Subtyping for mutable types in object-oriented programming languages

Krishna Kishore Dhara and Gary T. Leavens

TR #92-36
November, 1992

Keywords: subtype, mutation, abstract data type, message passing.
1992 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — logics of programs. F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — algebraic approaches to semantics, denotational sematics.

Submitted to the European Conference on Object-Oriented Programming, ECOOP '93.

© Krishna Kishore Dhara and Gary T. Leavens, 1992. All rights reserved.

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

Subtyping for mutable types in object-oriented programming languages

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

November 24, 1992

Abstract

Subtype relationships in object-oriented programming languages are studied to aid code reuse and reasoning about programs that use subtype polymorphism. We define what it means for one abstract data type to be a subtype of another. This definition allows for both mutation and aliasing. This work gives intuition for programmers and guidance to language designers.

1 Introduction

We believe that mutation, aliasing, and subtyping should be semantic notions[1], [14]. That is, they should depend on specifications and observable behavior. In this paper we present observable definition of subtype relationships for arbitrary deterministic, abstract data types. Unlike previous work on subtyping our definition allows for both mutation and aliasing.

In the next section we show why deciding whether one type is a subtype of another is a non-trivial issue. In section 3 we briefly describe related work and in section 4 we describe our model. In sections 5 and 6 we define a simulation relations and legal subtyping. In section 7 we define a polymorphic object-oriented language and in the next section we give some properties of subtype relationships. We end with a discussion on subtyping between different commonly used abstract data types and present our conclusions.

2 Examples of the Problem

The following discussion highlights the problem of reasoning about object-oriented programs that use subtyping, and shows that arbitrary types cannot be in a subtype relationship. Further, we discuss why subtyping among mutable types is different than subtyping among immutable types.

As an example of the kinds of mutable types we wish to treat formally, consider the specifications of the types `Point`(points), `cPoint`(colored points), and `Rect`(rectangle) as shown in Figures 1, 2, and 3.

*This work was supported in part by the United States of America's National Science Foundation under Grant CCR-9108654.

```
Point
    class ops [p-new]
    instance ops [abscissa, ordinate, set-abscissa, set-ordinate]
    based on sort P from PointTrait

    op p-new(i:int, j:int) returns(p:Point)
        ensures p = <i,j>

    op abscissa(p:Point) returns(x:int)
        ensures x = first(<i,j>)

    op ordinate(p:Point) returns(y:int)
        ensures y = ordinate(<i,j>)

    op addX(p:Point, i:int) returns(Void)
        modifies at most p
        ensures p′ = <i + first(p),second(p)>

    op addY(p:Point, i:int) returns(Void)
        modifies at most p
        ensures p′ = <first(p), second(p) + i>
```

Figure 1: The type specification `Point`.

```
cPoint
subtype of Point
    class ops [cp-new]
    instance ops [color, set-color]
    based on sort C from CPointTrait

    op cp-new(i:int, j:int, code:int) returns(c:cPoint)
        ensures c = <i,j, code>

    op color(c:cPoint) returns(code:int)
        ensures code = third(c)

    op set-color(c:cPoint, code:int) returns(Void)
        modifies at most c
        ensures c′ = <first(p), second(p), code>
```

Figure 2: The type specification `cPoint`.

```
Rect
    class ops [r-new]
    instance ops [lower-left, upper-right, set-lleft, set-uright]
    based on sort R from RectangleTrait

    op p-new(p1:Point, p2:Point) returns(r:Rect)
        ensures r = <p1, p2>

    op lower-left(r:Rect) returns(p:Point)
        ensures p = first(r)

    op upper-right(r:Rect) returns(p:Point)
        ensures p = second(r)

    op set-lleft(r:Rect, p:Point) returns (Void)
        modifies at most r
        ensures r′ = <p, second(r)>

    op set-uright(r:Rect, p:Point) returns (Void)
        modifies at most r
        ensures r′ = <first(r), p>
```

Figure 3: The type specification `Rect`.

For legal subtyping among immutable types, it is enough that every instance of a subtype behaves like an instance of a supertype object. Since there is no mutation of immutable types, if instances of subtype behave as expected in one state, they will behave that way in all states. However, for mutable types this is no longer true. Subtype objects acting like supertype objects should respond to messages that mutate the object in a similar way to the objects of supertype. That is, instances of subtypes should behave as expected in all states and across state transformations. If this is not the case and we reason based on the supertype's specification, we get unexpected results from a program. This is shown in the following example.

Example 2.1 *Consider a type* `wcPoint` *which is like* `cPoint` *except for the following operation.*

op *abscissa(w:*`wcPoint`*)* **returns** *(i:*`Int`*)*

modifies at most w

ensures $i = first(w) \wedge w' = <first(w) + 1, second(w), third(w)>$

This says that the operation abscissa on `wcPoint` *returns abscissa of* `wcPoint` *object and increments it by 1. Let us suppose that* `wcPoint` *is a subtype of* `Point`*. Consider the following code which mutates a variable of type* `Point`*.*

```
program ( p:Point) returns ( old:Int, i:Int);
```

```
    begin
        old := abscissa(p);
        addX(p, 10);
        i := abscissa(p);
    end;
```

If we call program with p an object of `Point` *we see that its abscissa is incremented by 10. If we call program with p an object of* `wcPoint`*, then we see that its abscissa is incremented by 11.*

When p denotes an object of type `wcPoint`, the value returned, i, does not conform to what we expected would happen for of `Point`. If `wcPoint` were a valid subtype of `Point`, then the operation *abscissa* would increment its abscissa component by 10. If `wcPoint` were a subtype of `Point` then the values of old and i would be the same. So the first problem is to define legal subtyping for mutable types that rules out such surprising behavior.

Subtypes can have extra operations on them and, some of these might be mutators. In the presence of these extra mutators, aliasing can result in a surprising behavior. As an example of this consider the following.

Example 2.2 *Consider a type* `mcPoint` *which is like* `cPoint` *except for the following operation.*

op *set-color(m:*`mcPoint`*,c:*`Int`*)* **returns** *(*`Void`*)*

modifies at most m

ensures $m' = <\mathit{first}(m) + c,\ \mathit{second}(m) + c,\ c>$

This operation mutates the argument m, setting all its components. Let us suppose that `mcPoint` *is a subtype of* `Point`*. Consider the following program.*

```
program ( p:Point, m:mcPoint) returns ( i:Int, j:Int);
    begin
        j := abscissa(p);
        set-color(m, 1);
        i := abscissa(p);
    end;
```

Passing the program p and m that are not aliased gives same values for i and j. But, passing it a point p and an mcpoint m that are aliased directly (denoting the same object), we see that the values of i and j are different.

