\mathbf{D}}$ that would be obtained by elaborating the declarations in Figure 2 is given in Figure 5.

Auxiliary Functions on Signatures Because signatures are constructed as part of the denotations of types and methods in π, we will need several auxiliary functions on signatures. These are defined below.

We write *SIGS* for the set of all signatures.

$$SIGS \stackrel{\text{def}}{=} \{\Sigma \mid \Sigma \text{ is a signature}\} \tag{4}$$

The notation $[i \mapsto OPS_i \cup \{g\}]OPS$, denotes the family $\langle OPS'_n : n \in Nat \rangle$ such that $OPS'_i = OPS_i \cup \{g\}$ and for all $j \neq i$, $OPS'_j = OPS_j$. We also write $[(g, \vec{S}) \mapsto T]ResType$, for

$$TYPES^{\mathbf{D}} \stackrel{\text{def}}{=} TYPES^{\pi} \cup \{\texttt{Point}, \texttt{Rect}\}$$

$OPS^{\mathbf{D}}$ and $ResType^{\mathbf{D}}$ added to Σ^{π}

`mkPoint`	:	$(\texttt{Int}, \texttt{Int}) \rightarrow \texttt{Point}$
`abscissa`	:	$(\texttt{Point}) \rightarrow \texttt{Int}$
`ordinate`	:	$(\texttt{Point}) \rightarrow \texttt{Int}$
`addX`	:	$(\texttt{Point}, \texttt{Int}) \rightarrow \texttt{Void}$
`addY`	:	$(\texttt{Point}, \texttt{Int}) \rightarrow \texttt{Void}$
`upRightOf`	:	$(\texttt{Point}, \texttt{Point}) \rightarrow \texttt{Bool}$
`pointEqual`	:	$(\texttt{Point}, \texttt{Point}) \rightarrow \texttt{Bool}$
`mkRect`	:	$(\texttt{Point}, \texttt{Point}) \rightarrow \texttt{Rect}$
`botLeft`	:	$(\texttt{Rect}) \rightarrow \texttt{Point}$
`topRight`	:	$(\texttt{Rect}) \rightarrow \texttt{Point}$
`horizMove`	:	$(\texttt{Rect}, \texttt{Int}) \rightarrow \texttt{Void}$
`vertMove`	:	$(\texttt{Rect}, \texttt{Int}) \rightarrow \texttt{Void}$

Figure 5: The signature $\Sigma^{\mathbf{D}}$. The operation symbols of $\Sigma^{\mathbf{D}}$ include all those of Σ^{π} and the ones listed above.

the family $\langle ResType'_n : n \in Nat \rangle$ such that $ResType'(g, \vec{S}) = T$, and for all $(g', \vec{U}) \neq (g, \vec{S})$ $ResType'(g', \vec{U}) = ResType(g, \vec{U})$.

The following auxiliary functions are used to add types and messages (the names and types of methods) to signatures. Both of these functions require the type or message being added to not already be in the signature; this is useful in ensuring that types and methods are not multiply declared in π.

$addType : TYPES \rightarrow SIGS \rightarrow SIGS_{\perp}$
$addType[\![T]\!]\ (TYPES, OPS, ResType) =$
  if $T \in TYPES$ **then** $\perp$ **else** $(TYPES \cup \{T\}, OPS, ResType)$
$addMessage$: Method-Name $\rightarrow TYPES^{*} \rightarrow TYPES \rightarrow SIGS \rightarrow SIGS_{\perp}$
$addMessage[\![g]\!][\![\vec{S}]\!][\![T]\!]\ (TYPES, OPS, ResType) =$
  if $ResType(g, \vec{S}) \neq \perp$ **then** $\perp$
  else $(TYPES,\ [length(\vec{S}) \mapsto OPS_{length(\vec{S})} \cup \{g\}]OPS,\ [(g, \vec{S}) \mapsto T]ResType)$

Special message names are used in the semantics below to create objects, and to set and get fields of objects. These messages have names of the form "creator(T)", "getter(I)", and "setter(I)". Later, for purposes of information hiding, we will need to delete these operations from a signature. They are deleted by the auxiliary function *hideInternalMessages* defined below. To avoid unnecessary detail, we assume that there is a predicate *isToBeHidden*, which is true for all message names of the form "creator(T)", "getter(I)", or "setter(I)", and false for all other message names. We write $OPS \setminus \{g \mid g \in OPS, isToBeHidden(g)\}$ for the obvious *Nat*-indexed family of operation symbols with such operation symbols removed. We also write $ResType \setminus \{(g, \vec{S}) \mapsto T \mid ResType(g, \vec{S}) = T, isToBeHidden(g)\}$ for the family $ResType'$ such that $ResType'(g, \vec{U}) = \perp$ if $isToBeHidden(g)$ and $ResType'(g, \vec{U}) = ResType(g, \vec{U})$ otherwise.

$hideInternalMessages : SIGS \to SIGS$
$hideInternalMessages\ (TYPES, OPS, ResType) =$
$\quad (TYPES,$
$\quad\ OPS \setminus \{g \mid g \in OPS, isToBeHidden(g)\},$
$\quad\ ResType \setminus \{(g, \vec{S}) \mapsto T \mid ResType(g, \vec{S}) = T, isToBeHidden(g)\})$

2.2.3 Algebras and Stores

An algebra that presents the interface Σ is called a Σ-algebra. Stores are defined simultaneously, but are not contained in Σ-algebras. This definition of algebra and stores below was inspired by work on models of types for interface specification languages [73] [8].

To explain a term used in the definition below, a *finite function*, $f : S \xrightarrow{\text{fin}} T$ is a function $S \to T_\perp$ such that f has a proper result (not $\perp$) only on a finite number of arguments. By the *domain* of a finite function, $dom(f)$, we mean the set of all arguments for which f's result is proper.

Definition 2.2 (Σ-algebra, *STORE*) *A Σ-algebra, $\mathbf{A}$, is a tuple,*

$$(SORTS^{\mathbf{A}}, ObjectTypes^{\mathbf{A}}, LOCS^{\mathbf{A}}, VALS^{\mathbf{A}}, TtoS^{\mathbf{A}}, OPS^{\mathbf{A}}, externVal^{\mathbf{A}}),$$

where

- $SORTS^{\mathbf{A}} \supseteq TYPES$ *is a set of sort symbols,*
- $ObjectTypes^{\mathbf{A}} \subseteq TYPES$ *is a set of object type symbols,*
- $LOCS^{\mathbf{A}}$ *is a family of sets, indexed by* $ObjectTypes^{\mathbf{A}}$*, representing typed locations,*
- $VALS^{\mathbf{A}}$ *is a family of abstract values indexed by* $SORTS^{\mathbf{A}}$*, such that for each* $T \in ObjectTypes^{\mathbf{A}}$*,* $VALS^{\mathbf{A}}_T = LOCS^{\mathbf{A}}_T$
- $TtoS^{\mathbf{A}} : ObjectTypes^{\mathbf{A}} \to SORTS^{\mathbf{A}}$ *is a function that gives a sort symbol for each object type symbol,*
- $OPS^{\mathbf{A}}$ *is a family of operation interpretations indexed by the natural numbers such that for each* $n \in Nat$ *and* $g \in OPS_n$*, there is a polymorphic partial function* $g^{\mathbf{A}} \in OPS^{\mathbf{A}}_n$ *where for each* $\vec{S} \in TYPES^n$ *and* $T \in TYPES$*, if* $ResType(g, \vec{S}) = T$ *then* $g^{\mathbf{A}}$ *satisfies*

$$g^{\mathbf{A}} : (VALS^{\mathbf{A}}_{\vec{S}} \times STORE[\mathbf{A}]) \to (VALS^{\mathbf{A}}_T \times STORE[\mathbf{A}])_\perp,$$

- *externVal*$^{\mathbf{A}}$ *is a family of functions indexed by VIS, such that for each* $T \in VIS$, $externVal^{\mathbf{A}}_T : VALS^{\mathbf{A}}_T \times STORE[\mathbf{A}] \to (EXTERNALS_T)_\perp$,

and

$$STORE[\mathbf{A}] \stackrel{\text{def}}{=} LOCS^{\mathbf{A}} \stackrel{\text{fin}}{\to} VALS^{\mathbf{A}} \tag{5}$$

is such that if $\sigma : STORE[\mathbf{A}]$, $l \in LOCS^{\mathbf{A}}_T$, *and* $l \in dom(\sigma)$, *then* $\sigma(l) \in VALS^{\mathbf{A}}_{TtoS^{\mathbf{A}}[T]}$.

An algebra may have more sort symbols than type symbols because it may be convenient to introduce hidden sorts in a model. (So, if we followed Goguen [17], we would call algebras "machines".) The set of object types is the set of types whose objects are modeled by locations. An immutable object might be modeled by a location whose value never changes or might be modeled by a value that is not a location. For each object type, the *TtoS* mapping gives the sort of its abstract values. A method is modeled by a partial function from a tuple of argument values and an initial store to a result value and a final store. (The notation $VALS^{\mathbf{A}}_{\vec{S}}$ means the cross product of the domains $VALS^{\mathbf{A}}_{S_i}$, that is $\Pi_{i=1}^{length\,\vec{S}} VALS^{\mathbf{A}}_{S_i}$. For example, $VALS^{\mathbf{A}}_{(S_1,S_2)}$ means $VALS^{\mathbf{A}}_{S_1} \times VALS^{\mathbf{A}}_{S_2}$.) Since methods are modeled by functions, our algebras do not model nondeterministic methods; this is another simplification. We require that such functions be polymorphic, because it affords considerable notational convenience; if we were modeling an object-oriented language with message passing, the polymorphism would be of the essence [33], but in this setting we could excise it at the cost of additional notation. Stores are finite functions because they are not defined for all potential locations. The store argument to $externVal^{\mathbf{A}}_T$ might be needed if the visible types were object types, and in that case the result might be $\perp$, if the location passed was not in the domain of the store.

We use $l : T$ to stand for a location $l \in LOCS^{\mathbf{A}}_T$ when the algebra $\mathbf{A}$ is clear from context. For this we also write "$l : T$ is a location." Similarly we use $v : T$ to stand for an abstract value $v \in VALS^{\mathbf{A}}_T$.

Example Algebras and Auxiliary Functions on Stores Figure 6 gives an example Σ^π-algebra, which is the built-in algebra for π, $\mathbf{A}^\pi$. In $\mathbf{A}^\pi$ the visible types are modeled as abstract values instead of objects. This example also shows how traditional multi-sorted algebras can be adapted to our definition.

A more interesting example algebra, $\mathbf{D}$, is given in Figure 7. It would be the denotation of the types and methods in Figure 2. We use the special names of the form "sortFor(`Point`)" in the semantics of π for the names of sorts. Note that the abstract values of rectangles, $VALS^{\mathbf{D}}_{\text{sortFor(Rect)}}$, contain point objects (i.e., locations). Because the visible types in $\mathbf{D}$ are modeled as pure values instead of objects, $\mathbf{D}$ can be thought of as a model of an implementation in a hybrid object-oriented language such as C++, CLOS, or Eiffel.

The operations of $\mathbf{D}$ are shown in Figure 8. In the figure we use the following notation, which we will also use in the semantics. This notation is not specific to $\mathbf{D}$, but will also be applied to any algebra constructed by our semantics, hence in the definitions of the notation, $\mathbf{D}$ should be thought of as generic.

We use *nextFree*$[T]$ to find the next free location of type T in a given store (where *TYPES* is the set of types in the signature of $\mathbf{D}$):

$$nextFree[T] : STORE[\mathbf{D}] \to LOCS^{\mathbf{D}}_T$$
$$nextFree[T]\ \sigma \stackrel{\text{def}}{=} l^T_{1+max(\{-1\}\cup\{i | l^S_i \in dom(\sigma), S \in TYPES\})}$$

$$SORTS^{\mathbf{A}^\pi} \stackrel{\text{def}}{=} TYPES$$

$$ObjectTypes^{\mathbf{A}^\pi} \stackrel{\text{def}}{=} \{\}$$

$$LOCS^{\mathbf{A}^\pi} \text{ is empty}$$

$$TtoS^{\mathbf{A}^\pi}[T] \text{ is the empty function, for each } T$$

$$\begin{array}{lcl} VALS^{\mathbf{A}^\pi}_{\texttt{Void}} & \stackrel{\text{def}}{=} & \{*\} \\ VALS^{\mathbf{A}^\pi}_{\texttt{Bool}} & \stackrel{\text{def}}{=} & \{true, false\} \\ VALS^{\mathbf{A}^\pi}_{\texttt{Int}} & \stackrel{\text{def}}{=} & \{0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\} \end{array}$$

$$\begin{array}{lcl} externVal^{\mathbf{A}^\pi}_{\texttt{Bool}}(v, \sigma) & \stackrel{\text{def}}{=} & v \\ externVal^{\mathbf{A}^\pi}_{\texttt{Int}}(v, \sigma) & \stackrel{\text{def}}{=} & v \end{array}$$

Operation Interpretations

$$\begin{array}{lcl} \texttt{nothing}^{\mathbf{A}^\pi}((), \sigma) & \stackrel{\text{def}}{=} & (*, \sigma) \\ \texttt{true}^{\mathbf{A}^\pi}((), \sigma) & \stackrel{\text{def}}{=} & (true, \sigma) \\ \texttt{false}^{\mathbf{A}^\pi}((), \sigma) & \stackrel{\text{def}}{=} & (false, \sigma) \\ \texttt{and}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 \wedge v_2, \sigma) \\ \texttt{or}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 \vee v_2, \sigma) \\ \texttt{not}^{\mathbf{A}^\pi}((v), \sigma) & \stackrel{\text{def}}{=} & (\neg(v), \sigma) \\ \texttt{0}^{\mathbf{A}^\pi}((), \sigma) & \stackrel{\text{def}}{=} & (0, \sigma) \\ \texttt{1}^{\mathbf{A}^\pi}((), \sigma) & \stackrel{\text{def}}{=} & (1, \sigma) \\ \texttt{add}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 + v_2, \sigma) \\ \texttt{mult}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 \times v_2, \sigma) \\ \texttt{negate}^{\mathbf{A}^\pi}((v), \sigma) & \stackrel{\text{def}}{=} & (\Leftrightarrow v_1, \sigma) \\ \texttt{equal}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 = v_2, \sigma) \\ \texttt{less}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 < v_2, \sigma) \\ \texttt{leq}^{\mathbf{A}^\pi}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 \leq v_2, \sigma) \end{array}$$

Figure 6: The Σ^π-algebra $\mathbf{A}^\pi$. In the definition of $SORTS^{\mathbf{A}^\pi}$, $TYPES$ means the $TYPES$ of Σ^π.

$$SORTS^{\mathbf{D}} \stackrel{\text{def}}{=} TYPES \bigcup \{\text{sortFor}(\texttt{Point}), \text{sortFor}(\texttt{Rect})\}$$

$$ObjectTypes^{\mathbf{D}} \stackrel{\text{def}}{=} \{\texttt{Point}, \texttt{Rect}\}$$

$$LOCS^{\mathbf{D}}_T \stackrel{\text{def}}{=} \{l^T_i \mid i \in Nat\}, \quad \text{for each } T \in ObjectTypes^{\mathbf{D}}$$

Type to Sort Mapping ($TtoS^{\mathbf{D}}$)

`Point`	$\mapsto$	sortFor(`Point`)
`Rect`	$\mapsto$	sortFor(`Rect`)

$$\begin{array}{lcl}
VALS^{\mathbf{D}}_{\texttt{Void}} & \stackrel{\text{def}}{=} & VALS^{\mathbf{A}^\pi}_{\texttt{Void}} \\
VALS^{\mathbf{D}}_{\texttt{Bool}} & \stackrel{\text{def}}{=} & VALS^{\mathbf{A}^\pi}_{\texttt{Bool}} \\
VALS^{\mathbf{D}}_{\texttt{Int}} & \stackrel{\text{def}}{=} & VALS^{\mathbf{A}^\pi}_{\texttt{Int}} \\
VALS^{\mathbf{D}}_{\text{sortFor(Point)}} & \stackrel{\text{def}}{=} & \{(v_x, v_y) \mid v_x, v_y \in VALS^{\mathbf{D}}_{\texttt{Int}}\} \\
VALS^{\mathbf{D}}_{\text{sortFor(Rect)}} & \stackrel{\text{def}}{=} & \{(l_{bl}, l_{tr}) \mid l_{bl}, l_{tr} \in LOCS^{\mathbf{D}}_{\texttt{Point}}\}
\end{array}$$

and for $T \in ObjectTypes^{\mathbf{D}}$, $VALS^{\mathbf{D}}_T \stackrel{\text{def}}{=} LOCS^{\mathbf{D}}_T$

$$externVal^{\mathbf{D}} \stackrel{\text{def}}{=} externVal^{\mathbf{A}^\pi}$$

Figure 7: The $\Sigma^{\mathbf{D}}$-algebra $\mathbf{D}$ (part 1). In this figure, $TYPES$ is from $\Sigma^{\mathbf{D}}$.

We use the function $alloc[T]$ to find a free location and initialize it with an abstract value of type T:

$$alloc[T] : (VALS^{\mathbf{D}}_{TtoS\mathbf{D}_{[T]}} \times STORE[\mathbf{D}]) \to VALS^{\mathbf{D}}_T \times STORE[\mathbf{D}]$$
$$alloc[T](v, \sigma) \stackrel{\text{def}}{=} \textbf{let } l = nextFree[T](\sigma) \textbf{ in } (l, [l \mapsto v]\sigma).$$

The notation $[l \mapsto v]\sigma$ is defined by

$$[l \mapsto v]\sigma \stackrel{\text{def}}{=} \lambda l_2 \,.\, \textbf{if } (l_2 = l) \textbf{ then } v \textbf{ else } (\sigma\ l_2). \tag{6}$$

We use $emptyStore : STORE[\mathbf{D}]$ for the store whose domain is empty (i.e., $\lambda l \,.\, \bot$).

Note that in Figure 8, in the interpretation of `mkRect`, the point arguments are put directly into the abstract value of the rectangle, producing indirect aliasing. The operations `botLeft` and `topRight` also produce aliasing.

Auxiliary functions on Algebras As with signatures, algebras are manipulated in the semantics of π, so we need various auxiliary functions on them.

Let $Alg(\Sigma)$ denote the class of all Σ-algebras.

$$Alg(\Sigma) \stackrel{\text{def}}{=} \{\mathbf{A} \mid \mathbf{A} \text{ is a } \Sigma\text{-algebra}\} \tag{7}$$

For an algebra $\mathbf{A}$, denote the components of the algebra by $SORTS^{\mathbf{A}}$, $ObjectTypes^{\mathbf{A}}$, etc. Also the following functions are defined to produce a new algebra with the given component changed, for each component. The function $setSORTS$ is shown below as an example; and the rest are entirely analogous.

$setSORTS : PowerSet(\text{Type-name}) \to Alg(\Sigma : SIGS) \to Alg(\Sigma)$
$setSORTS\ SORTS\ \mathbf{A} =$
$\quad (SORTS, ObjectTypes^{\mathbf{A}}, LOCS^{\mathbf{A}}, VALS^{\mathbf{A}}, TtoS^{\mathbf{A}}, OPS^{\mathbf{A}}, externVal^{\mathbf{A}})$

Because operation interpretations are polymorphic, we need a notation for defining the action of a polymorphic function on tuples of a given type. For this we use the notation $[\vec{S} \mapsto f]g^{\mathbf{A}}$, which is defined by the following.

$$([\vec{S} \mapsto f]g^{\mathbf{A}}) \stackrel{\text{def}}{=} \lambda(\vec{v}, \sigma) \,.\, \textbf{if } \vec{v} \in VALS^{\mathbf{A}}_{\vec{S}} \textbf{ then } f(\vec{v}, \sigma) \textbf{ else } g^{\mathbf{A}}(\vec{v}, \sigma) \tag{8}$$

We also use the following to add an interpretation of an operation for tuples of arguments of type $\vec{S}$ to $OPS^{\mathbf{A}}$.

