Weak Behavioral Subtyping for Types with Mutable Objects

Krishna Kishore Dhara and Gary T. Leavens

TR #94-21
November 1994

Keywords: object-oriented, behavioral subtype, abstract data type, mutation, aliasing, observable behavior, simulation relation, semantics, modular reasoning, supertype abstraction.

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 for publication.
© Krishna Kishore Dhara and Gary T. Leavens, 1994. Copies may be made for research and scholarly purposes, but not for direct commercial advantage. All rights reserved.

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

Weak Behavioral Subtyping for Types with Mutable Objects

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 14, 1994

Abstract

This paper studies the question of when one abstract data type (ADT) is a behavioral subtype of another, and proposes a model-theoretic notion of weak behavioral subtyping. Weak behavioral subtyping permits supertype abstraction to be a sound and modular reasoning principle in a language with mutation and limited forms of aliasing. The necessary restrictions on aliasing can be statically checked.

1 Introduction

Subtyping is a fundamental semantic concept in object-oriented (OO) languages. In this paper we study *behavioral subtyping*: when one ADT's objects act like those of another. Knowing the conditions on behavioral subtyping is important for guiding the design of ADTs. It is also critical for proving the soundness of logics for OO program verification.

Previous work on the model theory of behavioral subtyping has not allowed mutation and aliasing [2] [9] [8] [10]. But mutation and aliasing are important in practical OO programming, and many types occurring in practice have objects with mutable (time-varying) state. Although it is possible to imagine an OO language where aliasing is eliminated entirely, existing OO languages do permit aliasing. Unlike Liskov and Wing [11] [12], we do not allow arbitrary aliasing, but instead seek a middle ground that permits more useful behavioral subtype relationships.

The purpose of our study is ultimately to show how to reason in a modular fashion about OO programs. By *modular reasoning*, we mean reasoning such that conclusions about unchanged code remain valid when new behavioral subtypes are added to a program. One modular reasoning technique is *supertype abstraction*, in which one reasons about the effects of method sends using the properties of the static types of the subexpressions [9] [6]. The purpose and justification of a definition of behavioral subtyping is that it makes supertype abstraction sound.

Our technical approach showing that a definition of behavioral subtyping makes supertype abstraction sound is to capture the conclusions of reasoning via supertype abstraction in a set of expected behaviors. Behaviors that might occur because of subtyping are called *surprising* if they fall outside this set. Thus showing that a definition of "behavioral subtype" is *adequate* means showing that no surprising behavior is possible when subtyping relationships are required to satisfy the definition.

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

In this paper we define "weak behavioral subtyping." It is a weaker definition than either of Liskov and Wing's definitions [12] because it allows types with mutable objects (hereinafter *mutable types*) to be subtypes of immutable types. We sketch the semantics of a programming language with the necessary aliasing control, and show that weak behavioral subtyping is adequate in the sense described above. Finally we discuss related work and present some conclusions. Due to space limitations, we do not treat stronger definitions of behavioral subtyping.

2 A Motivating Example for the Problem

An example that motivates the problem is the following. Consider two types: `BoolSeq` and `StoreBool`. The type `BoolSeq` is a type of boolean sequences, which has only immutable objects. The messages one can send to a `BoolSeq` object are the following.

```
method fetch(s: BoolSeq, i: Int): Bool
method update(s: BoolSeq, i: Int, v: Bool): BoolSeq
```

The `update` method produces a new object, which has the same state as the argument `s`, except that in the `i`th position it contains `v`.

The type `StoreBool` has mutable objects. It has the following methods.

```
method fetch(s: StoreBool, i: Int): Bool
method store(s: StoreBool, i: Int, v: Bool): Void
```

There is no subtype relationship between `BoolSeq` and `StoreBool`. Suppose we wish to reason about part of a program in which we have the following identifiers available, where the type `BoolVar` is a type of boolean variable cells, which is like `Bool ref` in SML.

```
bseq: BoolSeq,  storb: StoreBool,  b: BoolVar
```

Now consider the following observation of a program state with such an environment. The observation consists of two commands and a constant declaration; the constant declaration gives the "output" of the observation. The messages `not` and `equal` have their standard meaning for `Bool` arguments. The messages `assign` and `value` mimic assignment and value access for program variables.

```
assign(b, fetch(bseq, 3));
store(storb, 3, not(value(b)));
const output:Bool = equal(value(b), fetch(bseq, 3))
```

What is the expected set of possible values for `output` in the above observation? The expected set depends on three points:

- Whether one's reasoning technique permits one to assume that identifiers of unrelated types (such as `BoolSeq` and `StoreBool`) cannot be directly aliased.
- Whether `bseq` and `storb` can be aliases for the same object.
- The notion of behavioral subtyping allowed.

These three points are not completely independent. Using Liskov and Wing's definitions of behavioral subtype [11] [12], `BoolSeq` and `StoreBool` cannot have a common subtype, because `BoolSeq` objects are immutable (and thus a common subtype would have to violate a history constraint, or would have a mutator corresponding to `store` that could not be explained). So using Liskov and Wing's definition prohibits `bseq` and `storb` from being directly aliased. If `bseq` and `storb` cannot be directly aliased, the set of expected results would be $\{true\}$.

If one's reasoning technique forces one to think about a case where `bseq` and `storb` might be directly aliased, then the set of expected results depends on the notion of behavioral subtyping used. If one had a weaker notion of behavioral subtyping than Liskov and Wing's, then it might be possible for `BoolSeq` and `StoreBool` to have a common subtype with mutable objects. Then `bseq` and `storb` could be directly aliased, and so presumably the set of expected results for the observation above would be $\{true, false\}$. We have not explored such reasoning techniques (which were suggested to us by Ian Maung). However, because objects of type `BoolSeq` are immutable, and the call of a mutator (`store`) is used in the program, it is difficult to imagine the specification of a (most general) common subtype of `BoolSeq` and `StoreBool`. Another problem we see is that, psychologically, programmers would tend to think that because `bseq` is immutable, the only possible result would be *true*, without considering aliasing. Thus such a reasoning technique might be error-prone if used informally. We leave the investigation of such a reasoning technique, and adequate notions of behavioral subtyping for it, as an open problem.