The results shown in the above example are surprising in that when m is an `mcPoint` and, p and m are aliased then $p := m$ does not act like a `Point`. Should we conclude that `mcPoint` cannot be a subtype of `cPoint` or is this a problem with the states in which we are executing? That is, is this a problem with subtyping or with aliasing between the subtype object and the supertype object?

So the questions are: how do we decide subtype relationships in general? What are the requirements needed for two abstract data types to have a subtype relationship in the presence of mutation and aliasing? In this paper we define a behavioral notion of subtyping to answer these questions.

3 Related Work

Bruce and Wegner [2] define subtyping in terms of binary relations on an algebra. They do not deal with incompletely specified data types. Leavens in [11], [14] defines subtyping using relations between algebras. These simulation relations were also extended to handle non-determinism. It was shown that no surprising results can occur when subtype objects are used in place of supertype objects. However, none of these deal with mutation.

In contrast to the above model-theoretic approaches, America uses a proof-theoretic approach to subtyping [1]. America defines subtyping with a transfer function and using pre and post conditions of the subtype and supertype operations.

In this paper we approach subtyping from a model-theoretic point of view. Simulation relations between algebras [11] are extended to deal with mutable types. This extension is not trivial as it also handles aliasing.

4 Algebraic Model

Our models of abstract types with mutable objects are somewhat nonstandard from the standpoint of denotational semantics, because they do not describe objects in terms of a few basic types, such as products, sums, and functions. Instead, they more closely resemble the models of equational algebraic specifications, having carrier sets and operations. Our particular models are most strongly inspired by the work of Wing [21] and Chen [5].

We take the view that objects can be thought of as typed memory cells containing abstract values. The values may contain locations, which are the names of memory cells. Abstract values can be thought of as mathematical abstractions of the concrete representations used in programs [9]. Locations can be thought of as the names of objects; these are sometimes called object identifiers. Locations are typed in the same sense that variables are typed; a location $l : T$ can store an abstract value of a subtype of T.

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

Definition 4.1 *A* signature*, Σ, is a tuple* $(TYPES, VIS, SORTS, VARS, TtoS, OPS, \leq, ResType)$, *where*

- *TYPES is a non-empty set of type symbols, such that* `Void` $\in$ *TYPES.*

- *$VIS \subseteq TYPES$ is a set of visible type symbols, such that* `Bool` $\in VIS$
- *$SORTS$ is a non-empty set of sort symbols,*
- *$VARS$ is a set of variable symbols, indexed by $TYPES$,*
- *$TtoS : TYPES \to SORTS$, is an an injective mapping,*
- *OPS, is a set of operation symbols, indexed by $TYPES^* \times TYPES$,*
- *$\leq$, is a preorder on types, called the presumed subtype relation,*
- *$ResType : OPS, TYPES^* \to TYPES$, is a a partial function which returns the nominal type of an operation symbol applied to a tuple of arguments with the given types. $ResType$ should be monotone, in the following sense [18]: for all $g \in OPS$, and for all tuple of types $\vec{S} \leq \vec{T}$, if $ResType(g, \vec{T})$ is defined, then so is $ResType(g, \vec{S}) \leq ResType(g, \vec{T})$.*

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

The monotonicity requirement, taken from Reynold's category sorted algebras [18], allows operations to model message passing. It permits a style of message passing similar to CLOS, with all the argument types determining the operation called [14].

Figure 4 gives an example signature, Σ_E. Its mutable types are `Point`, `cPoint` and `Rect`. That the objects of these types are mutable can be seen from the signature of operations such as `addX` and `set-ll`, which have no results.

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

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

Definition 4.2 *A Σ-algebra is a tuple, $(LOCS^A, |A|, OPS^A)$, where*

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

$$g^A : (DVAL^A_{S_1} \times \cdots \times DVAL^A_{S_n}) \times STORE[A] \to \bigcup_{U \leq T} DVAL^A_U \times STORE[A],$$

$$VIS_E \stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Int}\}$$
$$TYPES_E \stackrel{\text{def}}{=} \{\texttt{Void}, \texttt{Bool}, \texttt{Int}, \texttt{Point}, \texttt{cPoint}, \texttt{Rect}\}$$
$$SORTS_E \stackrel{\text{def}}{=} \{\texttt{Unit}, \texttt{Boolean}, \texttt{Integer}, \texttt{PointSort}, \texttt{cPointSort}, \texttt{RectSort}\}$$

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

Type to Sort Mapping ($TtoS$)

`Void`	$\mapsto$	`Unit`
`Bool`	$\mapsto$	`Boolean`
`Int`	$\mapsto$	`Integer`
`Point`	$\mapsto$	`PointSort`
`cPoint`	$\mapsto$	`cPointSort`
`Rect`	$\mapsto$	`RectSort`

Operation symbols (OPS) and ($ResType$)

