5-1997

Behavioral Subtyping in Object-Oriented Languages

Krishna Kishore Dhara
Iowa State University

Behavioral Subtyping in Object-Oriented Languages

Krishna Kishore Dhara

TR #97-09
May 1997

Keywords: object-oriented programming, behavioral subtype, abstract data type, mutation, aliasing, modular verification, supertype abstraction

1996 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented Programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, modules, packages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs, pre- and post-conditions, theory

A version of this technical report is published as the author's doctoral dissertation.

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

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my major professor, Gary Leavens, for his guidance and support throughout my graduate studies at ISU. He spent countless number of hours teaching me the importance of attention to detail, clarity of thought, and expressiveness in writing. He has been a great influence in my career thus far, and I hope it will continue in future also. To him I convey my deepest thanks.

I express my sincere thanks to the members of my dissertation committee. Albert Baker was a constant source of encouragement and he made me explain things in a more intuitive manner, which helped me while writing my dissertation. Don Pigozzi reviewed my work regularly in seminars and had several technical suggestions. Vasant Honavar and Johnathan Smith helped me explain my work to a broader audience.

Special thanks to Kelvin Nilsen for being a good critic, a close associate, and for providing insightful comments and suggestions on various topics. Thanks are also due to Kim Bruce for giving me an opportunity to present my work at the FOOL workshop. It was a good experience for me. Luca Cardelli's comments on my work were extremely beneficial. John Mitchell suggested presenting a language with alias restrictions. Others who helped with technical discussions are: Jeanette Wing, Ian Maung, and Rustan Leino. Thanks to the mentors, Mary Loomis, John McGregor, Eliot Moss, Satoshi Matsuoko, and MaryBeth Rosson, at the OOPSLA Doctoral Symposium for their valuable suggestions.

My deepest thanks to Rajesh Parekh for listening to my ideas, for his comments on my drafts, and for being such a great friend. Others I had the opportunity to know who I would like to thank are: Yoonsik Cheon, MaryBeth Gurski, Robyn Lutz, Mathew Markland, Akhilesh Tyagi, Tim Wahls, Jai Bhoothanath, Borislav Simov, Karthik Balakrishnan, Srinivas Aluru, Srini Namineni, Appaji Malla, Malleswar Kalla, Trish Stauble, Lynn Bremer, and several others at ISU who I do not have space to mention.

I would like to thank my parents, Sreenivasa and Prameela Dhara, my brother, Narendra Dhara, and my sister-in-law, Sudha Dhara, for their continued support and encouragement. I would like to specially mention Srinivasu Battula and Srinivas Mummidi with whom I shared my apartment for many years and felt it was my home, and Ram Challa with whom I shared many enjoyable discussions over several cups of coffee. Thanks to them and to many other friends who made my stay in Ames most enjoyable. I would also like to thank Raju Kakani for giving me an opportunity to be an active member of the Literacy Focus Group.

Many thanks to Arthur Oldehoeft and the Department of Computer Science, the National Science Foundation, Arne Hallam and the Economics Department, and the Graduate college for providing me financial support during my stay at ISU. I would also like to thank Bell Laboratories, Lucent Technologies for their support since November 1996.

Finally, I would like to thank all my teachers and God for providing me the resources to pursue (some of) the things I wanted to.

ABSTRACT

Modularity and code reuse are two important features of object-oriented programming. Modularity means that adding new components does not require reverification or respecification of existing components. A common form of reuse in object-oriented programs is to add new subtypes to existing types and to invoke already existing procedures with objects of these new types. In such cases, behavior of programs that contain these procedures also depend on the behavior of the new subtype objects. Reverifying the code that uses existing procedures whenever new types are added is not practical and is not modular. Thus, a notion of behavioral subtyping that allows sound modular reasoning is important for object-oriented programming.

In this dissertation, we study behavioral subtyping for arbitrary abstract data types in the prescence of mutation and aliasing. We propose two notions of behavioral subtyping. Strong behavioral subtypes have objects that act like supertype objects in all cases, whereas as weak behavioral subtypes have objects that only need to act like supertype objects when manipulated as supertype objects. Both these notions allow sound modular reasoning based on the static types of variables in programs. Weak behavioral subtyping allows conclusions about programs based on the effects of individual procedures but restricts certain forms of aliasing. Strong behavioral subtyping allows all forms of aliasing but permits conclusions based only on the history constraints of the types. History constraints are the reflexive and transitive properties preserved by objects of a type across different states of a program. We prove that both these behavioral subtype notions are sufficient for sound modular reasoning.

1. INTRODUCTION

Reuse and modularity are touted as important features in object-oriented (OO) programming. A common form of reuse is to add new types as subtypes to existing types. Subtype objects masquerade as supertype objects and increase the functionality of several already existing software components.[1] In such cases the correctness of the existing software components also depends on the newly created subtype objects. Reverifying the entire program, whenever new subtypes are added, is not practical and goes against the modularity principle of OO programming. The notion of subtyping plays an important role in modularity and correctness and hence, in the success of OO programming.

In [GHJV95], Gamma and his co-authors summarize a commonly used notion of subtyping based on only the interface or the structure of the types [Car91]. They define a type, S, a subtype of T if the interface of S contains the interface of T [GHJV95, page 13]. Though this notion is adequate to prevent any runtime type errors, it cannot guarantee correctness of OO programs.

A semantic notion of subtyping, based on the behavior of types, can be used to achieve modular and safe extension of OO software. Such a notion is termed *behavioral subtyping*. We say one type is a behavioral subtype of another if the subtype objects "behave like" supertype objects. This notion should allow modular reasoning. By *modular reasoning* we mean that conclusions about unchanged programs remain valid when new behavioral subtypes are added.

One technique for modular reasoning of OO programs is *supertype abstraction* [LW95]. This technique allows conclusions based on the static types of variables. To ensure that these conclusions do not change, only subtype objects that are based on behavior are allowed in place of supertype objects. The assumption here is that behavioral subtype objects do not produce any unexpected results when used in place of supertype objects. Hence for a modular reasoning technique that uses supertype abstraction any adequate notion of behavioral subtyping should guarantee that programs do not produce "surprising" results.

In [LW95], Leavens and Weihl give a model-theoretic definition of behavioral

[1]In Eiffel this is done by instances of conforming types and in `C++` by pointers to objects of publicly derived classes.

subtyping for immutable types,[2] which allows one to use supertype abstraction as a sound and modular reasoning principle. However, for modular reasoning about practical OO programs one needs to study behavioral subtype relations in the context of mutation.

Extending this notion of behavioral subtyping to mutable types is non-trivial. One complexity while dealing with mutable types is the notion of the state of a program. A subtype object that behaves like a supertype object in one state can behave differently in a different state. Another complexity is aliasing or interference. In the following sections, we show that aliasing is not an orthogonal issue to modular reasoning. The notion of behavioral subtyping for mutable types determines the kinds of aliasing allowed in a program and hence the kinds of modular reasoning techniques.

The main goal of this dissertation is to define behavioral subtyping between arbitrary mutable abstract data types (ADTs) in the presence of aliasing in OO languages. We define a new weaker notion of behavioral subtyping and formalize a notion of strong behavioral subtyping. Weak behavioral subtyping allows conclusions about programs based on the effects of invoking procedures or functions in programs that restrict certain forms of aliasing. Strong behavioral subtyping allows all forms of aliasing but can make only conclusions that are based on the history properties, which state reflexive and transitive properties of types. We also show soundness results, which guarantee that programs using subtype objects in place of supertype objects do not give unexpected results. Another contribution of this dissertation is the way that algebraic and denotational techniques are blended to model mutable ADTs.

1.1 Problem

The following examples illustrate reasoning problems in OO programs and the role of behavioral subtyping in solving them. Since the focus of our study is behavioral subtyping in the context of mutation and aliasing, we use ADTs with varying degrees of mutability in our examples.

1.1.1 Reasoning problem and behavioral subtyping

The type `BankAccount` used in Figure 1.1 models an account with a savings component, whose objects support the following methods.

```
  withdraw(b: BankAccount, m: MoneyObj): Void
```

[2]Types whose objects do not have any time varying state are referred as *immutable types*. Types whose objects have a time varying state are referred as *mutable types*.

```
function test_balance(b:BankAccount):Bool
  is
    const bal:  MoneyObj:= balance(b);
    const m:  MoneyObj:= mkMoneyObj(10)
  do
    withdraw(b, m)
  return
    equal(value(balance(b)), value(bal) - value(m));
```

Figure 1.1: The function `check_balance` returns `true` on `BankAccount` objects

```
balance(b: BankAccount): MoneyObj
deposit(b: BankAccount, m:MoneyObj): Void
```

The type `MoneyObj` models US currency as some number of pennies. The `value` method extracts the integer amount of pennies from money objects.

After designing and implementing the types `MoneyObj` and `BankAccount`, a programmer might want to implement a function on them. Figure 1.1 gives one such function, `check_balance`. The function returns `true` just when the new amount is exactly equal to the old amount minus the amount withdrawn. The method invocation `withdraw(b, m)` is dynamic, that is the code it runs depends on the runtime type of all its arguments, b and m.

Consider the following typical scenario in OO programming. After implementing and verifying `test_balance`, a programmer might add a new subtype `PlusAccount` that has both a savings and a checking components. OO programming languages then allow calls to `test_balance` with `PlusAccount` objects as arguments. Since message passing is dynamic, the methods `balance` and `withdraw` executed will be those of the type `PlusAccount`. But the reasoning done earlier that `test_balance` always returns `true` is based only on analysis of the `balance` and `withdraw` methods of `BankAccount`. So the question is, when `PlusAccount` is added, do we require a reverification of `test_balance`?

One approach to solve this problem is to use supertype abstraction as a tool for modular verification [LW95]. Using supertype abstraction and reasoning at the static types of variables, that is reasoning under the assumption that b is always a `BankAccount` object, the set of *expected results* of `test_balance` is {`true`}. If this set of expected results were to change with the addition of new types then we need to reverify when new subtypes of BankAccount are added. To prevent this reverification, the actual set of results when subtype objects are used in place of supertype objects should be guaranteed to be a subset of the expected set of results. That is, the set of results when PlusAccount objects are passed to test_balance should be {true}. If all the subtypes of BankAccount satisfy this requirement then one need not reverify test_balance whenever new subtypes, like PlusAccount, are added. This requirement that the set of results when subtype objects are passed should be a subset of the set of expected results is termed as "no surprises". The "no surprises" notion is a necessary condition for any notion of behavioral subtyping that is used for modular reasoning using supertype abstraction.

To see why "no surprises" is a necessary condition for modular reasoning, consider a new subtype, ChargeAccount, that is similar to BankAccount except for the behavior of the withdraw method. The ChargeAccount's withdraw method charges a transaction fee, which is deducted from its balance. If this new subtype object were passed in place of b to test_balance then, due to dynamic invocation, the result would be false. The conclusion that test_balance always returns true would not be valid. Hence we term this result as *surprising* because it is not in the expected set of results. To preserve the soundness of modular verification that is based on supertype abstraction, we disallow the subtype relationship between ChargeAccount and BankAccount.

So the overall question is: how can one decide behavioral subtype relations between arbitrary ADTs and guarantee "no surprises"?

1.1.2 Aliasing and behavioral subtyping

Another important point in the study of behavioral subtyping for mutable types is the effect of aliasing on the set of expected results, which in turn affects soundness. In this section we show how the assumptions on aliasing effect modular reasoning and behavioral subtyping.

The following discussion uses a new account type, FrozenAccount, whose objects are immutable. FrozenAccount objects support the following methods.

```
get_interest(f: FrozenAccount): Money
balance(f:FrozenAccount): Money
```

The types FrozenAccount and BankAccount are not related.

Figure 1.2 gives a test_deposit function that takes a FrozenAccount object f, a BankAccount object b, and returns a Bool. The test_deposit function withdraws 10 pennies from b and checks to see if that transaction had any affect on f.

What is the set of expected results of test_deposit? It depends on the following:

```
function test_deposit(f:FrozenAccount, b:BankAccount):Bool
  is
    const m :MoneyObj:= mkMoneyObj(10);
    const bal:  MoneyObj:= balance(f)
  do
    deposit(b, m);
  return
  return equal(value(balance(f)), value(bal));
end;
```

Figure 1.2: Function `test_deposit` to show that aliasing is not orthogonal to behavioral subtyping

- the assumptions about aliasing used by the reasoning technique.
- whether aliasing is permitted between f and b.
- the notion of behavioral subtyping.

These three points are not completely independent. The formals f and b can be aliased only if a common subtype is allowed. In such a case, if the reasoning technique uses supertype abstraction with the assumption that if f and b cannot be aliased then the set of expected results would be {`true`}. This is because f and b are not aliased and f is an immutable object. But if the reasoning technique assumes that f and b can be aliased and uses supertype abstraction, then the set of expected results would be {`true, false`}. But if no common behavioral subtype of `BankAccount` and `FrozenAccount` is permitted, then the set of expected results would be {`true`}.

If the set of expected result contains both `true` and `false` then the reasoning technique cannot conclude that `test_deposit` always returns `true`. But this is surprising because of our assumption that `FrozenAccount` is an immutable type and hence f cannot be changed. But in the above case, we fail to make such a conclusion. Is this because the reasoning technique is permitting all forms of aliasing or is it because the reasoning technique is permitting more behavioral subtypes?

To summarize, the two problems discussed in this section are: how to define notions of behavioral subtyping between arbitrary mutable ADTs that satisfy "no surprises" and what relation between aliasing and behavioral subtyping permits conclusions about OO programs such as objects of immutable types do not change.

1.2 Overview of the solutions

In this section, we present an overview of our solution and highlight the details needed in our formal presentation of the solution. We first look at the different options discussed in the above section.