The remaining case is where one's reasoning technique permits one to assume that identifiers of unrelated types cannot be directly aliased. Clearly in this case, such an assumption has to be enforced. If it is, then set of expected results of the observation above is $\{true\}$. However, in this case there is still the possibility that `BoolSeq` and `StoreBool` have a common subtype. Allowing common subtypes, such as `BoolArray`, would have a great practical benefit. (That benefit, however, should be weighed against any restrictions on aliasing.)

Thus our problem is twofold: to define a notion of behavioral subtyping that is weaker than Liskov and Wing's, and to state restrictions on aliasing such that it is adequate for reasoning. Here, by "reasoning" we mean model-based reasoning with supertype abstraction and the assumption that identifiers of unrelated types cannot be directly aliased. We refer to our notion of subtyping as "weak behavioral subtyping."

3 The Language INST and its Semantics

Our model-theoretic approach to solving the problem was described above. To carry out this approach, and to give the reader a concrete picture of the kind of languages to which our results apply, we need to define an OO programming language. The language used in this paper, INST, is a multimethod language, with an abstract syntax given in Figure 1. The instance variable assignment command ("I_1.I_2 := E"), and the object creation ("`new` I(E*)") and the instance variable access ("I_1.I_2") expressions can only be used directly within methods; they cannot be written in the main procedure (M). This provides a simple form of information hiding.

Abstract syntax:

P ∈ Program	TD ∈ TypeDecl	T ∈ TypeName	V ∈ VarDecl	MD ∈ MethDecl
F ∈ Formal	B ∈ Body	M ∈ MainProc	D ∈ Decl	E ∈ Expression
C ∈ Command				

P ::= TD* MD* M
TD ::= `type` I `subtype of` T* `instance variables` V* `end`
T ::= I
V ::= I `:` T
MD ::= `method` I_0 `(` F* `)` `:` T_r `alias types` I* `plus {` T* `}` `is` B
F ::= I_1 `:` T `alias types` I_2
B ::= D C `return` E
M ::= `main observe` D_1 C_1 `by` C_2 D_2
D ::= `const` I `:` T `=` E | D_1 `;` D_2
E ::= N | `nothing` | `true` | `false` | I | I `(` E* `)` | `new` I E* | E `.` I_2
C ::= E | `if` E_1 `then` C_1 `else` C_2 `fi` | C_1 `;` C_2 | I_1 `.` I_2 `:=` E

Figure 1: Abstract Syntax of INST. The nonterminal “I” is an identifier, and “N” a number. “TD*” is a sequence of zero or more “TD”s (with separators in concrete examples).

3.1 Denotational Semantics

The denotational semantics of INST is given in two parts [7]: the type and method declarations are compiled into a signature and an algebra over that signature. The semantics of expressions, declarations, and commands are parameterized by an algebra. For purposes of this paper, in which we wish to define observations that may observe states over algebras, the main procedure (M) consists of two sequences of declarations and commands. The first of which defines a state, and the second a function from algebras to observation of states over algebras. To run the program, one passes the algebra and state to the observation. The reason for splitting the semantics of the main procedure in this way is to indicate in what part supertype abstraction is used. Supertype abstraction would be used to reason about the part of the main procedure following the keyword `by`, which thus defines an observation.

The semantics of a program is shown formally below. Most of the notation has not been discussed yet, but it seemed helpful to show the valuation function for programs before launching into the details. Nonstandard notations not explained in this paragraph will be explained further below. The type of $\mathcal{P}$ is a dependent type. The signature Σ^{INST} and the Σ^{INST}-algebra $\mathbf{A}^{\mathrm{INST}}$ give the signature and semantics of the visible types (see Figures 4 and 6 in [7]). The valuation function for type declaration sequences, $\mathcal{TD}*$, adds to the signature and algebra primitive operations for each type declared; these primitive operations are used by the semantics of expressions and commands for creating objects and for accessing their instance variables. Once $\mathcal{MD}*$ has processed all the method declarations, these primitive methods are suppressed. A signature without the primitive operation symbols is produced by *hideInternalMessages*. The notation $\mathbf{A}'|_{(\mathit{hideInternalMessages}\ \Sigma')}$ is the reduct of $\mathbf{A}'$ without these primitives.

$$\mathcal{P} : \text{Program} \to ((\Sigma'' : \mathit{SIGS}) \times (\mathbf{A}'' : \mathit{Alg}(\Sigma'')) \times (H : \mathit{TENV}(\Sigma'')) \times \mathit{STATE}_H[\mathbf{A}'']$$
$$\times ((\mathbf{B} : \mathit{Alg}(\Sigma'')) \to \mathit{OBSERVATION}_H[\mathbf{B}]))_\perp$$

$\mathcal{P}[\![\text{TD* MD* M}]\!] =$
$\quad$ **let** $(\Sigma, \mathbf{A}) = \mathcal{TD}*[\![\text{TD*}]\!]\ \Sigma^{\mathrm{INST}}\ \mathbf{A}^{\mathrm{INST}}$ **in**

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 $(H, s_1, f) = \mathcal{M}''_{\Sigma}[\![\text{M}]\!]\ \mathbf{A}''$ **in** $(\Sigma'', \mathbf{A}'', H, s_1, f)$

Due to lack of space we do not give the details of the semantics of type and method declarations. Instead, we define the signatures and algebras that they denote, and then turn to the semantics of expressions, declarations, commands, and the main procedure.

To define observations, we fix a set of the visible (or built-in) types, $\mathit{VIS} = \{\texttt{Int}, \texttt{Bool}\}$. The externally visible values of these types are: $\mathit{EXTERNALS}_{\texttt{Int}} \overset{\text{def}}{=} \{0, 1, -1, 2, -2, \cdots\}$, and $\mathit{EXTERNALS}_{\texttt{Bool}} \overset{\text{def}}{=} \{\mathit{true}, \mathit{false}\}$.