`()`	:	$\langle\rangle \to$ `Void`
`true`	:	$\langle\rangle \to$ `Bool`
`false`	:	$\langle\rangle \to$ `Bool`
`and`	:	$\langle$`Bool`, `Bool`$\rangle \to$ `Bool`
`or`	:	$\langle$`Bool`, `Bool`$\rangle \to$ `Bool`
`not`	:	$\langle$`Bool`$\rangle \to$ `Bool`
`0`	:	$\langle\rangle \to$ `Int`
`1`	:	$\langle\rangle \to$ `Int`
`add`	:	$\langle$`Int`, `Int`$\rangle \to$ `Int`
`mult`	:	$\langle$`Int`, `Int`$\rangle \to$ `Int`
`negate`	:	$\langle$`Int`$\rangle \to$ `Int`
`equal`	:	$\langle$`Int`, `Int`$\rangle \to$ `Bool`
`p − new`	:	$\langle$`Int`, `Int`$\rangle \to$ `Point`
`abscissa`	:	$\langle$`Point`$\rangle \to$ `Int`
`ordinate`	:	$\langle$`Point`$\rangle \to$ `Int`
`addX`	:	$\langle$`Point`, `Int`$\rangle \to$ `Void`
`addY`	:	$\langle$`Point`, `Int`$\rangle \to$ `Void`
`c − new`	:	$\langle$`Int`, `Int`, `Int`$\rangle \to$ `cPoint`
`abscissa`	:	$\langle$`cPoint`$\rangle \to$ `Int`
`ordinate`	:	$\langle$`cPoint`$\rangle \to$ `Int`
`addX`	:	$\langle$`cPoint`, `Int`$\rangle \to$ `Void`
`addY`	:	$\langle$`cPoint`, `Int`$\rangle \to$ `Void`
`color`	:	$\langle$`cPoint`$\rangle \to$ `Int`
`set − color`	:	$\langle$`cPoint`, `Int`$\rangle \to$ `Void`
`mkRect`	:	$\langle$`Point`, `Point`$\rangle \to$ `Rect`
`lower − left`	:	$\langle$`Rect`$\rangle \to$ `Point`
`upper − right`	:	$\langle$`Rect`$\rangle \to$ `Point`
`set − lleft`	:	$\langle$`Rect`, `Int`$\rangle \to$ `Void`
`set − uright`	:	$\langle$`Rect`, `Int`$\rangle \to$ `Void`

Figure 4: The signature Σ_E.

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

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

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

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

and $DVAL^A$ is the set of denotable values,

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

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

The operations of an algebra are polymorphic, because they must be defined wherever *ResType* says they are defined. One can imagine that g^A looks at the actual types, and decides what to do, as in CLOS. This is easiest to understand if the sets of denotable values of each type are disjoint, but can be made to work even if they overlap [15]. We do not require carrier sets of subtypes be subsets of the carrier sets of supertypes in contrast to [4],[8].

The special abstract value $*$ of sort $TtoS[\texttt{Void}]$ is used for the result of an operation that would otherwise not have a value. An operation is modeled by a function from a sequence of denotable values and an initial store to a denotable value and a final store.

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

We now describe an example Σ_E-algebra, E. In this algebra, all operations take and return locations. As the locations have no interesting structure, we adopt the convention that for each type T:

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

The carrier sets for this algebra are defined in Figure 5. Our use of pairs in the carrier sets of rectangles and points in E is simply a convenience; we could also have used functions or stacks or some other mathematically defined abstract value set.

The operations of E are defined in Figure 6. We use $nextFree[T]$ to find the next free location of type T in a given store.

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

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

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

The notation $[l \mapsto v]\sigma$ is the function σ extended to bind l to v; that is $[l \mapsto v]\sigma \stackrel{\text{def}}{=} \lambda l_2 \,.\, (l_2 = l) \to v \,[\!]\, \sigma(l_2)$, where the notation $b \to e_1 [\!] e_2$ means if b is *true*, then e_1, else e_2. The notation $inLOCS(l)$ means the result of injecting a location l into the $LOCS$ summand of $DVAL$. We use this notation also in pattern matching as in Standard ML. For example, `add` takes two denotable values that are locations as arguments. Figure 6 gives a description of some of the operations of the algebra E. The omitted ones are defined similarly.

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

$$ENV[A] \quad = \quad VARS \stackrel{\text{fin}}{\to} DVAL^A \tag{4}$$

$$STATE[A] \quad = \quad (ENV[A] \times STORE[A])_\perp \tag{5}$$

In a state, variables of a supertype may denote objects of their subtypes. To capture this notion in the environment and store we place additional requirements on environments and stores.

Definition 4.3 *Let Σ be a signature, whose set of type symbols is $TYPES$, set of variable symbols is $VARS$ and whose presumed subtype relation is $\leq$. Let A be a Σ-algebra. Let X be a subset of $VARS$. A mapping $\eta{:}X \to DVAL^A$, is an* environment mapping *if and only if for every type $T \in TYPES$ and for every $x : T$ in X, $\eta(x)$ has a type S such that $S \leq T$.*

Similarly, a store mapping can map locations to abstract values of different sorts. To describe this subsorting, we define $\leq_{\text{sort}}$, as follows. For all types U and W, $TtoS(U) \leq_{\text{sort}} TtoS(W)$ if and only if $U \leq W$.

Definition 4.4 *Let Σ be a signature. Let $TYPES$ be its set of type symbols. Let A be a Σ-algebra. A mapping $\sigma : LOCS^A \to VALUES^A$, is a* store mapping *if and only if for every type $S \in TYPES$, and for every $l : S \in LOCS^A$, $\sigma(l)$ has a sort U such that $U \leq_{sort} TtoS(S)$.*

Nominal stores and environments are useful for reasoning about programs and are used in defining simulation relations.

Definition 4.5 *A state (η, σ) is* nominal *if both η and σ are nominal. An environment η is* nominal *if and only if for all $x : T \in dom(\eta)$, $\eta(x)$ has type T. Similarly a store σ is* nominal *if and only if for all $l : U \in dom(\sigma)$, $\sigma(l)$ has sort $TtoS(U)$.*