1.2.1 Different notions of behavioral subtyping

If one permits all forms of aliasing and would like to preserve certain constraints, such as immutability of `FrozenAccount`, on types, then the behavioral subtype notion obtained is similar to that of Liskov and Wing (the history constraint version) [LW94, Figure 4]. In such a case, there cannot be a common behavioral subtype between `FrozenAccount` and `BankAccount`. This is because if such a common subtype existed then it should satisfy the immutability constraint of `FrozenAccount` and the mutability of `BankAccount`. So f and b in `test_deposit` can never be aliased and the set of expected results is {`true`}. This notion permits fewer behavioral subtypes and hence referred as *strong behavioral subtyping.*

Strong behavioral subtyping disallows subtypes that have extra methods which violate the supertype's constraints. More specifically, it disallows mutable subtypes of immutable types. However, since mutable subtypes act like immutable types when viewed through the methods of the immutable supertypes, allowing mutable subtypes to immutable types does not, at first, seem to violate the soundness of supertype abstraction. Further, such a notion permits interesting and useful subtype relationships. As an example, using such relationships, one can pass a mutable array to a function (eg., `average`) that expects an immutable array. Such a notion of behavioral subtyping would permit more behavioral subtype relationships and hence referred as *weak behavioral subtyping.*

Weak behavioral subtyping would permit a common behavioral subtype of the types `FrozenAccount` and `BankAccount`. Hence the set of expected results depends on the assumptions on aliasing by the reasoning technique. If the reasoning technique forces one to think about a case when `f` and `b` are aliased, then the expected results of `test_deposit` would be { `true`, `false`}. This result is counter-intuitive because it is hard to imagine the specification of a common subtype to an immutable type, `FrozenAccount` and a mutable type, `BankAccount`, which satisfies the expected result above. Another problem is that it will be surprising to a programmer using `FrozenAccount` to expect `false` as a possible result because `FrozenAccount` is an immutable type. Hence we do not investigate reasoning techniques that allows all forms of aliasing and permits behavioral subtyping between mutable types and immutable types.

The remaining case is to permit a mutable subtypes to immutable types and allow the reasoning technique to restrict aliasing between unrelated types. That is, though a common behavioral subtype is allowed between `FrozenAccount` and `BankAccount`, but f and b cannot be aliased. Then the set of expected results for `test_deposit` is {`true`}. The tradeoff for weak behavioral subtyping would be the practical benefits of allowing mutable subtypes to immutable supertypes versus restrictions on aliasing.

1.2.2 Semantic conditions

For both the weak and the strong behavioral subtyping, the intuitive idea is that subtype objects should in a sense behave like supertype objects. The main objective in defining these notions is to capture this "behaves like" property such that "no surprises" is guaranteed.

1.2.2.1 Models of mutable types Our approach is model-theoretic. We capture the "behaves like" relation as a set of semantic properties on models of types. For this we need models of mutable types, which we refer as *mutation algebras*. Mutation algebras, as opposed to regular algebras [Wir90, GM87], contain stores as values. Using these stores, we model mutation as in denotational semantics. Further these algebras allow us to study observations on types independent of the language in which they are implemented. Models for mutable types are discussed in 2.

1.2.2.2 Simulation relations The semantics of a set of ADTs is given by a set of mutation algebras. To express the "behaves like" relation one might think that it would be enough to simply relate abstract values of the corresponding algebras. But these abstract values depend on the store, which map locations to values. Hence just relating environments that map identifiers to values does not take into account the mutation of the store. So one must relate environments, that map identifiers to abstract values in the context of stores. This idea is captured in *simulation relations*. Simulation relations capture the effects of operations on environments. They also ensure that two related environments are visibly equivalent, that is, simulation relations capture a notion of observable equivalence on the visible types such as `Integer` and `Bool`.

Since our main objective is to see the observable equivalence of environments that contain subtype objects in place of supertype objects, we define a "coercion" property. Coercion property ensures that, if certain aliasing restrictions are satisfied, every environment that contains subtype objects in place of supertype objects is related to an environment that contains only the supertype objects. We call such an environment, where every identifier and location denotes a value of its static type, *nominal environment*. Nominal environments do not contain any subtyping.

1.2.2.3 Weak behavioral subtyping We say that the declared subtype relationships are legal weak behavioral subtypes if for every implementation of the types (modeled by an algebra), **A**, there exists an implementation **B**, such that there is a simulation relation between **A** and **B**. From the coercion property, if the aliasing requirements are satisfied, every environment is related to a nominal environment that is observably equivalent. From this we can conclude that subtype objects "behave like" supertype objects. We use the notation $S \leq_{\mathrm{W}} T$ to denote that S is a weak behavioral subtype of T.

1.2.2.4 Strong behavioral subtyping Strong behavioral subtyping is defined with respect to a history constraint. History constraints [LW94] are like invariants but are defined across different stores. Recall that the strong behavioral subtype notion allows all forms of aliasing and does not allow subtypes that violate supertype's history constraints. To capture this, we define a model-theoretic equivalent of the history constraints and define a "constraint" property to ensure that all the operations on the subtype objects satisfy the history constraint. A declared subtype relationship is a legal strong behavioral subtype relation with respect to a history constraint if for every implementation of the types, there exists an implementation such that there is a simulation relation between them that satisfies the "constraint" property. This definition captures a behavioral subtype notion that prevents the extra methods in the subtype from violating the supertype's history constraint. We use the notation $S \leq_{\mathrm{S}} T$ to denote that S is a strong behavioral subtype of T.

1.2.2.5 Example As an example of a weak behavioral subtype relation, consider `PlusAccount` $\leq_{\mathrm{W}}$ `FrozenAccount`. We can define a simulation relation with a coercion property that maps an environment with an identifier `x:FrozenAccount` denoting a `PlusAccount` object to an environment with `x` denoting a `FrozenAccount` object, whose abstract value is the sum of the checking and savings component of the `PlusAccount`. If the `balance` method of `PlusAccount` returns the sum of its savings and checking components, then it satisfies the substitution property, because the effect of `balance` on a `PlusAccount` object (say with \$100 savings and \$100 checking) is the same as invoking `balance` on a `FrozenAccount` object (with \$200 as its savings component). Similarly other methods also satisfy the substitution property and the other properties. (Refer to Chapter 3 for a proof.) Hence, `PlusAccount` $\leq_{\mathrm{W}}$ `FrozenAccount`.

Because of the constraint property for strong behavioral subtyping `PlusAccount` is not a strong behavioral subtype of `FrozenAccount` with respect to a constraint that `FrozenAccount` objects are immutable. Note that this constraint does not affect the behavioral subtype relation between `PlusAccount` and `BankAccount`. Hence, we can show that `PlusAccount` $\leq_S$ `BankAccount` with respect to a constraint that `FrozenAccount` objects are immutable. Every $\leq_S$ is a $\leq_W$, but the converse is not true.

1.2.3 "No surprises"

To show the "no surprises" result and to give a concrete idea of the kind of languages to which our results apply, we define a multi-method OO programming language, $OBS^{\leq}$. To match the model theory discussed above, $OBS^{\leq}$ is split into two parts. The first part consists of type and method implementations and the second part consists of programs that use those types. We refer to the later part as the *main programs of* $\mathrm{OBS}^{\leq}$. The denotations of the main programs of $OBS^{\leq}$ are defined using algebras that represent the denotations of type and method implementations. Using the denotations of main programs, we show that expressions and statements in $OBS^{\leq}$ preserve simulations and prove that the results obtained when programs use subtype objects are a subset of the results when supertype objects are used.

Alias rules, which are similar to type checking rules, are provided for $OBS^{\leq}$ to restrict certain kinds of aliasing for weak behavioral subtypes. We define a semantic notion of *stAliasOk* that matches with the aliasing restrictions, which are adequate for reasoning using nominal environments, and show the soundness of the alias checking rules with respect to *stAliasOk*.

For weak behavioral subtyping, the expected results of a program are the set of results of all the invocations in the nominal environments, provided the aliasing restrictions are met, in all the implementations (algebras) of the types. Weak behavioral subtyping guarantees that for every state with subtyping in an implementation there is a corresponding state without subtyping in some implementation that is related, such that the states are observably equivalent. Since programs in $OBS^{\leq}$ preserve simulation relations, we can conclude that the resulting states, that is states with and without subtyping, are related. Hence, the set of actual results will be a subset of the expected results.

Similarly we show the "no surprises" for strong behavioral subtyping by taking the expected results to be the set of all invocations in all the states that along with the resulting states satisfy the constraint property. We show that for strong behavioral subtyping, expressions, declarations, and commands in the main programs of $OBS^{\leq}$ preserve the constraint property. Hence the actual results of a main program of $OBS^{\leq}$ is an expected result with respect to strong behavioral subtyping because the resulting state satisfies the constraint property. Hence, the actual set of results obtained will be a subset of the expected results.

1.3 Related work

Our work on the model-theory of behavioral subtyping is an extension of Leavens's work in [Lea89]. The simulation relations defined in [Lea89] do not have any provision for mutation. Similarly other model-theoretic approaches [BW90, LW90, LP92, LW95] do not deal with mutation and aliasing. So none of these approaches study the relation between aliasing and behavioral subtyping. Cusack [Cus91] has a notion of specialization that is similar to behavioral subtyping. Though her class schemas allow extra methods in the subtype and seem to deal with mutation, she does not study the relation between mutation, aliasing, and subtyping. Further, she requires the types of the arguments for the common methods in the subtype and supertype to be the same.

In contrast to the above model-theoretic approaches, America [Ame87, Ame91], and Liskov and Wing [LW93b, LW94] give proof-theoretic definitions of behavioral subtyping. America does not deal with extra mutators in subtypes. Liskov and Wing allow extra mutators provided that the extra mutators can be explained in terms of the supertype methods or if they do not violate any history constraints of the supertype. This rules out the possibility of mutable subtypes of immutable types. In [DL96], we weaken Liskov and Wing's constraint based notion to allow more behavioral subtypes that satisfy the supertype's history constraints. We believe that the strong behavioral subtyping defined in this dissertation is a model-theoretic equivalent of the notion defined in [DL96]. We leave the formal proof as a future work.

1.4 Outline of the dissertation

We present our models of mutable types in Chapter 2. In chapter 3 we define weak and strong behavioral subtype relationships with examples and comparisons to related work. To show that our notions of behavioral subtyping satisfies the established criteria of "no surprises", we define $OBS^{\leq}$, its semantics, alias checking rules for weak behavioral subtyping, and show the soundness of alias checking rules and soundness of supertype abstraction in Chapter 4. In Chapter 5, we define the sets of expected results and prove the corresponding "no surprises" results for weak and strong behavioral subtyping. Chapter 6 discusses our work in the context of reasoning about OO programs and Chapter 7 offers summary and conclusions.

2. ALGEBRAIC MODELS OF MUTABLE TYPES

In this chapter, we develop algebraic models of mutable types. We refer to these algebras as *mutation algebras*. Mutation algebras are extensions of standard algebraic models of immutable types [Wir90, GM87].

There are two applications of mutation algebras in our study of behavioral subtyping for mutable types. The first is to provide an algebraic model of types that is independent of any programming language. This allows us to study relations between different types or different implementations of types. One such relation, given in the next chapter, is a simulation relation that is used to define behavioral subtype relations.

Another application of mutation algebras is in the semantics of programming languages. We use mutation algebras to give semantics of our language, $OBS^{\leq}$, which is used to show the "no surprises" result for weak and strong behavioral subtyping. A "split" semantic technique is used for the semantics of $OBS^{\leq}$. The split semantics is a blend of algebraic and denotational semantic techniques. Programs in $OBS^{\leq}$ consist of two parts. The first part consists of type declarations and methods over the types, and the second part is a main program that uses the declared types and methods. The meaning of the first part is in a sense "compiled" into a mutation algebra and this algebra is used to describe the meaning of the main program. We present the semantics of main programs of $OBS^{\leq}$ in Chapter 4.

The next section gives a description of the visible types, which are used for observations and can be considered as the basic types in all mutation algebras. Section 2.2 presents the standard signatures and section 2.3 introduces mutation signatures. Mutation algebras are defined in Section 2.4. Section 2.5 formally defines notions of aliasing that are required in our study of behavioral subtyping and we discuss related work in Section 2.6.

Readers familiar with algebraic techniques can skim sections 2.1 and 2.2 and can go directly to section 2.3. Section 2.7 summarizes this chapter and presents Table 2.1 that tabulates the notation introduced in this chapter.

$$
\begin{array}{lcl}
\texttt{true} & : & (\texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{false} & : & (\texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{and} & : & (\texttt{Bool}, \texttt{Bool}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{or} & : & (\texttt{Bool}, \texttt{Bool}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{not} & : & (\texttt{Bool}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{0} & : & (\texttt{Store}) \rightarrow (\texttt{Integer}, \texttt{Store}) \\
\texttt{1} & : & (\texttt{Store}) \rightarrow (\texttt{Integer}, \texttt{Store}) \\
\texttt{add} & : & (\texttt{Integer}, \texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Integer}, \texttt{Store}) \\
\texttt{mult} & : & (\texttt{Integer}, \texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Integer}, \texttt{Store}) \\
\texttt{negate} & : & (\texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Integer}, \texttt{Store}) \\
\texttt{equal} & : & (\texttt{Integer}, \texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{less} & : & (\texttt{Integer}, \texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store}) \\
\texttt{leq} & : & (\texttt{Integer}, \texttt{Integer}, \texttt{Store}) \rightarrow (\texttt{Bool}, \texttt{Store})
\end{array}
$$

Figure 2.1: Operations for visible types, referred as *VISOPS*.

2.1 Visible types

To facilitate the study of visible behavior, we distinguish a subset of types as *visible* types; these are the types of values that can be "output" by a program [Sch91] [Nip86]. These are defined as follows.

$$VIS \stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Integer}\} \tag{2.1}$$

These types are used to define observable behavior of states. Hence, one needs to fix the set of operations on the visible types and the sets of externally visible values of each of these types. Figure 2.1 gives the names and signatures of operations on the visible types.

The carrier sets of externally visible values for visible types are defined as follows.

$$EXTERNALS_{\texttt{Bool}} = \{true, false\} \tag{2.2}$$

$$EXTERNALS_{\texttt{Integer}} = \{0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\} \tag{2.3}$$

Interpretations of operations on the *VIS* types are given in Figure 2.2.