$addOp[\vec{S}][T] : (\Sigma : SIGS) \to (A : Alg(\Sigma)) \to \text{Identifier}$
$\qquad \to ((VALS^{\mathbf{A}}_{\vec{S}} \times STORE[\mathbf{A}]) \to (VALS^{\mathbf{A}}_T \times STORE[\mathbf{A}])_\bot) \to Alg(\Sigma)_\bot$
$addOp[\vec{S}][T]\ (TYPES, OPS, ResType)\ \mathbf{A}\ [\![g]\!]\ f =$
$\quad \textbf{let } i = length\ \vec{S} \textbf{ in}$
$\quad \textbf{if } g \notin OPS_i \textbf{ then } \bot \textbf{ else } setOPS\ ([i \mapsto (OPS^{\mathbf{A}}_i \setminus \{g^{\mathbf{A}}\}) \cup \{[\vec{S} \mapsto f]g^{\mathbf{A}}\}]OPS^{\mathbf{A}})\ \mathbf{A}$

To handle the hiding of the operation interpretations for operations named "creator(T)", "setter(I)", or "getter(I)", we adapt the notion of a reduct of an algebra (see, for example, [14, Section 6.8]) to this setting. Since we will only be using reducts to hide such operations, we only define $\mathbf{A}|_{(hideInternalMessages\ \Sigma)}$ for a Σ-algebra $\mathbf{A}$. Let OPS and OPS' be the operation symbols of Σ and $(hideInternalMessages\ \Sigma)$, respectively. Since we are deleting only the specially named operations, it suffices to define $\mathbf{A}|_{(hideInternalMessages\ \Sigma)}$ as being the same as $\mathbf{A}$, except that for each $g \in (OPS \setminus OPS')$, for each $\vec{v}$ and σ, $g^{\mathbf{A}|_{(hideInternalMessages\ \Sigma)}}(\vec{v}, \sigma) = \bot$.

Operation Interpretations that are not also in $\mathbf{A}^{\pi}$

$$
\begin{array}{lcl}
\texttt{mkPoint}^{\mathbf{D}}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Point}]((v_1, v_2), \sigma) \\
\texttt{abscissa}^{\mathbf{D}}((l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = (\sigma\ l) \textbf{ in } (v_1, \sigma) \\
\texttt{ordinate}^{\mathbf{D}}((l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = (\sigma\ l) \textbf{ in } (v_2, \sigma) \\
\texttt{addX}^{\mathbf{D}}((l^{\texttt{Point}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = (\sigma\ l^{\texttt{Point}}) \textbf{ in} \\
 & & (*, [l^{\texttt{Point}} \mapsto (v_1 + v^{\texttt{Int}}, v_2)]\sigma) \\
\texttt{addY}^{\mathbf{D}}((l^{\texttt{Point}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = (\sigma\ l^{\texttt{Point}}) \textbf{ in} \\
 & & (*, [l^{\texttt{Point}} \mapsto (v_1, v_2 + v^{\texttt{Int}})]\sigma) \\
\texttt{upRightOf}^{\mathbf{D}}((l_1^{\texttt{Point}}, l_2^{\texttt{Point}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_{11}, v_{12}) = (\sigma\ l_1^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (v_{21}, v_{22}) = (\sigma\ l_2^{\texttt{Point}}) \textbf{ in} \\
 & & (v_{11} \leq v_{21} \wedge v_{12} \leq v_{22}, \sigma) \\
\texttt{pointEqual}^{\mathbf{D}}((l_1^{\texttt{Point}}, l_2^{\texttt{Point}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (v_{11}, v_{12}) = (\sigma\ l_1^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (v_{21}, v_{22}) = (\sigma\ l_2^{\texttt{Point}}) \textbf{ in} \\
 & & (v_{11} = v_{21} \wedge v_{12} = v_{22}, \sigma) \\
\texttt{mkRect}^{\mathbf{D}}((l_1, l_2), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_r, \sigma') = \mathit{alloc}[\texttt{Rect}]((l_1, l_2), \sigma) \textbf{ in} \\
 & & \textbf{let } (v_b, \sigma'') = \texttt{upRightOf}^{\mathbf{D}}(l_1, l_2) \textbf{ in} \\
 & & \textbf{if } \mathit{externVal}^{\mathbf{D}}_{\texttt{Bool}}(v_b, \sigma'') \textbf{ then } (l_r, \sigma'') \\
 & & \textbf{else } (l_r, [l_r \mapsto (l_2, l_1)]\sigma'') \\
\texttt{botLeft}^{\mathbf{D}}((l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = (\sigma l) \textbf{ in } (l_1, \sigma) \\
\texttt{topRight}^{\mathbf{D}}((l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = (\sigma l) \textbf{ in } (l_2, \sigma) \\
\texttt{horizMove}^{\mathbf{D}}((l^{\texttt{Rect}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl}, l_{tr}) = (\sigma\ l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (v', \sigma') = \texttt{addX}^{\mathbf{D}}((l_{bl}, v^{\texttt{Int}}), \sigma) \textbf{ in} \\
 & & \textbf{let } (v'', \sigma'') = \texttt{addX}^{\mathbf{D}}((l_{tr}, v^{\texttt{Int}}), \sigma') \textbf{ in} \\
 & & (*, \sigma'') \\
\texttt{vertMove}^{\mathbf{D}}((l^{\texttt{Rect}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl}, l_{tr}) = (\sigma\ l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (v', \sigma') = \texttt{addY}^{\mathbf{D}}((l_{bl}, v^{\texttt{Int}}), \sigma) \textbf{ in} \\
 & & \textbf{let } (v'', \sigma'') = \texttt{addY}^{\mathbf{D}}((l_{tr}, v^{\texttt{Int}}), \sigma') \textbf{ in} \\
 & & (*, \sigma'')
\end{array}
$$

Figure 8: The $\Sigma^{\mathbf{D}}$-algebra $\mathbf{D}$ (part 2).

2.2.4 Other Semantic Domains

Environments and States The state of a program is given by two mappings: an environment and a store. The type of stores is given in Equation (5) above. An environment maps identifiers to abstract values (which are thus the "denotable values"). The notation indicates that a state is a state over a particular algebra, as is an environment.

$$ENV[\mathbf{A}] \stackrel{\text{def}}{=} \text{Identifier} \stackrel{\text{fin}}{\rightarrow} VALS^{\mathbf{A}} \tag{9}$$

$$STATE[\mathbf{A}] \stackrel{\text{def}}{=} ENV[\mathbf{A}] \times STORE[\mathbf{A}] \tag{10}$$

Like stores, environments are typed mappings. That is, if $\eta : ENV[\mathbf{A}]$ and if $x : T \in dom(\eta)$, then $\eta(x) \in VALS^{\mathbf{A}}_T$. We write *emptyEnviron* for the empty finite function from identifiers to denotable values, and define the following functions on environments, for any Σ-algebra $\mathbf{A}$ [71].

$emptyEnviron : ENV[\mathbf{A}] = \lambda I \,.\, \bot$
$overlay : ENV[\mathbf{A}] \times ENV[\mathbf{A}] \rightarrow ENV[\mathbf{A}]$
$overlay(\eta_1, \eta_2) = \lambda I \,.\, \textbf{if } I \in dom(\eta_1) \textbf{ then } \eta_1[\![I]\!] \textbf{ else } \eta_2[\![I]\!]$
$bind : \text{Identifier} \times VALS^{\mathbf{A}} \rightarrow ENV[\mathbf{A}]$
$bind(I, v) = \lambda I' \,.\, \textbf{if } I' = I \textbf{ then } v \textbf{ else } \bot$

We also use the notation $[I \mapsto v]\eta$ as a shorthand for $overlay(bind(I, v), \eta)$.

Observations and Answers Recall that the second declaration, D_2, of the main procedure in a π program declares names for "outputs", each of which must have type `Bool` or `Int`. The denotation of the second part of the main procedure is a function from an algebra to an observation, which is itself a function from a state (defined, for example, by the first part of the main procedure) to a representation of this output. The output itself, the domain *ANSWERS*, is modeled as a finite function from the identifiers declared in the final declaration to their corresponding external values. One can think of the program as printing these values (labeled by each identifier's name).

$$OBSERVATION[\mathbf{A}] \stackrel{\text{def}}{=} STATE[\mathbf{A}] \rightarrow ANSWERS_\bot \tag{11}$$

$$ANSWERS \stackrel{\text{def}}{=} \text{Identifier} \stackrel{\text{fin}}{\rightarrow} EXTERNALS \tag{12}$$

Lists and Tuples For lists and tuples we follow the notation in [57]. For tuples, we use the usual pairing notation, and indicate indexing with a downward arrow; for example, $(x, y) \downarrow 2 = y$. As usual, tuples are written with vector notation, $\vec{v}$. Recall that the notation D^* means all finite tuples of D. We use the notation $[i \mapsto y]\vec{v}$ for the tuple such that $([i \mapsto y]\vec{v}) \downarrow i = y$ and for $j \neq i$, $([i \mapsto y]\vec{v}) \downarrow j = \vec{v} \downarrow j$.

For lists, $List(D)$ is the notation for the domain of lists of D, and we use the constant *nil*, and auxiliary functions *cons*, *hd*, and *tl* and the predicate *null*. To aid the reader, we use notations like $\hat{x}$ for lists. We also assume an auxiliary function *length*, that gives the length of a list. We use a *map* functional as in SML [50], as well as a *foldright* functional, which is defined below as the fixedpoint of a generator. (For simplicity of notation, we use the convention that the application of a function of a type such as $S \times T \rightarrow T_\bot$ to $\bot$ produces $\bot$.)

$foldright : \forall S.\forall T.(S \times T \rightarrow T_\bot) \rightarrow T \rightarrow List(S) \rightarrow T_\bot$

$foldright = fix\ \lambda G\ .\ \lambda f\ .\ \lambda b\ .\ \lambda \hat{x}\ .\ \textbf{if}\ null\ \hat{x}\ \textbf{then}\ b\ \textbf{else}\ f((hd\ \hat{x}),\ G\ f\ b\ (tl\ \hat{x}))$

The auxiliary function "*productize*" converts a list of n items into an n-tuple, preserving the ordering. The auxiliary function "*addToEnd*" adds an item to the end of a list. These satisfy the following property for all $1 \leq i \leq length(\hat{v}) + 1$.

$$\begin{aligned} & productize\ (addToEnd\ \hat{v}\ v') \downarrow i \\ & \quad = \quad \textbf{if}\ i = (length(\hat{v}) + 1)\ \textbf{then}\ v'\ \textbf{else}\ (productize(\hat{v}) \downarrow i) \end{aligned} \tag{13}$$

We also use the auxiliary function *indexOf* to find the 1-based index of an element in a list.

$indexOf : \forall T.T \to List(T) \to Nat_\perp$
$indexOf = fix\ \lambda G\ .\ \lambda x\ .\ \lambda \hat{x}\ .\ \textbf{if}\ null\ \hat{x}\ \textbf{then}\ \perp\ \textbf{else if}\ (x = hd\ \hat{x})\ \textbf{then}\ 1\ \textbf{else}\ 1 + (G\ x\ (tl\ \hat{x}))$

2.3 Valuation Functions

2.3.1 Programs

The denotation of a program reflects the split in the main procedure and in the semantics. It returns a signature, Σ'', a Σ''-algebra, $\mathbf{A}''$, a state over that algebra (defined by the first part of the main procedure, "D_1; C_1"), and a function from Σ''-algebras to observations. The signature and algebra are obtained by elaborating the program's type and method declarations, and then hiding the primitive operations used to create objects and access fields of objects. The denotation of the main procedure is obtained by using the restricted algebra and signature, which ensures that the code in the main procedure cannot (successfully) use the primitives on objects, but must use the methods defined in the method declarations. Therefore nothing in the main procedure can depend on these primitives. This also means that the signature and algebra returned do not reflect these details of the representation of objects used in this semantics.

Some remarks about the notation used below are also in order. The type of $\mathcal{P}$ is a dependent type [11] [58, Chapter 8]; for example, the signature of the algebra returned is the same as the signature returned. Recall that the signature Σ^π and the Σ^π-algebra $\mathbf{A}^\pi$ used below are defined in Figures 4 and 6. The valuation functions for type declarations, $\mathcal{TD}$, and method declarations, $\mathcal{MD}$, are defined below. The valuation function for the main procedure, $\mathcal{M}$, also defined below, has a currying that reflects: the possibility of using the signature to do static checking on the syntax (for example, type checking), and the lack of dependence of the denotation produced on the particular algebra. Recall that the primitive operation symbols for creating objects, and for accessing their fields, are taken out of the signature by auxiliary function *hideInternalMessages*, which is defined above in the auxiliary functions for signatures. The notation $\mathbf{A}'|_{(hideInternalMessages\ \Sigma')}$ is the reduct of $\mathbf{A}'$ without these primitives; this notation is defined above in the auxiliary functions for algebras.

As a simplification in the semantics, we use $\perp$ both for nontermination and for any error condition. As in [57], we use a strict **let** construct; for example, the value of **let** $x = \perp$ **in** 3 is $\perp$.

$$\begin{aligned} & \mathcal{P} : \mathrm{Program} \to ((\Sigma'' : SIGS) \times (\mathbf{A}'' : Alg(\Sigma'')) \times STATE[\mathbf{A}''] \\ & \qquad \times((\mathbf{B} : Alg(\Sigma'')) \to OBSERVATION[\mathbf{B}]))_\perp \\ & \mathcal{P}[\![\mathrm{TD\ MD\ M}]\!] = \\ & \quad \textbf{let}\ (\Sigma, \mathbf{A}) = \mathcal{TD}[\![\mathrm{TD}]\!]\ \Sigma^\pi\ \mathbf{A}^\pi\ \textbf{in} \end{aligned}$$

let $(\Sigma', \mathbf{A}') = \mathcal{MD}[\![\text{MD}]\!]\ \Sigma\ \mathbf{A}$ **in**
let $(\Sigma'', \mathbf{A}'') = (\mathit{hideInternalMessages}\ \Sigma',\ A'|_{(\mathit{hideInternalMessages}\ \Sigma')})$ **in**
let $(s_1, f) = \mathcal{M}\ \Sigma''\ [\![\text{M}]\!]\ \mathbf{A}''$ **in**
$(\Sigma'', \mathbf{A}'', s_1, f)$

As noted earlier, the "output" of a program can be recovered from the information in the denotation of the program. One can use the following auxiliary function to do that.

run : Program $\rightarrow \mathit{ANSWERS}_\perp$
$\mathit{run}[\![\text{P}]\!] =$ **let** $(\Sigma'', \mathbf{A}'', s_1, f) = \mathcal{P}[\![\text{P}]\!]$ **in** $f\ \mathbf{A}''\ s_1$

2.3.2 Type Declarations

The elaboration of a type declaration produces an "updated" signature and algebra. It adds messages to the signature, and operations to the algebra, for creating objects of the type, and getting and setting their fields. Such messages are used by the denotational semantics only; to prevent their being used by programs, we give them names that are not nameable by programmers. The method for creating objects of a type T is named "creator(T)", and the methods for getting and setting a field named I are named "getter(I)" and "setter(I)" respectively. Note also that types cannot be recursive as the formal list that is elaborated with a signature does not contain an object of the same type. The lack of recursive types is a simplification; to allow recursive types we would either have to add built-in support for variant representations (tagged unions) or mutually recursive type declarations, and also use a fix-point construction for constructing the carrier sets.

In the definition below, $\mathcal{F}*$ is the semantic function for formal lists (given below), the *addType* function is from the auxiliary functions for signatures, and the other auxiliary functions are described below.

$\mathcal{TD}$: Type-Declaration $\rightarrow (\Sigma : \mathit{SIGS}) \rightarrow \mathit{Alg}(\Sigma) \rightarrow ((\Sigma' : \mathit{SIGS}) \times \mathit{Alg}(\Sigma'))_\perp$
$\mathcal{TD}[\![]\!]\ \Sigma\ \mathbf{A} = (\Sigma, \mathbf{A})$
$\mathcal{TD}[\![$`type` I `fields (`$\hat{\text{F}}$`)`$]\!]\ \Sigma\ \mathbf{A} =$
  let $\hat{F}' = \mathcal{F}*\ \Sigma\ [\![\hat{\text{F}}]\!]$ **in**
  let $\Sigma' = \mathit{addMessagesForTypeDecl}[\![\text{I}]\!]\ \hat{F}'\ (\mathit{addType}[\![\text{I}]\!]\ \Sigma)$ **in**
  $(\Sigma',\ \mathit{compileTypeDecl}\ \Sigma'\ [\![\text{I}]\!]\ \hat{F}'\ \mathbf{A})$
$\mathcal{TD}[\![\text{TD}_1\ \text{;}\ \text{TD}_2]\!]\ \Sigma\ \mathbf{A} =$ **let** $(\Sigma', \mathbf{A}') = \mathcal{TD}[\![\text{TD}_1]\!]\ \Sigma\ \mathbf{A}$ **in** $\mathcal{TD}[\![\text{TD}_2]\!]\ \Sigma'\ \mathbf{A}'$

Auxiliary functions used to update the signature There are two sets of auxiliary functions used in the elaboration of type declarations. The first set of auxiliary functions is for adding messages to the signature. The function *addMessagesForTypeDecl* adds the creator message, and the messages to get and set each field, to the signature.

$\mathit{addMessagesForTypeDecl}$: Identifier $\rightarrow$ *List*(Identifier $\times$ Type-Name) $\rightarrow \mathit{SIGS} \rightarrow \mathit{SIGS}_\perp$
$\mathit{addMessagesForTypeDecl}[\![\text{I}]\!]\ \hat{F}\ \Sigma =$
  $\mathit{addGetsAndSets}[\![\text{I}]\!]\ \hat{F}\ (\mathit{addMessage}[\![\text{creator(I)}]\!]\ (\mathit{formalTypes}\ \hat{F})\ [\![\text{I}]\!]\ \Sigma)$

The function *formalTypes* extracts the types from the denotation of a formal list.