Operations

$$
\begin{array}{lcl}
\texttt{()}^E((), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Void}](*, \sigma) \\
\texttt{true}^E((), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Bool}](\mathit{true}, \sigma) \\
\texttt{or}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l_2)), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Bool}](\sigma(l_1) \vee \sigma(l_2), \sigma) \\
\texttt{0}^E((), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Int}](0, \sigma) \\
\texttt{add}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l_2)), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Int}](\sigma(l_1) + \sigma(l_2), \sigma) \\
\texttt{p} - \texttt{new}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l_2)), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Point}]((l_1, l_2), \sigma) \\
\texttt{abscissa}^E(\mathit{inLOCS}(l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = \sigma(l) \textbf{ in } (\mathit{inLOCS}(l_1), \sigma) \\
\texttt{addX}^E((\mathit{inLOCS}(l^{\texttt{Point}}), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = \sigma(l^{\texttt{Point}}) \textbf{ in} \\
 & & \textbf{let } (\mathit{inLOCS}(l_x), \sigma') = \\
 & & \ \textbf{in } \texttt{add}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) \\
 & & \mathit{alloc}[\texttt{Void}](*, [l^{\texttt{Point}} \mapsto (l_x, l_2)]\sigma') \\
\texttt{cp} - \texttt{new}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l_2), \mathit{inLOCS}(l_3)), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{cPoint}]((l_1, l_2, l_3), \sigma) \\
\texttt{abscissa}^E(\mathit{inLOCS}(l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2, l_3) = \sigma(l) \textbf{ in } (\mathit{inLOCS}(l_1), \sigma) \\
\texttt{addX}^E((\mathit{inLOCS}(l^{\texttt{cPoint}}), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2, l_3) = \sigma(l^{\texttt{cPoint}}) \textbf{ in} \\
 & & \textbf{let } (\mathit{inLOCS}(l_x), \sigma') = \\
 & & \ \textbf{in } \texttt{add}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) \\
 & & \mathit{alloc}[\texttt{Void}](*, [l^{\texttt{cPoint}} \mapsto (l_x, l_2, l_3)]\sigma') \\
\texttt{color}^E(\mathit{inLOCS}(l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2, l_3) = \sigma(l) \textbf{ in } (\mathit{inLOCS}(l_3), \sigma) \\
\texttt{set} - \texttt{color}^E((\mathit{inLOCS}(l), l^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2, l_3) = \sigma(l) \\
 & & \mathit{alloc}[\texttt{Void}](*, [l \mapsto (l_1, l_2, l^{\texttt{Int}})]\sigma) \\
\texttt{mkRect}^E((\mathit{inLOCS}(l_1), \mathit{inLOCS}(l_2)), \sigma) & \stackrel{\text{def}}{=} & \mathit{alloc}[\texttt{Rect}]((l_1, l_2), \sigma) \\
\texttt{botLeft}^E(\mathit{inLOCS}(l), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_1, l_2) = \sigma(l) \textbf{ in } (\mathit{inLOCS}(l_1), \sigma) \\
\texttt{horizMove}^E((\mathit{inLOCS}(l^{\texttt{Rect}}), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (l_{bl}, l_{tr}) = \sigma(l^{\texttt{Rect}}) \textbf{ in} \\
 & & \textbf{let } (\mathit{inLOCS}(l)', \sigma') = \\
 & & \ \textbf{in } \texttt{addX}^E((\mathit{inLOCS}(l_{bl}), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma) \\
 & & \textbf{let } (\mathit{inLOCS}(l''), \sigma'') = \\
 & & \ \textbf{in } \texttt{addX}^E((\mathit{inLOCS}(l_{tr}), \mathit{inLOCS}(l^{\texttt{Int}})), \sigma') \\
 & & \mathit{alloc}[\texttt{Void}](*, \sigma'')
\end{array}
$$

Figure 6: Some operations for the example Σ_E-algebra, E.

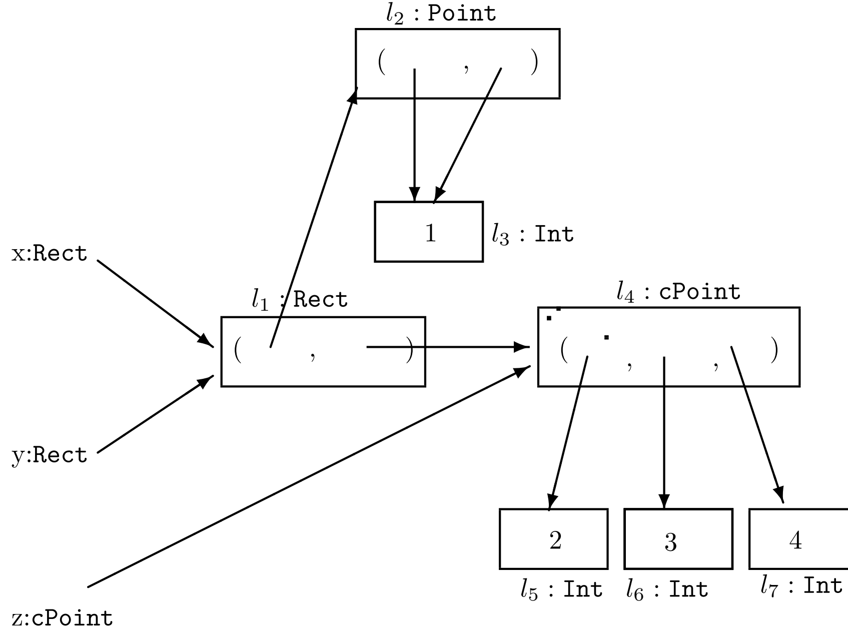

Figure 7: Picture of a non-nominal state over the algebra E

The significance of nominal and non-nominal states is that in a nominal state the location bound to a variable of type U and its abstract value, are of type U and sort $TtoS(U)$ respectively. In a non-nominal environment the variable's abstract value can belong to a sort $TtoS(W)$, where $W \leq U$. A non-nominal state is shown in Figure 7. Non-nominal states arise when one exploits subtyping.

Since locations are typed, we use $l : T$ to stand for a location $l \in LOCS_T^A$ when the algebra A is clear from context. For this we also write "$l : T$ is a location." Similarly we use $d : T$ to stand for a denotable value $d \in DVAL_T^A$. Similarly, we denote variables by $x : T$, and write "$x : T$ is a variable" to mean $x \in VARS_T$, when the set $VARS$ is clear from context.

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

$$
\begin{array}{l}
absVal : DVAL^A \times STORE[A] \rightarrow VALUES[A] \\
absVal(d, \sigma) \stackrel{\text{def}}{=} \textbf{cases } d \textbf{ of } isVALUES(v) \ \rightarrow \ v [\!] isLOCS(l) \ \rightarrow \ \sigma(l) \textbf{end}
\end{array}
$$

An example of a non-nominal state over the Σ_E-algebra E is pictured in Figure 7. In the formal model of this picture, the environment η_E would be defined as follows.

$$
\begin{array}{rcl}
\eta_E(\mathtt{x}) & = & inLOCS(l_1^{\mathtt{Rect}}) \\
\eta_E(\mathtt{y}) & = & inLOCS(l_1^{\mathtt{Rect}}) \\
\eta_E(\mathtt{z}) & = & inLOCS(l_4^{\mathtt{cPoint}})
\end{array}
$$

In Figure 7, we have arranged the picture so that each location has a unique index across all types.