2.2 Signatures

Signatures describe the interface of a set of types. They contain type names, a presumed subtype relation between type names, and operation symbols. (Method

$$
\begin{array}{lcl}
\mathtt{true}^{\mathbf{A}}(\sigma) & \stackrel{\text{def}}{=} & (\mathit{true}, \sigma) \\
\mathtt{false}^{\mathbf{A}}(\sigma) & \stackrel{\text{def}}{=} & (\mathit{false}, \sigma) \\
\mathtt{and}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 \wedge v_2, \sigma) \\
\mathtt{or}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 \vee v_2, \sigma) \\
\mathtt{not}^{\mathbf{A}}(v, \sigma) & \stackrel{\text{def}}{=} & (\neg(v), \sigma) \\
\mathtt{0}^{\mathbf{A}}(\sigma) & \stackrel{\text{def}}{=} & (0, \sigma) \\
\mathtt{1}^{\mathbf{A}}(\sigma) & \stackrel{\text{def}}{=} & (1, \sigma) \\
\mathtt{add}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 + v_2, \sigma) \\
\mathtt{mult}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 \times v_2, \sigma) \\
\mathtt{negate}^{\mathbf{A}}(v, \sigma) & \stackrel{\text{def}}{=} & (\Leftrightarrow v_1, \sigma) \\
\mathtt{equal}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 = v_2, \sigma) \\
\mathtt{less}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 < v_2, \sigma) \\
\mathtt{leq}^{\mathbf{A}}(v_1, v_2, \sigma) & \stackrel{\text{def}}{=} & (v_1 \leq v_2, \sigma)
\end{array}
$$

Figure 2.2: Operation interpretations for *VIS* types for all algebras, **A** and all stores, σ.

names are historically called operation symbols in this context.) By *presumed subtype relations*, we mean subtype relations that are declared by programmers. Overloading of operation symbols based on argument types is permitted. Overloading will also be useful in applying these algebras to object-oriented languages with message passing (which supports a kind of dynamic overloading [WB89]). Signatures also contain a function, *ResType*, which keeps track of the upper bound of the return type of operations. That is, operations are allowed to return subtype objects of their return type. These signatures are thus a simplified form of the signatures used in category-sorted algebras [Rey80] [Rey85].

Definition 2.2.1 (Signature) *A* signature*, Σ, is a tuple,*

$$(\mathit{TYPES}, \mathit{OPS}, \leq, \mathit{ResType}),$$

where

- *TYPES is a set of type symbols,*
- *OPS is a set of operation symbols,*
- *$\leq$ is a pre-order on TYPES, and*
- *$\mathit{ResType}$ is a family of partial functions such that $\mathit{ResType} : \mathit{OPS} \times \mathit{TYPES} \to \mathit{TYPES}_{\perp}$, and $\mathit{ResType}$ is monotone. That is, for all $n \geq 0$ and $g \in \mathit{OPS}$, and for all types $\vec{S} \leq \vec{T}$, if $\mathit{ResType}(g, \vec{T}) \neq \perp$ then $\mathit{ResType}(g, \vec{S}) \neq \perp$ and $\mathit{ResType}(g, \vec{S}) \leq \mathit{ResType}(g, \vec{T})$.*

For notational convenience, in the rest of dissertation, we use signatures as if they are closed under the formation of tuple types. That is, for all $T_1, \cdots, T_n \in \mathit{TYPES}$, we consider $(T_1, \cdots, T_n) \in \mathit{TYPES}$. Similarly, $(S_1, \cdots, S_n) \leq (T_1, \cdots, T_n)$ only if $S_1 \leq T_1, \cdots$, and $S_n \leq T_n$.

Figure 2.3 gives an example signature, $\mathrm{E}\Sigma^{\mathbf{E}}$. The set of types of $\mathrm{E}\Sigma^{\mathbf{E}}$ includes `Void`, the visible types `Integer` and `Bool`, `MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount`, and `Store`. The binary relation, $\leq_{\mathrm{W}}$, defines subtype relationships between these types.

The type `Void` is used to indicate that a method has no results. The overloaded operator, `withdraw`, mutates a `PlusAccount` or a `BankAccount`, and returns nothing and the changed store. To model mutation, as in denotational semantics we use `Store` both as an argument and as a result for operations. (Stores are discussed in the next section along with operations on them.)

Algebras may have additional hidden types and operations. These hidden types are referred to as *sorts* and are used to implement external types. A signature that

$$\mathit{TYPES} \overset{\text{def}}{=} \{\texttt{Void}, \texttt{Bool}, \texttt{Integer}, \texttt{Store}, \texttt{MoneyObj}, \texttt{FrozenAccount}, \texttt{BankAccount}, \texttt{PlusAccount}\}$$

$$\leq_{\mathrm{W}} \overset{\text{def}}{=} \{(\texttt{PlusAccount}, \texttt{FrozenAccount}), (\texttt{PlusAccount}, \texttt{BankAccount})\} \cup \{(T, T) \mid T \in \mathit{TYPES}\}$$

OPS and *ResType*

`nothing`	:	$(\texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`value`	:	$(\texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Integer}, \texttt{Store})$
`change`	:	$(\texttt{MoneyObj}, \texttt{Integer}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`balance`	:	$(\texttt{FrozenAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`get_interest`	:	$(\texttt{FrozenAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`balance`	:	$(\texttt{BankAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`withdraw`	:	$(\texttt{BankAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`interest`	:	$(\texttt{BankAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`deposit`	:	$(\texttt{BankAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`balance`	:	$(\texttt{PlusAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`withdraw`	:	$(\texttt{PlusAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`interest`	:	$(\texttt{PlusAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`deposit`	:	$(\texttt{PlusAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`get_interest`	:	$(\texttt{PlusAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`check_deposit`	:	$(\texttt{PlusAccount}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{Void}, \texttt{Store})$
`check_balance`	:	$(\texttt{PlusAccount}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`mkMoneyObj`	:	$(\texttt{Integer}, \texttt{Store}) \to (\texttt{MoneyObj}, \texttt{Store})$
`mkFrozenAccount`	:	$(\texttt{MoneyObj}, \texttt{Store}) \to (\texttt{FrozenAccount}, \texttt{Store})$
`mkBankAccount`	:	$(\texttt{MoneyObj}, \texttt{Store}) \to (\texttt{BankAccount}, \texttt{Store})$
`mkPlusAccount`	:	$(\texttt{MoneyObj}, \texttt{MoneyObj}, \texttt{Store}) \to (\texttt{PlusAccount}, \texttt{Store})$

Figure 2.3: A signature, $\mathrm{E}\Sigma^{\mathbf{E}} = (\mathit{TYPES}, \leq_{\mathrm{W}}, \mathit{OPS} \cup \mathit{VISOPS}, \mathit{ResType})$, for various account types. *ResType* for *VISOPS* is given in the previous figure.

also describes these internal sorts and their operations is called an *internal signature*. Algebras can be thought of as having both an internal signature, for implementation, and an external signature, a common interface. Such signatures are called *hierarchical signatures* and are similar to those in [Wir90, page 734]. Since internal operations can operate on external types, we require that the internal signature of a hierarchical signature contain the external signature.

Definition 2.2.2 (Hierarchical signature) *A* hierarchical signature *consists of a pair,* $(I\Sigma, E\Sigma)$*, of an internal signature* $I\Sigma$ *and an external signature* $E\Sigma$ *such that* $I\Sigma.TYPES \supseteq E\Sigma.TYPES$*,* $(I\Sigma.\leq) \supseteq (E\Sigma.\leq)$*,* $I\Sigma.OPS \supseteq E\Sigma.OPS$*, and* $I\Sigma.ResType \supseteq E\Sigma.ResType$*.*

To differentiate between the internal and the external components of the signature, we use $ITYPES$ for $\mathrm{I}\Sigma.TYPES$ and $\mathrm{E}TYPES$ for $\mathrm{E}\Sigma.TYPES$. Similarly, use $\mathrm{I}OPS$ for $\mathrm{I}\Sigma.OPS$ and $\mathrm{E}OPS$ for $\mathrm{E}\Sigma.OPS$, and $\mathrm{I}ResType$ for $\mathrm{I}\Sigma.ResType$ and $\mathrm{E}ResType$ for $\mathrm{E}\Sigma.ResType$. However, we overload $\leq$ to refer to both $\mathrm{I}\Sigma.\leq$ and $\mathrm{E}\Sigma.\leq$.

2.3 Mutation signatures

As in denotational semantics, mutation is modeled using locations and stores. Locations can be thought of as object identities; they can be used to extract an object's value from a store. That is the abstract values of a mutable type are locations, which are mapped by a store to values of the corresponding internal *sort* for the mutable type. For example, the abstract values of `MoneyObj` objects are locations, which are mapped by a store to an integer. When a `MoneyObj`, is mutated, the store is modified so that the `MoneyObj`'s abstract value, which is a location, denotes a new integer in the modified store.

Since we model mutation algebraically, we require an abstract notion of stores, that is stores cannot be treated just as functions (as is common in denotational semantics, e.g., [Sch86]). The following discussion motivates the need for special types and operations to handle mutation and stores abstractly.

In a given algebra, a subset of $\mathrm{E}TYPES$ will be implemented as object types; that is as types whose carrier set is a set of locations. One could identify these object types by specifying them in the signature but doing that would make our signatures nonstandard. Instead we make a convention that all user defined types are object types and use a consistent notation to denote the corresponding sort. For an object type T, sortFor(T) is used as the corresponding internal sort. Using a common notation for all the internal sorts for object types allows us to easily talk about the subtype relationship between these internal sorts. The abstract values of these internal sorts vary from algebra to algebra allowing different implementations.

Hence, for any signature $\mathrm{E}\Sigma$, the set of object types can be defined as follows.

$$\mathit{ObjectTypes}(\mathrm{E}\Sigma) \overset{\text{def}}{=} \{T \mid T \in \mathrm{E}\mathit{TYPES} \Leftrightarrow \{\mathtt{Integer}, \mathtt{Bool}, \mathtt{Store}, \mathtt{Void}\}\}$$

The set of object types for $\mathrm{E}\Sigma^{\mathbf{E}}$, $\mathit{ObjectTypes}(\mathrm{E}\Sigma^{\mathbf{E}})$, is {`MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount`} (refer to Figure 2.4). Readers familiar with `C++` can think of an object type, T, as the reference of type, `T &`, and sortFor(T) as T. We drop the argument $\mathrm{E}\Sigma$ for *ObjectTypes* whenever the signature is obvious from the context.

Hierarchical signatures do not require signatures to have any specific types and operations. But, to model mutation we require `Store` both as an argument and as a result type for all external operations, to model mutation. We also need a type `Void` for external operations that have no return value other than a store. Hence, for mutation, we require hierarchical signatures to contain a type `Store`, a type `Void`, some internal operations on `Store`, and an external operation `nothing` on `Void`. We refer such signatures as *mutation signatures.*

The operation `nothing` allows one to return a value of type `Void`. Since we do not manipulate the value of `Void`, we do not require any additional operations on `Void`. Thus `Void` can be modeled as a one-point set.

In the definition of mutation signatures below, though we require `Store` to be an external type, the operations on it are internal. This might seem strange but since stores map values of external types (object types) to values of their internal sorts, exposing the operations of stores would expose the internal sorts. This is because operations on store, like `lookup`, return values of their internal sorts. Since internal operations need not take and return store as an argument, making the operation on stores internal allows a model of store that is closer to denotational semantics. So we treat `Store` as an external type and operations on `Store` as internal.

Definition 2.3.1 (Mutation signature) *A* mutation signature*, $(I\Sigma, E\Sigma)$ is a hierarchical signature, such that $E\Sigma$ satisfies the following conditions:*

- $\mathtt{Store}, \mathtt{Void} \in \mathit{ETYPES}$, *and* $\mathit{VIS} \subset \mathit{ETYPES}$,
- *each* $g \in \mathit{EOPS}$ *takes a store as an argument and returns store as a result, i.e., it has a signature of the form* $(S_1, \cdots, S_n, \mathtt{Store}) \rightarrow (T, \mathtt{Store})$,
- $\mathit{VISOPS} \subseteq \mathit{EOPS}$,
- $\mathtt{nothing} \in \mathit{EOPS}$ *such that* $\mathit{ResType}(\mathtt{nothing}, (\mathtt{Store})) = (\mathtt{Void}, \mathtt{Store})$,

and $I\Sigma$ *satisfies the following conditions:*

- $\texttt{Set}[\texttt{Loc}], \texttt{Loc} \in$ *ITYPES,*
- *for each* $T \in$ *ObjectTypes,* $\mathrm{sortFor}(T) \in$ *ITYPES,*
- *for each* $S, T \in$ *ObjectTypes, if* $S \leq T$ *then* $\mathrm{sortFor}(S) \leq \mathrm{sortFor}(T)$*,*
- *for each* $T \in$ *ObjectTypes,* $T \leq \texttt{Loc}$*,*
- *for each* $T \in$ *ObjectTypes,* `emptyStore`*,* `isInDom`*,* `lookup`*,* `update`*,* $\texttt{alloc}[T]$ $\in$ *IOPS such that*

$$\begin{array}{lcl} \mathit{ResType}(\texttt{emptyStore}, ()) & = & \texttt{Store} \\ \mathit{ResType}(\texttt{isInDom}, (T, \texttt{Store})) & = & \texttt{Bool} \\ \mathit{ResType}(\texttt{lookup}, (T, \texttt{Store})) & = & \mathrm{sortFor}(T) \\ \mathit{ResType}(\texttt{update}, (T, \mathrm{sortFor}(T), \texttt{Store})) & = & \texttt{Store} \\ \mathit{ResType}(\texttt{alloc}[T], (\mathrm{sortFor}(T), \texttt{Store})) & = & (T, \texttt{Store}) \end{array}$$

- $\texttt{emptySet}, \texttt{isIn}, \texttt{addSet} \in$ *IOPS such that that*

$$\begin{array}{lcl} \mathit{ResType}(\texttt{emptySet}, ()) & = & \texttt{Set}[\texttt{Loc}] \\ \mathit{ResType}(\texttt{isIn}, (\texttt{Loc}, \texttt{Set}[\texttt{Loc}])) & = & \texttt{Bool} \\ \mathit{ResType}(\texttt{addSet}, (\texttt{Loc}, \texttt{Set}[\texttt{Loc}])) & = & \texttt{Set}[\texttt{Loc}] \end{array}$$

- *for each* $T \in$ *ObjectTypes there exists a* $\texttt{containedObjs}[\mathrm{T}]$ *and a* $\texttt{isNominal}[T]$ *in IOPS such that*

$$\begin{array}{lcl} \mathit{ResType}(\texttt{containedObjs}[\mathrm{T}], (T, \texttt{Store})) & = & \texttt{Set}[\texttt{Loc}] \\ \mathit{ResType}(\texttt{isNominal}[T], (T, \texttt{Store})) & = & \texttt{Bool} \end{array}$$

Names of the form $\texttt{alloc}[T]$ may seem strange, but adding the type information to the name allows more flexibility in allocating objects and permits more subtyping. For instance, if we had only `alloc`, then passing a sortFor(`PlusAccount`) value to it would result in a `PlusAccount` location. But with two operations alloc[BankAccount] and alloc[PlusAccount], passing a sortFor(`PlusAccount`) value, v, to the operation alloc[BankAccount] can result in a `BankAccount` location with a value v and passing v to alloc[PlusAccount] results in a `PlusAccount` location with a value v.