Signatures are roughly as in Reynolds's category sorted algebras [13], with the addition of information about aliasing that is used in our static restrictions on aliasing.

Definition 3.1 (*SIGS*, **signature)** *The set SIGS, consists of all* signatures*, Σ, which are tuples $(\mathit{TYPES}, \leq, \mathit{OPS}, \mathit{ResType}, \mathit{RetAliasRel})$ such that:*

- *TYPES is a set of type symbols such that $\mathit{VIS} \subseteq \mathit{TYPES}$ and $\texttt{Void} \in \mathit{TYPES}$.*
- *$\leq$ is a preorder on TYPES, such that if $S \leq T$ and $T \in \mathit{VIS}$, then $S = T$.*
- *OPS is a family of sets of operation symbols, indexed by natural numbers,*
- *ResType is a family of partial functions indexed by the natural numbers, such that for each natural number n, $\mathit{ResType}_n : \mathit{OPS}_n \times \mathit{TYPES}^n \to \mathit{TYPES}_\perp$, and ResType is monotone. That is, for all $g \in \mathit{OPS}$, and for all tuples of types $\vec{S} \leq \vec{T}$, if $\mathit{ResType}(g, \vec{T}) \neq \perp$ then $\mathit{ResType}(g, \vec{S}) \neq \perp$ and $\mathit{ResType}(g, \vec{S}) \leq \mathit{ResType}(g, \vec{T})$.*
- *RetAliasRel limits the types of identifiers that may be directly aliased to the result of a method (based on the types of identifiers aliased to the actuals). RetAliasRel is a family of partial functions indexed by the natural numbers, such that for each natural number n, $\mathit{RetAliasRel}_n : \mathit{OPS}_n \times \mathit{TYPES}^n \times \mathit{PowerSet}(\mathit{TYPES})^n \to \mathit{PowerSet}(\mathit{TYPES})_\perp$.*

Our models of abstract types with mutable objects are algebraic [14] [4] [7]. Objects are modeled by typed locations containing values, which may in turn contain locations. We define algebras and stores simultaneously, because the operations of an algebra take and return a store [7].

Definition 3.2 (*Alg*(Σ), **Σ-algebra,** *STORE*) *The set Alg(Σ), consists of all* Σ-algebras*, $\mathbf{A} = (\mathit{SORTS}^{\mathbf{A}}, \mathit{ObjectTypes}^{\mathbf{A}}, \mathit{LOCS}^{\mathbf{A}}, \mathit{VALS}^{\mathbf{A}}, \mathit{TtoS}^{\mathbf{A}}, \mathit{OPS}^{\mathbf{A}}, \mathit{externVal}^{\mathbf{A}})$, such that:*

- *$\mathit{SORTS}^{\mathbf{A}} \supseteq \mathit{TYPES}$ is a set of sort symbols,*
- *$\mathit{ObjectTypes}^{\mathbf{A}} \subseteq \mathit{TYPES}$ is a set of object type symbols,*
- *$\mathit{LOCS}^{\mathbf{A}}$ is a family of sets, indexed by $\mathit{ObjectTypes}^{\mathbf{A}}$, representing typed locations,*
- *$\mathit{VALS}^{\mathbf{A}}$ is a family of abstract values indexed by $\mathit{SORTS}^{\mathbf{A}}$, such that for each $T \in \mathit{ObjectTypes}^{\mathbf{A}}$, $\mathit{VALS}^{\mathbf{A}}_T = \mathit{LOCS}^{\mathbf{A}}_T$*
- *$\mathit{TtoS}^{\mathbf{A}} : \mathit{ObjectTypes}^{\mathbf{A}} \to \mathit{SORTS}^{\mathbf{A}}$ 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 $g^{\mathbf{A}} \in OPS_n^{\mathbf{A}}$ where for each $\vec{S} \in TYPES^n$ and $T \in TYPES$, if $ResType(g, \vec{S}) = T$ then $g^{\mathbf{A}}$ satisfies $g^{\mathbf{A}} : (VALS^{\mathbf{A}}_{\vec{S}} \times STORE[\mathbf{A}]) \to ((\cup_{U \leq T} VALS^{\mathbf{A}}_T) \times STORE[\mathbf{A}])_\perp$,*
- *$externVal^{\mathbf{A}}$ 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 $STORE[\mathbf{A}] \overset{\text{def}}{=} LOCS^{\mathbf{A}} \overset{\text{fin}}{\to} VALS^{\mathbf{A}}$ is such that if $\sigma : STORE[\mathbf{A}]$ and $l \in LOCS^{\mathbf{A}}_T \cap dom(\sigma)$, then $\sigma(l) \in \cup_{U \leq T} VALS^{\mathbf{A}}_{TtoS^{\mathbf{A}}[U]}$.

We write $l : T$ as an abbreviation for $l \in LOCS^{\mathbf{A}}_T$. A store $\sigma \in STORE[\mathbf{A}]$ is *nominal* if and only if for all locations $l : T \in dom(\sigma)$, $\sigma(l) \in VALS^{\mathbf{A}}_T$.

The set $TENV(\Sigma)$ of *type environments* over a signature Σ is defined by $TENV(\Sigma) = \text{Identifier} \overset{\text{fin}}{\to} TYPES$. Let H stand for a type environment below.

A state consists of an environment and a store. The set $ENV_H[\mathbf{A}]$ of *H-environments over* $\mathbf{A}$ is the set of all mappings, $\eta : \text{Identifier} \overset{\text{fin}}{\to} VALS^{\mathbf{A}}$, such that for every $T \in TYPES$, if $H(x) = T$ then $x \in dom(\eta)$ and $\eta(x) \in \cup_{S \leq T} VALS^{\mathbf{A}}_S$. A H-environment $\eta \in ENV_H[\mathbf{A}]$ is *nominal* if and only if for every type $T \in TYPES$, if $H(x) = T$ then $\eta(x) \in VALS^{\mathbf{A}}_T$. The set, $STATE_H[\mathbf{A}]$, of *H-states over* $\mathbf{A}$ is defined by $STATE_H[\mathbf{A}] \overset{\text{def}}{=} ENV_H[\mathbf{A}] \times STORE[\mathbf{A}]$. A H-state (η, σ) is *nominal* if and only if both η and σ are nominal. We write $ENV[\mathbf{A}]$ for $\cup_{H \in TENV(\Sigma)} ENV[\mathbf{A}]$, and $STATE[\mathbf{A}]$ for $ENV[\mathbf{A}] \times STORE[\mathbf{A}]$.