For this example, the store mapping, σ_E would be defined as follows.

$$\begin{array}{lcl} \sigma_E(l_1^{\texttt{Rect}}) & = & (l_2^{\texttt{Point}}, l_4^{\texttt{cPoint}}) \\ \sigma_E(l_2^{\texttt{Point}}) & = & (l_3^{\texttt{Int}}, l_3^{\texttt{Int}}) \\ \sigma_E(l_6^{\texttt{cPoint}}) & = & (l_5^{\texttt{Int}}, l_6^{\texttt{Int}}, l_7^{\texttt{Int}}) \\ \sigma_E(l_3^{\texttt{Int}}) & = & 1 \\ \sigma_E(l_5^{\texttt{Int}}) & = & 2 \\ \sigma_E(l_6^{\texttt{Int}}) & = & 3 \\ \sigma_E(l_7^{\texttt{Int}}) & = & 4 \end{array}$$

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

In the example algebra E all variables point to locations. We could describe another algebra which has the same signature with variable of type integer and boolean pointing directly to abstract values instead of locations.

5 Simulation Relations

The intuitive idea of subtyping is that each object of subtype should behave like some object of each of its supertypes. The notion "behaves like" is expressed as a relation on the states that contain the objects. The reason for expressing the notion "behaves like" as relation on states is because we not only want the abstract values to be related but the states should also have the similar aliasing. What we mean by similar aliasing is that if state s_1 is related to s_2, and if variables x, y are aliased in s_1, then x and y must be aliased in s_2. Without this, operations that change the state would not preserve the relationships. The main property of such relationships is that they are preserved by operations of algebras so that after the operation is applied the resultant states behave similarly. This is captured in the "substitution" property.

Definition 5.1 (simulation relation) *Let C and A be Σ-algebras. A Σ-*simulation relation $\mathcal{R}$ from C to A *is a binary relation on states*

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

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

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

bindable: *for each type T for each type S such that $S \leq T$, for each variable $x : T$, and for each variable $y : S \in \mathrm{dom}(\eta_C)$,*

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

Substitution: *Let g be a program operation such that ResType(g, $\vec{S}$) = T. Let $\vec{x} : \vec{S} \in dom(\eta_C)$. Let $g^C(\eta_C(\vec{x}), \sigma_C) = (d_C, \sigma'_C)$ and let $g^A(\eta_A(\vec{x}), \sigma_A) = (d_A, \sigma'_A)$. Then for all $y : T$,*

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

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

$$(\eta_C, \sigma_C)\ \mathcal{R}\ (\eta_A, \sigma_A) \Rightarrow (\eta'_C, \sigma'_C)\ \mathcal{R}\ (\eta'_A, \sigma'_A), \tag{8}$$

***VIS*-identical:** *for all $T \in$ VIS, for all $x : T \in dom(\eta_C)$,*

$$(\eta_C, \sigma_C)\mathcal{R}(\eta_A, \sigma_A) \quad \Rightarrow \quad absVal(\eta_C(x), \sigma_C) = absVal(\eta_A(x), \sigma_A) \tag{9}$$

Coercion: *there exists a nominal state $(\eta'_A, \sigma'_A) \in STATE[A]$, such that*

$$(\eta_C, \sigma_C)\ \mathcal{R}\ (\eta'_A, \sigma'_A) \tag{10}$$

Bistrict: $(\eta_C, \sigma_C)\ \mathcal{R} \perp \Leftrightarrow (\eta_C, \sigma_C) = \perp$ *and* $\perp \mathcal{R}\ (\eta_A, \sigma_A) \Leftrightarrow (\eta_A, \sigma_A) = \perp$.

The substitution property ensures that homomorphic relationships between states are preserved by operations. It is expressed by binding a variable to the denotable value returned by the operation in each algebra, and then requiring that the resulting extended final states be related.

The "*VIS*-identical" property ensures that a simulation relation is the identity on objects of visible types. Distinct elements of visible type cannot be related. Further, at each visible type V, $\mathcal{R}_{TtoS(V)}$ is the identity on the carrier set of V.

The "Coercion property" is similar to the requirement that each object of a subtype simulate some object of the supertype for immutable types. The coercion property ensures that each non-nominal state simulates a nominal state.

The "bistrict" condition ensures that the relation is preserved in the case of non-termination.

Since our simulation relations are defined on states instead of abstract values they preserve aliasing between any two variables. That is, if two variables are aliased in one state then they must be aliased in a related state. This is because if two variables, say $x : T$ and $y : S$ were aliased in a state s_A and not aliased in s_N, one could observe a change in x by using operations on y in s_A while there would be no change in x in s_N when y is mutated. This would violate the substitution property. So such states cannot be related.

Though we allow aliasing between different types, the coercion property does not allow direct aliasing between variables of a proper subtype and variables of a supertype; that is, variables of a proper subtype and its supertype cannot denote the same object. If this kind of aliasing were present, there could not be a nominal environment with similar aliasing between those two variables, because in the nominal environment only variables of the same type can be directly aliased. Similarly, at the component level also we cannot have a direct aliasing between objects of proper subtypes and

objects of supertypes. However, all other kinds of direct aliasing are allowed, for example, direct aliasing between variables of the same type or direct aliasing between variables of unrelated types.

A trivial example of a simulation relation is the identity relation on states of E. The full version of this paper will show a non-trivial simulation relation between two different algebras.

6 Legal Subtype Relationships

In this section we define what it means for mutable, abstract data types to have a legal subtype relationship. We use simulation relations to define legal subtype relations. Legal subtype relations are defined among deterministic, first-order, mutable, abstract types as characterized by type specifications — not just the built-in types of a particular programming language. The formal definition of subtype relations is parameterized by the semantics of a specification. This allows the subtype definition to be independent of particular specification languages. Furthermore, since the semantics of a specification is a set of algebraic models, this definition handles incompletely specified types.

Definition 6.1 (Legal Subtyping) *Let SPEC be a set of Σ-algebras. The presumed subtype relationship $\leq$ on types is a* legal subtype relation for *SPEC if and only if for each $B \in SPEC$ there is some $A \in SPEC$ such that there is a Σ-simulation from B to A.*

The requirement that Σ-simulation property on states of the two algebras B,$A \in SPEC$ does test the properties of presumed subtype relationships. This is because the coercion property of simulation relations ensures that every non-nominal state in B simulates a nominal state in A.