For each object type T, we require two internal operations containedObjs[T] and $\texttt{isNominal}[T]$ in our study of behavioral subtyping in the context of aliasing. Often, as discussed in the next section, we need an alias graph of a store that links an object and its containing objects. For such a graph, we need to identify the components of an object. Since our goal is to hide the implementation details of the object, we need internal operations, which when given an object and a store return the set of contained objects. For these operations to be polymorphic, they need to satisfy monotonicity. Hence, we introduce a new internal type, Loc, as a supertype of all object types and require that the result type of containedObjs[T] be Set[Loc]. Note that we do not require any operations on Loc and it can be considered as just the common supertype of object types.

For some types, like the Collection [Gol84, Coo92] types, the contained objects vary dynamically. For example, the contained objects in a List type vary with mutation, that is adding a new object to a list changes its contained objects. Hence, we cannot have a fixed tuple type as the return type of containedObjs[List]. So we introduce a sort to contain a set of locations, Set[Loc].

For each object type T, given a value of T and a store, containedObjs[T] returns a set of objects that are in the abstract value denoted by the value in the store. If the carrier set of sortFor(MoneyObj) is a set of integers, then containedObjs[MoneyObj] applied to a MoneyObj object and a store returns an empty set, because it contains an integer value, not a location. If the carrier set of sortFor(BankAccount) is a set consisting of money objects, then the containedObjs[BankAccount] on a BankAccount returns either a singleton set with a MoneyObj or the contained objects of sortFor(PlusAccount) (because PlusAccount $\leq_{\mathrm{W}}$ BankAccount). This information is used to construct an alias graph (in later sections).

Another internal operation that is required for our study of behavioral subtyping is isNominal[T]. For our study, we compare results of operations in stores with subtyping to the results of operations in stores without subtyping. By stores with subtyping, we mean stores in which locations of type T denote values of sortFor(S) where $S \leq T$ whereas in stores without subtyping locations of type T denote values of only sortFor(T). Given a location and a store, isNominal[T] should return true if the location and all the objects that are reachable from the location do not denote any subtype values. If this is true for all locations in a store then we can conclude that the store contains no subtyping. More details, including motivation for isNominal[T], are provided in the next chapter.

Figure 2.4 gives an example mutation signature. The subtype relation $\leq_{\mathrm{W}}$ is overloaded for both types (external types) and sorts (internal types). The internal operations alloc[MoneyObj], alloc[FrozenAccount], alloc[BankAccount], and alloc[PlusAccount] are used to create MoneyObj, FrozenAccount, BankAccount, and PlusAccount objects respectively. Making these operations internal, hides implementation details of object types, such as, that a PlusAccount contains a pair of MoneyObj objects. These are different from the external operations like, mkMoneyObj, which do not give away any implementation details. The additional operations in IΣ are operations on Store.

$$\mathit{ITYPES} \overset{\text{def}}{=} (\mathit{TYPES} \cup \{\text{sortFor}(\texttt{MoneyObj}), \text{sortFor}(\texttt{FrozenAccount}), \text{sortFor}(\texttt{BankAccount}), \text{sortFor}(\texttt{PlusAccount}) \; \texttt{Loc}, \texttt{Set}[\texttt{Loc}]\})$$

$$\begin{aligned} \mathrm{I}\Sigma.\leq_{\mathrm{W}} \overset{\text{def}}{=} & \{(\texttt{PlusAccount}, \texttt{FrozenAccount}), (\texttt{PlusAccount}, \texttt{BankAccount}), \\ & (\text{sortFor}(\texttt{PlusAccount}), \text{sortFor}(\texttt{FrozenAccount})), \\ & (\text{sortFor}(\texttt{PlusAccount}), \text{sortFor}(\texttt{BankAccount})) \\ & (\texttt{MoneyObj}, \texttt{Loc}), (\texttt{FrozenAccount}, \texttt{Loc}) \\ & (\texttt{BankAccount}, \texttt{Loc}), (\texttt{PlusAccount}, \texttt{Loc})\} \\ \cup \; & \{(S, S) \mid S \in \mathit{ITYPES}\} \end{aligned}$$

I*OPS* and *ResType*

E*OPS* ⊂ I*OPS* and *EResType* ⊂ I*ResType*

`containedObjs[MoneyObj]`	:	`(MoneyObj, Store)` $\to$ `(Set[Loc])`
`containedObjs[FrozenAccount]`	:	`(FrozenAccount, Store)` $\to$ `(Set[Loc])`
`containedObjs[BankAccount]`	:	`(BankAccount, Store)` $\to$ `(Set[Loc])`
`containedObjs[PlusAccount]`	:	`(PlusAccount, Store)` $\to$ `(Set[Loc])`
`isNominal[MoneyObj]`	:	`(MoneyObj, Store)` $\to$ `Bool`
`isNominal[FrozenAccount]`	:	`(FrozenAccount, Store)` $\to$ `Bool`
`isNominal[BankAccount]`	:	`(BankAccount, Store)` $\to$ `Bool`
`isNominal[PlusAccount]`	:	`(PlusAccount, Store)` $\to$ `Bool`
`emptyStore`	:	`()` $\to$ `(Store)`
`emptySet`	:	`()` $\to$ `(Set[Loc])`
`isIn`	:	`(Loc, Set[Loc])` $\to$ `Bool`
`addSet`	:	`(Loc, Set[Loc])` $\to$ `Set[Loc]`
$\forall T \in \mathit{ObjectTypes}$		
`isInDom`	:	$(T,$ `Store`$)$ $\to$ `Bool`
`lookup`	:	$(T,$ `Store`$)$ $\to$ sortFor(T)
`update`	:	$(T, \text{sortFor}(T),$ `Store`$)$ $\to$ `(Store)`
`alloc[`T`]`	:	$(\text{sortFor}(T),$ `Store`$)$ $\to$ $(T,$ `Store`$)$

Figure 2.4: A mutation signature, $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$, where EΣ is shown in the previous figures. Note that $\mathit{ObjectTypes}((\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}})$ = {`MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount`}.

2.4 Mutation algebras

We first define standard Σ-algebras and then define mutation algebras. Algebras that satisfy a signature Σ are called Σ-algebras. These are the variety of algebras that model the types and operations of a signature Σ.

Definition 2.4.1 (Σ-algebra) *Let Σ be a signature. A Σ-algebra, $\mathbf{A}$, is a pair $(VALS^{\mathbf{A}}, OPS^{\mathbf{A}})$, where*

- *$VALS^{\mathbf{A}}$ is a family of sets of abstract values, indexed by $TYPES$, and*
- *$OPS^{\mathbf{A}}$ is a set of operation interpretations such that for each $g \in OPS$, there is a polymorphic partial function $g^{\mathbf{A}} \in OPS^{\mathbf{A}}$ where for each $\vec{S} \in TYPES$ and $T \in TYPES$, if $ResType(g, \vec{S}) = T$ then $g^{\mathbf{A}}$ satisfies*

$$g^{\mathbf{A}} : VALS^{\mathbf{A}}_{\vec{S}} \rightarrow (\cup_{U \leq T} VALS^{\mathbf{A}}_{U\,\perp}).$$

- *For all $T \in VIS$, $VALS^{\mathbf{A}}_T = EXTERNALS_T$.*

The last condition allows comparisons between values of visibles types in different algebras, and is important for the discussion of observable behavior.

We use $VALS^{\mathbf{A}}_T$ to denote the abstract values of T in $\mathbf{A}$. The notation $\widehat{VALS}^{\mathbf{A}}_T$ abbreviates the set of all abstract values of the subtypes of T; that is, $\widehat{VALS}^{\mathbf{A}}_T = \bigcup_{U \leq T}(VALS^{\mathbf{A}}_U)$. For operations with zero arguments, we use $g^{\mathbf{A}}$ to denote $g^{\mathbf{A}}()$. For example, $\texttt{emptyStore}^{\mathbf{A}}$ is used for $\texttt{emptyStore}^{\mathbf{A}}()$.

A EΣ-algebra, $\mathbf{A}$, is a $(I\Sigma, E\Sigma)$-*hierarchical algebra* if it is a IΣ-algebra. We are interested in the class of algebras that correspond to the mutation signatures, that is, that satisfy certain properties on stores. These algebras, referred to as *mutation algebras*, are a subset of hierarchical algebras.

Definition 2.4.2 (Mutation algebra) *Let $(I\Sigma, E\Sigma)$ be a mutation signature. A $(I\Sigma, E\Sigma)$-algebra, $\mathbf{A}$, is a $(I\Sigma, E\Sigma)$-mutation algebra if*

- *for all $\sigma \in VALS^{\mathbf{A}}_{\texttt{Store}}$, $T \in ObjectTypes$, $l, l' \in VALS^{\mathbf{A}}_T$, and for all $v \in \widehat{VALS}^{\mathbf{A}}_{\mathrm{sortFor}(T)}$ each of the following is true:*

$$\begin{array}{lcl}
\texttt{isInDom}^{\mathbf{A}}(l, \texttt{emptyStore}^{\mathbf{A}}) & = & \mathit{false} \\
\texttt{isInDom}^{\mathbf{A}}(l, \texttt{update}^{\mathbf{A}}(l', v, \sigma)) & = & (l = l') \vee \texttt{isInDom}^{\mathbf{A}}(l, \sigma) \\
\texttt{lookup}^{\mathbf{A}}(l, \texttt{update}^{\mathbf{A}}(l', v, \sigma)) & = & \textbf{if } (l = l') \textbf{ then } v \textbf{ else } \texttt{lookup}^{\mathbf{A}}(l, \sigma) \\
\texttt{lookup}^{\mathbf{A}}(l, \texttt{emptyStore}^{\mathbf{A}}) & = & \perp
\end{array}$$

$$\begin{array}{l}
\textbf{let } (l, \sigma') = \texttt{alloc}[T]^{\mathbf{A}}(v, \sigma) \textbf{ in } (\texttt{isInDom}^{\mathbf{A}}(l, \sigma) = \mathit{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \sigma' = \texttt{update}^{\mathbf{A}}(l, v, \sigma))
\end{array}$$

- *for all* $s \in \mathit{VALS}^{\mathbf{A}}_{\texttt{Set[Loc]}}$, $T \in \mathit{ObjectTypes}$, *and* $v, v' \in \mathit{VALS}^{\mathbf{A}}_{T}$ *each of the following are true:*

$$\begin{aligned}
\texttt{isIn}^{\mathbf{A}}(v, \texttt{emptySet}^{\mathbf{A}}) &= \mathit{false} \\
\texttt{isIn}^{\mathbf{A}}(v, \texttt{addSet}^{\mathbf{A}}(v', s)) &= \textbf{if } (v = v') \textbf{ then } \mathit{true} \textbf{ else } \texttt{isIn}^{\mathbf{A}}(v, s)
\end{aligned}$$

To ensure that the type `Store` model stores as in denotational semantics, we specify properties for operations on stores. That is, the operation $\texttt{update}^{\mathbf{A}}(l, v, \sigma)$, returns a new store after binding l to v. The operation, $\texttt{lookup}^{\mathbf{A}}$ on l on such a store should return v.

Typically, in denotational semantics stores are modeled as functions. To enable such a view for the stores in our algebra, we adopt sugared forms for the operations on stores; these are used when the algebra is clear from the context. We use $l \in \mathit{Domain}(\sigma)$ for $\texttt{isInDom}^{\mathbf{A}}(l, \sigma)$, $(\sigma\ l)$ for $\texttt{lookup}^{\mathbf{A}}(l, \sigma)$, and $[l \mapsto v]\sigma$ for $\texttt{update}^{\mathbf{A}}(l, v, \sigma)$.

We also use $v \in v'$ for $\texttt{isIn}^{\mathbf{A}}(v, v')$.

Figure 2.5 gives the carrier sets for an example $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$-mutation algebra **E**. Note that the values of all the object types, that is the types `MoneyObj` and all the account types, are typed locations. The carrier sets of sortFor(`FrozenAccount`) and sortFor(`BankAccount`) contain money locations while sortFor(`PlusAccount`) contains a pair of money locations. Modeling account types as money locations instead of integers permits interesting types of aliasing in the store.

Figure 2.6 shows the operation interpretations of **E**. For the sake of clarity, we expand the interpretations of polymorphic operations as if they are different operations. That is, the operations interpretation of `balance` is expanded as three interpretations based on the type of the argument. To obtain the interpretation of the unsugared form, we test the argument's type and branch into the corresponding interpretation. For example, `balance` is expanded as follows.

$$\begin{aligned}
\texttt{balance}^{\mathbf{E}}(v, \sigma) \stackrel{\text{def}}{=}\ & \textbf{if } (v \in \mathit{VALS}^{\mathbf{E}}_{\texttt{FrozenAccount}}) \textbf{ then } \texttt{alloc[MoneyObj]}^{\mathbf{E}}((\sigma\ v), \sigma) \\
& \textbf{else if } (v \in \mathit{VALS}^{\mathbf{E}}_{\texttt{BankAccount}}) \\
& \quad \textbf{then } \texttt{alloc[MoneyObj]}^{\mathbf{E}}((\sigma\ v), \sigma) \\
& \textbf{else if } (v \in \mathit{VALS}^{\mathbf{E}}_{\texttt{PlusAccount}}) \textbf{ then let } (m_1, m_2) = (\sigma\ v) \textbf{ in} \\
& \qquad \texttt{alloc[MoneyObj]}^{\mathbf{E}}((\sigma\ m_1) + (\sigma\ m_2), \sigma) \\
& \textbf{else } \bot
\end{aligned}$$

Figure 2.7 gives the operations interpretations for internal operations of **E**.