$\mathit{formalTypes}$: *List*(Identifier $\times$ Type-Name) $\rightarrow$ *List*(Type-Name)

$$formalTypes\ \hat{F} = map\ (\lambda(I,T)\,.\,T)\ \hat{F}$$

The function *addGetsAndSets* adds messages to get and set each field to the signature. Recall that "getter(I_0)" and "setter(I_0)" are just special names that are not nameable by programmers.

$$\begin{array}{l}
addGetsAndSets : \text{Identifier} \to List(\text{Identifier} \times \text{Type-Name}) \to SIGS \to SIGS_\perp \\
addGetsAndSets[\![\text{I}]\!]\ \hat{F}\ \Sigma = \\
\quad (foldright \\
\qquad (\lambda((I_0,T_0),\Sigma')\,. \\
\qquad\quad (addMessage[\![\text{getter}(I_0)]\!]\ ([\![\text{I}]\!])\ [\![T_0]\!] \\
\qquad\qquad (addMessage[\![\text{setter}(I_0)]\!]\ ([\![\text{I}]\!],[\![T_0]\!])\ [\![\texttt{Void}]\!]\ \Sigma'))) \\
\qquad \Sigma \\
\qquad \hat{F})
\end{array}$$

Auxiliary functions used to update the algebra The second set of auxiliary functions is used to produce the new algebra with the methods for creating objects of the type, and accessing their fields. Aside from making up the methods, the work is done in *compileTypeDecl* itself. It uses *formalTypes* from above, the *set*... auxiliary functions on algebras defined above, and the functions *addGetAndSetOps* and *addCreatorOp* defined just below. For each type, the values of the type are locations, which the store maps to tuples of values of the fields (in the order in which they were declared). Each user-defined type is thus represented as an object type, and its objects are locations and thus potentially mutable. The domain of abstract values for the type (V below) is a tuple of the values of the fields of the object. Recall that "sortFor(I)" is just a way of forming a sort name from a type name.

$$\begin{array}{l}
compileTypeDecl : (\Sigma : SIGS) \to \text{Identifier} \to List(\text{Identifier} \times \text{Type-Name}) \\
\qquad\qquad \to Alg(\Sigma) \to Alg(\Sigma)_\perp \\
compileTypeDecl\ \Sigma\ [\![\text{I}]\!]\ \hat{F}\ \mathbf{A} = \\
\quad \textbf{let}\ \vec{T} = productize\ (formalTypes\ \hat{F})\ \textbf{in} \\
\quad \textbf{let}\ L = \{l_i^{\text{I}} \mid i \in Nat\}\ \textbf{in} \\
\quad \textbf{let}\ V = VALS_{\vec{T}}^{\mathbf{A}}\ \textbf{in} \\
\quad \textbf{let}\ \mathbf{A}' = (setTtoS\ ([\text{I} \mapsto \text{sortFor(I)}]\,TtoS^{\mathbf{A}}) \\
\qquad\qquad (setVALS\ ([\text{sortFor(I)} \mapsto V]([\text{I} \mapsto L]\,VALS^{\mathbf{A}})) \\
\qquad\qquad\quad (setLOCS\ ([\text{I} \mapsto L]LOCS^{\mathbf{A}}) \\
\qquad\qquad\qquad (setObjectTypes\ (\{\text{I}\} \cup ObjectTypes^{\mathbf{A}}) \\
\qquad\qquad\qquad\quad (setSORTS\ (\{\text{sortFor(I)}\} \cup SORTS^{\mathbf{A}})\ \mathbf{A}))))\ \textbf{in} \\
\quad addGetAndSetOps\ \Sigma\ [\![\text{I}]\!]\ \hat{F}\ (addCreatorOp\ \Sigma\ [\![\text{I}]\!]\ \vec{T}\ \mathbf{A})
\end{array}$$

The auxiliary function *addGetAndSetOps* uses the *addOp* auxiliary function defined for algebras to add each get and set operation. The operation interpretations (methods) themselves are created by passing appropriate parameters to the auxiliary functions *fetch* and *set*.

$$\begin{array}{l}
addGetAndSetOps : (\Sigma : SIGS) \to \text{Identifier} \to List(\text{Identifier} \times \text{Type-Name}) \\
\qquad\qquad \to Alg(\Sigma) \to Alg(\Sigma)_\perp \\
addGetAndSetOps\ \Sigma\ [\![\text{I}]\!]\ \hat{F}\ \mathbf{A} = \\
\quad (foldright
\end{array}$$

$$
\begin{array}{l}
(\lambda((I_0, T_0), \mathbf{A}')\,. \\
\qquad \textbf{let } \mathbf{A}'' = \mathit{addOp}[(\mathrm{I})][T_0]\ \Sigma\ \mathbf{A}'\ [\![\mathrm{getter}(I_0)]\!]\ (\mathit{fetch}[\mathrm{I}][T_0]\ [\![I_0]\!]\ \hat{F})\ \textbf{in} \\
\qquad \mathit{addOp}[(\mathrm{I}, T_0)][\texttt{Void}]\ \Sigma\ \mathbf{A}''\ [\![\mathrm{setter}(I_0)]\!]\ (\mathit{set}[\mathrm{I}][T_0]\ [\![I_0]\!]\ \hat{F})) \\
\mathbf{A} \\
\hat{F})
\end{array}
$$

The fields of an object are tuples of values, stored in the order in which the fields are declared. The denotation of the field declarations is used by the auxiliary function on lists, *indexOf*, to obtain the index into the tuple. Both *fetch* and *set* return a function which can be used in an algebra to interpret a method. The function produced by *set* updates the store, and returns nothing. Returning nothing is denoted by returning *, which the reader will recall is the only abstract value of the type `Void` in $\mathbf{A}^{\pi}$ (Figure 6). It is fine to assume that the algebra in question is based on $\mathbf{A}^{\pi}$, because *set* is only used in giving denotations to methods, and so is not involved in giving the denotation of the main procedure. Alternatively, we could have written "$\texttt{nothing}^{\mathbf{A}}((), [l \mapsto \vec{v}]\sigma)$" below instead of "$(*, [l \mapsto \vec{v}]\sigma)$", which is equivalent (because $\mathbf{A}$ must be based on $\mathbf{A}^{\pi}$ in this context).

$$
\begin{array}{l}
\mathit{fetch}[T][T'] : \mathrm{Identifier} \to \mathit{List}(\mathrm{Identifier} \times \mathrm{Type\text{-}Name}) \\
\qquad \to \mathit{VALS}^{\mathbf{A}}_{T} \times \mathit{STORE}[\mathbf{A}] \to (\mathit{VALS}^{\mathbf{A}}_{T'} \times \mathit{STORE}[\mathbf{A}])_{\perp} \\
\mathit{fetch}[T][T']\ [\![\mathrm{I}]\!]\ \hat{F} = \\
\qquad \lambda(l, \sigma)\,.\ \textbf{if } (l \notin \mathit{LOCS}^{\mathbf{A}}_{T})\ \textbf{then } \perp \\
\qquad\qquad \textbf{else let } i = \mathit{indexOf}\ (\mathrm{I}, T')\ \hat{F}\ \textbf{in} \\
\qquad\qquad\qquad ((\sigma\ l) \downarrow i,\ \sigma) \\
\mathit{set}[T][T'] : \mathrm{Identifier} \to \mathit{List}(\mathrm{Identifier} \times \mathrm{Type\text{-}Name}) \\
\qquad \to (\mathit{VALS}^{\mathbf{A}}_{T} \times \mathit{VALS}^{\mathbf{A}}_{T'}) \times \mathit{STORE}[\mathbf{A}] \to (\mathit{VALS}^{\mathbf{A}}_{T'} \times \mathit{STORE}[\mathbf{A}])_{\perp} \\
\mathit{set}[T][T']\ [\![\mathrm{I}]\!]\ \hat{F} = \\
\qquad \lambda((l, v), \sigma)\,.\ \ \textbf{if } (l \notin \mathit{LOCS}^{\mathbf{A}}_{T})\ \textbf{then } \perp \\
\qquad\qquad\qquad \textbf{else let } i = \mathit{indexOf}\ (\mathrm{I}, T')\ \hat{F}\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{let } \vec{v}' = [i \mapsto v](\sigma\ l)\ \textbf{in} \\
\qquad\qquad\qquad\qquad (*, [l \mapsto \vec{v}']\sigma)
\end{array}
$$

The function *addCreatorOp* adds an operation interpretation for the operation that creates an object. The operation interpretation, f below, uses the *alloc* function defined above on stores to allocate a location of the type holding the tuple of arguments of the creator.

$$
\begin{array}{l}
\mathit{addCreatorOp} : (\Sigma : \mathit{SIGS}) \to \mathrm{Identifier} \to \mathrm{Type\text{-}Name}^* \to \mathit{Alg}(\Sigma) \to \mathit{Alg}(\Sigma)_{\perp} \\
\mathit{addCreatorOp}\ \Sigma\ [\![I]\!]\ \vec{T}\ \mathbf{A} = \\
\qquad \textbf{let } f = (\lambda(\vec{v}, \sigma)\,.\ \textbf{if } \vec{v} \notin \mathit{VALS}^{\mathbf{A}}_{\vec{T}}\ \textbf{then } \perp\ \textbf{else } \mathit{alloc}[\mathrm{I}](\vec{v}, \sigma))\ \textbf{in} \\
\qquad \mathit{addOp}[\vec{T}][\mathrm{I}]\ \Sigma\ \mathbf{A}\ [\![\mathrm{creator}(\mathrm{I})]\!]\ f
\end{array}
$$

2.3.3 Formal Lists, Formals, and Type Names

The denotations of formal lists, formals, and type names are all relatively straightforward. The error checking they perform is encapsulated in $\mathcal{T}$, which checks to be sure the type named is defined in the signature. We assume that a Formal-List is actually a list.

$$
\begin{array}{l}
\mathcal{F}* : \mathit{SIGS} \to \mathrm{Formal\text{-}List} \to \mathit{List}(\mathrm{Identifier} \times \mathrm{Type\text{-}Name})_{\perp} \\
\mathcal{F}*\ \Sigma\ [\![\hat{F}]\!] = \mathit{map}\ (\mathcal{F}\ \Sigma)\ [\![\hat{F}]\!]
\end{array}
$$

$\mathcal{F} : SIGS \to \text{Formal} \to (\text{Identifier} \times \text{Type-Name})_\perp$
$\mathcal{F}\ \Sigma\ [\![\text{I} : \text{T}]\!] = \textbf{let}\ T' = \mathcal{T}\ \Sigma\ [\![\text{T}]\!]\ \textbf{in}\ (\text{I}, T')$

$\mathcal{T} : SIGS \to \text{Type-Name} \to \text{Identifier}_\perp$
$\mathcal{T}\ (TYPES, OPS, ResType)\ [\![T]\!] = \textbf{if}\ T \in TYPES\ \textbf{then}\ T\ \textbf{else}\ \perp$

2.3.4 Method Declarations

The valuation function for method declarations does not allow recursive methods, although this could be corrected by using a fixed-point construction. It uses the semantic function for bodies, $\mathcal{B}$, and an auxiliary function *bindActuals*, both of which are defined below.

$\mathcal{MD} : \text{Method-Declaration} \to (\Sigma : SIGS) \to Alg(\Sigma) \to ((\Sigma' : SIGS) \times Alg(\Sigma'))_\perp$
$\mathcal{MD}[\![]\!]\ \Sigma\ \mathbf{A} = (\Sigma, \mathbf{A})$
$\mathcal{MD}[\![\texttt{method}\ \text{I}(\hat{\text{F}}):\text{T}\ \{\text{B}\}]\!]\ \Sigma\ \mathbf{A} =$
  $\textbf{let}\ \hat{F}' = \mathcal{F}*\ \Sigma\ [\![\hat{\text{F}}]\!]\ \textbf{in}$
  $\textbf{let}\ T' = \mathcal{T}\ \Sigma\ [\![\text{T}]\!]\ \textbf{in}$
  $\textbf{let}\ f = (\lambda(\vec{v}, \sigma)\,.\,\mathcal{B}\ \Sigma\ [\![\text{B}]\!]\ \mathbf{A}\ (bindActuals[\mathbf{A}]\ \vec{v}\ \hat{F}',\ \sigma))\ \textbf{in}$
  $\textbf{let}\ \vec{S} = productize\ (formalTypes\ \hat{F}')\ \textbf{in}$
  $\textbf{let}\ \Sigma' = addMessage[\![\text{I}]\!]\ \vec{S}\ T'\ \Sigma\ \textbf{in}$
  $\textbf{let}\ \mathbf{A}' = addOp[\vec{S}][T']\ \Sigma'\ \mathbf{A}\ [\![\text{I}]\!]\ f\ \textbf{in}$
  $(\Sigma',\ \mathbf{A}')$
$\mathcal{MD}[\![\text{MD}_1\ ;\ \text{MD}_2]\!]\ \Sigma\ \mathbf{A} = \textbf{let}\ (\Sigma', \mathbf{A}') = \mathcal{MD}[\![\text{MD}_1]\!]\ \Sigma\ \mathbf{A}\ \textbf{in}\ \mathcal{MD}[\![\text{MD}_2]\!]\ \Sigma'\ \mathbf{A}'$

The binding of actuals to formals creates an environment. The folding process in the call to *foldright* passes the λ-abstraction an element of $\hat{F}$, which is a pair of an identifier and a type name, the forming environment, and the index of the element of the list $\hat{F}$; hence the notation (I_i, S_i) used below is accurate.

$bindActuals[\mathbf{A}] : (\ VALS^{\mathbf{A}})^* \to List(\text{Identifier} \times \text{Type-Name}) \to ENV[\mathbf{A}]_\perp$
$bindActuals[\mathbf{A}]\ \vec{v}\ \hat{F} =$
  $\textbf{let}\ \vec{S} = productize\ (formalTypes\ \hat{F})\ \textbf{in}$
  $\textbf{if}\ \vec{v} \notin VALS^{\mathbf{A}}_{\vec{S}}\ \textbf{then}\ \perp$
  $\textbf{else}\ \ \textbf{let}\ (\eta', n) =$
    $(foldright$
      $(\lambda((I_i, S_i), (\eta, i))\,.\,([I_i \mapsto (\vec{v} \downarrow i)]\eta,\ i \Leftrightarrow 1))$
      $(emptyEnviron,\ length\ \hat{F})$
      $\hat{F})$
  $\textbf{in}\ \eta'$

2.3.5 Bodies

The denotation of a method body could also be used for the bodies of procedures or local blocks, if π had procedures or local blocks. As such it is written in the tradition of denotational semantics, and relies on an algebra to do its work, and does not change the algebra.

$\mathcal{B} : (\Sigma : SIGS) \to \text{Body} \to (\mathbf{A} : Alg(\Sigma)) \to STATE[\mathbf{A}] \to (\ VALS^{\mathbf{A}} \times STORE[\mathbf{A}])_\perp$

$\mathcal{B}\ \Sigma\ [\![\mathrm{D\ C\ ;\ \texttt{return}\ E}]\!]\ \mathbf{A}\ s =$
$\quad$ **let** $(\eta_1, \sigma_1) = \mathcal{D}\ \Sigma\ [\![\mathrm{D}]\!]\ \mathbf{A}\ s$ **in**
$\quad$ **let** $\sigma_2 = \mathcal{C}\ \Sigma\ [\![\mathrm{C}]\!]\ \mathbf{A}\ (\eta_1, \sigma_1)$ **in**
$\quad \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_1, \sigma_2)$

2.3.6 Declarations of Constants

A constant declaration evaluates its expression, and returns a new state with a binding of the declared identifier to the expression's value. The signature is used to check that the type has been declared.

$\mathcal{D} : (\Sigma : SIGS) \to \mathrm{Declaration} \to (\mathbf{A} : Alg(\Sigma)) \to STATE[\mathbf{A}] \to STATE[\mathbf{A}]_\perp$
$\mathcal{D}\ \Sigma\ [\![]\!]\ \mathbf{A}\ s = s$
$\mathcal{D}\ \Sigma\ [\![\texttt{const}\ \mathrm{I}\texttt{:}\mathrm{T} = \mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma) =$
$\quad$ **let** $T' = \mathcal{T}\ \Sigma\ [\![\mathrm{T}]\!]$ **in**
$\quad$ **let** $(v, \sigma') = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma)$ **in**
$\quad$ **if** $v \notin VALS^{\mathbf{A}}_{T'}$ **then** $\perp$ **else** $([\mathrm{I} \mapsto v]\eta, \sigma')$
$\mathcal{D}\ \Sigma\ [\![\mathrm{D}_1\ ;\ \mathrm{D}_2]\!]\ \mathbf{A}\ s = \mathcal{D}\ \Sigma\ [\![\mathrm{D}_2]\!]\ \mathbf{A}\ (\mathcal{D}\ \Sigma\ [\![\mathrm{D}_1]\!]\ \mathbf{A}\ s)$

2.3.7 Expressions, Expression Lists

The meaning of an expression is found by either looking up an identifier in the environment, or by using the algebra and store to evaluate an operation. The semantic function for numerals, $\mathcal{N}$, has the following type.

$$\mathcal{N} : (\Sigma : SIGS) \to \mathrm{Numeral} \to (\mathbf{A} : Alg(\Sigma)) \to STORE[\mathbf{A}] \to (VALS^{\mathbf{A}}_{\mathtt{Int}} \times STORE[\mathbf{A}])_\perp .$$

The store is needed by algebras that represent integers using locations, as the denotations for numerals should be independent of the algebra. The details of $\mathcal{N}$ are left an exercise for the reader. The denotations for `true`, `false`, and `nothing` use the appropriate method in the algebra; hence these semantic equations are also independent of the particular algebra. Similarly, the meaning of a method call uses an operation in an algebra. While object creation and field access use operations in the algebra also, these are the specially named operations "creator(T)" and "getter(I)" that are specific to this semantics; hence calls to these special operations reflect a dependence of those expressions on the algebra. This is why such expressions are not allowed in the main procedure.

$\mathcal{E} : (\Sigma : SIGS) \to \mathrm{Expression} \to (\mathbf{A} : Alg(\Sigma)) \to STATE[\mathbf{A}] \to (VALS^{\mathbf{A}} \times STORE[\mathbf{A}])_\perp$
$\mathcal{E}\ \Sigma\ [\![\mathrm{I}]\!]\ \mathbf{A}\ (\eta, \sigma) = (\mathbf{let}\ v = \eta[\![\mathrm{I}]\!]\ \mathbf{in}\ (v, \sigma))$
$\mathcal{E}\ \Sigma\ [\![\mathrm{N}]\!]\ \mathbf{A}\ (\eta, \sigma) = \mathcal{N}\ \Sigma\ [\![\mathrm{N}]\!]\ \mathbf{A}\ \sigma$
$\mathcal{E}\ \Sigma\ [\![\texttt{true}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{true}^{\mathbf{A}}((), \sigma)$
$\mathcal{E}\ \Sigma\ [\![\texttt{false}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{false}^{\mathbf{A}}((), \sigma)$
$\mathcal{E}\ \Sigma\ [\![\texttt{nothing}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{nothing}^{\mathbf{A}}((), \sigma)$
$\mathcal{E}\ \Sigma\ [\![\mathrm{g(\hat{E})}]\!]\ \mathbf{A}\ (\eta, \sigma) =$
$\quad$ **let** $(\hat{v}, \sigma') = \mathcal{E}{*}\ \Sigma\ [\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta, \sigma)$ **in** $\mathrm{g}^{\mathbf{A}}(\mathit{productize}\ \hat{v}, \sigma')$
$\mathcal{E}\ \Sigma\ [\![\texttt{new}\ \mathrm{T(\hat{E})}]\!]\ \mathbf{A}\ (\eta, \sigma) =$
$\quad$ **let** $T' = \mathcal{T}\ \Sigma\ [\![\mathrm{T}]\!]$ **in**
$\quad$ **let** $(\hat{v}, \sigma') = \mathcal{E}{*}\ \Sigma\ [\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta, \sigma)$ **in** $\mathrm{creator}(T')^{\mathbf{A}}(\mathit{productize}\ \hat{v}, \sigma')$
$\mathcal{E}\ \Sigma\ [\![\mathrm{I}_1\ .\ \mathrm{I}_2]\!]\ \mathbf{A}\ (\eta, \sigma) =$

$$\textbf{let } (v,\sigma') = \mathcal{E}\ \Sigma\ [\![\mathrm{I}_1]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}\ \ \mathrm{getter}(\mathrm{I}_2)^{\mathbf{A}}(v,\sigma')$$