The identity relation on types is trivially a subtype relation on types because simulation is reflexive.

Example 6.2 *Let SPEC be a set $\{E\}$, then the presumed subtype relationship between* `Point` *and* `cPoint` *is a legal subtype relationship for SPEC.*

7 A polymorphic object-oriented language, π

We define a polymorphic object-oriented language π to show some properties of simulation relationships and legal subtypes. Operations of π support message passing, and variables in π may denote objects of a subtype of the declared type, thus allowing subtype polymorphism.

7.1 The Language π

The syntax of π is given in Figure 8. The semantics of π is given in Figure 9. The main notations in this section are from [19]. Expressions, commands and programs in π are strict. The notation $(\mathcal{C}[\![C]\!](A)(s))|\mathrm{D}^*{}_2$ means that if $\mathcal{C}[\![C]\!](A)(s) = (\eta, \sigma)$, then $(\mathcal{C}[\![C]\!](A)(s))|\mathrm{D}^*{}_2 = \lambda x \in \mathrm{D}^*{}_2 . absVal(x, \sigma)$. That is, a program returns a function from its result values to their abstract values. This can be thought of as a labeled printing of the results. The output variables of a program,

Abstract Syntax:

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

P ::= `program` (D*$_1$) `returns` (D*$_2$) ; C
D* ::= | D D*
D ::= V : T
C ::=E | V := E | `begin` V : T := E; C `end` | C$_1$; C$_2$ | `if` E$_1$ `then` C$_1$ `else` C$_2$ `fi`
| `while` E `do` C `od` | `skip`
E ::= V | g(E*)
E* ::= | E* E

Figure 8: Syntax of π.

D*$_2$, must have a visible type; that is, $x : T \in$ D*$_2$ implies $T \in \mathit{VIS}$. We define the input variables and the output variables of a program as follows.

$$inVars[\![\texttt{program (D*}_1\texttt{) returns D*}_2\texttt{; C}]\!] = \text{D*}_1$$

$$outVars[\![\texttt{program (D*}_1\texttt{) returns D*}_2\texttt{; C}]\!] = \text{D*}_2$$

We use π to show properties of simulation relationships and subtype relationships in the next section.

8 Legal Subtyping means no surprises

In this section we state some important properties of simulation relations and using these we show that legal subtyping means "no surprises". By no surprises we mean that the results expected from a program according to the supertype's specification, should also happen when the program is executed in a non-nominal state.

The following lemma is similar to the fundamental theorem of logical relations, or the homomorphism theorem for standard algebras.

Lemma 8.1 *Let C and A be Σ-algebras. Let $\mathcal{R}$ be a Σ-simulation relations from C to A. Then the simulation relation $\mathcal{R}$ is preserved by expressions and commands in π.*

We have an important corollary for this theorem that simulation is preserved by programs.

$ANSWERS[\mathrm{A}] = VARS^A \rightarrow VAL^A$
$\mathcal{P}$: program $\rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow ANSWERS[A]$
$\mathcal{P}[\![\texttt{program (D}^*{}_1\texttt{) returns D}^*{}_2\texttt{; C}]\!](A)(s) = (\mathcal{C}[\![C]\!](A)(s))|\mathrm{D}^*{}_2$

$\mathcal{C}$: Command $\rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow STATE[A]$
$\mathcal{C}[\![\mathrm{E}]\!](A)(\eta, \sigma) = \mathbf{let}\ (l, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}\ \ (\eta, \sigma')$
$\mathcal{C}[\![\mathrm{V}\ \texttt{:=}\ \mathrm{E}]\!](A)(\eta, \sigma) = \mathbf{let}\ (l : T, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}$
$\quad ([\mathrm{V} \mapsto l]\eta, \sigma')$
$\mathcal{C}[\![\texttt{begin V : T := E; C end}]\!](A)(\eta, \sigma) = \mathbf{let}\ (l, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}$
$\quad \mathcal{C}[\![\mathrm{C}]\!](A)([\mathrm{V} \mapsto l]\eta, \sigma') - V$
$\mathcal{C}[\![\mathrm{C}_1\texttt{;}\ \mathrm{C}_2]\!](A)(s) = \mathcal{C}[\![\mathrm{C}_2]\!](A)(\mathcal{C}[\![\mathrm{C}_1]\!](A)(s))$
$\mathcal{C}[\![\texttt{if}\ \mathrm{E}_1\ \texttt{then}\ \mathrm{C}_1\ \texttt{else}\ \mathrm{C}_2\ \texttt{fi}]\!](A)(\eta, \sigma) =$
$\quad \mathbf{let}\ (l, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}$
$\quad\quad absVal(l, \sigma') \rightarrow (\mathcal{C}[\![\mathrm{C}_1]\!](A)(\eta, \sigma'))[\!](\mathcal{C}[\![\mathrm{C}_2]\!](A)(\eta, \sigma'))$
$\mathcal{C}[\![\texttt{While E do C od}]\!](A)(\eta, \sigma) =$
$\quad \mathrm{fix}(\lambda \mathrm{f}.(\mathbf{let}\ (l, \sigma') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma)\ \mathbf{in}$
$\quad\quad absVal(l, \sigma') \rightarrow f(A, (\mathcal{C}[\![\mathrm{C}]\!](A)(\eta, \sigma')))[\!](\eta, \sigma')))$
$\mathcal{C}[\![\texttt{skip}]\!](A)(\eta, \sigma) = (\eta, \sigma)$

$\mathcal{E}$: Expression $\rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow LOCS^A \times STATE[A]$
$\mathcal{E}[\![\mathrm{V}]\!](A)(\eta, \sigma) = (\eta[\![\mathrm{V}]\!], \sigma)$
$\mathcal{E}[\![\mathrm{g(E^*)}]\!](A)(\eta, \sigma) = \mathbf{let}\ (\hat{l}, \sigma') = \mathcal{E}*[\![\mathrm{E}^*]\!](A)(\eta, \sigma)\ \mathbf{in}\ \ \mathrm{g}^A(\mathrm{productize}(\hat{l}), \sigma')$