$$
\begin{aligned}
\mathit{VALS}^{\mathbf{E}}_{\texttt{Void}} &\stackrel{\text{def}}{=} \{*\} \\
\mathit{VALS}^{\mathbf{E}}_{\texttt{Bool}} &\stackrel{\text{def}}{=} \{\mathit{true}, \mathit{false}\} \\
\mathit{VALS}^{\mathbf{E}}_{\texttt{Integer}} &\stackrel{\text{def}}{=} \{0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\} \\
\mathit{VALS}^{\mathbf{E}}_{T} &\stackrel{\text{def}}{=} \mathit{LOCS}^{\mathbf{E}}_{T}, \text{for each } T \in \mathit{ObjectTypes} \\
\mathit{VALS}^{\mathbf{E}}_{\text{sortFor}(\texttt{MoneyObj})} &\stackrel{\text{def}}{=} \mathit{VALS}^{\mathbf{E}}_{\texttt{Integer}} \\
\mathit{VALS}^{\mathbf{E}}_{\text{sortFor}(\texttt{FrozenAccount})} &\stackrel{\text{def}}{=} \mathit{VALS}^{\mathbf{E}}_{\texttt{MoneyObj}} \\
\mathit{VALS}^{\mathbf{E}}_{\text{sortFor}(\texttt{BankAccount})} &\stackrel{\text{def}}{=} \mathit{VALS}^{\mathbf{E}}_{\texttt{MoneyObj}} \\
\mathit{VALS}^{\mathbf{E}}_{\text{sortFor}(\texttt{PlusAccount})} &\stackrel{\text{def}}{=} \mathit{VALS}^{\mathbf{E}}_{\texttt{MoneyObj}} \times \mathit{VALS}^{\mathbf{E}}_{\texttt{MoneyObj}} \\
\mathit{VALS}^{\mathbf{E}}_{\texttt{Store}} &\stackrel{\text{def}}{=} \mathit{LOCS}^{\mathbf{E}} \rightarrow \mathit{VALS}^{\mathbf{E}}{}_{\perp} \\
\mathit{VALS}^{\mathbf{E}}_{\texttt{Loc}} &\stackrel{\text{def}}{=} \{l_i \mid i \in \text{Nat}\} \\
\mathit{VALS}^{\mathbf{E}}_{\texttt{Set[Loc]}} &\stackrel{\text{def}}{=} \texttt{Loc} \rightarrow \texttt{Bool} \\
\mathit{VALS}^{\mathbf{E}}_{(S_1 \times S_2 \cdots S_n)} &\stackrel{\text{def}}{=} (\mathit{VALS}^{\mathbf{E}}_{S_1} \times \cdots \times \mathit{VALS}^{\mathbf{E}}_{S_n}), \\
&\qquad \text{for each } S_1, \cdots, S_n \in \mathit{ITYPES}
\end{aligned}
$$

where

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

Figure 2.5: Abstract values of the $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$ mutation algebra $\mathbf{E}$.

$$\begin{array}{lcl}
\mathtt{mkMoneyObj}^{\mathbf{E}}(v^i, \sigma) & \overset{\text{def}}{=} & \mathtt{alloc[MoneyObj]}^{\mathbf{E}}(v^i, \sigma) \\
\mathtt{value}^{\mathbf{E}}(m^m, \sigma) & \overset{\text{def}}{=} & ((\sigma\ m^m),\ \sigma) \\
\mathtt{mkFrozenAccount}^{\mathbf{E}}(m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m', \sigma') = \mathtt{alloc[MoneyObj]}((\sigma\ m^m), \sigma)^{\mathbf{E}} \textbf{ in} \\
 & & \mathtt{alloc[FrozenAccount]}^{\mathbf{E}}(m', \sigma') \\
\mathtt{balance}^{\mathbf{E}}(f^f, \sigma) & \overset{\text{def}}{=} & \mathtt{alloc[MoneyObj]}^{\mathbf{E}}((\sigma\ (\sigma\ f^f)), \sigma) \\
\mathtt{mkBankAccount}^{\mathbf{E}}(m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m', \sigma') = \mathtt{alloc[MoneyObj]}((\sigma\ m^m), \sigma)^{\mathbf{E}} \textbf{ in} \\
 & & \mathtt{alloc[BankAccount]}^{\mathbf{E}}(m', \sigma') \\
\mathtt{balance}^{\mathbf{E}}(b^b, \sigma) & \overset{\text{def}}{=} & \mathtt{alloc[MoneyObj]}^{\mathbf{E}}((\sigma\ (\sigma\ b^b)), \sigma) \\
\mathtt{deposit}^{\mathbf{E}}(b^b, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } m' = (\sigma\ b^b) \textbf{ in} \\
 & & \mathtt{update}^{\mathbf{E}}(m', (\sigma\ m^m) + (\sigma\ m'), \sigma) \\
\mathtt{withdraw}^{\mathbf{E}}(b^b, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } m' = (\sigma\ b^b) \textbf{ in} \\
 & & \textbf{let } (v', \sigma) = (\sigma\ m') \textbf{ in} \\
 & & \textbf{let } (v, \sigma) = (\sigma\ m^m) \textbf{ in} \\
 & & \textbf{if } v' > v \textbf{ then } \mathtt{update}^{\mathbf{E}}(m', (v' \Leftrightarrow v), \sigma) \textbf{ else } \bot \\
\mathtt{mkPlusAccount}^{\mathbf{E}}(m_1^m, m_2^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_1', \sigma_1') = \mathtt{alloc[MoneyObj]}((\sigma\ m_1^m), \sigma)^{\mathbf{E}} \textbf{ in} \\
 & & \textbf{let } (m_2', \sigma_2') = \mathtt{alloc[MoneyObj]}((\sigma\ m_2^m), \sigma_1')^{\mathbf{E}} \textbf{ in} \\
 & & \mathtt{alloc[PlusAccount]}^{\mathbf{E}}((m_1', m_2'), \sigma_2') \\
\mathtt{balance}^{\mathbf{E}}(p^p, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_1, m_2) = (\sigma\ p^p) \textbf{ in} \\
 & & \mathtt{alloc[MoneyObj]}((\sigma\ m_1) + (\sigma\ m_2), \sigma) \\
\mathtt{withdraw}^{\mathbf{E}}(p^p, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ p^p) \textbf{ in} \\
 & & \textbf{let } v_s = (\sigma\ m_s) \textbf{ in} \\
 & & \textbf{let } v_c = (\sigma\ m_c) \textbf{ in} \\
 & & \textbf{let } v' = (\sigma\ m^m) \textbf{ in} \\
 & & \textbf{if } v_s > v' \textbf{ then } (*, [m_s \mapsto (v_s \Leftrightarrow v')]\sigma) \\
 & & \textbf{else if } (v_s + v_l) > v' \\
 & & \qquad \textbf{then } (*, [m_s \mapsto 0][m_c \mapsto (v_s + v_l \Leftrightarrow v')]\sigma) \\
 & & \qquad \textbf{else } \bot \\
 & \ldots &
\end{array}$$

Figure 2.6: Operation interpretations of the $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$-mutation algebra $\mathbf{E}$. More details can be found in the Appendix A. The superscript i, m, f, b, and p denote values of `Integer`, `MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount`.

$$\begin{array}{lcl}
\texttt{containedObjs[MoneyObj]}^{\mathbf{E}}(m^m, \sigma) & \overset{\text{def}}{=} & (\lambda(m).\mathit{false}) \\
\texttt{containedObjs[FrozenAccount]}^{\mathbf{E}}(f^f, \sigma) & \overset{\text{def}}{=} & \textbf{let } m = (\sigma\ f^f) \textbf{ in} \\
 & & (\lambda(m').\textbf{if } (m' = m) \textbf{ then } \mathit{true} \\
 & & \qquad\qquad \textbf{else } \mathit{false}) \\
\texttt{containedObjs[BankAccount]}^{\mathbf{E}}(b^b, \sigma) & \overset{\text{def}}{=} & \textbf{let } m = (\sigma\ b^b) \textbf{ in} \\
 & & (\lambda(m').\textbf{if } (m' = m) \textbf{ then } \mathit{true} \\
 & & \qquad\qquad \textbf{else } \mathit{false}) \\
\texttt{containedObjs[PlusAccount]}^{\mathbf{E}}(p^p, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_1, m_2) = (\sigma\ p^p) \textbf{ in} \\
 & & (\lambda(m').\textbf{if } (m' = m_1) \textbf{ then } \mathit{true} \\
 & & \textbf{else if } (m' = m_2) \textbf{ then } \mathit{true} \\
 & & \qquad\qquad \textbf{else } \mathit{false}) \\
\forall T \in \mathit{ObjectTypes} & & \\
\texttt{isNominal}[T]^{\mathbf{E}}(v^T, \sigma) & \overset{\text{def}}{=} & \textbf{if } (v^T \in \mathit{VALS}^{\mathbf{E}}_T) \textbf{ then } \mathit{true} \\
 & & \qquad\qquad \textbf{else } \mathit{false} \\
 & \ldots &
\end{array}$$

Figure 2.7: Operation interpretations of internal operations of $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$-mutation algebra **E**. Additional details can be found in Appendix A.

$(\sigma\ l_1^{\texttt{BankAccount}}) = (l_3^{\texttt{MoneyObj}}, l_4^{\texttt{MoneyObj}}),\ (\sigma\ l_2^{\texttt{BankAccount}}) = (l_4^{\texttt{MoneyObj}}, l_5^{\texttt{MoneyObj}})$
$(\sigma\ l_3^{\texttt{MoneyObj}}) = 100, (\sigma\ l_4^{\texttt{MoneyObj}}) = 100, (\sigma\ l_5^{\texttt{MoneyObj}}) = 200$

Figure 2.8: A sample store of $\mathbf{E}$, that is $\sigma \in \mathit{VALS}^{\mathbf{E}}_{\texttt{Store}}$.

2.5 Aliasing in stores

From the examples in Chapter 1 we have seen that behavioral subtyping depends on aliasing between objects of different types. In this subsection, we define what it means for two locations to be aliased in a store. Since we are interested only in direct aliasing between locations of different types, we limit our discussion to only to direct aliases.

First, we define an alias graph of a store. Let $\mathbf{A}$ be a $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra and let $\sigma \in \mathit{VALS}^{\mathbf{A}}_{\texttt{Store}}$. Then the notation $\texttt{aliasG}(\sigma)$ is used for the alias graph of σ. It has nodes and edges as follows. The nodes of $\texttt{aliasG}(\sigma)$ are all values v such that $v \in \mathit{Domain}(\sigma)$ holds. Recall that $v \in \mathit{Domain}(\sigma)$ is sugar for $\texttt{isInDom}^{\mathbf{E}}(v, \sigma)$, when $v \in \mathit{VALS}^{\mathbf{E}}$ and $\sigma \in \mathit{VALS}^{\mathbf{E}}_{\texttt{Store}}$. The directed edges of $\texttt{aliasG}(\sigma)$ are pairs of (l, v), for every $l \in \mathit{VALS}^{\mathbf{A}}_T$, $v \in \mathit{VALS}^{\mathbf{A}}_U$, where $T, U \in \mathit{ObjectTypes}$ and $v \in \texttt{containedObjs[T]}^{\mathbf{A}}(l, \sigma)$.

Definition 2.5.1 (Direct aliasing) *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra. Let* $\sigma \in \mathit{VALS}^{\mathbf{A}}_{\texttt{Store}}$*. Let* $l_1 \in \mathit{VALS}^{\mathbf{A}}_U$ *and* $l_2 \in \mathit{VALS}^{\mathbf{A}}_V$*, where* $U, V \in \mathit{ObjectTypes}$*. Then* l_1 and l_2 are direct aliases in σ *if there exists a node* v *in* $\texttt{aliasG}(\sigma)$ *such that* (l_1, v) *and* (l_2, v) *are edges of* $\texttt{aliasG}(\sigma)$*.*

Figure 2.9 gives the alias graph of a store given in Figure 2.8. The locations $l_1^{\texttt{BankAccount}}$ and $l_2^{\texttt{BankAccount}}$ in σ, given in Figure 2.9, are direct aliases because of the edges $(l_1^{\texttt{BankAccount}}, l_4^{\texttt{MoneyObj}})$ and $(l_2^{\texttt{BankAccount}}, l_4^{\texttt{MoneyObj}})$. We use this notion of direct aliasing in our study of weak behavioral subtyping.

2.6 Related work

In this section, we present a brief description of some work related to mutation algebras. Mutation algebras introduced in this chapter are extensions to standard algebraic techniques described, for example, in [EM85], [GD94].

Gougen and Diaconescu [GD94] treat states as terms of an algebra and use them to describe properties of data types. States map variables to values and the values are immutable. One does not have a notion of object identities and hence no mutation

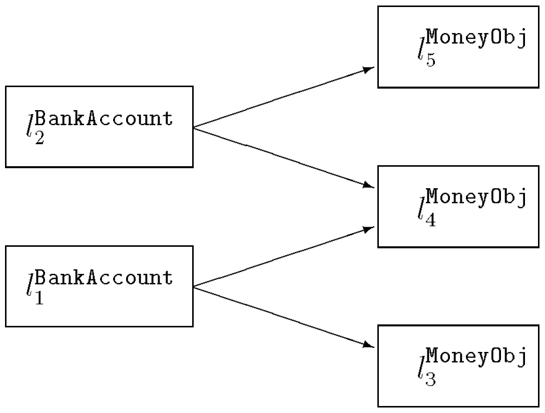
$l_5^{\texttt{MoneyObj}}$
$l_2^{\texttt{BankAccount}}$
$l_4^{\texttt{MoneyObj}}$
$l_1^{\texttt{BankAccount}}$
$l_3^{\texttt{MoneyObj}}$

Figure 2.9: Alias graph of σ given in the previous figure. Locations $l_1^{\texttt{BankAccount}}$ and $l_2^{\texttt{BankAccount}}$ directly alias a location.

and aliasing of objects. However, their treatment of states as hidden sorts is similar to our `Store`, though we require `Store` as an external type to model mutation.

Evolving algebras [Gur91] have a fixed carrier set that gives values to terms in a state and function updates that keep track of the changes of structures. For a given algorithm one writes the transformations on structures. Evolving algebras are more operational in nature than our mutation algebras. However, they are not algebraic in the sense that the purpose of the algebras in evolving algebras is only to provide the structures and hence, one does not define any homomorphism between these algebras. We require homomorphisms between different algebras to define behavioral subtype relations. Further, our approach is more denotational. In our split semantics (given in Chapter 4), we treat mutation algebras as arguments to the valuation functions of the denotational semantics. It is also not clear how to achieve this split in the context of operational semantics as the configurations depend on specific algebras.

Another interesting work that explicitly treats the with modeling mutable types is *D-oids* [AZ93]. Essentially, a *D-oid* consists of a set of instant structures and dynamic operations. These instant structures can be thought of as configurations (states) that contain objects along with their operations. A dynamic operation is a mapping between these configurations, which also maps higher-level objects or structures in the static framework. Static structures are the standard algebras used for immutable types. Object identities across different configurations are preserved by a tracking map. Hence the notion of object identity is more abstract than our notion of location. Like evolving algebras, D-oids are operational in nature but the underlying structures and their transformations are studied algebraically and a category of d-oids is constructed. However, our approach is more denotational, so having explicit locations helps in a better understanding of subtype relationships since it is closer to the way mutation is handled in denotational semantics.