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

$$\mathcal{E}* : (\Sigma : SIGS) \to \text{Expression-List} \to (\mathbf{A} : Alg(\Sigma)) \to STATE[\mathbf{A}] \to (List(VALS^{\mathbf{A}}) \times STORE[\mathbf{A}])_\perp$$

$$\mathcal{E}*\ \Sigma\ [\![]\!]\ \mathbf{A}\ (\eta,\sigma) = (nil,\sigma)$$

$$\mathcal{E}*\ \Sigma\ [\![\hat{\mathrm{E}}\ \mathrm{E}_n]\!]\ \mathbf{A}\ (\eta,\sigma) =$$
$$\quad \textbf{let } (\hat{v},\sigma') = \mathcal{E}*\ \Sigma\ [\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}$$
$$\quad \textbf{let } (v_n,\sigma_n) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}_n]\!]\ \mathbf{A}\ (\eta,\sigma')\ \textbf{in}$$
$$\quad ((addToEnd\ \hat{v}\ v_n),\ \sigma_n)$$

2.3.8 Commands

The semantic functions for commands are fairly straight-forward. Note, however that the denotation of the field update command depends on the special operation "setter(I)", and thus is specific to this semantics. Hence the field update command is not allowed in the main procedure. On the other hand, in an `if`-command, the external value of the test is used, which makes the semantics for `if` independent of the algebra.

$$\mathcal{C} : (\Sigma : SIGS) \to \text{Command} \to (\mathbf{A} : Alg(\Sigma)) \to STATE[\mathbf{A}] \to STORE[\mathbf{A}]_\perp$$

$$\mathcal{C}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma) = \textbf{let } (v,\sigma') = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}\ \ \sigma'$$

$$\mathcal{C}\ \Sigma\ [\![\mathrm{I}_1\,.\,\mathrm{I}_2\ \texttt{:=}\ \mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma) =$$
$$\quad \textbf{let } (v',\sigma') = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}$$
$$\quad \textbf{let } (v_1,\sigma_1) = \mathcal{E}\ \Sigma\ [\![\mathrm{I}_1]\!]\ \mathbf{A}\ (\eta,\sigma')\ \textbf{in}$$
$$\quad \textbf{let } (v_*,\sigma_2) = \mathrm{setter}(\mathrm{I}_2)^{\mathbf{A}}((v_1,v'),\sigma_1)\ \textbf{in}$$
$$\quad \sigma_2$$

$$\mathcal{C}\ \Sigma\ [\![\mathrm{C}_1\texttt{;}\ \mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma) =$$
$$\quad \textbf{let } \sigma_1 = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}\ \ \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma_1)$$

$$\mathcal{C}\ \Sigma\ [\![\texttt{if}\ \mathrm{E}_1\ \texttt{then}\ \mathrm{C}_1\ \texttt{else}\ \mathrm{C}_2\ \texttt{fi}]\!]\ \mathbf{A}\ (\eta,\sigma) =$$
$$\quad \textbf{let } (v,\sigma') = \mathcal{E}\ \Sigma\ [\![\mathrm{E}_1]\!]\ \mathbf{A}\ (\eta,\sigma)\ \textbf{in}$$
$$\quad \textbf{if } externVal^{\mathbf{A}}_{\texttt{Bool}}(v,\sigma')\ \textbf{then } (\mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta,\sigma'))\ \textbf{else } (\mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma'))$$

2.3.9 Main Procedure

The meaning of the main procedure is a pair of a state and a function from algebras to observations. The state returned, (η,σ'), is produced by the first part of the main procedure, D_1 and C_1. The function from algebras to observations is defined by the second part of the main procedure, C_2 and D_2; this function takes an algebra, $\mathbf{B}$, and a state over $\mathbf{B}$, and, starting from the given state, computes a final state, and returns a finite function. The finite function is defined on the names declared in D_2, and for each such name gives its external value in the final state. The auxiliary function *typeEnvAndCheckVisible*, defined below, checks that all the declarations in D_2 are declarations of constants of visible type, and if so returns a type environment. This type environment is a finite function from identifiers to their types. Since the domain of this finite function, H, is just the names in D_2, this ensures that only those names are "output" by the program. (The domain $OBSERVATION[\mathbf{B}]$ is defined in Section 2.2.4 above.)

$$\mathcal{M} : (\Sigma : SIGS) \to \text{Program} \to (\mathbf{A} : Alg(\Sigma))$$

$$\rightarrow (STATE[\mathbf{A}] \times (\mathbf{B} : Alg(\Sigma) \rightarrow OBSERVATION[\mathbf{B}]))_\perp$$

$\mathcal{M}\ \Sigma\ [\![\ \texttt{main}\ \{\texttt{observe}\ \mathrm{D}_1\ \mathrm{C}_1\ \texttt{by}\ \mathrm{C}_2\ \mathrm{D}_2\}]\!]\ \mathbf{A} =$
  let $(\eta, \sigma) = \mathcal{D}\ \Sigma\ [\![\mathrm{D}_1]\!]\ \mathbf{A}\ (emptyEnviron, emptyStore)$ **in**
  let $\sigma' = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta, \sigma)$ **in**
  let $H = typeEnvAndCheckVisible[\![\mathrm{D}_2]\!]$ **in**
  let $f\ =\ (\lambda \mathbf{B}\ .\ \lambda(\eta_\mathbf{B}, \sigma_\mathbf{B})\ .$
    let $\sigma'_\mathbf{B} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ B\ (\eta_\mathbf{B}, \sigma_\mathbf{B})$ **in**
    let $\eta''_\mathbf{B}, \sigma''_\mathbf{B} = \mathcal{D}\ \Sigma\ [\![\mathrm{D}_2]\!]\ \mathbf{B}\ (\eta_\mathbf{B}, \sigma'_\mathbf{B})$ **in**
    $\lambda[\![I]\!]\ .$ **let** $T = H[\![I]\!]$ **in** $externVal^{\mathbf{B}}_T(\eta''_\mathbf{B}[\![I]\!], \sigma''_\mathbf{B}))$
  in $((\eta, \sigma'),\ f)$

The following auxiliary function returns checks declarations to see that they are declaring constants of types `Bool` or `Int`, and returns a finite function from the names declared to their types. Since this kind of finite function is a "type environment" (as opposed to a value environment), we use the auxiliary functions on environments defined above (and ask the reader's pardon for not redefining them with new types).

$typeEnvAndCheckVisible$: Declaration $\rightarrow$ (Identifier $\stackrel{\text{fin}}{\rightarrow}$ $VIS)_\perp$
$typeEnvAndCheckVisible[\![]\!] = emptyEnviron$
$typeEnvAndCheckVisible[\![\texttt{const}\ \mathrm{I}{:}\mathrm{T} = \mathrm{E}]\!] =$ **if** $T \notin VIS$ **then** $\perp$ **else** $bind(\mathrm{I}, \mathrm{T})$
$typeEnvAndCheckVisible[\![\mathrm{D}_1\ ;\ \mathrm{D}_2]\!] =$
  $overlay(typeEnvAndCheckVisible[\![\mathrm{D}_2]\!], typeEnvAndCheckVisible[\![\mathrm{D}_1]\!])$

2.4 Discussion

In this section we give an example of the semantics, and discuss some issues relating to our semantics of π and our semantic technique.

First we bring together the pieces of our running example, the program in Figure 2. The denotation of this program is a four-tuple. Recall that the first element of this tuple is the signature, $\Sigma^{\mathbf{D}}$, which was shown in Figure 5, and that the second element is the algebra, $\mathbf{D}$, which was shown in Figures 7 and 8. The third element is a state over $\mathbf{D}$, and the fourth is a function from states over $\Sigma^{\mathbf{D}}$-algebras to observations. The state is $(\eta_\mathbf{D}, \sigma_\mathbf{D})$, where the environment $\eta_\mathbf{D}$ satisfies:

$$\begin{aligned}
\eta_\mathbf{D}(\mathtt{w}) &= l_3^{\mathtt{Rect}} \\
\eta_\mathbf{D}(\mathtt{x}) &= l_4^{\mathtt{Rect}} \\
\eta_\mathbf{D}(\mathtt{y}) &= l_4^{\mathtt{Rect}} \\
\eta_\mathbf{D}(\mathtt{z}) &= l_0^{\mathtt{Point}}
\end{aligned}$$

and the store, $\sigma_\mathbf{D}$ satisfies:

$$\begin{aligned}
\sigma_\mathbf{D}(l_3^{\mathtt{Rect}}) &= (l_1^{\mathtt{Point}}, l_2^{\mathtt{Point}}) \\
\sigma_\mathbf{D}(l_4^{\mathtt{Rect}}) &= (l_1^{\mathtt{Point}}, l_0^{\mathtt{Point}}) \\
\sigma_\mathbf{D}(l_0^{\mathtt{Point}}) &= (2, 5) \\
\sigma_\mathbf{D}(l_2^{\mathtt{Point}}) &= (3, 4) \\
\sigma_\mathbf{D}(l_1^{\mathtt{Point}}) &= (1, 1).
\end{aligned}$$

This state is pictured in Figure 9. The function, f^{obs}, that is the fourth element of the denotation of the program in Figure 2 is given in Figure 10.

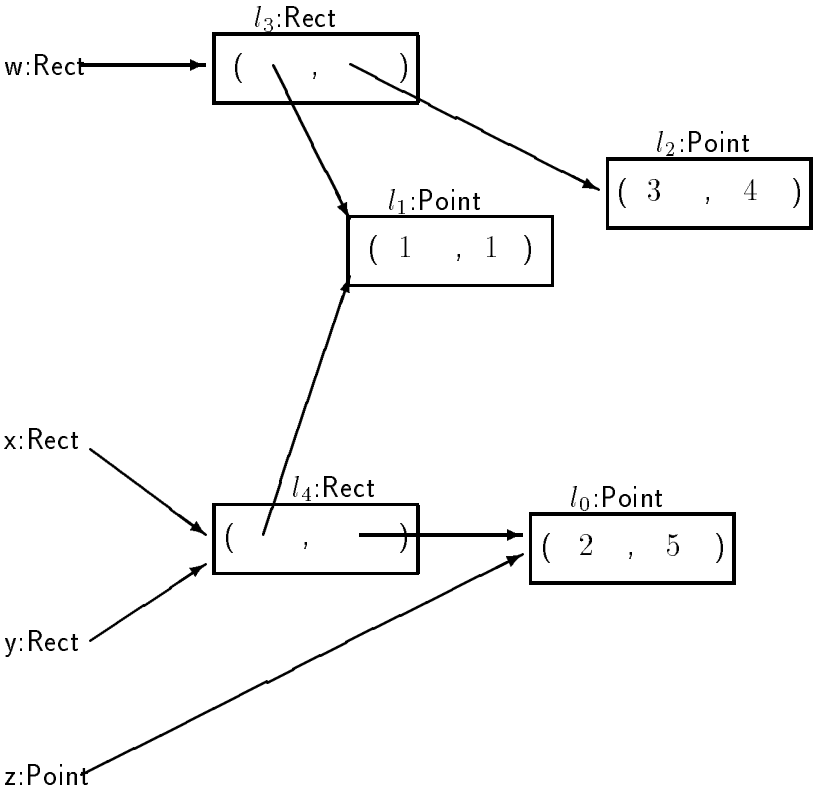

Figure 9: Picture of the state $(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$ over the algebra $\mathbf{D}$.

$$
\begin{array}{l}
f^{obs} = \textbf{let } H = \lambda[\![I]\!] \,.\, \textbf{if } I \in \{\texttt{shouldBe1}, \texttt{shouldBe5}, \texttt{shouldBe4}\} \textbf{ then } \texttt{Int} \textbf{ else } \bot \textbf{ in} \\
\quad (\lambda \mathbf{B} \,.\, \lambda(\eta, \sigma_0) \,. \\
\qquad \textbf{let } v_y = \eta\ \texttt{y} \textbf{ in} \\
\qquad \textbf{let } (v_{tr(y)}, \sigma_1) = \texttt{topRight}^{\mathbf{B}}((v_y), \sigma_0) \textbf{ in} \\
\qquad \textbf{let } (v_2, \sigma_1') = \mathcal{N}\ \Sigma\ [\![\texttt{2}]\!]\ \mathbf{B}\ \sigma_1 \textbf{ in} \\
\qquad \textbf{let } (v_5, \sigma_1'') = \mathcal{N}\ \Sigma\ [\![\texttt{5}]\!]\ \mathbf{B}\ \sigma_1' \textbf{ in} \\
\qquad \textbf{let } (v_p, \sigma_2) = \texttt{mkPoint}^{\mathbf{B}}((v_2, v_5), \sigma_1'') \textbf{ in} \\
\qquad \textbf{let } (v_b, \sigma_3) = \texttt{pointEqual}^{\mathbf{B}}((v_{tr(y)}, v_p), \sigma_2) \textbf{ in} \\
\qquad \textbf{let } \sigma_5 = \\
\qquad\quad \textbf{if } externVal^{\mathbf{B}}_{\texttt{Bool}}(v_b, \sigma_3) \\
\qquad\quad \textbf{then} \\
\qquad\qquad \textbf{let } v_w = \eta\ \texttt{w} \textbf{ in} \\
\qquad\qquad \textbf{let } v_1 = \mathcal{N}\ \Sigma\ [\![\texttt{1}]\!]\ \mathbf{B} \textbf{ in} \\
\qquad\qquad \textbf{let } (v_{ax}, \sigma_4) = \texttt{addX}^{\mathbf{B}}((v_w, v_1), \sigma_3) \textbf{ in} \\
\qquad\qquad \sigma_4 \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{let } (v_*, \sigma_4) = \texttt{nothing}^{\mathbf{B}}((), \sigma_3) \textbf{ in} \\
\qquad\qquad \sigma_4 \\
\qquad \textbf{let } (v_{bl(y)}, \sigma_6) = \texttt{botLeft}^{\mathbf{B}}((v_y), \sigma_5) \textbf{ in} \\
\qquad \textbf{let } (v_{a(bl(y))}, \sigma_7) = \texttt{abscissa}^{\mathbf{B}}((v_{bl(y)}), \sigma_6) \textbf{ in} \\
\qquad \textbf{let } \eta_7 = [\texttt{shouldBe1} \mapsto v_{a(bl(y))}]\eta \textbf{ in} \\
\qquad \textbf{let } v_z = \eta_7\ \texttt{z} \textbf{ in} \\
\qquad \textbf{let } (v_{o(z)}, \sigma_8) = \texttt{ordinate}^{\mathbf{B}}((v_z), \sigma_7) \textbf{ in} \\
\qquad \textbf{let } \eta_8 = [\texttt{shouldBe5} \mapsto v_{o(z)}]\eta_7 \textbf{ in} \\
\qquad \textbf{let } v_w = \eta_8\ \texttt{w} \textbf{ in} \\
\qquad \textbf{let } (v_{tr(w)}, \sigma_9) = \texttt{topRight}^{\mathbf{B}}((v_w), \sigma_8) \textbf{ in} \\
\qquad \textbf{let } (v_{a(tr(w))}, \sigma_{10}) = \texttt{abscissa}^{\mathbf{B}}((v_{tr(w)}), \sigma_9) \textbf{ in} \\
\qquad \textbf{let } \eta_{10} = [\texttt{shouldBe4} \mapsto v_{a(tr(w))}]\eta_8 \textbf{ in} \\
\qquad \lambda[\![I]\!] \,.\, \textbf{let } T = H[\![I]\!] \textbf{ in } externVal^{\mathbf{B}}_{T}(\eta_{10}[\![I]\!], \sigma_{10})
\end{array}
$$

Figure 10: The function, f^{obs}, that is produced as the fourth element of the semantics of the example program. Note that σ_5 is the store after executing the `if`-command following the `by` in the example program, and (η_{10}, σ_{10}) is the state after elaborating the declarations at the end of the program.

The above semantics may seem complex, and there is certainly more notation than in a traditional denotational semantics. Clearly the algebraic structures are also more complex than is traditional in the study of, for example, equational specification. Some of the notational complexities arise from our penchant for multi-sorted algebras and fairly exact types. But the majority come from the split in the semantics. First, we split the usual environment in two — the two pieces being the algebra (which can be thought of as an environment for methods, as it gives an interpretation to each operation symbol) and the usual (value) environment. We also split the domains in two – the two pieces being the carrier sets of the algebra and the usual domains for expressions, answers, etc. (which are not much in evidence in such a simple language). The notation is also made more complex by our passing the signature around explicitly, as a thing to be manipulated by the semantics; this could be easily fixed by writing the signature as a subscript, as in "$\mathcal{C}_\Sigma[\![\mathrm{C}_1;\mathrm{C}_2]\!]$ **A** s", but the signature is explicitly manipulated by the semantic functions for programs, type declarations, and method declarations, so we prefer the notation used in the paper.

What justifies this notational complexity, the split semantics, and ultimately any semantic description technique[1], is its usefulness in furthering the study of programming languages. We offer an example of the utility of the technique in the next section, where we study implementation concepts for ADTs. The main advantage of this framework is that one can study the semantics of either half of the semantics somewhat separately.

Although we have not done so, studying ways to give semantics to type and method declarations, or other languages for specifying such algebras might be fruitful. One could, for example, define the algebras not by programming, but by some specification method [18] or logic programming technique. Note especially that in Figure 8, we have given what amounts to an equational presentation of an algebra with mutable objects. The kind of mutation allowed is not just one-level mutation either; that is, it is not just variables containing pure values. Because of the store, one can have interesting aliasing relationships. Thus we believe the conventional view that equational specifications are not suitable for studying mutation might need revision. This idea is, of course, inspired by denotational semantics and functional programming ideas. However, we are not aware of any work studying equational specifications of mutable types which uses it. (Relationships with related work we do know about is discussed in Section 4 below.)

Our main interest, however, is in studying the denotational half of the semantics — the semantics of expressions, commands, etc. We are especially interested in studying behavioral properties of algebraic models with mutable types in the context of object-oriented programming. In such studies we often define languages that compute over models of ADTs, allowing us to start with the algebraic structures and compare their properties (for example, [30]). A small example of such a study appears in the next section.

However, we hasten to point out that others are also interested in questions of behavioral properties of programs. For example, full abstraction is such a question. Recent work on full abstraction has focused on the semantics of blocks and local variables. The work of O'Hearn and Tennent [47] [48] [65] and the work of Sieber [62] [63] [61] use logical relations (i.e., higher-order extensions of the kind of simulation relations we study in the next section), to obtain a restricted domain of procedure denotations. The idea is to restrict the domain of procedure denotations so that a procedure can only affect certain variables, not the entire store [39]. Reddy has investigated another approach to the full abstraction problem for local variables that uses techniques akin to object-oriented programming to prevent unwanted

[1] Yuri Gurevich, personal communication, March 1994.

modifications to the store [52]. We speculate that it might be possible to characterize, perhaps algebraically, varieties of algebras that have behavior that is appropriate, and that this might be of some interest in the study of full abstraction for programming languages with local variables. More probably, such a split semantics will be of use for studies of full abstraction for programming languages with ADT constructs and for object-oriented languages.

Granting the potential interest in our semantic description technique, the next logical question to ask is whether it has intrinsic limitations compared to standard ways of doing denotational semantics. We leave the detailed study of such questions for future work, but briefly raise some obvious issues.

The first issue is whether some of the features we left out of π are intrinsically beyond our techniques. While not exhausting this question by any means, we discuss two such extensions here: procedures and block structure.

We believe that it would be possible to add procedure declarations to the language with little conceptual difficulty. One simply has to pass a separate environment for procedures, or modify the domain of environments by introducing a domain of denotable values, one of whose summands would be a domain for procedures. Indeed, we have worked out such an extension, and it is interesting to note that the domain for (call-by-value) procedures (that return a result) would be something like the following.

$$Procedure[\Sigma] = (\mathbf{A} : Alg(\Sigma)) \to (VALS^{\mathbf{A}^*} \times STORE[\mathbf{A}]) \to (VALS^{\mathbf{A}} \times STORE[\mathbf{A}])_\perp$$

This emphasizes that such procedures would take an algebra (of the appropriate signature) as an argument, and compute over it. Such a domain would also serve for first-class procedures (call-by-value λ-expressions), except that instead of passing and returning only values from the algebra, one would pass and return denotable values, which would include such procedures as a summand. Alternatively, one could model first-class procedures as objects, and give them types and values in the algebra. (This would more closely correspond to Smalltalk [19].) We leave it as future work to work out the details of such approaches. Recursive procedures seem to pose no particular difficulties; and we leave as future work the working out of the appropriate domain theory for constructing algebras with recursive methods and recursive data. Work on continuous algebras may be of use in solving such problems [16] [70] [20] [43] [74, Section 3.3.3].