$\mathcal{E}*$: Expression-List $\rightarrow A : Alg(\Sigma) \rightarrow STATE[A] \rightarrow (List((LOCS^A)) \times STATE[A])$
$\mathcal{E}*[\![]\!](A)(\eta, \sigma) = (nil, \sigma)$
$\mathcal{E}*[\![\mathrm{E}^*\ \mathrm{E}]\!](A)(\eta, \sigma) =$
$\quad \mathbf{let}\ (\hat{l}, \sigma') = \mathcal{E}*[\![\mathrm{E}^*]\!](A)(\eta, \sigma)\ \mathbf{in}$
$\quad\quad \mathbf{let}\ (l_n, \sigma'') = \mathcal{E}[\![\mathrm{E}]\!](A)(\eta, \sigma')\ \mathbf{in}$
$\quad\quad\quad (\mathrm{addToEnd}(\hat{l}, l_n), \sigma'')$

$\forall 1 \leq i \leq \mathrm{length}(\hat{d}).\mathrm{productize}(\mathrm{addToEnd}(\hat{d}, d')) \downarrow i = (i = \mathrm{length}(\hat{d})) \rightarrow d'[\!](\mathrm{productize}(\hat{d}) \downarrow i)$

$FV[\![\mathrm{E}^*\ \mathrm{E}]\!] = FV[\![\mathrm{E}^*]\!] \cup FV[\![\mathrm{E}]\!]$
$FV[\![\mathrm{V}]\!] = \{\mathrm{V} : \mathrm{T}\}$ (assuming V is declared to have type T), and
$FV[\![\mathrm{g(E^*)}]\!] = FV[\![\mathrm{E}^*]\!]$
$FV[\![\texttt{begin V:T := E; C end}]\!] = (FV[\![\mathrm{E}]\!] \cup FV[\![\mathrm{C}]\!]) \setminus \{\mathrm{V{:}T}\}$
$FV[\![\texttt{program (D}^*{}_1\texttt{) returns (D}^*{}_2\texttt{); C}]\!] = \mathrm{toSet}(\mathrm{D}^*{}_1)$

Figure 9: Semantics of π

Corollary 8.2 *Let C and A be Σ-algebras. Let $s_C \in STATE[C]$ and $s_A \in STATE[A]$ be given. For each program P, if $s_C \mathcal{R} s_A$, then $\mathcal{P}[\![P]\!](C)(s_C) = \mathcal{P}[\![P]\!](A)(s_A)$.*

The result of a program P in a state s over an algebra A is defined as $\mathcal{P}[\![P]\!](A)(s)$. To make judgments whether subtype relationships give us the right kind of results or not we need to define a set of expected results. The *set of expected results of a program P for SPEC* is the union over all $A \in SPEC$ and nominal states $s \in STATE[A]$ of $\mathcal{P}[\![P]\!](A)(s)$.

Since the set of expected results of an observation is defined using nominal states, it is possible that in a non-nominal state one might observe surprising results.

Definition 8.3 (Surprising result) *Let SPEC be a set of Σ-algebras. Let $A \in SPEC$. A result, $\mathcal{P}[\![P]\!](A)(\eta, \sigma)$, is* surprising *with respect to SPEC if and only if $\mathcal{P}[\![P]\!](A)(\eta, \sigma)$ is not an element of the set of expected results of P for SPEC.*

As discussed in section 2, instances of legal subtype when passed as instances of supertypes cannot give surprising results.

Theorem 8.4 (Subtyping means no surprises) *Let SPEC be a set of Σ-algebras. Let P be a program in π. Let $C \in SPEC$. Suppose $\leq$ is a legal subtype relationship for SPEC. Then for all states $(\eta_C, \sigma_C) \in STATE[C]$, $\mathcal{P}[\![P]\!](C)(\eta_C, \sigma_C)$ is not surprising for SPEC.*

Proof: Let P be a program and let $(\eta_C, \sigma_C) \in STATE(C)$.

Since $\leq$ is a legal subtype relation for SPEC, there is some $A \in$ SPEC and some Σ-simulation relation $\mathcal{R}'$ from C to A. By the coercion properties we know that there is a nominal environment $(\eta_A, \sigma_A) \in STATE(A)$ such that $(\eta_C, \sigma_C) \mathcal{R}' (\eta_A, \sigma_A)$.

Let $r'_C = \mathcal{P}[\![P]\!](C)(\eta_C, \sigma_C)$ and $r'_A = \mathcal{P}[\![P]\!](A)(\eta_A, \sigma_A)$. By corollary 8.2, $r'_C = r'_A$. But r'_A is an expected result as it is the result of P taken over a nominal state. So r'_C is also an expected result. ∎

The conclusion of Theorem 8.4 may not hold if the subtype relationship is not legal. Suppose there is a presumed subtype relationship between `Point` and `wcPoint` as given in Example 1.2. From the observation of a similar program in π we get the values of i and j to be 10, while the actual results are 10 and 11 respectively.

9 Discussion

Since simulation relations take care of aliasing and mutation we can study subtype relations between a variety of abstract data types with mutable objects. Though subtype relations depend on the exact, formal specification of these types, nevertheless we discuss the insights that our formal framework gives us informally.

9.1 Tuple types

Our definition of legal subtype relations also applies to immutable types. With appropriate models, we can show that Cardelli's rules for immutable record subtyping are in accord with our definitions. That is, his rules in [3] for subtyping of immutable record types follow as a special case from our definition of legal subtype relationships.

If we consider mutable pairs and triples, then a mutable triple type whose elements are of the same types as corresponding elements of a mutable pair type is a legal subtype in our sense. This agrees with Cardelli's rules for mutable record subtyping in [3]. Cardelli justifies his rule on grounds of type theory, while ours follows from the model theory of the associated specifications.

We can consider three dimensional objects as subtypes of two dimensional objects. That is, if a three dimensional point can be a subtype of a two dimensional point. The z component of the three dimensional point is unobservable from the specification of a two dimensional point. Extending the same intuition to other objects, a cube can be a subtype of a square.

9.2 Mutable Vs Immutable types

A mutable type can be a subtype of an immutable types obtained by disallowing mutation on its objects. This corresponds to a similar discussion that a semi-mutable triple is a subtype of an immutable pair by Liskov and Wing [17]. They define subtyping from a proof-theoretic point of view.