In [Wag92], Wagner gives semantics for a language with objects, classes, and inheritance. For a given a class system, he defines a state and gives semantics of a language that uses it. One can think of these states as algebras. This work is different both philosophically and technically from mutation algebras. Our goal in defining mutation algebras is to model a set of types, which may contain mutable objects. Wagner's work concentrates on an algebraic notion for a set of classes and to model inheritance. The technical difference comes from the fact that while we try to hide the internal details of objects, it is not an issue in Wagner's semantics. In fact, he needs a mapping (ι) that gives out the internal representation of object, which enables him to transform subclass objects to superclass objects and back. Another way of looking at this comparison is while our work models subtype polymorphism, Wagner's work gives a mathematical explanation for a set of classes with inheritance.

Other related work includes action semantics [Mos92], [Wat91], which specify data using algebraic techniques. Action semantics, unlike denotational semantics, is more operational in nature. It would be interesting to look at applying the concepts of actions in our mutation algebras and split semantics.

Mason and Talcott [MT91], [MT92] studied the semantics of functional languages with mutation, like LISP [Mas86]. Their approach is mainly operational and uses equational logics for proving program equivalences. Further, they limit interferences (aliasing) to within objects. They work on axioms over a particular language while we work on algebras and operations of the algebras. Our notion of equivalence (simulations defined in the next chapter) is defined using algebras and visible types of the algebras. Using operations of algebra rather than language features is essential to define behavioral subtyping independent of any programming language.

2.7 Summary

In this chapter we presented an algebraic model for mutation that closely mimics denotational semantics. We introduced mutation signature that contain stores, locations, and operations on them. To hide all the implementation details from external view, we define mutation signatures as hierarchical signatures, where only the external signature is visible. A simulation relation on algebras with a common external signature is defined in the next chapter.

Table 2.1: Overview of the notations adopted

Notation	Description
VIS	Visible types
$VISOPS$	Operations on visible types
IΣ	Internal Signature
EΣ	External Signature
E$TYPES$	External types also referred as types
$ITYPES$	Internal types also referred as sorts
EOPS	External Ops
IOPS	Internal Ops
$\leq$	Presumed Subtype relation, overloaded for types and sorts
sortFor(T)	Internal type or sort for object type T
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \cdots$	Mutation algebras
l	Values of object types, also referred as locations
$\widehat{VALS}^{\mathbf{A}}_{T}$	$\cup_{S \leq T} VALS^{\mathbf{A}}_{S}$
$g^{\mathbf{A}}$	$g^{\mathbf{A}}()$
$\sigma, \sigma', \sigma_{\mathbf{A}}$	Stores
$l \in Domain(\sigma)$	$\mathtt{isInDom}^{\mathbf{A}}(l, \sigma)$
$(\sigma\ l)$	$\mathtt{lookup}^{\mathbf{A}}(l, \sigma)$
$[l \mapsto v]\sigma$	$\mathtt{update}^{\mathbf{A}}(l, v, \sigma)$
$v \in l$	$\mathtt{isIn}^{\mathbf{A}}(v, l)$

We conclude by presenting Table 2.1, which, for convenience, gives a quick overview of the notations adopted in this chapter.

3. WEAK AND STRONG BEHAVIORAL SUBTYPING

This chapter defines weak and strong behavioral subtyping. Presenting these notions before introducing any programming language allows us to discuss behavioral subtyping in a general setting. This chapter forms the core of this dissertation.

Intuitively, for a type to be a behavioral subtype of another, subtype objects should "behave like" supertype objects. This "behaves like" notion requires that every subtype object should be related, in a certain sense, to a supertype object, and that subtype methods preserve these relationships. That is, behavioral subtyping requires some form of relation between subtype and supertype objects. Such a relationship would allow one to substitute subtype objects in place of supertype objects without surprising behavioral changes [Lis88]. We refer to such relations as *simulation relations*.

These simulation relations form an important component in the definition of weak and strong behavioral subtyping. To define simulations, the first thing one would consider is to relate subtype values to supertype values. That is, in our example, relate the abstract values of `PlusAccount` to that of `BankAccount` values. Since the abstract values of `PlusAccount` and `BankAccount` are locations, relating a `PlusAccount` object to a `BankAccount` object amounts to relating their corresponding locations. But mutating a `PlusAccount` object means changing the value associated with the location in the corresponding store and not its location. Because locations do not change, just relating locations cannot capture the effects of mutation on the `PlusAccount` object and the related `BankAccount` object. So for methods to preserve the "behaves like" property between locations, the corresponding stores, where the mutation is modeled, should be related.

Since our goal is to study behavioral subtyping in the context of mutation and aliasing, we also need to consider the effects of mutation on identifiers that are aliases. But stores cannot see the aliasing between identifiers, so we introduce a notion of an environment and define simulations over environments.

In the next section, we define environments and in section 3.2 we formalize these notions. In section 3.3, we define a nominal environment and discuss the alias requirements for nominal environments in section 3.4. Using nominal environments we define simulation relations in section 3.5. Weak behavioral subtyping is defined in section 3.6 and an example is presented in 3.7. We define strong behavioral subtyping in 3.8 and an example in the following section. We discuss these definitions in Section 3.10 and compare them with some related work in Section 3.11.

3.1 Environments

Environments are introduced in this section. Environments permit all forms of aliasing among identifiers and locations (objects). Since stores are just like any other values in mutation algebras, we assign a store to a special identifier, `store`, in the environment. Since these abstract values depend on a particular algebra, we refer to the environments over a particular algebra. Environments can be thought of as states of a program, where the identifiers, other than `store`, represent identifiers of a program and `store` represents the current store. These notions are also used in the semantics of programs given in the next chapter.

The set TENV(Σ) of type environments over an external signature Σ is defined by TENV(Σ) = Identifier $\rightarrow$ $\mathit{TYPES}_{\perp}$, where Identifier is the set of all identifiers. We use H to denote a (typical) type environment.

The set $\mathit{Env}_H[\mathbf{A}]$ of *H-environments over* $\mathbf{A}$ is the set of all mappings, η : Identifier $\rightarrow$ ${\mathit{VALS}^{\mathbf{A}}}_{\perp}$ such that if $H(x) = T$ then $x \in \mathit{Domain}(\eta)$ and $(\eta\ x) \in \widehat{\mathit{VALS}}^{\mathbf{A}}_{T}$. We use the notation $[I \mapsto v]\eta$ as a shorthand to mean updating the value of I with v in η. That is, $([I_2 \mapsto v]\eta, I_1) = \textbf{if } I_1 = I_2 \textbf{ then } v \textbf{ else } (\eta\ I_1)$.

Valid environments capture, in a certain sense, the well formedness of environments over algebras.

Definition 3.1.1 (valid environments) *Let* $\mathbf{A}$ *be a* $(E\Sigma, I\Sigma)$*-mutation algebra and let* H *be a type environment. A* H*-environment,* η*, over* $\mathbf{A}$ *is* valid *if and only if* $H(\texttt{store}) = \texttt{Store}$*, and for all* $T \in (\mathit{ETYPES} \Leftrightarrow \{\texttt{Store}\})$*, and for all* $x \in \mathit{Domain}(H)$ *such that* $H(x) = T$*, if* $(\eta\ x) \in \mathit{Domain}(\eta\ \texttt{store})$ *then for each value* $v \in \texttt{containedObjs[T]}^{\mathbf{A}}((\eta\ x), (\eta\ \texttt{store}))$*,* $v \in \mathit{Domain}(\eta\ \texttt{store})$*.*

We consider only valid environments for the rest of this dissertation.

3.2 Homomorphic relations

Homomorphic relations are defined on environments over mutation algebras. These mutation algebras contain arbitrary abstract data types and a few visible types. Recall that in our model the set of visible types consists of `Integer` and `Bool`. Homomorphic relations defined in this section capture when an algebra simulates another. These homomorphic relations are similar to the generalized homomorphic relations defined in [LP94] in the sense that they require "substitution", "*VIS*-identical", and "bistrict" properties to be satisfied by related environments. The difference is in our explicit treatment of stores. More discussion on the comparison can be found in section 3.11.

Homomorphic relations capture the basic properties of "behaves like" for subtype objects and are further constrained to define weak and strong behavioral subtyping below. The additional properties are stated in the definitions of weak and strong behavioral subtypes.

Definition 3.2.1 (homomorphic relation) *Let* $\mathbf{A}$ *be a* $(I\Sigma', E\Sigma)$*-mutation algebra and* $\mathbf{C}$ *be a* $(I\Sigma'', E\Sigma)$*-mutation algebra. An* EΣ-homomorphic relation $\mathcal{R}$ from $\mathbf{C}$ to $\mathbf{A}$ *is a family of binary relations on environments,* $\langle \mathcal{R}_H : H \in \mathit{TENV}(E\Sigma)\rangle$*, such that for each type environment* $H \in \mathit{TENV}(E\Sigma)$*,*

$$\mathcal{R}_H \subseteq \mathit{Env}_H[\mathbf{C}]_\perp \times \mathit{Env}_H[\mathbf{A}]_\perp,$$

and if for each pair of valid H*-environments,* $\eta_{\mathbf{C}}$ *and* $\eta_{\mathbf{A}}$*, the following properties hold:*

substitution: *for each type* $\vec{S}$*, for each type* T*, for each operation symbol* $g \in \mathit{EOPS}$ *such that* $g : \vec{S} \to (T, \texttt{Store})$*, and for all* $\vec{x}$ *such that* $H(\vec{x}) = \vec{S}$*,*

- *for each identifier* y*, if* $g^{\mathbf{C}}(\eta_{\mathbf{C}}\ \vec{x}) \neq \perp$ *and* $g^{\mathbf{A}}(\eta_{\mathbf{A}}\ \vec{x}) \neq \perp$*, and if* $g^{\mathbf{C}}(\eta_{\mathbf{C}}\ \vec{x}) = (r_{\mathbf{C}}, \sigma'_{\mathbf{C}})$ *and* $g^{\mathbf{A}}(\eta_{\mathbf{A}}\ \vec{x}) = (r_{\mathbf{A}}, \sigma'_{\mathbf{A}})$ *then*

  $\eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}}$
  $\Rightarrow$
  $[y \mapsto r_{\mathbf{C}}][\texttt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C}}\ \mathcal{R}_{[y\mapsto T][\texttt{store}\mapsto\texttt{Store}]H}\ [y \mapsto r_{\mathbf{A}}][\texttt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}$

- $(g^{\mathbf{C}}(\eta_{\mathbf{C}}\ \vec{x}) = \perp) \Leftrightarrow (g^{\mathbf{A}}(\eta_{\mathbf{A}}\ \vec{x}) = \perp)$.

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

bistrict: $\perp \mathcal{R}_H \perp$*, and whenever* $\eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}}$ *and either* $\eta_{\mathbf{C}}$ *or* $\eta_{\mathbf{A}}$ *is* $\perp$*, then so is the other.*

bindable: *for each type* $T \neq \texttt{Store}$*, for each identifier* y *such that* $H(y) = T$*, for each* x*,*

$$\eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}} \Rightarrow [x \mapsto (\eta_{\mathbf{C}}\ y)]\eta_{\mathbf{C}}\ \mathcal{R}_{[x\mapsto T]H}\ [x \mapsto (\eta_{\mathbf{A}}\ y)]\eta_{\mathbf{A}}$$

shrinkable: *if* $H' \subseteq H$*,* $\eta'_C \in \mathit{Env}_{H'}[\mathbf{C}]$*, and* $\eta'_A \in \mathit{Env}_{H'}[\mathbf{A}]$ *such that* $\eta'_{\mathbf{C}} \subseteq \eta_{\mathbf{C}}$*,* $\eta'_{\mathbf{A}} \subseteq \eta_{\mathbf{A}}$*, and* $\eta'_{\mathbf{C}}$*,* $\eta'_{\mathbf{A}}$ *are valid* H'*-environments, then*

$$\eta_{\mathbf{A}}\, \mathcal{R}_H\, \eta_{\mathbf{A}} \Rightarrow \eta'_{\mathbf{C}}\, \mathcal{R}_{H'}\, \eta'_{\mathbf{A}}$$

The substitution property of homomorphic relations ensures that operations in related algebras preserve homomorphic relations. That is, this property ensures that method invocations in a programming language will preserve homomorphic relations. For this the results of operations along with the effects on other identifiers should be related. Hence, we relate the resulting environments after pushing the results of the operation into the environment. Since the store is also part of the result we push the store also in the resulting environment and relate the resulting environments at the appropriate type contexts.

The substitution property also ensures that if resulting environment after the invocation of an operation in one environment is $\perp$, then the corresponding invocation in a related environment will also be $\perp$. Note that we require the results of operations in both the related environments to be $\perp$ if one of them is $\perp$. We could allow a method on subtype objects to return $\perp$ in fewer cases than the method on supertype objects. That is, the subtypes are in a sense more defined than the supertypes. But the deterministic models of types we have in this dissertation limits us from allowing such subtypes.

The '*VIS*-identical' property states that in related environments, identifiers of visible types denote the same value. Recall, that all mutation algebras have the same carrier sets for visible types. This property allows us to compare visible results, results that can be printed out in a programming language.

The bistrict property states that two environments are related if and only if both are undefined.

The bindable and shrinkable properties allow homomorphic relations to be preserved while entering and leaving different scopes in a programming language. If the environments are related before an assigning an identifier to another in a programming language, then bindable property ensures that the resulting environments after the assignment are related.

We provide examples of homomorphic relations in later sections.

Aliasing between two locations or identifiers in an environment can be observed through a series of operations. We refer to such aliasing as observable aliasing. We claim that observable aliasing between locations or identifiers is preserved by homomorphic relations. One could prove the claim formally by defining observations and observable aliasing on environments. Since the purpose of this claim is only to help in understanding homomorphic relations, we do not wish to elaborate on this issue and hence, we only provide a discussion to support our claim.