The main procedure (M) returns a type environment, a state, and a function from algebras to observations. This function is defined by the second half of the main procedure. An observation takes a state, such as the one produced by the first half of the main procedure, and "prints" the values of the identifiers in D$_2$. *H-observations* are defined as follows.

$$OBSERVATION_H[\mathbf{A}] \overset{\text{def}}{=} (\cup_{H' \supseteq H} STATE_{H'}[\mathbf{A}]) \to ANSWERS_\perp \quad (1)$$

$$ANSWERS \overset{\text{def}}{=} \text{Identifier} \overset{\text{fin}}{\to} EXTERNALS \quad (2)$$

The identifiers declared in D$_2$ must have visible type. This condition is checked by *typeEnvAndCheckVisible*, which produces a type environment if they are visible (and $\perp$ otherwise).

```
$\mathcal{M} : (\Sigma : SIGS) \to \text{MainProc} \to (\mathbf{A} : Alg(\Sigma))$
    $\to ((H : TENV(\Sigma)) \times STATE_H[\mathbf{A}] \times ((\mathbf{B} : Alg(\Sigma)) \to OBSERVATION_H[\mathbf{B}]))_\perp$
$\mathcal{M}_\Sigma[\![$ main observe $D_1\ C_1$ by $C_2\ D_2]\!]\ \mathbf{A} =$
    let $H = typeEnv[\![D_1]\!]$ in
    let $(\eta, \sigma) = \mathcal{D}_\Sigma[\![D_1]\!]\ \mathbf{A}\ (emptyEnviron, emptyStore)$ in
    let $\sigma' = \mathcal{C}_\Sigma[\![C_1]\!]\ \mathbf{A}\ (\eta, \sigma)$ in
    let $H' = typeEnvAndCheckVisible[\![D_2]\!]$ in
    let $f = (\lambda \mathbf{B} . \lambda(\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})$ .
            let $\sigma'_{\mathbf{B}} = \mathcal{C}_\Sigma[\![C_2]\!]\ B\ (\eta_{\mathbf{B}}, \sigma_{\mathbf{B}})$ in
            let $\eta''_{\mathbf{B}}, \sigma''_{\mathbf{B}} = \mathcal{D}_\Sigma[\![D_2]\!]\ \mathbf{B}\ (\eta_{\mathbf{B}}, \sigma'_{\mathbf{B}})$ in
            $\lambda[\![I]\!]$ . let $T = H'[\![I]\!]$ in $externVal^{\mathbf{B}}_T(\eta''_{\mathbf{B}}[\![I]\!], \sigma''_{\mathbf{B}}))$
    in $(H,\ (\eta, \sigma'),\ f)$
```

For a given signature, Σ, an expression has a meaning which depends on a Σ-algebra. We do not show the semantics for the expressions of the form "`new I(E*)`" or "I$_1$.I$_2$", because these cannot occur in the main procedure, and so play no role in defining observations.

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

$\mathcal{E}_\Sigma[\![\mathrm{N}]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathcal{N}_\Sigma[\![\mathrm{N}]\!]\ \mathbf{A}\ \sigma$
$\mathcal{E}_\Sigma[\![\texttt{nothing}]\!]\ \mathbf{A}\ (\eta,\sigma) = \texttt{nothing}^{\mathbf{A}}((),\sigma)$
$\mathcal{E}_\Sigma[\![\texttt{true}]\!]\ \mathbf{A}\ (\eta,\sigma) = \texttt{true}^{\mathbf{A}}((),\sigma)$
$\mathcal{E}_\Sigma[\![\texttt{false}]\!]\ \mathbf{A}\ (\eta,\sigma) = \texttt{false}^{\mathbf{A}}((),\sigma)$
$\mathcal{E}_\Sigma[\![\mathrm{I}]\!]\ \mathbf{A}\ (\eta,\sigma) = (\mathbf{let}\ v = \eta[\![\mathrm{I}]\!]\ \mathbf{in}\ (v,\sigma))$
$\mathcal{E}_\Sigma[\![\mathrm{I}(\hat{\mathrm{E}})]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathbf{let}\ (\hat{v},\sigma') = \mathcal{E}{*}_\Sigma[\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}\ \ \mathrm{I}^{\mathbf{A}}(\mathit{productize}\ \hat{v},\sigma')$
$\mathcal{E}{*} : (\Sigma : \mathit{SIGS}) \to \text{Expression-List} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to \mathit{STATE}[\mathbf{A}]$
$\qquad\qquad \to (\mathit{List}(\mathit{VALS}^{\mathbf{A}}) \times \mathit{STORE}[\mathbf{A}])_\perp$
$\mathcal{E}{*}_\Sigma[\![]\!]\ \mathbf{A}\ (\eta,\sigma) = (\mathit{nil},\sigma)$
$\mathcal{E}{*}_\Sigma[\![\hat{\mathrm{E}}\ \mathrm{E}_n]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathbf{let}\ (\hat{v},\sigma') = \mathcal{E}{*}_\Sigma[\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}$
$\qquad\qquad \mathbf{let}\ (v_n,\sigma_n) = \mathcal{E}_\Sigma[\![\mathrm{E}_n]\!]\ \mathbf{A}\ (\eta,\sigma')\ \mathbf{in}\ \ ((\mathit{addToEnd}\ \hat{v}\ v_n),\ \sigma_n)$

The semantics of commands is straightforward. Declarations bind identifiers to objects.