An immutable type cannot mutate or observe the mutation in objects of its mutable subtype with respect to its operations. So, according to our definition, objects of its subtype when viewed from supertype's perspective are immutable. This works because we also prohibit direct aliasing between supertype and its mutable subtype. So every object of a mutable subtype acts like an instance of this immutable supertype. This notion also generalizes to partially mutable types as follows.

Consider a set of matrix types $A_{i,j}$, where the subscripts i and j denote a level of mutation. That is, $A_{0,0}$ indicate that no components of A are mutable and $A_{2,3}$ indicate that the first two rows and first two columns are mutable. The subtype hierarchy for this is shown in the Figure 10. In the figure an arrow from $A_{i+1,j+1}$ to $A_{i,j+1}$ indicate that $A_{i+1,j+1}$ is a subtype of $A_{i,j+1}$. This is because objects of type $A_{i+1,j+1}$ when viewed from the specification of $A_{i,j+1}$ act as if only i rows and $j+1$ columns are mutable. There is no relationship between $A_{i,j+1}$ and $A_{i+1,j}$ as objects of type $A_{i+1,j}$ can mutate observe the $i+1$ row of objects of type $A_{i,j+1}$ which is surprising for $A_{i+1,j+1}$ and vice versa.

9.3 Collection Hierarchy

Our definition of subtyping allows relationships between various collections. As an example consider a `Set`, `Stack`, `Queue` and `Bag`. `Stack`, `Queue` and `Bag` can be subtypes of `Set`. Standard subtype relationships in the collection hierarchy [6] are legal in our sense.

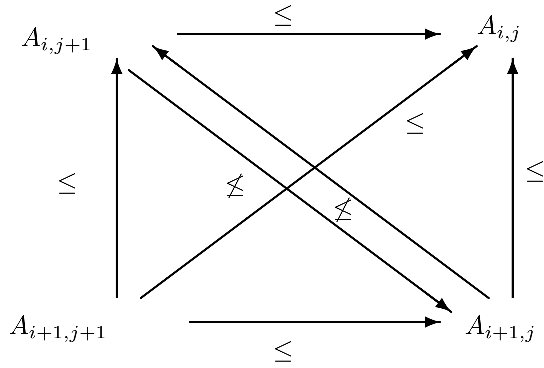

Figure 10: Subtyping in different kinds of mutable arrays

10 Conclusions

The important contribution of this work is a new definition of subtyping for arbitrary deterministic abstract data types in the presence of mutation and aliasing. Our definition of subtype relations are based on the intuitive notion that each instance of a subtype simulates some instance of each of its supertypes. Based on our formalization, we prove that legal subtype relationships preclude surprising behavior.

The significant feature of our work is that we allow for both mutation and aliasing in our definition of legal subtype relationships. Most previous studies only dealt with immutable data, and hence were not directly applicable to the majority of object-oriented programs. The work of Bertrand Meyer and Pierre America on subtyping finds model-theoretic support here.

A new aspect of subtyping which we have drawn attention to is the utility of limiting direct aliasing between a subtype and a proper supertype. While we allow aliasing among component objects (indirect aliasing), the prohibition of direct aliasing between a subtype and a supertype gives more legal subtype relationships then would otherwise be possible. For example, the type `mcPoint` can be a legal subtype of the type `Point` (see section 2). Although the extra mutator of `mcPoint` would cause surprising behavior if an `mcPoint` variable were directly aliased to `Point` variable, since this cannot arise, no surprising behavior is possible. Without this restriction on aliasing, one either needs a stronger definition of legal subtype relations [17] or a different way of reasoning about the correctness of programs.

Because of our restrictions on aliasing, one can reason about object-oriented programs in a modular fashion [14], [11]. That is, to prove the correctness of a program that uses subtyping and message passing one:

- proves the program meets its pre and post-conditions using the static (nominal) type information in the program text, reasoning essentially as usual,
- proves that the subtype relationships are legal, and

- proves that the aliasing restrictions hold.

Such reasoning is modular in the sense that when new subtypes are added to the program, the proof that the program meets its pre and post-conditions does not have to be repeated. That proof remains valid if the new subtypes are legal.

References

[1] Pierre America. A Behavioural Approach to Subtyping in Object-Oriented Programming Languages. *Technical Report*, Philips Research Laboratories, 443, Apr 1989.

[2] Kim B Bruce and Peter Wegner. An Algebraic Model of Subtype and Inheritance. 75-96, Advances in Database Programming Languages Addison-Wesley, 1990.

[3] Luca Cardelli. A semantics of Mutiple Inheritance. Information and Computation, 76, 138-164, Ferbruary/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[4] Luca Cardelli. Typeful Programming. 431-507, Formal Descirption of Programming Concepts Springer-Verlag, 1991.

[5] Jolly Chen. The larch/generic interface language. Technical report, Massachusetts Institute of Technology, May 1989. The author's Bachelor's thesis.

[6] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes SIGPLAN 27(10), Oct 1992.

[7] Krishna Kishore Dhara. Subtyping among mutable types in object-oriented programming languages. Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa 50011, May 1992.

[8] Joseph A. Gougen and Jose Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics Research Directions in Object-Oriented Programming, MIT press, Cambridge, Mass. 417-477, 1987.

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

[10] Gary T. Leavens. Verifying Object-Oriented Programs that use Subtypes. *Massachusetts Institute of Technology, Laboratory for Computer Science*, 439, Feb 1989.

[11] Gary T. Leavens. Modular Verification of Object-Oriented Programs with Subtypes. *Technical Report*, Iowa State University, 90-09, July 1990.

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

[13] 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. *Mathematical Foundations of Programming Semantics '91* Springer-Verlag, 598, 1992 144-167. Lecture Notes in Computer Science series.

[14] Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract) *OOPSLA ECOOP '90 Proceedings*, 212-223, SIGPLAN 25, Oct 90.

[15] Gary T. Leavens and William E. Weihl. Subtying, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs. Technical Report 92-28. Deparment of Computer Science, Iowa State University.

[16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[17] Barbara Liskov and Jeannette Marie Wing. A semantic notion of subtyping Unpublished draft, MIT Laboratory for Computer Science 1992.

[18] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Proc. IFIP Congress '83, Paris*, Sep 1983.

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

[20] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2/3):85–97, May/June 1985.

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