To see that homomorphic relations preserve observable aliasing between stores, let two locations $(\eta_{\mathbf{C}}\ x), (\eta_{\mathbf{C}}\ y)$ be aliases and let the corresponding locations $(\eta_{\mathbf{A}}\ x)$ and $(\eta_{\mathbf{A}}\ y)$ not be aliased. For the sake of contradiction, let the H-environments $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{A}}$ be related. Some observable mutations of $(\eta_{\mathbf{C}}\ x)$ result in a change to y in $\eta_{\mathbf{C}}$ but the same mutations on $(\eta_{\mathbf{A}}\ x)$ do not result in any change to y in $\eta_{\mathbf{A}}$. This violates the substitution property because the resulting environments, after the mutations, should be related because of our assumption that $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{A}}$ are related. Further from our assumption, the mutations are observable, so one can compare the resulting environments to see that y changed in $\eta_{\mathbf{C}}$ and did not in $\eta_{\mathbf{A}}$. Hence, our assumption that aliasing in related environments can be different contradicts the substitution property. Thus homomorphic relations preserve observable aliasing between stores

3.3 Nominal values and nominal environments

For our study of behavioral subtyping, to capture the "behaves like" property for subtype objects, we compare environments with subtyping to environments without any subtyping. That is homomorphic relations should relate environments with subtyping, to environments without any subtyping. In this section, we define exactly what we mean by environments without any subtyping. We refer to such environments as *nominal environments* because all the identifiers and locations denote objects/values of their nominal (static) type.

The purpose of nominal environments is to allow calls only to operations of the static types of the objects. This notion allows comparisons with the runtime calls that depend on the dynamic types of objects. Nominal environments are defined based on a notion of nominality of values. This notion is given by the `isNominal`$[T]$ operations of the corresponding mutation algebra.

Definition 3.3.1 (nominal values) *Let* $(I\Sigma, E\Sigma)$ *be a mutation signature and let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra.*

For $T \in \mathit{ObjectTypes}$*, a* value v is nominal for type T in store σ *if and only if* `isNominal`$[T]^{\mathbf{A}}(v, \sigma)$.

For $T \notin \mathit{ObjectTypes}$*, a* value v is nominal for T in σ *if and only if* $v \in \mathit{VALS}^{\mathbf{A}}_{T}$.

The following discussion motivates the two requirements in the above definition. Values of visible types are always nominal for their respective types. However, for other types there can be non-nominal values. To see this consider a `FrozenAccount` location, $l_{\texttt{FrozenAccount}}$, denoting a sortFor(`PlusAccount`) value in store $\sigma_{\mathbf{E}}$. This is allowed because `PlusAccount` $\leq_{\mathrm{W}}$ `FrozenAccount`. However, if $(\sigma_{\mathbf{E}}\ l_{\texttt{FrozenAccount}}) \notin \mathit{VALS}^{\mathbf{E}}_{\mathrm{sortFor}(\texttt{FrozenAccount})}$, then $l_{\texttt{FrozenAccount}}$ is not nominal for `FrozenAccount` in σ; that is `isNominal[FrozenAccount]`$^{\mathbf{E}}(l_{\texttt{FrozenAccount}}, \sigma)$ is not true. The location $l_{\texttt{FrozenAccount}}$ should be nominal for `FrozenAccount` in store σ only if its abstract value, $(\sigma\ l_{\texttt{FrozenAccount}})$ is in $\mathit{VALS}^{\mathbf{E}}_{\mathrm{sortFor}(\texttt{FrozenAccount})}$.

While defining the `isNominal`$[T]$ for $T \in \mathit{ObjectTypes}$, one should note that that the contained objects should also be nominal. In our example, because `MoneyObj` does not have any subtypes, we do not need specifically define nominality for the contained

`MoneyObj` objects. So all `MoneyObj` values are nominal in any store. However, if there were a subtype of `MoneyObj`, then any $l_{\texttt{FrozenAccount}}$ would be nominal only if its abstract value were of sort sortFor(`FrozenAccount`) and the `MoneyObj` value denoted by the sortFor(`FrozenAccount`) value were of sort sortFor(`MoneyObj`).

For non-object types, a value v is nominal for a type T in a store σ if and only if it is in the carrier set of that type, when the carrier sets are disjoint. That is, subtype values are non-nominal for the supertype in any store unless the carrier sets overlap.

We use this notion of nominal values to define a nominal environment as follows.

Definition 3.3.2 (nominal environment) *Let $(I\Sigma, E\Sigma)$ be a mutation signature and let $\mathbf{A}$ be a $(I\Sigma, E\Sigma)$-mutation algebra. Let H be a type environment and let η be a valid H-environment over $\mathbf{A}$.*

Then η is nominal *if and only if for every $T \in H$, for every $x \in Domain(H)$ such that $H(x) = T$, $(\eta\ x) \in VALS^{\mathbf{A}}_T$ and $(\eta\ x)$ is nominal for type T in $(\eta$ `store`$)$.*

The following argument motivates our notion of nominal environments. Our idea, from a programming language perspective, is to ensure that an operation invocation in nominal environments can be thought of as static invocation. That is because in a nominal environment, the static types of objects and their dynamic types are the same. But in a non-nominal environment, since identifiers/objects can denote values of their subtypes, their dynamic types can be different from their static types, so methods cannot be dispatched statically. So comparing a method invocation in nominal environments and in non-nominal environments, where due to dynamic invocation different methods can execute, allows us to check whether invocations on the subtype objects "behave like" invocation on the supertype objects.

To see this, let H be a type environment and $\mathbf{A}$ be a $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra. Let η_1, η_2 be valid H-states over $\mathbf{A}$. Informally, consider an operation invocation g on the denotations of a set of identifiers $\vec{x}$ such that $H(\vec{x}) = \vec{T}$. Then the actual invocation would be $g^{\mathbf{A}}((\eta_1\ \vec{x}), (\eta_1$ `store`$))$, where g is defined on arguments with types $(\vec{T}$, `Store`$)$. But if $(\eta_1\ \vec{x})$ is not nominal for $\vec{T}$ in $(\eta_1$ `store`$)$, then because of dynamic invocation, the operation g invoked in $\mathbf{A}$ is that of a subtype of tuple of types $\vec{T}$. Hence, we can compare the results of different operation interpretations that correspond to the different method invocations in the nominal and non-nominal environments.

Recall that in the discussion of nominal values we recommend that even the contained objects should be nominal. To see why this is recommended, consider the case when g is implemented by invoking an operation f using the contained objects of its arguments. To compare the invocation of g in the context of supertype and subtype values, we would like even this invocation to be static. That is we would like g to invoke f at the nominal types of the contained objects. This comparison ensures us a way to decide whether methods when invoked with subtypes act as methods in the absence of any subtype values. Hence we require even the contained objects to be nominal. However, since the static reasoning of method invocations is based on nominal environments, if the nominality is defined differently then the conclusions based on nominal environments also change with the notion of nominality.

In summary, nominal environments allow one to create environments as if there were no subtyping in the environment. Hence, we can compare the results of observations with subtyping to observations without subtyping, and can decide subtype objects "behave like" supertype objects. When the carrier sets of two types, particularly related types, overlap, then a value can belong to two different types. But the property of nominality ensures that, given a type, the method invoked on nominal objects can be determined statically.

3.4 Aliasing and nominal environments

While relating certain algebras, there is a conflict between capturing the "behaves like" notion and the nominality of environments. In particular, we are referring to the set of algebras in which the carrier sets of the subtypes and the carrier sets of supertypes are all disjoint. The reason for considering such algebras is two fold. For every set of type specifications, it is not clear whether there exists an algebra in which the carrier sets of subtypes are part of the carrier set of the supertypes. The second reason is often homomorphic relations can be defined on a single algebra providing better intuition on the effects of methods on subtype objects and on supertype objects in that algebra. In this section, we discuss problems with aliasing and nominality in environments over algebras that contain disjoint carrier sets.

The reason for such a conflict is that homomorphic relations, as discussed earlier, should preserve aliasing. To see how the requirement of preserving aliasing restricts the kinds of environments that can be related, consider two locations of different types directly aliased in a store. Also suppose that the carrier sets of these two types are disjoint, which is allowed by mutation algebras. Then any environment consisting of such a store cannot have a related nominal environment. This is because in general it is not possible to both preserve the aliasing between such locations and ensure that abstract values are nominal for their respective types. Hence, in general, combining the requirement "behaves like" and nominal environments results in the problem that there cannot be homomorphic relation between any two algebras that allow all forms of aliasing.

One way to solve this problem is to allow all kinds of aliasing and relax the condition that the reasoning technique requires a nominal environment. This approach results in strong behavioral subtyping that allows all forms of aliasing but allows fewer types to be related.

Another solution to this problem would be to restrict aliasing such that nominal environments are possible. Such a restriction leads to weak behavioral subtyping. The following kinds of aliasing restrictions are required for nominal environments.

- An identifier cannot be a direct alias to a location or another identifier of different type (part (a) of Figure 3.1).
- Two locations of different types cannot be direct aliases (part (b) of Figure 3.1).
- If two locations of the same type are direct aliases in a store then they should denote the same abstract value (part (c) of Figure 3.1).

The first two conditions are necessary because any environment containing such an alias cannot be related to a nominal environment that preserves the aliasing. That is in the right side of Figure 3.1 parts (a) and (b), because the carrier sets are disjoint, it is not possible to have both $x : T$ or $l_1 : T$ to be nominal and still be aliased to $y : S$ and $l_2 : S$ respectively.

The third condition is subtle and seems less motivated than the other two. To see why the third condition is important, consider a partial subtype alias, that is two locations directly alias only to one component of their subtype's abstract value. This is illustrated on the right side of Figure 3.1(c). The abstract values denoted by the two `FrozenAccount` locations $l_1^{\texttt{FrozenAccount}}$ and $l_2^{\texttt{FrozenAccount}}$ are (m_s, m) and (m, m_c) respectively, where m, m_s, and m_c are the three money objects on the left side of Figure 3.1(c). The partial direct alias is due to the sharing of the money object m. If such an aliasing occurs, then in general we cannot have an abstract value that preserves the aliasing between the locations and still is nominal. We cannot coerce $l_1^{\texttt{FrozenAccount}}$ and $l_2^{\texttt{FrozenAccount}}$ to a sortFor(`FrozenAccount`) value while still preserving the partial direct alias.

However, if the direct alias to subtype objects is not partial, then we can lift the subtype values to their supertype values and still preserve the aliasing. This is illustrated on the left hand side of Figure 3.1(c). In this case, $l_1^{\texttt{FrozenAccount}}$ and $l_2^{\texttt{FrozenAccount}}$ denote the same abstract value (m_s, m_c), where m_s and m_c are the two money objects. In this case, we can lift (m_s, m_c) to a nominal value of sort sortFor(`FrozenAccount`), and still preserve the direct aliasing between $l_1^{\texttt{FrozenAccount}}$ and $l_2^{\texttt{FrozenAccount}}$. Then the related environment can have a store where the two locations, $l_1^{\texttt{FrozenAccount}}$ and $l_2^{\texttt{FrozenAccount}}$, are aliased to a sortFor(`FrozenAccount`) value.

The following discussion formalizes the alias restrictions on environments.

First, we define the notion of an *alias type set*, which given a type environment, an algebra, a value v, and a valid environment, η, returns a set consisting of static types of identifiers or locations through which v can be reached in η. The reachability

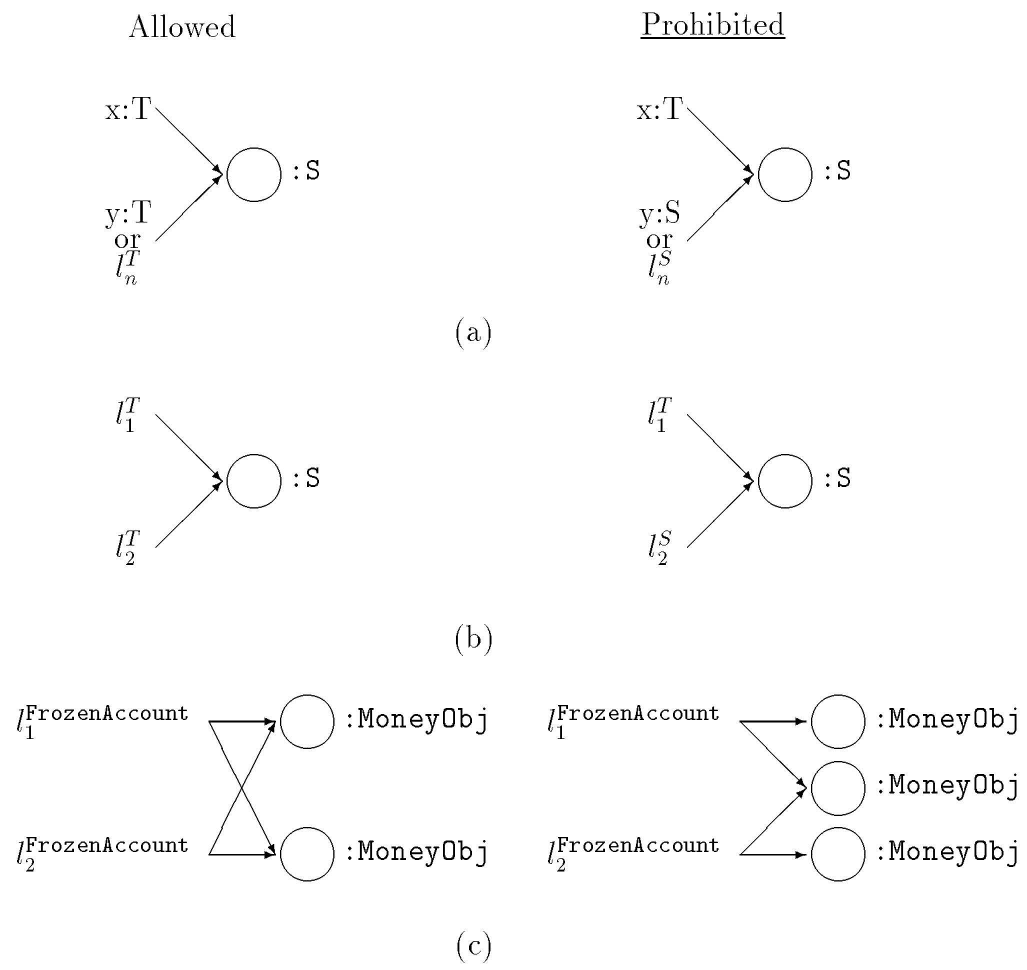

Figure 3.1: A comparison of the kinds of direct aliases that are allowed and are prohibited for weak behavioral subtyping, where $S \leq_{\mathrm{w}} T$ (but $S \neq T$). The notation l_n^T represents a location of type T.

of a location is defined based on the alias graph of a store defined in Section 2.5. Let $\mathbf{A}$ be a $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra and η be a valid H-environment over $\mathbf{A}$. Then a value l is *reachable* from x in a H-environment, η over $\mathbf{A}$ if and only if $x \in \mathit{Domain}(H)$ is such that $(\eta\ x) = l$ or $((\eta\ x), l)$ is a path in $\texttt{aliasG}(\eta\ \texttt{store})$.

$$\mathit{aliasTypeSet}(H, \mathbf{A}, v, \eta) \stackrel{\text{def}}{=} \begin{cases} \{T \mid T \in \mathit{ObjectTypes}, H(x) = T, (\eta\ x) = v\} \\ \cup\ \{T \mid T \in \mathit{ObjectTypes}, l \in \mathit{VALS}^{\mathbf{A}}_{T}, \\ \quad \text{if } l \text{ is reachable from some } x \in \mathit{Domain}(H) \\ \quad \text{such that } (l, v) \text{ is an edge in } \texttt{aliasG}(\eta\ \texttt{store})\} \end{cases}$$

For any value if the alias type set contains more than one type, then the corresponding environment violates the alias restrictions. We use this condition to define environments that satisfy the alias restrictions discussed above.