$\mathcal{C} : (\Sigma : \mathit{SIGS}) \to \text{Command} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to \mathit{STATE}[\mathbf{A}] \to \mathit{STORE}[\mathbf{A}]_\perp$
$\mathcal{C}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathbf{let}\ (v,\sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}\ \ \sigma'$
$\mathcal{C}_\Sigma[\![\mathrm{C}_1\texttt{;}\ \mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathbf{let}\ \sigma_1 = \mathcal{C}_\Sigma[\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}\ \ \mathcal{C}_\Sigma[\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma_1)$
$\mathcal{C}_\Sigma[\![\texttt{if}\ \mathrm{E}_1\ \texttt{then}\ \mathrm{C}_1\ \texttt{else}\ \mathrm{C}_2\ \texttt{fi}]\!]\ \mathbf{A}\ (\eta,\sigma) =$
$\qquad \mathbf{let}\ (v,\sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}_1]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}$
$\qquad \mathbf{if}\ \mathit{externVal}^{\mathbf{A}}_{\texttt{Bool}}(v,\sigma')\ \mathbf{then}\ (\mathcal{C}_\Sigma[\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta,\sigma'))\ \mathbf{else}\ (\mathcal{C}_\Sigma[\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta,\sigma'))$
$\mathcal{D} : (\Sigma : \mathit{SIGS}) \to \text{Decl} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to \mathit{STATE}[\mathbf{A}] \to \mathit{STATE}[\mathbf{A}]_\perp$
$\mathcal{D}_\Sigma[\![]\!]\ \mathbf{A}\ s = s$
$\mathcal{D}_\Sigma[\![\texttt{const}\ \mathrm{I}\texttt{:}\mathrm{T}\ \texttt{=}\ \mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma) = \mathbf{let}\ T' = \mathcal{T}_\Sigma[\![\mathrm{T}]\!]\ \mathbf{in}$
$\qquad\qquad \mathbf{let}\ (v,\sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta,\sigma)\ \mathbf{in}$
$\qquad\qquad \mathbf{if}\ v \notin \cup_{U \le T'} \mathit{VALS}^{\mathbf{A}}_{T'}\ \mathbf{then}\ \perp\ \mathbf{else}\ ([\mathrm{I} \mapsto v]\eta,\sigma')$
$\mathcal{D}_\Sigma[\![\mathrm{D}_1\ \texttt{;}\ \mathrm{D}_2]\!]\ \mathbf{A}\ s = \mathcal{D}_\Sigma[\![\mathrm{D}_2]\!]\ \mathbf{A}\ (\mathcal{D}_\Sigma[\![\mathrm{D}_1]\!]\ \mathbf{A}\ s)$

The semantics of bodies (of methods) will also be used to help define the restrictions on aliasing. The type of a body B is the type of the return expression.

$\mathcal{B} : (\Sigma : \mathit{SIGS}) \to \text{Body} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to \mathit{STATE}[\mathbf{A}] \to (\mathit{VALS}^{\mathbf{A}} \times \mathit{STORE}[\mathbf{A}])_\perp$
$\mathcal{B}_\Sigma[\![\mathrm{D}\ \mathrm{C}\ \texttt{return}\ \mathrm{E}]\!]\ \mathbf{A}\ s = \mathbf{let}\ (\eta_1,\sigma_1) = \mathcal{D}_\Sigma[\![\mathrm{D}]\!]\ \mathbf{A}\ s\ \mathbf{in}$
$\qquad\qquad \mathbf{let}\ \sigma_2 = \mathcal{C}_\Sigma[\![\mathrm{C}]\!]\ \mathbf{A}\ (\eta_1,\sigma_1)\ \mathbf{in}\ \ \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_1,\sigma_2)$

3.2 Enforcing Restrictions on Aliasing

For our notion of weak behavioral subtyping to be adequate, we need to prevent direct aliasing between related, but distinct, types. Since we also want to be able to reason modularly, we need to also prevent direct aliasing between identifiers of unrelated types, because two unrelated types might, at some later time, have a common subtype. Thus, in this section, we define restrictions on aliasing such that identifiers of different types cannot be directly aliased. We do this by an abstract interpretation of INST programs, which tracks the set of types that may be aliased to each expression result. This set of types is called an *alias type set.*

The definitions below rely on results of a sequence of declarations, commands, and an expression that could be executed in the main procedure. For this purpose we define the set "MBody" as the subset of Body that includes only declarations, commands, and expressions that can be written in the main procedure.

A location l is *reachable* in a H-state s over an Σ-algebra $\mathbf{A}$ if and only if there exists a type T, and a body $B \in \text{MBody}$ such that $\Sigma; H \vdash B : T$ and $(l, \sigma) = \mathcal{B}_\Sigma[\![B]\!]\ \mathbf{A}\ s$. The notation $\Sigma; H \vdash B : T$ means that B has type T (see Figure 2).

The alias type set of a location in a H-state over an algebra is defined by the following.

$$\begin{aligned} &\mathit{aliasTypeSet}_\Sigma(H, \mathbf{A}, l, s) \\ &\ \stackrel{\text{def}}{=} \{T \mid T \in \mathit{TYPES},\ B \in \text{MBody},\ \Sigma; H \vdash B : T,\ (l, \sigma) = \mathcal{B}_\Sigma[\![B]\!]\ \mathbf{A}\ s\}. \end{aligned} \tag{3}$$

If a location's alias type set contains at most one type, then it can only be aliased by identifiers of the same type.

$$\mathit{atMostOneType}(r) \stackrel{\text{def}}{=} (S \in r \land T \in r) \Rightarrow S = T \tag{4}$$

Alias legality means that every reachable location has this property.

Definition 3.3 (alias legality, $\mathit{stAliasOk}_\Sigma$**)** *Let* $\mathbf{A}$ *be a* Σ*-algebra and* $s \in \mathit{STATE}_H[\mathbf{A}]$*. Then* s *is* alias legal*, written* $\mathit{stAliasOk}_\Sigma(H, \mathbf{A}, s)$*, if and only if for all reachable locations* l *in* s*,* $\mathit{atMostOneType}(\mathit{aliasTypeSet}_\Sigma(H, \mathbf{A}, l, s))$*.*

Figure 2 gives the type and alias checking rules for expressions, declarations, commands that can appear in the main procedure, and for bodies containing such (i.e., $M \in \text{MBody}$). For expressions, the notation $\Sigma; H \vdash E : T :: r$ means E has static type T and r is an upper bound on the alias type set of the result of E. For declarations, $\Sigma; H \vdash D \Longrightarrow H'$ means H' is the type environment after elaborating D, and that the binding does not produce illegal aliasing. For bodies in MBody, we do not give the alias type set of the result, because it is not needed in this paper.