Another feature not in π is block structure, more specifically local declarations of types and methods. The semantics could be modified to allow the declaration of types and methods in local blocks without conceptual difficulties; however, in doing so it is more difficult to see the separation between the two halves of the semantics, as they would both be present in each block in the same way they are present in π as a whole.

3 Simulation Relations

As an example of the utility of our split semantics, we offer the following simple study of simulation between states over algebraic models. In this study we ignore type and method declarations, and focus on the behavior of their denotations, as observed by commands and declarations in the main procedure. Since we will be ignoring the details of type and method declarations, in what follows, let $\Sigma = (TYPES, OPS, ResType)$ stand for an arbitrary signature.

The simulation relations we study help one decide when one ADT behaves like another [59] [46] [28] [31]. This is useful in theoretical contexts [42]; also the general idea of

simulation and refinement of ADTs is important for optimization. If the abstraction of a faster implementation simulates a slower one's behavior, then one can replace the slower implementation by the faster implementation.

To define simulation relations appropriate for π and its semantics, one might think that the appropriate notion would simply relate abstract values. However, that would not take locations, and hence aliasing, 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. Even relating locations together with stores is not enough, since that would not take aliasing among identifiers into account. So the formulation of simulation relations we present relates states over one algebra to states over another algebra. (In this respect it is similar to the base case of the logical relations used, for example, in [47] [48] [65]. They are extensions of the relations used in [33].)

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

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

such that for each $(\eta_\mathbf{C}, \sigma_\mathbf{C}) \in STATE[\mathbf{C}]$ *and for each* $(\eta_\mathbf{A}, \sigma_\mathbf{A}) \in STATE[\mathbf{A}]$*, the following properties hold:*

well-formed: $(\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,(\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow (dom\ \eta_\mathbf{C}) \subseteq (dom\ \eta_\mathbf{A})$,

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

$$(\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,(\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow ([x \mapsto (\eta_\mathbf{C}\ y)]\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,([x \mapsto (\eta_\mathbf{A}\ y)]\eta_\mathbf{A}, \sigma_\mathbf{A}), \tag{14}$$

bistrict: $\perp\,\mathcal{R}\,\perp$*, and whenever* $s\,\mathcal{R}\,s'$ *and either* s *or* s' *is* $\perp$*, then so is the other,*

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* $\vec{x} : \vec{S} \in (dom\ \eta_\mathbf{C})$*, and for each identifier* $y : T$*,*

$$\begin{aligned}(\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,(\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow\ & \\ & (\textbf{let } (r_\mathbf{C}, \sigma'_\mathbf{C}) = g^\mathbf{C}((\eta_\mathbf{C}\ \vec{x}), \sigma_\mathbf{C}) \textbf{ in } ([y \mapsto r_\mathbf{C}]\eta_\mathbf{C},\ \sigma'_\mathbf{C})) \\ & \mathcal{R} \\ & (\textbf{let } (r_\mathbf{A}, \sigma'_\mathbf{A}) = g^\mathbf{A}((\eta_\mathbf{A}\ \vec{x}), \sigma_\mathbf{A}) \textbf{ in } ([y \mapsto r_\mathbf{A}]\eta_\mathbf{A},\ \sigma'_\mathbf{A})),\end{aligned} \tag{15}$$

shrinkable: *if* $(\eta'_\mathbf{C}, \sigma'_\mathbf{C}) \subseteq (\eta_\mathbf{C}, \sigma_\mathbf{C})$*, and* $(\eta'_\mathbf{A}, \sigma'_\mathbf{A}) \subseteq (\eta_\mathbf{A}, \sigma_\mathbf{A})$*, and* $(dom\,\eta'_\mathbf{C}) \subseteq (dom\,\eta'_\mathbf{A})$*, then* $(\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,(\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow (\eta'_\mathbf{C}, \sigma'_\mathbf{C})\,\mathcal{R}\,(\eta'_\mathbf{A}, \sigma'_\mathbf{A})$,

EXTERNALS-**identical:** *for each type* $T \in VIS$*, for each identifier* $x : T \in (dom\ \eta_\mathbf{C})$*, if* $(\eta_\mathbf{C}, \sigma_\mathbf{C})\,\mathcal{R}\,(\eta_\mathbf{A}, \sigma_\mathbf{A})$*, then*

$$externVal^\mathbf{C}((\eta_\mathbf{C}\ x), \sigma_\mathbf{C}) = externVal^\mathbf{A}((\eta_\mathbf{A}\ x), \sigma_\mathbf{A}).$$

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

The *EXTERNALS*-identical property ensures that a simulation is the identity on external representations.

In the shrinkable property, $(\eta'_\mathbf{C}, \sigma'_\mathbf{C}) \subseteq (\eta_\mathbf{C}, \sigma_\mathbf{C})$ means that for all types T, and for all identifiers $x : T$, $x : T \in (dom\ \eta'_\mathbf{C}) \Rightarrow (\eta'_\mathbf{C}\ x) = (\eta_\mathbf{C}\ x)$ and for all locations $l : T$, $l : T \in (dom\ \sigma'_\mathbf{C}) \Rightarrow (\sigma'_\mathbf{C}\ l) = (\sigma_\mathbf{C}\ l)$.

3.1 Examples of Simulation Relations

As a trivial example, the identity relation on $STATE[\mathbf{D}]_\perp \times STATE[\mathbf{D}]_\perp$ is a $\Sigma^{\mathbf{D}}$-simulation relation from our example algebra $\mathbf{D}$ to itself.

To build a more interesting example, we give a different $\Sigma^{\mathbf{D}}$-algebra, $\mathbf{E}$, in Figures 12 and 13. This algebra is built on a different Σ^{π}-algebra, $\mathbf{B}^{\pi}$ given in Figure 11. These algebras use objects for the booleans and integers. In this sense they represent what a "pure object-oriented" semantics might produce, and thus would be more faithful models of languages like Smalltalk, where everything is (at least conceptually) an object.

An example of a state over $\mathbf{E}$ that we would relate to our example state $(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$ is the state $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$, where $\eta_{\mathbf{E}}$ is such that

$$\begin{array}{lcl} (\eta_{\mathbf{E}}\ \mathtt{w}) & = & l_{10}^{\mathtt{Rect}} \\ (\eta_{\mathbf{E}}\ \mathtt{x}) & = & l_{9}^{\mathtt{Rect}} \\ (\eta_{\mathbf{E}}\ \mathtt{y}) & = & l_{9}^{\mathtt{Rect}} \\ (\eta_{\mathbf{E}}\ \mathtt{z}) & = & l_{8}^{\mathtt{Point}} \end{array}$$

and $\sigma_{\mathbf{E}}$ is defined as follows.

$$\begin{array}{lcl} (\sigma_{\mathbf{E}}\ l_{10}^{\mathtt{Rect}}) & = & (l_{6}^{\mathtt{Point}}, l_{7}^{\mathtt{Point}}) \\ (\sigma_{\mathbf{E}}\ l_{9}^{\mathtt{Rect}}) & = & (l_{6}^{\mathtt{Point}}, l_{8}^{\mathtt{Point}}) \\ (\sigma_{\mathbf{E}}\ l_{8}^{\mathtt{Point}}) & = & (l_{2}^{\mathtt{Int}}, l_{5}^{\mathtt{Int}}) \\ (\sigma_{\mathbf{E}}\ l_{7}^{\mathtt{Point}}) & = & (l_{3}^{\mathtt{Int}}, l_{4}^{\mathtt{Int}}) \\ (\sigma_{\mathbf{E}}\ l_{6}^{\mathtt{Point}}) & = & (l_{1}^{\mathtt{Int}}, l_{1}^{\mathtt{Int}}) \\ (\sigma_{\mathbf{E}}\ l_{5}^{\mathtt{Int}}) & = & 5 \\ (\sigma_{\mathbf{E}}\ l_{4}^{\mathtt{Int}}) & = & 4 \\ (\sigma_{\mathbf{E}}\ l_{3}^{\mathtt{Int}}) & = & 3 \\ (\sigma_{\mathbf{E}}\ l_{2}^{\mathtt{Int}}) & = & 2 \\ (\sigma_{\mathbf{E}}\ l_{1}^{\mathtt{Int}}) & = & 1 \end{array}$$

A picture of this state is given in Figure 14.

We now give an example of a simulation relation between states of $\mathbf{E}$ and states of $\mathbf{D}$ that would relate, for example, $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$ to $(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$. 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 are related. Given two stores over $\mathbf{E}$ and $\mathbf{D}$, the function

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

returns a predicate that tests two 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 $T \in \{\mathtt{Bool}, \mathtt{Int}\}$, for each pair of stores $(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})$, for each $v_{\mathbf{E}} \in VALS_T^{\mathbf{E}}$ and $v_{\mathbf{D}} \in VALS_T^{\mathbf{D}}$:

$$\mathcal{S}'_T(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(v_{\mathbf{E}}, v_{\mathbf{D}}) \stackrel{\text{def}}{=} (externVal_T^{\mathbf{E}}(v_{\mathbf{E}}, \sigma_{\mathbf{E}}) = externVal_T^{\mathbf{D}}(v_{\mathbf{D}}, \sigma_{\mathbf{D}})). \qquad (16)$$

For clarity, we emphasize that we are using a nonstrict interpretation of equality in which, for example, $\perp = 3$ is *false*. Therefore, if $externVal_T^{\mathbf{E}}(v_{\mathbf{E}}, \sigma_{\mathbf{E}})$ is $\perp$ and if $externVal_T^{\mathbf{D}}(v_{\mathbf{D}}, \sigma_{\mathbf{D}})$ is proper, then the result of $\mathcal{S}'_T(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(v_{\mathbf{E}}, v_{\mathbf{D}})$ is *false*.

$$SORTS^{\mathbf{B}^\pi} \stackrel{\text{def}}{=} TYPES \cup \{\text{sortFor}(\texttt{Bool}), \text{sortFor}(\texttt{Int})\}$$

$$ObjectTypes^{\mathbf{B}^\pi} \stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Int}\}$$

$$LOCS_T^{\mathbf{B}^\pi} \stackrel{\text{def}}{=} \{l_i^T \mid i \in Nat\}, \quad \text{for each } T \in ObjectTypes^{\mathbf{B}^\pi}$$

Type to Sort Mapping ($TtoS^{\mathbf{B}}$)

`Bool`	$\mapsto$	sortFor(`Bool`)
`Int`	$\mapsto$	sortFor(`Int`)

$$\begin{aligned}
VALS^{\mathbf{B}^\pi}_{\texttt{Void}} &\stackrel{\text{def}}{=} \{*\} \\
VALS^{\mathbf{B}^\pi}_{\text{sortFor(Bool)}} &\stackrel{\text{def}}{=} \{true, false\} \\
VALS^{\mathbf{B}^\pi}_{\text{sortFor(Int)}} &\stackrel{\text{def}}{=} \{0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\}
\end{aligned}$$

and for $T \in ObjectTypes^{\mathbf{B}^\pi}$, $VALS_T^{\mathbf{B}^\pi} \stackrel{\text{def}}{=} LOCS_T^{\mathbf{B}^\pi}$

$$\begin{aligned}
externVal^{\mathbf{B}^\pi}_{\texttt{Bool}}(l, \sigma) &\stackrel{\text{def}}{=} (\sigma\ l) \\
externVal^{\mathbf{B}^\pi}_{\texttt{Int}}(l, \sigma) &\stackrel{\text{def}}{=} (\sigma\ l)
\end{aligned}$$

Operation Interpretations

$$\begin{aligned}
\texttt{nothing}^{\mathbf{B}^\pi}((), \sigma) &\stackrel{\text{def}}{=} (*, \sigma) \\
\texttt{true}^{\mathbf{B}^\pi}((), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}](true, \sigma) \\
\texttt{false}^{\mathbf{B}^\pi}((), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}](false, \sigma) \\
\texttt{and}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}]((\sigma\ l_1) \wedge (\sigma\ l_2), \sigma) \\
\texttt{or}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}]((\sigma\ l_1) \vee (\sigma\ l_2), \sigma) \\
\texttt{not}^{\mathbf{B}^\pi}((l), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}](\neg(\sigma\ l), \sigma) \\
\texttt{0}^{\mathbf{B}^\pi}((), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Int}](0, \sigma) \\
\texttt{1}^{\mathbf{B}^\pi}((), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Int}](1, \sigma) \\
\texttt{add}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Int}]((\sigma\ l_1) + (\sigma\ l_2), \sigma) \\
\texttt{mult}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Int}]((\sigma\ l_1) \times (\sigma\ l_2), \sigma) \\
\texttt{negate}^{\mathbf{B}^\pi}((l), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Int}](\Leftrightarrow(\sigma\ l), \sigma) \\
\texttt{equal}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}]((\sigma\ l_1) = (\sigma\ l_2), \sigma) \\
\texttt{less}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}]((\sigma\ l_1) < (\sigma\ l_2), \sigma) \\
\texttt{leq}^{\mathbf{B}^\pi}((l_1, l_2), \sigma) &\stackrel{\text{def}}{=} alloc[\texttt{Bool}]((\sigma\ l_1) \leq (\sigma\ l_2), \sigma)
\end{aligned}$$

Figure 11: The Σ^π-algebra $\mathbf{B}^\pi$. Recall that *TYPES* means the *TYPES* of Σ^π.

$$SORTS^{\mathbf{E}} \stackrel{\text{def}}{=} TYPES \cup \{\text{sortFor}(\texttt{Int}), \text{sortFor}(\texttt{Bool}), \text{sortFor}(\texttt{Point}), \text{sortFor}(\texttt{Rect})\}$$

$$ObjectTypes^{\mathbf{E}} \stackrel{\text{def}}{=} ObjectTypes^{\mathbf{B}^{\pi}} \cup \{\texttt{Point}, \texttt{Rect}\}$$

$$LOCS^{\mathbf{E}}_{T} \stackrel{\text{def}}{=} \{l^{T}_{i} \mid i \in Nat\}, \quad \text{for each } T \in ObjectTypes^{\mathbf{E}}$$

Type to Sort Mappings ($TtoS^{\mathbf{E}}$) added to $TtoS^{\mathbf{B}^{\pi}}$

$\texttt{Point} \mapsto \text{sortFor}(\texttt{Point})$

$\texttt{Rect} \mapsto \text{sortFor}(\texttt{Rect})$

Abstract values added to $VALS^{\mathbf{B}^{\pi}}$

$$VALS^{\mathbf{E}}_{\text{sortFor}(\texttt{Point})} \stackrel{\text{def}}{=} \{(l_x, l_y) \mid l_x, l_y \in LOCS^{\mathbf{E}}_{\texttt{Int}}\}$$

$$VALS^{\mathbf{E}}_{\text{sortFor}(\texttt{Rect})} \stackrel{\text{def}}{=} \{(l_{bl}, l_{tr}) \mid l_{bl}, l_{tr} \in LOCS^{\mathbf{E}}_{\texttt{Point}}\}$$

and for $T \in ObjectTypes^{\mathbf{E}}$, $VALS^{\mathbf{E}}_{T} \stackrel{\text{def}}{=} LOCS^{E}_{T}$

$$externVal^{\mathbf{E}} \stackrel{\text{def}}{=} externVal^{\mathbf{B}^{\pi}}$$

Figure 12: The $\Sigma^{\mathbf{D}}$-algebra $\mathbf{E}$ (part 1). In this figure, $TYPES$ is from $\Sigma^{\mathbf{D}}$.

Operation Interpretations that are not also in $\mathbf{B}^{\pi}$

$$
\begin{array}{lcl}
\texttt{mkPoint}^{\mathbf{E}}((l_1,l_2),\sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Point}]((l_1,l_2),\sigma) \\
\texttt{abscissa}^{\mathbf{E}}((l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l) \textbf{ in } (l_1,\sigma) \\
\texttt{ordinate}^{\mathbf{E}}((l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l) \textbf{ in } (l_2,\sigma) \\
\texttt{addX}^{\mathbf{E}}((l^{\texttt{Point}},l^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{new},\sigma') = \texttt{add}^{\mathbf{E}}((l_1,l^{\texttt{Int}}),\sigma) \textbf{ in} \\
 & & (*,[l^{\texttt{Point}} \mapsto (l_{new},l_2)]\sigma') \\
\texttt{addY}^{\mathbf{E}}((l^{\texttt{Point}},l^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{new},\sigma') = \texttt{add}^{\mathbf{E}}((l_2,l^{\texttt{Int}}),\sigma) \textbf{ in} \\
 & & (*,[l^{\texttt{Point}} \mapsto (l_1,l_{new})]\sigma') \\
\texttt{upRightOf}^{\mathbf{E}}((l_1^{\texttt{Point}},l_2^{\texttt{Point}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{11},l_{12}) = (\sigma\ l_1^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{21},l_{22}) = (\sigma\ l_2^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{b1},\sigma') = (\texttt{leq}^{\mathbf{B}}(l_{11},l_{21})) \textbf{ in} \\
 & & \textbf{let } (l_{b2},\sigma'') = (\texttt{leq}^{\mathbf{B}}(l_{12},l_{22})) \textbf{ in} \\
 & & \texttt{and}^{\mathbf{B}}((l_{b1},l_{b2}),\sigma'') \\
\texttt{pointEqual}^{\mathbf{E}}((l_1^{\texttt{Point}},l_2^{\texttt{Point}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{11},l_{12}) = (\sigma\ l_1^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{21},l_{22}) = (\sigma\ l_2^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (l_{b1},\sigma') = (\texttt{equal}^{\mathbf{B}}(l_{11},l_{21})) \textbf{ in} \\
 & & \textbf{let } (l_{b2},\sigma') = (\texttt{equal}^{\mathbf{B}}(l_{12},l_{22})) \textbf{ in} \\
 & & \texttt{and}^{\mathbf{B}}((l_{b1},l_{b2}),\sigma'') \\
\texttt{mkRect}^{\mathbf{E}}((l_1,l_2),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_r,\sigma') = \mathit{alloc}[\texttt{Rect}]((l_1,l_2),\sigma) \textbf{ in} \\
 & & \textbf{let } (l_b,\sigma'') = \texttt{upRightOf}^{\mathbf{E}}(l_1,l_2) \textbf{ in} \\
 & & \textbf{if } \mathit{externVal}^{\mathbf{E}}_{\texttt{Bool}}(l_b,\sigma'') \textbf{ then } (l_r,\sigma'') \\
 & & \textbf{else } (l_r,[l_r \mapsto (l_2,l_1)]\sigma'') \\
\texttt{botLeft}^{\mathbf{E}}((l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l) \textbf{ in } (l_1,\sigma) \\
\texttt{topRight}^{\mathbf{E}}((l),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1,l_2) = (\sigma\ l) \textbf{ in } (l_2,\sigma) \\
\texttt{horizMove}^{\mathbf{E}}((l^{\texttt{Rect}},l^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr}) = (\sigma\ l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (v',\sigma') = \texttt{addX}^{\mathbf{E}}((l_{bl},l^{\texttt{Int}}),\sigma) \textbf{ in} \\
 & & \textbf{let } (v'',\sigma'') = \texttt{addX}^{\mathbf{E}}((l_{tr},l^{\texttt{Int}}),\sigma') \textbf{ in} \\
 & & (*,\sigma'') \\
\texttt{vertMove}^{\mathbf{E}}((l^{\texttt{Rect}},l^{\texttt{Int}}),\sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl},l_{tr}) = (\sigma\ l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (v',\sigma') = \texttt{addY}^{\mathbf{E}}((l_{bl},l^{\texttt{Int}}),\sigma) \textbf{ in} \\
 & & \textbf{let } (v'',\sigma'') = \texttt{addY}^{\mathbf{E}}((l_{tr},l^{\texttt{Int}}),\sigma') \textbf{ in} \\
 & & (*,\sigma'')
\end{array}
$$

Figure 13: The $\Sigma^{\mathbf{D}}$-algebra $\mathbf{E}$ (part 2).