To see the practical implications of our technique for restricting aliasing, it is useful to consider how the property that identifiers of distinct types are not directly aliased would be established in the body of a method, after binding actuals to formals. One option would be to prohibit any direct aliasing among the actuals in a call. This is more restrictive than we need, because aliasing between formals of the same type is not a problem. Instead, we require that the programmer write enough methods so that any call with directly aliased actuals will be handled by a method implementation where the formals corresponding to those actuals have the same type. For example, consider a method `foo` with two arguments, S and T, and suppose that U is the greatest lower bound of S and T in the subtype ordering; then the program would also need a method named `foo` with both arguments of type U.

Because we do not work with methods in this paper, and because we work with algebras that may not result from INST programs, we need to impose an equivalent condition that calling an operation in an algebra cannot result in illegal aliases. To prevent illegal aliases in the result state, the H-state, s, that results from a call to $g^{\mathbf{A}}$ must satisfy $\mathit{stAliasOk}_\Sigma(H, \mathbf{A}, s)$. To prevent the result itself from being directly aliased with identifiers of different types, the actual alias type set of the result must be smaller than that declared.

Definition 3.4 (preserves alias legality) *Let* $\mathbf{A}$ *be a* Σ*-algebra. Let* H *be a type environment. Then* $\mathbf{A}$ preserves alias legality *if and only if for each* H*-state* (η, σ) *such that* $\mathit{stAliasOk}_\Sigma(H, \mathbf{A}, (\eta, \sigma))$*, for each operation* $g \in \mathit{OPS}$*, for each tuple of types* $\vec{S}$*, if* $\mathit{RetAliasRel}(g, \vec{S}, \vec{r}) = \hat{r}$*,* $\vec{v} \in \mathit{VALS}^{\mathbf{A}}_{\vec{S}}$*, and* $(l, \sigma') = g^{\mathbf{A}}(\vec{v}, \sigma)$*, then:*

$$\mathit{stAliasOk}_\Sigma(H, \mathbf{A}, (\eta, \sigma'))\ \land\ (l \in \mathit{LOCS}^{\mathbf{A}} \Rightarrow \mathit{aliasTypeSet}_\Sigma(H, \mathbf{A}, l, (\eta, \sigma')) \subseteq \hat{r}).$$

$$
\begin{array}{llll}
[\mathrm{Num}] & \Sigma; H \vdash \mathrm{N} : \mathtt{Int} :: \{\} & [\mathrm{nothing}] & \Sigma; H \vdash \mathtt{nothing} : \mathtt{Void} :: \{\} \\
[\mathrm{true}] & \Sigma; H \vdash \mathtt{true} : \mathtt{Bool} :: \{\} & [\mathrm{false}] & \Sigma; H \vdash \mathtt{false} : \mathtt{Bool} :: \{\} \\
[\mathrm{ident}] & \Sigma; H \vdash \mathrm{I} : H(\mathrm{I}) :: H(\mathrm{I}) & & \text{if } \mathrm{I} \in \mathit{dom}(H)
\end{array}
$$

$$
[\mathrm{call}] \quad \frac{\Sigma; H \vdash \vec{\mathrm{E}} : \vec{S} :: \vec{r}, \quad \Sigma.\mathit{ResType}(\mathrm{I}, \vec{S}) = T, \quad \Sigma.\mathit{RetAliasRel}(\mathrm{I}, \vec{S}, \vec{r}) = r'}{\Sigma; H \vdash \mathrm{I}(\vec{\mathrm{E}}) : T :: r'}
$$

$$
[\mathrm{decl}] \quad \frac{\Sigma; H \vdash \mathrm{E} : S :: r, \quad S \leq \mathrm{T}, \quad r \subseteq \{\mathrm{T}\}}{\Sigma; H \vdash \mathtt{const}\ \mathrm{I} :\mathrm{T} \ \mathtt{=}\ \mathrm{E} \Longrightarrow [\mathrm{I} \mapsto \mathrm{T}]H}
$$

$$
[\mathrm{decl\ list}] \quad \frac{\Sigma; H \vdash \mathrm{D}_1 \Longrightarrow H', \quad \Sigma; H' \vdash \mathrm{D}_2 \Longrightarrow H''}{\Sigma; H \vdash \mathrm{D}_1\,\mathtt{;}\ \mathrm{D}_2 \Longrightarrow H''}
$$

$$
[\mathrm{ExpCom}] \quad \frac{\Sigma; H \vdash \mathrm{E} : S :: r}{\Sigma; H \vdash \mathrm{E} \ \surd}
$$

$$
[\mathrm{Cond}] \quad \frac{\Sigma; H \vdash \mathrm{E} : \mathtt{Bool} :: r, \quad \Sigma; H \vdash \mathrm{C}_1 \ \surd, \quad \Sigma; H \vdash \mathrm{C}_2 \ \surd}{\Sigma; H \vdash \mathtt{if}\ \mathrm{E}\ \mathtt{then}\ \mathrm{C}_1\ \mathtt{else}\ \mathrm{C}_2\ \mathtt{fi} \ \surd}
$$

$$
[\mathrm{Seq}] \quad \frac{\Sigma; H \vdash \mathrm{C}_1 \ \surd, \quad \Sigma; H \vdash \mathrm{C}_2 \ \surd}{\Sigma; H \vdash \mathrm{C}_1 \ \mathtt{;}\ \mathrm{C}_2 \ \surd}
$$