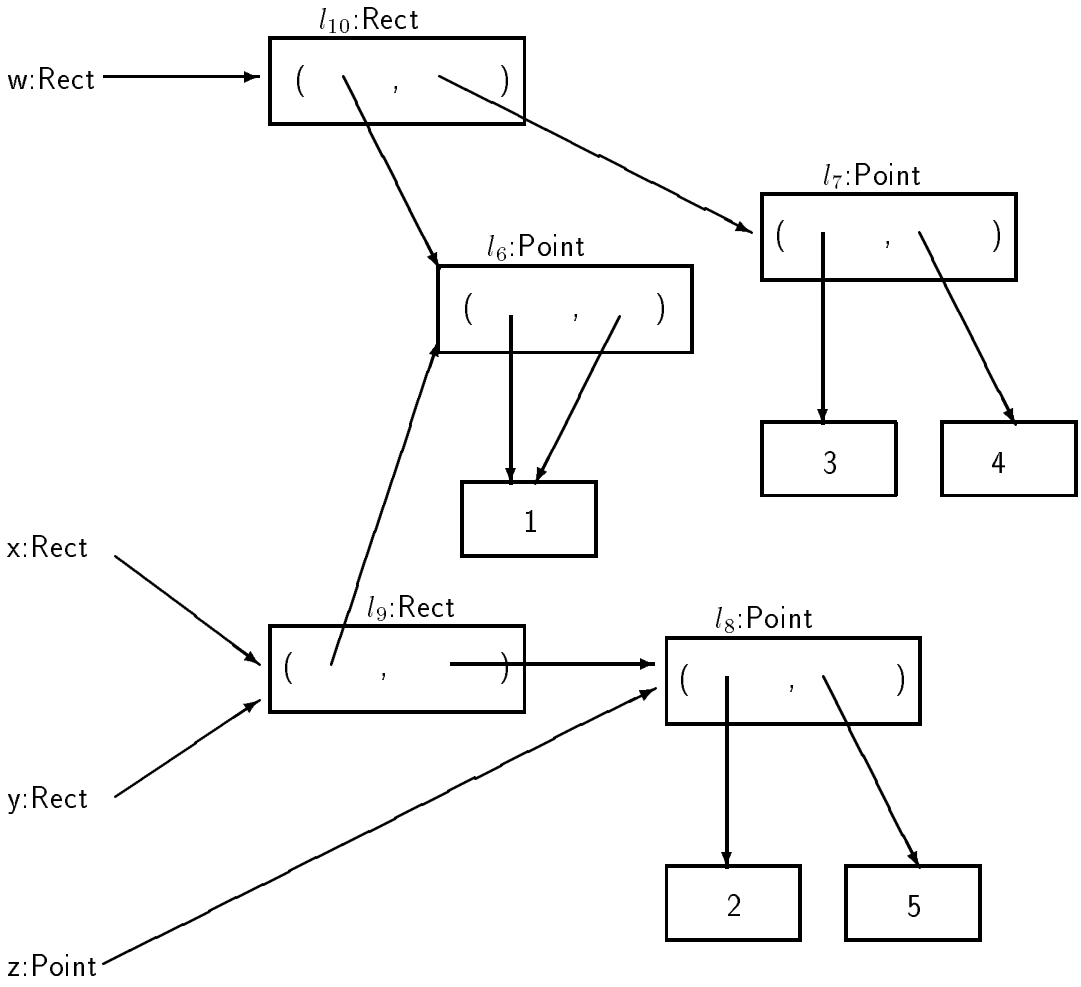

Figure 14: Picture of the state $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$ over the algebra $\mathbf{E}$.

For each pair of stores $(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})$, for each $v_{\mathbf{E}} \in VALS^{\mathbf{E}}_{\texttt{Void}}$ and $v_{\mathbf{D}} \in VALS^{\mathbf{D}}_{\texttt{Void}}$:

$$\mathcal{S}'_{\texttt{Void}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(v_{\mathbf{E}}, v_{\mathbf{D}}) \stackrel{\text{def}}{=} \mathit{true} \tag{17}$$

Point: For each pair of stores $(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})$ and locations $(l_{\mathbf{E}}, l_{\mathbf{D}}) \in (LOCS^{\mathbf{E}}_{\texttt{Point}} \times LOCS^{\mathbf{E}}_{\texttt{Point}})$,

$$\mathcal{S}'_{\texttt{Point}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_{\mathbf{E}}, l_{\mathbf{D}}) \stackrel{\text{def}}{=} \begin{cases} \mathit{true}, & \text{if } l_{\mathbf{E}} \notin \mathit{dom}(\sigma_{\mathbf{E}}) \wedge l_{\mathbf{D}} \notin \mathit{dom}(\sigma_{\mathbf{D}}) \\ \mathit{false}, & \text{if } (l_{\mathbf{E}} \in \mathit{dom}(\sigma_{\mathbf{E}})) \neq (l_{\mathbf{D}} \in \mathit{dom}(\sigma_{\mathbf{D}})) \\ b, & \text{if } (\sigma_{\mathbf{E}}\ l_{\mathbf{E}}) = (l_x, l_y),\ (\sigma_{\mathbf{D}}\ l_{\mathbf{D}}) = (v'_x, v'_y), \text{ and} \\ & b = (\mathcal{S}'_{\texttt{Int}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_x, v'_x) \wedge \mathcal{S}'_{\texttt{Int}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_y, v'_y)). \end{cases} \tag{18}$$

Rect: For each pair of stores $(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})$ and locations $(l_{\mathbf{E}}, l_{\mathbf{D}}) \in (LOCS^{\mathbf{E}}_{\texttt{Rect}} \times LOCS^{\mathbf{E}}_{\texttt{Rect}})$,

$$\mathcal{S}'_{\texttt{Rect}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l, l') \stackrel{\text{def}}{=} \begin{cases} \mathit{true}, & \text{if } l_{\mathbf{E}} \notin \mathit{dom}(\sigma_{\mathbf{E}}) \wedge l_{\mathbf{D}} \notin \mathit{dom}(\sigma_{\mathbf{D}}) \\ \mathit{false}, & \text{if } (l_{\mathbf{E}} \in \mathit{dom}(\sigma_{\mathbf{E}})) \neq (l_{\mathbf{D}} \in \mathit{dom}(\sigma_{\mathbf{D}})) \\ b, & \text{if } (\sigma_{\mathbf{E}}\ l_{\mathbf{E}}) = (l_{\mathbf{E},bl}, l_{\mathbf{E},tr}),\ (\sigma_{\mathbf{D}}\ l_{\mathbf{D}}) = (l_{\mathbf{D},bl}, l_{\mathbf{D},tr}), \text{ and} \\ & b = (\mathcal{S}'_{\texttt{Point}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_{\mathbf{E},bl}, l_{\mathbf{D},bl}) \wedge \mathcal{S}'_{\texttt{Point}}(\sigma_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_{\mathbf{E},tr}, l_{\mathbf{D},tr})). \end{cases} \tag{19}$$

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 a $\Sigma^{\mathbf{D}}$-algebra. This directed graph has as its nodes: the identifiers in $(\mathit{dom}\ \eta)$, the locations in $(\mathit{dom}\ \sigma)$ 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 $(\eta\ x) = l$.
- From a location $l : \texttt{Rect}$ to locations $l', l'' : \texttt{Point}$ if $(l', l'') = (\sigma\ l)$.

We write $\mathit{AliasG}(\eta, \sigma)$ for this graph. As an example, the $\mathit{AliasG}(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$ is shown in Figure 15. It is also a picture of $\mathit{AliasG}(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$.

We consider one aliasing graph to be mimicked in 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 pairs of node and edge sets representing two graphs, then $f = (f_n, f_e)$ is a graph homomorphism if and only if

$$(n, n') \in E_1 \Rightarrow (f_e\ (n, n')) = ((f_n\ n), (f_n\ n')). \tag{20}$$

We now have enough machinery to define an interesting $\Sigma_{\mathbf{E}}$-simulation relation from $\mathbf{E}$ to $\mathbf{D}$, which we will call $\mathcal{R}'$.

Definition 3.2 ($\mathcal{R}'$) *The $\Sigma^{\mathbf{D}}$-simulation relation $\mathcal{R}' \subseteq \mathit{STATE}[\mathbf{E}]_\perp \times \mathit{STATE}[\mathbf{D}]_\perp$ is defined such that $\perp \mathcal{R}' \perp$ and $(\eta_1, \sigma_1)\mathcal{R}'(\eta_2, \sigma_2)$ if and only if the following conditions all hold:*

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

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

Lemma 3.3 *The relation $\mathcal{R}'$ is a $\Sigma^{\mathbf{D}}$-simulation relation from $\mathbf{E}$ to $\mathbf{D}$.*

Proof Sketch: See Appendix A. ∎

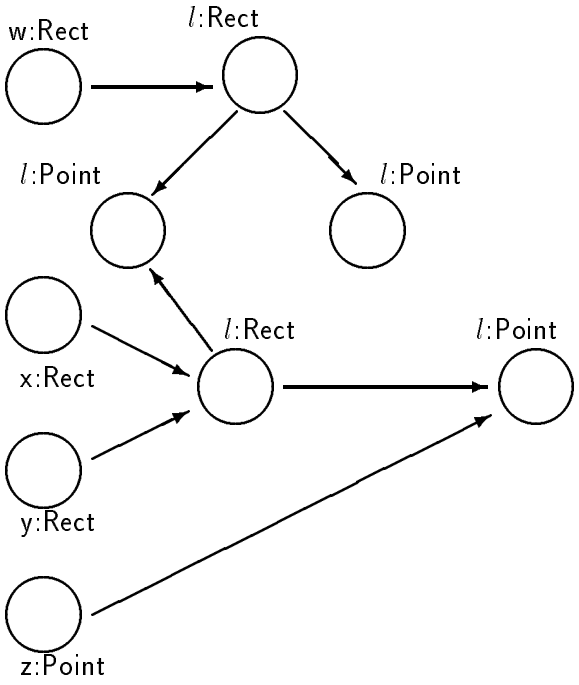

Figure 15: Aliasing graph for the states $(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$ and $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$.

3.2 Simulation is Preserved in the Main Procedure

In this section we show that simulation relations are preserved by the constructs of π that can appear in the main procedure, and thus by observations written in π. Since object creation expressions, field access expressions, and field update commands use primitives of the algebra that are specific to the semantics given in Section 2, they cannot be applied to all algebras and cannot appear in the main procedure. To avoid constantly describing the expressions, commands, and declarations that can appear in the main procedure, we make the following definition.

Definition 3.4 (main procedure expression, command, declaration) *In π, an expression (command) is a* main procedure expression (command) *if and only if it satisfies the grammar for* E′ (C′) *in Figure 16. A declaration of π is a* main procedure declaration *if and only if it satisfies the grammar for* D′ *in Figure 16 and in addition declares only constants of types* `Bool` *or* `Int`.

We also need a simple type system for π. The type of a main procedure expression is based on the type information in the algebra's signature, and a type environment, written H, obtained from the constant declarations in the main procedure. The notation $\Sigma; H \vdash E : T$ means E has type T in H; in this case we say that E is *well H-typed.* A command is *well H-typed* if and only if each of its subexpressions is well H-typed. We use the notation $\Sigma; H \vdash [\![D]\!] \Longrightarrow H'$ to mean that D is well H-typed, and produces the type environment H' when elaborated. See Appendix B for details.

We relate type environments to value environments in the following way.

Abstract Syntax of Main Procedure Expressions, Commands, Declarations:

$\mathrm{E}' \in$ Main-Proc-Expression $\quad \mathrm{E}'^* \in$ Main-Proc-Expression-List
$\mathrm{C}' \in$ Main-Proc-Command $\quad \mathrm{D}' \in$ Main-Proc-Declaration

$\mathrm{E}' ::= \mathrm{I} \mid \mathrm{N} \mid \texttt{true} \mid \texttt{false} \mid \texttt{nothing} \mid \mathrm{g(E}'^*\mathrm{)}$
$\mathrm{E}'^* ::= \mid \mathrm{E}'^* \; \mathrm{E}'$
$\mathrm{C}' ::= \mathrm{E}' \mid \mathrm{C}'_1 ; \; \mathrm{C}'_2 \mid \texttt{if } \mathrm{E}'_1 \texttt{ then } \mathrm{C}'_1 \texttt{ else } \mathrm{C}'_2 \texttt{ fi}$
$\mathrm{D}' ::= \mid \texttt{const I : T} = \mathrm{E}' \mid \mathrm{D}_1 \; ; \; \mathrm{D}_2$

Figure 16: Syntax of main procedure expressions, commands and declarations. All other syntax is as in the standard abstract syntax for π.

Definition 3.5 (H-environment, H-state) *Let* $\mathbf{A}$ *be a* Σ*-algebra and let* H *be a type environment. A value envionment,* $\eta \in ENV[\mathbf{A}]$*, is an* H-environment *if and only if for each* $I \in dom(H)$*, and for each type* T*, if* $H[\![I]\!] = T$ *then* $\eta[\![I]\!] \in VALS^{\mathbf{A}}_T$*. A state,* (η, σ) *is an* H-state *if and only if* η *is an* H*-environment.*

The following lemma says that simulation relations are preserved by the evaluation of a main procedure expression in related states. Recall that we use **let** as a strict binding mechanism.

Lemma 3.6 *Let* $\mathbf{B}$ *and* $\mathbf{A}$ *be* Σ*-algebras. Let* $\mathcal{R}$ *be a* Σ*-simulation relation from* $\mathbf{B}$ *to* $\mathbf{A}$*. Let* H *be a type environment.*

Then for all H*-states* $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \in STATE[\mathbf{B}]$ *and* $(\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \in STATE[\mathbf{A}]$ *such that* $(dom\ \eta_{\mathbf{B}}) \subseteq (dom\ \eta_{\mathbf{A}})$*, for each type* T*, for each main procedure expression* E *such that* $\Sigma; H \vdash \mathrm{E} : T$*, for each identifier* $y : T$*,*

$$
\begin{aligned}
&(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\, \mathcal{R}\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \Rightarrow \\
&\qquad (\textbf{let } (v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \textbf{in } ([y \mapsto v_{\mathbf{B}}]\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})) \\
&\qquad \mathcal{R} \\
&\qquad (\textbf{let } (v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})\ \textbf{in } ([y \mapsto v_{\mathbf{A}}]\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})).
\end{aligned}
$$

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

Let H-states $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \in STATE[\mathbf{B}]$ and $(\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \in STATE[\mathbf{A}]$ be given such that $(dom\ \eta_{\mathbf{B}}) \subseteq (dom\ \eta_{\mathbf{A}})$. Let T and E be given such that $\Sigma; H \vdash \mathrm{E} : T$. Let $y : T$ be given. Suppose that $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\, \mathcal{R}\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$.

(basis) If E is an identifier I of type T, then the result follows from the assumption and the bindable property. If E is a numeral, N, or one of `true`, `false`, or `nothing`, then the result follows from the substitution property.

(inductive step) Suppose that E has the form g(E*). Since g(E*) has type T, it must be that $\Sigma; H \vdash \mathrm{E}* : \vec{S}$ and $ResType(\mathrm{g}, \vec{S}) = T$, 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 $(\eta_{\mathbf{B},i-1}, \sigma_{\mathbf{B},i-1}) \in STATE[\mathbf{B}]$ and for all $(\eta_{\mathbf{A},i-1}, \sigma_{\mathbf{A},i-1}) \in STATE[\mathbf{A}]$ such that $(dom\ \eta_{\mathbf{B},i-1}) \subseteq (dom\ \eta_{\mathbf{A},i-1})$, for each type S_i and for each expression E_i such that $\Sigma; H \vdash \mathrm{E}_i : S_i$, and for each identifier $z_i : S_i$,

$$
\begin{array}{l}
(\eta_{\mathbf{B},i-1}, \sigma_{\mathbf{B},i-1})\ \mathcal{R}\ (\eta_{\mathbf{A},i-1}, \sigma_{\mathbf{A},i-1}) \Rightarrow \\
\qquad (\textbf{let}\ (d_i, \sigma_{\mathbf{B},i}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}_i]\!]\ \mathbf{B}\ (\eta_{\mathbf{B},i-1}, \sigma_{\mathbf{B},i-1})\ \textbf{in}\ ([z_i \mapsto d_i]\eta_{\mathbf{B},i-1}, \sigma_{\mathbf{B},i})) \\
\qquad \mathcal{R} \\
\qquad (\textbf{let}\ (e_i, \sigma_{\mathbf{A},i}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}_i]\!]\ \mathbf{A}\ (\eta_{\mathbf{A},i-1}, \sigma_{\mathbf{A},i-1})\ \textbf{in}\ ([z_i \mapsto e_i]\eta_{\mathbf{A},i-1}, \sigma_{\mathbf{A},i})).
\end{array}
$$

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. If at any stage, one result is $\perp$, then by the bistrict property, so is the other; otherwise the resulting states are related by the inductive hypothesis, and then the substitution property gives the desired result. The details are found in Appendix C. ∎

We do not know whether the converse of the above lemma holds. However, because the binding of the result of an expression to an identifier and a dynamic type check is what happens in the semantics of a constant declaration, the following corollary is immediate from the above lemma.

Corollary 3.7 *Let* **B** *and* **A** *be* Σ*-algebras, and* $\mathcal{R}$ *be a* Σ*-simulation relation from* **B** *to* **A**. *Let* H *be a type environment.*

For all H*-states* $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \in STATE[\mathbf{B}]$ *and for all* $(\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \in STATE[\mathbf{A}]$ *such that* $(dom\ \eta_{\mathbf{B}}) \subseteq (dom\ \eta_{\mathbf{A}})$, *for each main procedure declaration,* D, *such that* $\Sigma; H \vdash [\![\mathrm{D}]\!] \Longrightarrow H'$,

$$
\begin{array}{l}
((\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})) \Rightarrow \\
\qquad (\mathcal{D}\ \Sigma\ [\![\mathrm{D}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}))\quad \mathcal{R}\quad (\mathcal{D}\ \Sigma\ [\![\mathrm{D}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})).
\end{array}
$$
∎

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

Theorem 3.8 *Let* **B** *and* **A** *be* Σ*-algebras. Let* $\mathcal{R}$ *be a* Σ*-simulation relation from* **B** *to* **A**. *Let* H *be a type environment.*

For all H*-states* $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \in STATE[\mathbf{B}]$ *and* $(\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \in STATE[\mathbf{A}]$, *for all main procedure commands* C *such that* C *is well* H*-typed,*

$$
\begin{array}{l}
(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \Rightarrow \\
\qquad (\textbf{let}\ \sigma'_{\mathbf{B}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \textbf{in}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})) \\
\qquad \mathcal{R} \\
\qquad (\textbf{let}\ \sigma'_{\mathbf{A}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})\ \textbf{in}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}}))
\end{array}
$$

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

Let H-states $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \in STATE[\mathbf{B}]$ and $(\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \in STATE[\mathbf{A}]$ be given. Let C be given such that C is well H-typed. Suppose that $(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$.

(basis) Suppose that C is an expression, E. By Lemma 3.6, the result in one algebra is $\perp$ if and only if it is $\perp$ in the other. If the result is $\perp$, the required property follows. Otherwise, let $(v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = \mathcal{E}\ \Sigma\ [\![\mathbf{E}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})$, and let $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}\ \Sigma\ [\![\mathbf{E}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$. Then we can show the result as follows.

$(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$

$\Rightarrow$ $\langle$by Lemma 3.6$\rangle$
$([y \mapsto v_{\mathbf{B}}]\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathcal{R}\ ([y \mapsto v_{\mathbf{A}}]\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\Rightarrow$ $\langle$by shrinkable property of simulation relations$\rangle$
$(\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\Leftrightarrow$ $\langle$by definition of command denotation of an expression$\rangle$
$(\mathbf{let}\ \sigma'_{\mathbf{B}} = \mathcal{C}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathbf{in}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}'}))$
$\mathcal{R}$
$(\mathbf{let}\ \sigma'_{\mathbf{A}} = \mathcal{C}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})\ \mathbf{in}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}}))$

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

1. Suppose C is "C_1; C_2". Then the result follows by two applications of the inductive hypothesis.

2. Suppose C is "`if` E `then` C_1 `else` C_2 `fi`". Let y : `Bool` be a fresh identifier. Again, by Lemma 3.6, the result of E is $\perp$ in one algebra if and only if it is $\perp$ in the other, and again if it is $\perp$ we are done. Otherwise, let $(v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})$ and $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$. Then by the Lemma 3.6,

$$([y \mapsto v_{\mathbf{B}}]\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathcal{R}\ ([y \mapsto v_{\mathbf{A}}]\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}}). \tag{21}$$

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

$$externVal^{\mathbf{B}}_{\mathtt{Bool}}(v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = externVal^{\mathbf{A}}_{\mathtt{Bool}}(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}). \tag{22}$$

Hence the result of the test is the same in both **B** and in **A**. So starting with Equation (21), the result is shown as follows.

$([y \mapsto v_{\mathbf{B}}]\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathcal{R}\ ([y \mapsto v_{\mathbf{A}}]\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\Rightarrow$ $\langle$by the shrinkable property of simulation relations$\rangle$
$(\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathcal{R}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\Rightarrow$ $\langle$by the inductive hypothesis for C_1 and $\mathrm{C}_2\rangle$
$((\mathbf{let}\ \sigma''_{\mathbf{B}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathbf{in}\ (\eta_{\mathbf{B}}, \sigma''_{\mathbf{B}}))$
$\mathcal{R}\ (\mathbf{let}\ \sigma''_{\mathbf{A}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})\ \mathbf{in}\ (\eta_{\mathbf{A}}, \sigma''_{\mathbf{A}})))$
$\wedge$ $((\mathbf{let}\ \sigma''_{\mathbf{B}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathbf{in}\ (\eta_{\mathbf{B}}, \sigma''_{\mathbf{B}}))$
$\mathcal{R}\ (\mathbf{let}\ \sigma''_{\mathbf{A}} = \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})\ \mathbf{in}\ (\eta_{\mathbf{A}}, \sigma''_{\mathbf{A}})))$
$\Leftrightarrow$ $\langle$by Equation (22)$\rangle$
$(\mathbf{let}\ \sigma''_{\mathbf{B}}\ =\ \mathbf{if}\ externVal^{\mathbf{B}}_{\mathtt{Bool}}(v_{\mathbf{B}}, \sigma'_{\mathbf{B}})$
$\mathbf{then}\ \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})\ \mathbf{else}\ \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})$
$\mathbf{in}\ (\eta_{\mathbf{B}}, \sigma''_{\mathbf{B}}))$
$\mathcal{R}$ $(\mathbf{let}\ \sigma''_{\mathbf{A}}\ =\ \mathbf{if}\ externVal^{\mathbf{A}}_{\mathtt{Bool}}(v_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\mathbf{then}\ \mathcal{C}\ \Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})\ \mathbf{else}\ \mathcal{C}\ \Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})$
$\mathbf{in}\ (\eta_{\mathbf{A}}, \sigma''_{\mathbf{A}}))$
$\Leftrightarrow$ $\langle$by definition of $\mathcal{C}\rangle$
$(\mathbf{let}\ \sigma'_{\mathbf{B}} = \mathcal{C}\ \Sigma\ [\![\mathtt{if}\ \mathrm{E}\ \mathtt{then}\ \mathrm{C}_1\ \mathtt{else}\ \mathrm{C}_2\ \mathtt{fi}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathbf{in}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}}))$
$\mathcal{R}$ $(\mathbf{let}\ \sigma'_{\mathbf{A}} = \mathcal{C}\ \Sigma\ [\![\mathtt{if}\ \mathrm{E}\ \mathtt{then}\ \mathrm{C}_1\ \mathtt{else}\ \mathrm{C}_2\ \mathtt{fi}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})\ \mathbf{in}\ (\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}}))$

■

To summarize this section, with the split semantics one can investigate simulation relations between states using only the semantics of expressions, constant declarations, and commands in π. The detailed semantics of type and method declarations are suppressed, because one only deals with their denotations: algebras of a given signature.

4 Related Work

Our algebraic models are based, in part, on the work of Wing [73] and Chen [8]. These authors did not investigate simulation relations, or the use of mixed algebraic and denotational semantics.

Several authors use Kripke (i.e., possible world) models to give semantics to mutation and object identities [18] [1] [72] [25]. One could consider our algebras to be Kripke models if we were to include the store as part of the algebra. However, we believe that keeping the store outside the algebra leads to a cleaner separation of the algebra and the semantics of π. More importantly, it allows the denotational semantics to retain its characteristic referential transparency. Since Kripke models form a category, there are homomorphic functions on such models; however we do not know of anyone who has investigated relationships preserved by programs for Kripke models. For homomorphic relationships on Kripke models to be preserved by programs, the environment would also have to be included in the Kripke model, which would further destroy our separation between the language semantics and the semantics of the data types.

Evolving algebras [21] [22] are algebraic structures whose operations may be updated. The relationship between our algebraic models and evolving algebras seems to be that our state plus our algebra is very similar to an evolving algebra. The differences are that in giving a semantics to type and method declarations, the signature and everything else about our algebras changes, then in giving semantics to expressions and commands, only the state's environment and store can change. Once the type and method declarations are processed, in the "denotational half" of the semantics one can think of identifying the part of the Gurevich's algebra that evolves with our environment and store. Fixing one part of the "algebra" that evolves, namely not our algebra but our environment and store, makes our semantics more of a blend of denotational semantics and algebraic model theory than evolving algebras, as one can see by comparing our semantics with the evolving algebra semantics for C given in [23].

Action semantics [45] [71, Chapters 7–8] also uses algebraic techniques to specify data more abstractly than standard denotational semantics. However, in action semantics, the locations are not part of the algebra, and so the datatypes that one specifies must be immutable. Our semantic techniques allow one to obtain some of the benefits of action semantics without adopting its idiosyncratic notation. The idea of computing over an algebra is also found in [66].

Our separation between the algebra and the semantics of π is similar to what Mason and Talcott have studied the semantics of LISP [36] and other languages with mutation [37] [38]. Their work mainly uses operational semantics, and as such is complimentary to our denotational/algebraic approach. Mason and Talcott focus on equational logics for reasoning about programs that use mutation, whereas our work has not progressed to a reasoning calculus.

5 Future Work

In view of the use of dynamic logic [26] in some of the related work on Kripke models [72] [25], it would be interesting to investigate the relationship between our techniques and dynamic logic. Perhaps dynamic logic or other related axiomatic specification techniques [6] would be useful in specifying our models, or in developing their theory. Another possibility is developing the theory more fully by defining the appropriate category or institution [55].

Another direction for future work is to extend our results to more interesting languages. We mentioned some language extensions that would be interesting for investigating the utility of the split semantics in Section 2.4 above. One could also extend our results about simulation relations to languages with higher types; that is, one could investigate logical relations that use something like our simulation relations for a base case.

Our models have features for investigating observability notions, such as observable equivalence; we are trying to use these features to precisely define a notion of "observable aliasing" for objects of abstract data types [29]. But the main thrust of our investigations with these techniques is to extend our notions of legal behavioral subtyping in object-oriented programming to types with mutable objects [12] [13]. As a first step, one could use something like our simulation relations to algebraically characterize when one set of ADTs with mutable objects implements another.

6 Conclusions

The "split semantics" technique presented in this paper is a blend of algebraic and denotational semantics. We hope that it offers some of the advantages of both techniques, and that it might prove beneficial when investigating the semantics of languages with ADTs and mutable objects.

Our models of ADTs 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 ADTs with mutation. In support of this we have offered a notion of homomorphic relation (our simulation relations) and have shown that it is preserved by expressions and commands in a simple language. The addition of more realistic features to the language, such as loops, does not destroy this fundamental property [12] [13]. The semantics of the simple language, and the proof of this property demonstrate the utility of these techniques. When doing the proof, we ignored half the language's semantics, and started with the denotations of type and method declarations — that is, we started with our algebraic models and only used the (denotational-style) semantics of expressions and commands.

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 believe that our approach to describing the semantics of languages with ADTs — the split semantics — is a fruitful way to do semantics for such languages. Traditional denotational semantics "compiles" the implementations of ADTs into an undifferentiated mass of functions, Cartesian products, etc. (For example, see [56].) 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. This has made it difficult to apply the ideas of algebraic model theory to such languages, and has separated the worlds of algebraic and denotational semantics. By adding more structure to the semantics of a language with ADTs, that is, by compiling to an algebraic structure, one can have both a better organized semantics, and the possibility of using algebraic techniques to study the properties of either half of the language.

Acknowledgements

Thanks to Carolyn Talcott and Ian Mason for electronic discussions, comments on an earlier draft, and references. Thanks to Jacek Lesczylowski and Uday Reddy for comments and discussions. Thanks to Yuri Gurevich for discussions about the relation of this work to evolving algebras. Thanks to Don Pigozzi for many discussions, which have been helpful in formulating these ideas. Thanks also to the students of "Programming Languges 1" (Com S 541) of Spring semester 1994, whose feedback on initial versions of this work helped us explain it better.

Appendices

A Postponed Proof Sketch for Lemma 3.3

The following is the proof sketch of Lemma 3.3, which was postponed from the text. Recall that $\mathcal{R}'$ is defined in Definition 3.2, that $\Sigma^{\mathbf{D}}$ is defined in Figure 5, and that the $\Sigma^{\mathbf{D}}$-algebras $\mathbf{D}$ and $\mathbf{E}$ are defined in Figures 7, 8, 12, and 13.

Proof Sketch: Let $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}}) \in STATE[\mathbf{E}]$, and $(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}}) \in STATE[\mathbf{D}]$ be given. The proof proceeds by showing that $\mathcal{R}'$ has all the properties of a simulation relation.

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

bindable: Let T be a type of $\Sigma^{\mathbf{D}}$, let $x : T$ be an identifier, and let $y : T \in (dom\ \eta_{\mathbf{E}})$ be given. Suppose that $(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})\mathcal{R}'(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$. We show that $([x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})\mathcal{R}'([x \mapsto (\eta_{\mathbf{D}}\ y)]\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$ by using the definition of $\mathcal{R}'$. The only tricky part is to show that there is an injective graph homomorphism from $AliasG([x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$ to $AliasG([x \mapsto (\eta_{\mathbf{D}}\ y)]\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$. By construction of $\mathcal{R}'$, there is an injective graph homomorphism $f = (f_n, f_e)$ from $AliasG(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$ to $AliasG(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$. We define the required injective graph homomorphism $f' = (f'_n, f'_e)$ as follows. Let $f'_n \stackrel{\text{def}}{=} [x \mapsto x]f_n$; that is, $f'_n(x) = x$. If $(\eta_{\mathbf{E}}\ y) = l$ and $(\eta_{\mathbf{D}}\ y) = l'$, then let $f'_e \stackrel{\text{def}}{=} [(x, l) \mapsto (x, l')]f_e$. By definition, f' is a graph homomorphism. To show that f' is injective, we must show that f'_n and f'_e are injective. Since f_n is injective and since f_n is the identity on the identifiers in $(dom\ \eta_{\mathbf{E}})$, 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 $LOCS^E$.

$$
\begin{array}{ll}
 & (f'_e\ (x, l)) = (f'_e\ (u, l_u)) \\
\Rightarrow & \langle \text{by the homomorphism property} \rangle \\
 & ((f'_n\ x), (f'_n\ l)) = ((f'_n\ u), (f'_n\ l_u)) \\
\Rightarrow & \langle \text{by definition of equality for edges} \rangle \\
 & ((f'_n\ x) = (f'_n\ u)) \wedge ((f'_n\ l) = (f'_n\ l_u)) \\
\Rightarrow & \langle \text{by injectivity of } f'_n \rangle \\
 & (x = u) \wedge (l = l_u)
\end{array}
$$

bistrict: By construction, $\mathcal{R}'$ is bistrict.

substitution: To show that $\mathcal{R}'$ satisfies substitution property we must show it for all operations. The following example shows how the proof goes. 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 take as our example the operation `addX`.

We assume without loss of generality that the following hold.

$$\begin{array}{lcl}(\eta_{\mathbf{E}}\ \mathtt{p}) & = & l^{\mathtt{Point}}_{\mathbf{E}}\\ (\eta_{\mathbf{D}}\ \mathtt{p}) & = & l^{\mathtt{Point}}_{\mathbf{D}}\\ (\eta_{\mathbf{E}}\ \mathtt{i}) & = & l^{\mathtt{Int}}_{\mathbf{E}}\\ (\eta_{\mathbf{D}}\ \mathtt{i}) & = & v^{\mathtt{Int}}_{\mathbf{D}}\end{array}$$

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

Suppose that

$$(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})\ \mathcal{R}'\ (\eta_{\mathbf{D}}, \sigma_{\mathbf{D}}). \tag{23}$$

Consider the results of the calling `addX` in each algebra.

$$\begin{array}{lcl}\mathtt{addX}^{\mathbf{E}}((\eta_{\mathbf{E}}\ \mathtt{p}, \eta_{\mathbf{E}}\ \mathtt{i}), \sigma_{\mathbf{E}}) & \stackrel{\text{def}}{=} & \mathtt{addX}^{\mathbf{E}}((l^{\mathtt{Point}}_{\mathbf{E}}, l^{\mathtt{Int}}_{\mathbf{E}}), \sigma_{\mathbf{E}})\\ & = & \textbf{let } (l_1, l_2) = (\sigma_{\mathbf{E}}\ l^{\mathtt{Point}}) \textbf{ in}\\ & & \textbf{let } (l_{new}, \sigma'_{\mathbf{E}}) = \mathtt{add}^{\mathbf{E}}((l_1, l^{\mathtt{Int}}_{\mathbf{E}}), \sigma_{\mathbf{E}}) \textbf{ in}\\ & & (*, [l^{\mathtt{Point}}_{\mathbf{E}} \mapsto (l_{new}, l_2)]\sigma'_{\mathbf{E}})\\ \mathtt{addX}^{\mathbf{D}}((\eta_{\mathbf{D}}\ \mathtt{p}, \eta_{\mathbf{D}}\ \mathtt{i}), \sigma_{\mathbf{D}}) & = & \mathtt{addX}^{\mathbf{D}}((l^{\mathtt{Point}}_{\mathbf{D}}, v^{\mathtt{Int}}_{\mathbf{D}}), \sigma_{\mathbf{D}})\\ & \stackrel{\text{def}}{=} & \textbf{let } (v_1, v_2) = (\sigma_{\mathbf{D}}\ l^{\mathtt{Point}}_{\mathbf{D}}) \textbf{ in}\\ & & (*, [l^{\mathtt{Point}}_{\mathbf{D}} \mapsto (v_1 + v^{\mathtt{Int}}_{\mathbf{D}}, v_2)]\sigma_{\mathbf{D}})\end{array}$$

If the result of calling `addX` in $\mathbf{E}$ is $\perp$, this can only be because $l^{\mathtt{Point}}_{\mathbf{E}}$, l_1, or $l^{\mathtt{Int}}_{\mathbf{E}}$ is not in the domain of $\sigma_{\mathbf{E}}$. If $l^{\mathtt{Point}}_{\mathbf{E}} \notin dom(\sigma_{\mathbf{E}})$, then by definition of $\mathcal{R}'$, in particular the definition of $\mathcal{S}'_{\mathtt{Point}}$, it must be that $l^{\mathtt{Point}}_{\mathbf{D}} \notin dom(\sigma_{\mathbf{D}})$. Similarly using the definition of $\mathcal{S}'$, one can show that if the result of calling `addX` in $\mathbf{E}$ is $\perp$, then the result of calling `addX` in $\mathbf{D}$ must also be $\perp$. If both are $\perp$, then the substitution property holds; so we need not consider this case further.

The remaining case is if the result of calling `addX` in $\mathbf{E}$ is proper. There are two subcases: one is when the result in $\mathbf{D}$ is $\perp$, and the other when the result in $\mathbf{D}$ is proper. Suppose the result in $\mathbf{D}$ is $\perp$. Then by definition of $\mathbf{D}$, it must be that $l^{\mathtt{Point}}_{\mathbf{D}} \notin dom(\sigma_{\mathbf{D}})$, but then by definition of $\mathcal{S}'$, $l^{\mathtt{Point}}_{\mathbf{E}} \notin dom(\sigma_{\mathbf{E}})$, which is a contradiction, because (it is now assumed) the result of calling `addX` in $\mathbf{E}$ is proper. Hence it must be that, in this case, both results are proper.