$$
[\mathrm{MBody}] \quad \frac{\Sigma; H \vdash \mathrm{D} \Longrightarrow H', \quad \Sigma; H' \vdash \mathrm{C} \ \surd, \quad \Sigma; H' \vdash \mathrm{E} : T :: r}{\Sigma; H \vdash \mathrm{D}\ \mathrm{C}\ \mathtt{return}\ \mathrm{E} : T}
$$

$$
[\mathrm{Main}] \quad \frac{\begin{array}{c}\Sigma; \{\} \vdash \mathrm{D}_1 \Longrightarrow H', \quad \Sigma; H' \vdash \mathrm{C}_1 \ \surd, \\ \Sigma; H' \vdash \mathrm{C}_2 \ \surd, \quad \Sigma; H' \vdash \mathrm{D}_2 \Longrightarrow H'', \quad \mathit{checkVisible}(\mathrm{D}_2)\end{array}}{\Sigma \vdash \mathtt{main}\ \mathtt{observe}\ \mathrm{D}_1\ \mathrm{C}_1\ \mathtt{by}\ \mathrm{C}_2\ \mathrm{D}_2 \ \surd}
$$

Figure 2: Type and alias checking rules for the main procedure part of INST.

For algebras that preserve alias legality, the alias checking rules are sound.

Lemma 3.5 *Let M be a main procedure of INST. Let $\mathbf{A}$ be a Σ-algebra. If $\mathbf{A}$ preserves alias legality and $(H, s, f) = \mathcal{M}_\Sigma[\![M]\!]\mathbf{A}$, then $(\Sigma \vdash M \ \surd) \Rightarrow \mathit{stAliasOk}_\Sigma(H, \mathbf{A}, s)$.*

4 Weak Behavioral Subtyping

The intuitive idea of behavioral subtyping is that each object of a subtype should behave like some object of its supertypes. One might think to express "behaves like" it would be enough to simply relate abstract values. However, this would not take locations and hence aliasing into consideration. One cannot relate just locations either, because the abstract values stored in locations also determine behavior. Relating locations along with the store does not account for aliasing between identifiers in the environment. So one must relate whole states. This idea is captured by the definition of simulation relations below.

4.1 Simulation Relations

The following formulation of simulation relations uses techniques from [**?**].

Definition 4.1 (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 family of binary relations on states, $\langle \mathcal{R}_H : H \in \mathit{TENV}(\Sigma)\rangle$, such that $\mathcal{R}_H \subseteq \mathit{STATE}_H[\mathbf{C}]_\bot \times \mathit{STATE}_H[\mathbf{A}]_\bot$ and for each type environment H each $(\eta_\mathbf{C}, \sigma_\mathbf{C}) \in \mathit{STATE}_H[\mathbf{C}]$, and each $(\eta_\mathbf{A}, \sigma_\mathbf{A}) \in \mathit{STATE}_H[\mathbf{A}]$, the following properties hold:*

bindable: *for each identifier x, for each type T, and for each identifier y such that $H(y) = T$, $(\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})\, \mathcal{R}_H\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \Rightarrow ([x \mapsto (\eta_{\mathbf{C}}\, y)]\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})\, \mathcal{R}_{[x \mapsto T]H}\, ([x \mapsto (\eta_{\mathbf{A}}\, y)]\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$,*

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 $\vec{y}$ such that $H(\vec{y}) = \vec{S}$, and for each identifier x,*

$$\begin{aligned}
&(\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})\, \mathcal{R}_H\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \Rightarrow \\
&\qquad (\mathbf{let}\ (r_{\mathbf{C}}, \sigma'_{\mathbf{C}}) = g^{\mathbf{C}}((\eta_{\mathbf{C}}\, \vec{y}), \sigma_{\mathbf{C}})\ \mathbf{in}\ ([x \mapsto r_{\mathbf{C}}]\eta_{\mathbf{C}},\ \sigma'_{\mathbf{C}})) \\
&\qquad \mathcal{R}_{[x \mapsto T]H}\ (\mathbf{let}\ (r_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = g^{\mathbf{A}}((\eta_{\mathbf{A}}\, \vec{y}), \sigma_{\mathbf{A}})\ \mathbf{in}\ ([x \mapsto r_{\mathbf{A}}]\eta_{\mathbf{A}},\ \sigma'_{\mathbf{A}})),
\end{aligned} \tag{5}$$

coercion: *there is a nominal state $(\eta'_A, \sigma'_A) \in STATE_H[\mathbf{A}]$, such that $(\eta_C, \sigma_C)\mathcal{R}_H(\eta'_A, \sigma'_A)$,*

EXTERNALS-**identical:** *for each type $T \in VIS$, for each identifier x such that $H(x) = T$, if $(\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})\, \mathcal{R}_H\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$, then $externVal^{\mathbf{C}}((\eta_{\mathbf{C}}\, x), \sigma_{\mathbf{C}}) = externVal^{\mathbf{A}}((\eta_{\mathbf{A}}\, x), \sigma_{\mathbf{A}})$,*

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

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

The substitution property says that simulation relationships between states are preserved by operations. It is expressed by binding an identifier to the value returned by the operation in each algebra, and then requiring that the resulting extended states be related. The coercion property is similar to the requirement that each object of a subtype should simulate some object of its supertypes. It ensures that each state simulates a state that does not use subtyping. The *EXTERNALS*-identical property says that a simulation relation is identity on values of visible types. This is used to compare the outputs of observations.

Simulation relations preserve aliasing. That is, if two identifiers, x and y, are directly aliased in a state $s_{\mathbf{C}}$, and if $s_{\mathbf{C}}\, \mathcal{R}_H\, s_{\mathbf{A}}$, then x and y must be directly aliased in $s_{\mathbf{A}}$. If this were not the case, then one could observe changes in x by using operations on y in $s_{\mathbf{C}}$, while in $s_{\mathbf{A}}$ the same changes to y would not be observable through x. But this would violate the substitution property.

A careful reader might observe that the requirement that every state should be simulated by a nominal state in the "coercion" property eliminates certain kinds of direct aliasing. More precisely, it eliminates direct aliasing between identifiers of different types. The reason for this is the following. Suppose $S \neq T$, and consider a state in which and $x : T$ and $y : S$ were directly aliased. Then to satisfy the coercion property, such a state would have to be related to one where x and y both denoted objects of their types, and thus could not be directly aliased. This motivates the alias restrictions we impose on INST. These restrictions do allow a weaker form of behavioral subtyping than Liskov and Wing's.