In the case that both results are proper, we use σ_{E_r} and σ_{D_r} to refer to the stores in these final (or result) states, i.e., $\sigma_{E,r} = [l^{\mathtt{Point}}_{\mathbf{E}} \mapsto (l_{new}, l_2)]\sigma'_{\mathbf{E}}$ and $\sigma_{D,r} = [l^{\mathtt{Point}}_{\mathbf{D}} \mapsto (v_1 + v^{\mathtt{Int}}_{\mathbf{D}}, v_2)]\sigma_{\mathbf{D}}$. We will also refer to the store $\sigma'_{\mathbf{E}}$ to refer to the intermediate store defined in the course of evaluating $\mathtt{addX}^{\mathbf{E}}((\eta_{\mathbf{E}}\ \mathtt{p}, \eta_{\mathbf{E}}\ \mathtt{i}), \sigma_{\mathbf{E}})$. With this notation, what we have to show for the substitution property is as follows.

$$([y \mapsto *]\eta_{\mathbf{E}}, \sigma_{E_r})\ \mathcal{R}'\ ([y \mapsto *]\eta_{\mathbf{D}}, \sigma_{D_r}) \tag{24}$$

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

- By hypothesis, $(dom\ \eta_{\mathbf{E}}) \subseteq (dom\ \eta_{\mathbf{D}})$, and thus

$$(dom\ ([y \mapsto *]\eta_{\mathbf{E}})) \subseteq (dom\ ([y \mapsto *]\eta_{\mathbf{D}})).$$

- To show that the abstract values for each identifier are similar in the final states, let T be a type and let $z : T$ be an identifier in $(dom\ ([y \mapsto *]\eta_{\mathbf{E}}))$. We do this by cases.
  - Suppose that there is no path from z to $l_{\mathbf{E}}^{\mathtt{Point}}$ in the graph $AliasG(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$. Then the abstract value of z is unchanged. Hence, by the hypothesis:

    $$\mathcal{S}'_T(\sigma_{E_r}, \sigma_{D_r})((\eta_{\mathbf{E}}\ z), (\eta_{\mathbf{D}}\ z)). \tag{25}$$

  - If z is $\mathtt{p} : \mathtt{Point}$, we need to show $\mathcal{S}'_{\mathtt{Point}}(\sigma_{E_r}, \sigma_{D_r})(\eta_{\mathbf{E}}\ \mathtt{p}, \eta_{\mathbf{D}}\ \mathtt{p}))$. 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.

    $$\mathcal{S}'_{\mathtt{Int}}(\sigma'_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_{new}, v_1 + v_{\mathbf{D}}^{\mathtt{Int}}) \tag{26}$$

    $$\mathcal{S}'_{\mathtt{Int}}(\sigma'_{\mathbf{E}}, \sigma_{\mathbf{D}})(l_2, v_2) \tag{27}$$

    Condition (27) holds by hypothesis, which implies that the abstract values of $\mathtt{p}$ are related by $\mathcal{S}'_{\mathtt{Int}}$ in the original states. Condition (26) follows from the definition of $\mathtt{add}^{\mathbf{E}}$, and the relationships of the abstract values in the original states.
  - Otherwise z is an identifier that is not $\mathtt{p}$, but from which $\mathtt{p}$ is reachable along some path of $AliasG(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$. Note that $AliasG([y \mapsto *]\eta_{\mathbf{E}}, \sigma_{E_r}) = AliasG(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$, 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 $\mathtt{addX}$ in both algebras.
    - If T, the type of z, is $\mathtt{Point}$ then z and $\mathtt{p}$ must denote the same location in $[y \mapsto *]\eta_{\mathbf{E}}$. The same situation must hold in $[y \mapsto *]\eta_{\mathbf{E}}$, because there is a homomorphism from $AliasG([y \mapsto *]\eta_{\mathbf{E}}, \sigma_{E_r})$ to $AliasG([y \mapsto *]\eta_{\mathbf{D}}, \sigma_{D_r})$. Since z denotes the same location as $\mathtt{p}$ in both final states, the above argument shows that the abstract values of z are similar.
    - If T is $\mathtt{Rect}$, then one (or both!) of the corners of z is the same location as denoted by $\mathtt{p}$. 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 $AliasG(\eta_{\mathbf{E}}, \sigma_{\mathbf{E}})$ to $AliasG(\eta_{\mathbf{D}}, \sigma_{\mathbf{D}})$. We construct a new injective homomorphism $f' = (f'_n, f'_e)$ from $AliasG([y \mapsto *]\eta_{\mathbf{E}}, \sigma_{E_r})$ to $AliasG([y \mapsto *]\eta_{\mathbf{D}}, \sigma_{D_r})$ by simply letting $f'_n = f_n$ and $f'_e = f_e$. This suffices because the only new identifier does not have a type that matters, and because no new locations of type $\mathtt{Point}$ or $\mathtt{Rect}$ are introduced.

shrinkable: That $\mathcal{R}'$ satisfies the shrinkable property follows in a straightforward manner from its definition.

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

∎

[ident] $\Sigma; H \vdash I : T$ if $H[\![I]\!] = T$

[numeral] $\Sigma; H \vdash N$: `Int`

[true] $\Sigma; H \vdash$ `true` : `Bool`

[false] $\Sigma; H \vdash$ `false` : `Bool`

[nothing] $\Sigma; H \vdash$ `nothing` : `Void`

[method call] $$\frac{\Sigma; H \vdash E* : \vec{S},\ \Sigma \vdash \mathit{ResType}(\mathrm{g}, \vec{S}) = T}{\Sigma; H \vdash \mathrm{g}\texttt{(E*)} : T}$$

[empty decl] $\Sigma; H \vdash [\![]\!] \Longrightarrow \mathit{emptyEnviron}$

[const decl] $$\frac{\Sigma; H \vdash E : T}{\Sigma; H \vdash [\![\texttt{const I:T = E}]\!] \Longrightarrow [I \mapsto T]H}$$

[decls] $$\frac{\Sigma; H \vdash [\![\mathrm{D}_1]\!] \Longrightarrow H_1,\ \Sigma; H \vdash [\![\mathrm{D}_2]\!] \Longrightarrow H_2}{\Sigma; H \vdash [\![\mathrm{D}_1\texttt{;}\ \mathrm{D}_2]\!] \Longrightarrow H_2}$$

Figure 17: Type Inference Rules for π main procedure expressions and rules for inferring type environments for constant declarations.

B Types of Main Procedure Expressions and Declarations

Formal type inference rules for main procedure expressions and declarations are given in Figure 17. In the figure, H is a type environment, which is a finite function from identifiers to types. Such a type environment is produced by constant declarations; the formal rules for deriving a type environment from a constant declaration are given in the figure. The notation $\Sigma \vdash \mathit{ResType}(\mathrm{g}, \vec{S}) = T$ indicates that *ResType* comes from Σ.

C Omitted Details of the Proof of Lemma 3.6

The following proof has the omitted details for lemma 3.6. It picks up in the inductive step right where the proof in the main text leaves off.

Proof: Let the $z_i : S_i$ be distinct identifiers. We construct new states $(\eta_{\mathbf{B},n}, \sigma_{\mathbf{B},n})$ and $(\eta_{\mathbf{A},n}, \sigma_{\mathbf{A},n})$ such that the following hold for each $1 \leq i \leq n$:

$$(\hat{d}_n, \sigma_{\mathbf{B},n}) = \mathcal{E}* \ \Sigma \ [\![\mathrm{E*}]\!] \ \mathbf{B} \ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \quad (28)$$

$$\eta_{\mathbf{B},n}(z_i) = (\mathit{productize}\ \hat{d}_n) \downarrow i \quad (29)$$

$$(\hat{e}_n, \sigma_{\mathbf{A},n}) = \mathcal{E}* \ \Sigma \ [\![\mathrm{E*}]\!] \ \mathbf{A} \ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \quad (30)$$

$$\eta_{\mathbf{A},n}(z_i) = (\mathit{productize}\ \hat{e}_n) \downarrow i \quad (31)$$

$$(\eta_{\mathbf{B},n}, \sigma_{\mathbf{B},n}) \quad \mathcal{R} \quad (\eta_{\mathbf{A},n}, \sigma_{\mathbf{A},n}) \tag{32}$$

Once this is done, we can calculate as follows.

$$\begin{array}{ll} & \mathcal{E}\ \Sigma\ [\![\mathrm{g}(\mathrm{E}^*)]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \\ = & \langle \text{by definition of } \mathcal{E} \text{ and } \sigma_{\mathbf{B},n} \rangle \\ & \mathbf{let}\ (\hat{d}_n, \sigma_{\mathbf{B},n}) = \mathcal{E}*\ \Sigma\ [\![\mathrm{E}^*]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathbf{in}\ \mathrm{g}^{\mathbf{B}}((\mathit{productize}\ \hat{d}_n), \sigma_{\mathbf{B},n}) \\ = & \langle \text{by definition of } \eta_{\mathbf{B},n} \rangle \\ & \mathbf{let}\ (\hat{d}_n, \sigma_{\mathbf{B},n}) = \mathcal{E}*\ \Sigma\ [\![\vec{z}]\!]\ \mathbf{B}\ (\eta_{\mathbf{B},n}, \sigma_{\mathbf{B},n})\ \mathbf{in}\ \mathrm{g}^{\mathbf{B}}((\mathit{productize}\ \hat{d}_n), \sigma_{\mathbf{B},n}) \\ = & \langle \text{by definition of } \mathcal{E} \rangle \\ & \mathcal{E}\ \Sigma\ [\![\mathrm{g}(\vec{z})]\!]\ \mathbf{B}\ (\eta_{\mathbf{B},n}, \sigma_{\mathbf{B},n}) \end{array}$$

Similarly,

$$\mathcal{E}\ \Sigma\ [\![\mathrm{g}(\vec{z})]\!]\ \mathbf{A}\ (\eta_{\mathbf{A},n}, \sigma_{\mathbf{A},n}) = \mathcal{E}\ \Sigma\ [\![\mathrm{g}(\mathrm{E}^*)]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}).$$

Therefore, by the substitution property and Equation (32), the desired result follows.

$$\begin{array}{l} (\mathbf{let}\ (v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{g}(\mathrm{E}^*)]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})\ \mathbf{in}\ ([y \mapsto v_{\mathbf{B}}]\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})) \\ \mathcal{R} \\ (\mathbf{let}\ (v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}\ \Sigma\ [\![\mathrm{g}(\mathrm{E}^*)]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})\ \mathbf{in}\ ([y \mapsto v_{\mathbf{A}}]\eta_{\mathbf{A}}, \sigma'_{\mathbf{A}})) \end{array} \tag{33}$$

So it remains to construct the states $(\eta_{\mathbf{B},n}, \sigma_{\mathbf{B},n})$ and $(\eta_{\mathbf{A},n}, \sigma_{\mathbf{A},n})$ and the lists $\hat{d}_n$ and $\hat{e}_n$ with the required properties. These are constructed by induction on n. The construction only mentions the case where all the subexpressions have proper results; if at any stage one of the subexpressions has $\perp$ for a result in one algebra, by the bistrictness of $\mathcal{R}$ it must also be $\perp$, in the other, and then by the definition of π, the required relationship would hold.

For the basis, if $n = 0$, then E* is empty, so let $(\eta_{\mathbf{B},0}, \sigma_{\mathbf{B},0}) = (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})$, $(\eta_{\mathbf{A},0}, \sigma_{\mathbf{A},0}) = (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$, $\hat{d}_0 = \mathit{nil}$, and $\hat{e}_0 = \mathit{nil}$. 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 $(\eta_{\mathbf{B},j-1}, \sigma_{\mathbf{B},j-1})$, $(\eta_{\mathbf{A},j-1}, \sigma_{\mathbf{A},j-1})$, $\hat{d}_{j-1}$, and $\hat{e}_{j-1}$ satisfy the required properties. The required stores, along with values that will be used shortly, are constructed as follows.

$$(d_j, \sigma_{\mathbf{B},j}) \stackrel{\text{def}}{=} \mathcal{E}\ \Sigma\ [\![\mathrm{E}_j]\!]\ \mathbf{B}\ (\eta_{\mathbf{B},j-1}, \sigma_{\mathbf{B},j-1}) \tag{34}$$

$$(e_j, \sigma_{\mathbf{A},j}) \stackrel{\text{def}}{=} \mathcal{E}\ \Sigma\ [\![\mathrm{E}_j]\!]\ \mathbf{A}\ (\eta_{\mathbf{A},j-1}, \sigma_{\mathbf{A},j-1}). \tag{35}$$

We define the required environments and lists as follows.

$$\hat{d}_j \stackrel{\text{def}}{=} (\mathit{addToEnd}\ \hat{d}_{j-1}\ d_j) \tag{36}$$

$$\hat{e}_j \stackrel{\text{def}}{=} (\mathit{addToEnd}\ \hat{e}_{j-1}\ e_j) \tag{37}$$

$$\eta_{\mathbf{B},j} \stackrel{\text{def}}{=} [z_j \mapsto d_j]\eta_{\mathbf{B},j-1} \tag{38}$$

$$\eta_{\mathbf{A},j} \stackrel{\text{def}}{=} [z_j \mapsto e_j]\eta_{\mathbf{A},j-1}. \tag{39}$$

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

$$\begin{array}{ll} & \mathcal{E}*\ \Sigma\ [\![\mathrm{E}_1 \cdots \mathrm{E}_{j-1}\ \mathrm{E}_j]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}}) \\ = & \langle \text{by definition of } \mathcal{E}* \text{ and inductive hypothesis} \rangle \\ & \mathbf{let}\ (d_j, \sigma_{\mathbf{B},j}) = \mathcal{E}\ \Sigma\ [\![\mathrm{E}_j]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B},j-1})\ \mathbf{in}\ (\mathit{addToEnd}\ \hat{d}_{j-1}\ d_j), \sigma_{\mathbf{B},j}) \end{array}$$

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

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

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

$\eta_{\mathbf{B},j}(z_i)$
$=$ $\langle$by definition of $\eta_{\mathbf{B},j}$ and environment extension$\rangle$
if $(z_i = z_j)$ **then** d_j **else** $\eta_{\mathbf{B},j-1}(z_i)$
$=$ $\langle$by inductive hypothesis for $\eta_{\mathbf{B},j-1}$ and $\hat{d}_{j-1}\rangle$
if $(z_i = z_j)$ **then** d_j **else** $productize(\hat{d}_{j-1}) \downarrow i$
$=$ $\langle$by distinctness of the $z_k\rangle$
if $(i = j)$ **then** d_j **else** $productize(\hat{d}_{j-1}) \downarrow i$
$=$ $\langle$by Equation (13), i.e., the defining property of *productize* and *addToEnd*$\rangle$
$(productize(addToEnd\ \hat{d}_{j-1}\ d_j)) \downarrow i$
$=$ $\langle$by construction of $\hat{d}_j\rangle$
$(productize\ \hat{d}_j) \downarrow i$

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

Equation (32) thus follows directly from the main inductive hypothesis, because of the inductive assumption that $(\eta_{\mathbf{B},j-1}, \sigma_{\mathbf{B},j-1})\mathcal{R}(\eta_{\mathbf{A},j-1}, \sigma_{\mathbf{A},j-1})$. ∎

References

[1] A. J. Alencar and J. A. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, New York, N.Y., 1991.

[2] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[3] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Publishing Co., New York, N.Y., 1984. Revised Edition.

[5] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. ISBN 0-387-54060-1.

[6] Hans-Juergen Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[7] Kim B. Bruce and Peter Wegner. An algebraic model of subtype and inheritance. In Francois Bançilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., August 1990.

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

[9] A. Church. *The Calculi of Lambda Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, N.J., 1941. Reprinted by Klaus Reprint Corp., New York in 1965.

[10] William R. Cook. A denotational semantics of inheritance. Technical Report CS-89-33, Department of Computer Science, Brown University, Providence, Rhode Island, May 1989.

[11] N. G. de Bruijn. A survey of the project automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, Inc., New York, N.Y., 1980.

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

[13] Krishna Kishore Dhara and Gary T. Leavens. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 1992. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[14] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, N.Y., 1985.

[15] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[16] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[17] Joseph A. Goguen. Realization is universal. *Math. Systems Theory*, 6(4):359–374, 1973.

[18] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. The MIT Press, Cambridge, Mass., 1987.

[19] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[20] I. Guessarian. *Algebraic Semantics*, volume 99 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1981.

[21] Yuri Gurevich. Evolving algebras: A tutorial introduction. *Bulletin of the EATCS*, 43:264–284, February 1991.

[22] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[23] Yuri Gurevich and James K. Huggins. The semantics of the c programming language. In E. Börger et al., editors, *Computer Science Logic, Proceedings, 1992*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer-Verlag, New York, N.Y., 1992. Errata to appear in the 1993 Computer Science Logic proceedings. Corrected copy obtained from the authors.

[24] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[25] R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91, Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, Volume 2, Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 60–82. Springer-Verlag, New York, N.Y., April 1991.

[26] Dexter Kozen and J. Tiuryn. Logics of programs. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, New York, N.Y., 1990.

[27] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[28] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[29] Gary T. Leavens and Krishna Kishore Dhara. A foundation for the model theory of abstract data types with mutation and aliasing (preliminary version). Technical Report 92-35, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 1992. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[30] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. Technical Report 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.

[31] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. In Stephen Brookes, editor, *Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[32] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). *ACM SIGPLAN Notices*, 25(10):212–223, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[33] Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, August 1994. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[34] Gary Todd Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, Massachusetts Institute of Technology, December 1988. Published as MIT/LCS/TR-439 in February 1989.

[35] Barbara H. Liskov and Stephen N. Zilles. Specification techniques for data abstraction. *Transactions on Software Engineering*, 1(1), March 1975.

[36] I. A. Mason. *The Semantics of Destructive Lisp*. PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[37] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–328, July 1991.

[38] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on logic in computer science*. IEEE, 1992.

[39] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proc.* 15^{th} *Annual ACM Symp. on Principles of Programming Languages, San Diego*, pages 191–203. ACM, 1988.

[40] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., 1988.

[41] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[42] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, N.Y., 1990.

[43] B. Möller. On the algebraic specification of infinite objects—ordered and continuous models of algebraic types. *Acta Informatica*, 22:537–578, 1985.

[44] Peter D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, N.Y., 1990.

[45] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, N.Y., 1992.

[46] Tobias Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[47] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 217–238. Cambridge University Press, Cambridge, England, 1992.

[48] Peter W. O'Hearn and Robert D. Tennent. Relational parametricity and local variables. In *Proc.* 20^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 171–184, 1993.

[49] D. L. Parnas. A technique for the specification of software modules with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[50] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[51] F. Parisi Presicce and A. Pierantonio. An algebraic view of inheritance and subtyping in object oriented programming. In A. van Lamsweerde and A. Fugetta, editors, *ESEC '91: 3rd European Software Engineering Conference, Milan, Italy*, volume 550 of *Lecture Notes in Computer Science*, pages 364–379. Springer-Verlag, October 1991.

[52] Uday S. Reddy. Passivity and independence. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science, Paris, France*, pages 342–352. IEEE, July 1994.

[53] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[54] John C. Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[55] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 337–356.

[56] Scheifler. A denotational semantics of CLU. Technical Report TR-201, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1978.

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

[58] David A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.

[59] Oliver Schoett. Behavioural correctness of data representations. *Science of Computer Programming*, 14(1):43–57, June 1990.

[60] Oliver Schoett. An observational subset of first-order logic cannot specify the behavior of a counter. In C. Choffrut and M. Jantzen, editors, *STACS 91 8th Annual Symposium on Theoretical Aspects of Computer Science Hamburg, Germany, February 1991 Proceedings*, volume 480 of *Lecture Notes in Computer Science*, pages 499–510. Springer-Verlag, New York, N.Y., 1991.

[61] K. Sieber. Call-by-value and non-determinism. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 376–390, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.

[62] Kurt Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Proc. LMS Symposium on Applications of Categories in Computer Science, Durham 1991*, LMS Lecture Note Series 177, pages 258–269. Cambridge University Press, 1992.

[63] Kurt Sieber. New steps towards full abstraction for local variables. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages*, pages 88–100, Copenhagen, Denmark, 1993. Published as Yale University, Dept. of Comp. Sci. Technical Report YALEU/DCS/RR-968.

[64] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[65] R. D. Tennent. Correctness of data representations in Algol-like languages. In A. W. Roscoe, editor, *A Classical Mind, Essays in Honour of C. A. R. Hoare*, chapter 23, pages 405–417. Prentice-Hall International, 1994.

[66] J. V. Tucker and J. I. Zucker. Toward a general theory of computation and specification over abstract data types. In S. G. Akl, F. Fiala, and W. W. Koczkodaj, editors, *Advances in Computing and Information ICCI '90, International Conference on Computing and Information, Niagara Falls, Canada, May 1990*, volume 468 of *Lecture Notes in Computer Science*, pages 129–133. Springer-Verlag, New York, N.Y., 1991.

[67] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, Kensington, Australia, 1992. Draft of February 1992 obtained from the Author.

[68] Mark Utting and Ken Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, N.Y., 1992.

[69] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76. ACM, January 1989.

[70] E. G. Wagner, J. W. Thatcher, and J. B. Wright. Programming languages as mathematical objects. In J. Winkowski, editor, *Mathematical Foundations of Computer Science*, volume 10 of *Lecture Notes in Computer Science*, pages 84–101. Springer-Verlag, New York, N.Y., 1978.

[71] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice-Hall, New York, N.Y., 1991.

[72] R. J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masanaga, editors, *Deductive and object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 431–452. Springer-Verlag, New York, N.Y., 1991.

[73] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[74] Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, N.Y., 1990.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR93-21b
Submission Date: September 6, 1994