4.2 Weak Behavioral Subtypes

The following definition of a weak behavioral subtype relation characterizes when a specification of several ADTs has a subtype relation ($\leq$) that is adequate for modular reasoning. Since we do not discuss the forms of type specifications, we use their denotations, which are sets of algebras that preserve alias legality.

Definition 4.2 (weak behavioral subtyping) *Let SPEC be a set of Σ-algebras such that each $\mathbf{A}$ in SPEC preserves alias legality. The presumed subtype relationship $\leq$ on types (of Σ) is a* weak behavioral subtype relation for *SPEC if and only if for each $\mathbf{B} \in SPEC$ there is some $\mathbf{A} \in SPEC$ such that there is a Σ-simulation from $\mathbf{B}$ to $\mathbf{A}$.*

This definition, for example, permits the types `BoolSeq` and `StoreBool` to have a common subtype. Hence it allows types with immutable objects to have subtypes that have mutable objects. One can even have a hierarchy of weak behavioral subtypes, with increasing degrees of mutability. As an example, a completely mutable array is a weak behavioral subtype of partially mutable array, which in turn is a weak behavioral subtype of an immutable array. The subtype relationships between the various collection types in Cook's hierarchy [5] are weak behavioral subtypes in our sense. For immutable tuple types, our definition matches Cardelli's rules [3].

Because this definition permits $\mathbf{B}$ and $\mathbf{A}$ to be different algebras, it works for incomplete specifications: those with observably different models. Such incomplete specifications are important in practice, so that a subtype can be more completely specified than its supertypes.

Not every presumed subtype relation is a weak behavioral subtype relation, because of the coercion and substitution properties of simulation relations.

4.3 Weak Behavioral Subtyping means No Surprises

We now show that the definition of weak behavioral subtyping is adequate for modular reasoning with supertype abstraction. We do this in a model-theoretic fashion, by first defining the set of expected results of an observation, or rather of a function from algebras to observations.

Definition 4.3 (expected results) *Let SPEC be a set of Σ-algebras that preserve alias legality. Let H be a type environment. Let f be a function from Σ-algebras to H-observations. Then the set of* expected results of f *for SPEC is the union over all $\mathbf{A} \in SPEC$ and all $s_{\mathbf{A}} \in STATE_H[\mathbf{A}]$, such that $s_{\mathbf{A}}$ is nominal, of $(f\ \mathbf{A}\ s_{\mathbf{A}})$.*

A result is *surprising* if it is not expected. Surprising results can occur if one uses a presumed subtype relation that does not satisfy the definition of weak behavioral subtyping, and observes a state that is not nominal.

Theorem 4.4 (no surprises) *Let SPEC be a set of Σ-algebras that preserve alias legality. Let H be a type environment. Let $f : (\mathbf{B} : Alg(\Sigma)) \to OBSERVATION_H[\mathbf{B}]$ be such that there is some $\mathbf{A} \in SPEC$ and some main procedure M in INST such that $(H, s, f) = \mathcal{M}_\Sigma[\![M]\!]\ \mathbf{A}$.*

Then for all $\mathbf{C} \in SPEC$, and for all $s_{\mathbf{C}} \in STATE_H[\mathbf{C}]$, if $\leq$ is weak behavioral subtype relationship for SPEC, then $(f\ \mathbf{C}\ s_{\mathbf{C}})$ is an expected result for SPEC.

Proof Sketch: Because $\leq$ is a weak behavioral subtype relation for *SPEC*, there is an $\mathbf{A}' \in SPEC$ and a Σ-simulation relation, $\mathcal{R}$, from $\mathbf{C}$ to $\mathbf{A}'$. Using structural induction, show that simulations are preserved by commands and declarations. Then in the semantics of the main procedure, the resulting states in the observation part ($(\eta''_{\mathbf{B}}, \sigma''_{\mathbf{B}})$ in the semantics of the main procedure) are related. So by the *EXTERNALS*-identical property, the resulting answer functions must give the same result for each identifier (namely for those in D_2 of the main procedure M). ∎

5 Discussion and Related Work

The main contribution of our work is a new definition of subtyping for arbitrary deterministic abstract data types in the presence of mutation and aliasing. This definition is weaker than Liskov and Wing's definitions [11] [12], because it allows types with immutable objects to have subtypes with mutable objects. This flexibility seems to be important in practice. The price to be paid, however, is that the language must restrict aliasing. We have given suitable aliasing restrictions, which disallow direct aliasing between identifiers of different types. We believe that such aliasing restrictions may actually be of some practical benefit, as they allow naive reasoning to be sound.

Most other model-theoretic approaches [2], [9] [8] [10] do not deal with mutation and aliasing. In contrast to our approach, America [1] and Liskov and Wing [11] [12] give proof-theoretic definitions of behavioral subtyping. We leave for future work a direct comparison between our definition and such proof-theoretic definitions, and formulating the model-theoretic equivalent of Liskov and Wing's definitions.

Acknowledgements

Thanks to Ian Maung discussions about modular reasoning and to Don Pigozzi, Barbara Liskov, Jeannette Wing, and the participants at the 1991 *Foundations of OO Languages* workshop for discussions about our work on this topic. Thanks to John Mitchell for his suggestion that we should present more of the language, which helped clarify this work.

References

[1] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., January 1989. Superseded by a later version in April 1989.

[2] 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.

[3] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, NY, 1991.

[4] 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).

[5] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. *ACM SIGPLAN Notices*, 27(10):1–15, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

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

[7] Gary T. Leavens and Krishna Kishore Dhara. Blended algebraic and denotational semantics for ADT languages. Technical Report 93-21b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 1994. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[8] 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.

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

[10] 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. Full version of a paper in *Acta Informatica*, volume 32, number 8, pages 705–778. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[11] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[12] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[13] 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, New York, NY, January 1980.

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