Definition 3.4.1 (*storeAliasOk*^{W}**,** *stAliasOk*) *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra. Let* $\sigma \in \mathit{VALS}^{\mathbf{A}}_{\texttt{Store}}$*. Then* $\mathit{storeAliasOk}^{W}(\mathbf{A}, \sigma)$ *is true if and only if for each* $U, V \in \mathit{ETYPES}$ *and for each* $l_1 \in \mathit{VALS}^{\mathbf{A}}_{V}$ *and* $l_2 \in \mathit{VALS}^{\mathbf{A}}_{U}$ *such that* $l_1, l_2 \in \mathit{Domain}(\sigma)$*, if* l_1 *and* l_2 *are direct aliases in* σ *then* $U = V$ *and* $(\sigma\ l_1) = (\sigma\ l_2)$*.*

Let H *be a type environment such that* $\texttt{store} \in \mathit{Domain}(H)$*, and let* η *be a valid* H*-environment. Then* $\mathit{stAliasOk}(H, \mathbf{A}, \eta)$ *is true if and only if for every* $x : T \in H$*,* $\mathit{aliasTypeSet}(H, \mathbf{A}, (\eta\ x)) = \{T\}$ *and* $\mathit{storeAliasOk}^{W}(\mathbf{A}, (\eta\ \texttt{store}))$*.*

All other kinds of aliasing, that is two identifiers of the same type denoting the same location, two locations of the same type denoting the same location (nominal or non-nominal), and (partial) direct aliases to a value of its nominal type, allow a related nominal environment. Figure 3.1 compares the kinds of aliasing allowed to the kinds of aliasing restricted for a sound notion of weak behavioral subtyping.

3.5 Simulation relations

Recall that homomorphic relation defined in Section 3.2 capture the "behaves like" notion for objects. But the intuitive idea for behavioral subtyping is that subtype objects should behave like supertype objects. The following definition extends homomorphic relations to capture this notion.

Definition 3.5.1 (simulation relations) *Let* $\mathbf{A}$ *be a* $(I\Sigma', E\Sigma)$*-mutation algebra and* $\mathbf{C}$ *be a* $(I\Sigma'', E\Sigma)$*-mutation algebra. An* $\mathrm{E}\Sigma$-simulation relation $\mathcal{R}$ from $\mathbf{C}$ to $\mathbf{A}$ *is a homomorphic relation from* $\mathbf{C}$ *to* $\mathbf{A}$ *such that the following condition is satisfied:*

coercion: *if* $\mathit{stAliasOk}(H, \mathbf{C}, \eta_{\mathbf{C}})$ *then there exists a nominal* H*-environment over* $\mathbf{A}$*,* $\eta_{\mathbf{A}}$*, such that* $\eta_{\mathbf{C}}\ \mathcal{R}_H\ \eta_{\mathbf{A}}$*. Otherwise, there exists a valid* H*-environment,* $\eta'_{\mathbf{A}}$*, over* $\mathbf{A}$ *such that* $\eta_{\mathbf{C}}\ \mathcal{R}_H\ \eta'_{\mathbf{A}}$*.*

If the aliasing constraints are satisfied then the “coercion” property ensures that there is a related environment where all the subtype objects are coerced into supertype objects. The homomorphic relation between such environments ensure that subtype objects “behave like” supertype objects.

Since the substitution property allows comparisons between method invocations on subtype objects and supertype objects, one can have a case when the subtype operation behaves differently than a supertype operation. In such cases, the identity relation on environments is not a simulation. The following example illustrates that not every identity is a simulation relation.

Example 3.5.2 *Let $E\Sigma^{\mathbf{F}}$ be an external signature that contains the visible types, the* `Store`*,* `BankAccount`*, and a* `ChargeAccount`*. The type* `ChargeAccount` *is similar to* `BankAccount` *except for the behavior of the* `balance` *method. The* `ChargeAccount`*'s* `balance` *method charges a transaction fee, which is deducted from its balance and the deducted balance is returned. Let* `ChargeAccount` *be declared as a subtype of* `BankAccount`*.*

Let $\mathbf{F}$ be a $(E\Sigma^{\mathbf{F}}, I\Sigma^{\mathbf{F}})$-mutation algebra. Let the carrier sets of `BankAccount` *and* `ChargeAccount` *be $(0, m_1, m_2)$ and $(1, m_1, m_2)$ respectively, where m_1 and m_2 are two* `MoneyObj` *objects. The tag 0 or 1 is used to keep the carrier sets of* `BankAccount` *and* `ChargeAccount` *disjoint.*

Let $\mathcal{R}^{\mathbf{F}}$ be an identity relation on environments. Let H be a type context such that $Domain(H) = \{x, \texttt{store}\}$ and $H(x) = \texttt{BankAccount}$. Let η_1 be a valid environment such that $stAliasOk(H, \mathbf{F}, \eta_1)$ and $(\eta_1\ x) \in VALS^{\mathbf{F}}_{\texttt{ChargeAccount}}$.

If $\mathcal{R}^{\mathbf{F}}$ is a simulation relation, there exists a nominal state, η_2, such that $\eta_1 \mathcal{R}^{\mathbf{F}}_H \eta_2$. But, since $\mathcal{R}^{\mathbf{F}}$ is an identity relation on environments, $\eta_1 = \eta_2$. Hence, $(\eta_2\ x)$ denotes a `ChargeAccount` *object, so η_2 is not nominal. So $\mathcal{R}^{\mathbf{F}}$ is not a simulation relation.*∎

The following discussion shows that even adding a coercion, from subtype objects to supertype objects, to the identity relation on environments is not a simulation relation. That is, we show that even an identity relation that is extended to satisfy the coercion property is not a simulation relation.

Suppose the relation, $\mathcal{R}^{\mathbf{F}}$, is a simulation with a coercion that converts non-nominal `ChargeAccount` values to `BankAccount` by changing the tag from 1 to 0. Then there exists a η_2 such that η_2 is nominal and $\eta_1\ \mathcal{R}_H\ \eta_2$.

Applying the substitution property for `balance` with arguments $(x, \texttt{Store})$, for some y we have

$$[y \mapsto v_1][\texttt{store} \mapsto \sigma'_1]\eta_1\ \mathcal{R}^{\mathbf{F}}_{[y \mapsto \texttt{Integer}]\texttt{storeStore}H}\ [y \mapsto v_2][\texttt{store} \mapsto \sigma'_2]\eta_2.$$

From the *VIS*-identical property, we have $v_1 = v_2$.

Since, η_2 is a nominal environment, $(\eta_2\ x) \in VALS^{\mathbf{F}}_{\texttt{BankAccount}}$. So the operation `balance` does not charge any transaction fee and hence the amount returned v_2 is different from v_1 returned by `balance` for a `ChargeAccount`. But this contradicts *VIS*-identical property. Hence, our assumption that the identity relation, $\mathcal{R}^{\mathbf{F}}$, extended to satisfy the coercion property is a simulation relation is false.

3.6 Weak behavioral subtypes

Weak behavioral subtyping is defined on the denotations of specifications of types, which are sets of algebras. The existence of a simulation relation between an algebra and itself ensures that objects of subtype behave like objects of supertype. This conclusion is based on the substitution property and the coercion property, which relate the effects of operations on subtype and supertype objects. Hence, the existence of a simulation relation between an algebra and itself is sufficient to ensure that subtype objects act like supertype objects. But we give a more general definition for behavioral subtyping that permits relationships between incompletely specified data types to be supertypes of more completely specified types. For example, `Bag`, with an under specified `get` operation, can be a supertype of `Stack`, with a completely-specified `get`. The `get` operation of a `Stack` is more defined than a `Bag` in the sense that it gets the last inserted element. However, to permit more defined subtypes, we need to consider a set of algebras that satisfies the denotations of specifications of types. Such a set allows one to find at least one implementation of the incompletely-specified type that is simulated by a more completely specified subtype. That is, the implementation of `Bag` with a `get` operation that selects the last inserted item is simulated by objects of `Stack`.

The set of all implementations of a given set of type specifications is referred to as *SPEC*. Weak behavioral subtyping is defined with respect to *SPEC*.

Definition 3.6.1 (weak behavioral subtyping) *Let $E\Sigma$ be a signature and let SPEC be a set of $E\Sigma$-algebras, such that each $\mathbf{A}$ in SPEC is a $(I\Sigma, E\Sigma)$-mutation algebra for some $I\Sigma$. The presumed subtype relation $\leq$ of $E\Sigma$ 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 $E\Sigma$-simulation relation, $\mathcal{R}$, from $\mathbf{B}$ to $\mathbf{A}$.*

Note that the only property of simulation relations that deals directly with the operations is the substitution property, which compares the effects of operations on subtype and supertype objects. Simulations does not place any constraints on the extra operations on the subtypes. Hence, we can have types with extra mutators, that are subtypes of immutable types. That is, `PlusAccount` can be a weak behavioral subtype of `FrozenAccount` (unlike the case for strong behavioral subtyping).

For a reasoning technique based on weak behavioral subtyping to be practical, each program state should satisfy *stAliasOk*. If that were not the case then for some algebras the environment at any give state of the program could not be related to a nominal environment because it would violate aliasing restrictions. In such cases, modular reasoning using supertype abstraction would be unsound. We show that the semantics of $OBS^{\leq}$ given in the next chapter allows reasoning with weak behavioral subtyping by constructing only states that satisfy *stAliasOk*.

3.7 Example of a weak behavioral subtype relation

In this section we show that the presumed subtype relation, $\leq_{\mathrm{W}}$, given in Figure 2.4 is a weak behavioral subtype relation for $\{\mathbf{E}\}$. To show this, we first define a relation, R^w between algebra $\mathbf{E}$ (given in Figures 2.5 and 2.6) and itself. Then we prove that R^w is a simulation relation. This proof is tedious but is constructive. In addition, we believe that this kind of proof might work in general for other mutation algebras with different external signatures.

We define a function that captures similarity between abstract values, and an environment homomorphism that maps environments with similar aliasing. Using these two notions, we define a relation that preserves both the similarity between the abstract values of identifiers and the aliasing in related environments. We show that that this relation is a simulation relation.

We first define an abstraction function A_T for each $T \in \mathit{ObjectTypes}^{\mathrm{E}\Sigma^{\mathbf{E}}}$ that maps the abstract values of various object types to an integer. The type of each A_T is:

$$A_T : (\,\widehat{VALS^{\mathbf{E}}_T} \times VALS^{E}_{\texttt{Store}}) \to \texttt{Integer}_{\perp}$$

and the particular versions are defined as follows:

$$\begin{array}{lcl}
A_{\texttt{MoneyObj}}(v,\sigma) & = & (\sigma\ v) \\
A_{\texttt{FrozenAccount}}(v,\sigma) & = & \textbf{if } (\sigma\ v) \in VALS^{\mathbf{E}}_{\mathrm{sortFor}(\texttt{FrozenAccount})} \\
 & & \textbf{then } A_{\texttt{MoneyObj}}(\sigma\ v) \\
 & & \textbf{else let } (m_s, m_c) = (\sigma\ v) \textbf{ in} \\
 & & \quad A_{\texttt{MoneyObj}}(\sigma\ m_s) + A_{\texttt{MoneyObj}}(\sigma\ m_c) \\
A_{\texttt{BankAccount}}(v,\sigma) & = & \textbf{if } (\sigma\ v) \in VALS^{\mathbf{E}}_{\mathrm{sortFor}(\texttt{BankAccount})} \textbf{ then } A_{\texttt{MoneyObj}}(\sigma\ v) \\
 & & \textbf{else let } (m_s, m_c) = (\sigma\ v) \textbf{ in} \\
 & & \quad A_{\texttt{MoneyObj}}(\sigma\ m_s) + A_{\texttt{MoneyObj}}(\sigma\ m_c) \\
A_{\texttt{PlusAccount}}(v,\sigma) & = & \textbf{let } (m_s, m_c) = (\sigma\ v) \textbf{ in} \\
 & & (A_{\texttt{MoneyObj}}(\sigma\ m_s) + A_{\texttt{MoneyObj}}(\sigma\ m_c))
\end{array}$$

We define the notion of similarity, S^w, between abstract values of two types of $\mathbf{E}$ and use that notion in defining simulation relations. For each $T \in (\mathrm{E}\mathit{TYPES} \Leftrightarrow$

$\{\texttt{Store}\}$),

$$\mathrm{S}^w{}_T : (\mathit{VALS}^{\mathbf{E}}_{\texttt{Store}} \times \mathit{VALS}^{\mathbf{E}}_{\texttt{Store}}) \to (\widehat{\mathit{VALS}}^{\mathbf{E}} \times \widehat{\mathit{VALS}}^{\mathbf{E}}) \to \texttt{Bool}_\perp$$

is a predicate that tests the similarity of two values in their respective stores. It is defined as follows. Let $\sigma_1, \sigma_2 \in \mathit{VALS}^{\mathbf{E}}_{\texttt{Store}}$.

For all $T \in (\mathit{VIS} \cup \{\texttt{Void}\})$ and for each $v_1, v_2 \in \mathit{VALS}^{\mathbf{E}}_{\mathrm{sortFor}(T)}$:

$$\mathrm{S}^w{}_T(\sigma_1, \sigma_2)(v_1, v_2) \overset{\mathrm{def}}{=} (v_1 = v_2) \tag{3.1}$$

We extend S^w pointwise to tuples. That is, $\vec{U}, \vec{V} \in \mathit{TYPES}$ such that $\vec{U} \leq_{\mathrm{W}} \vec{V}$, for each pair $\sigma_1, \sigma_2 \in \mathit{VALS}^{\mathbf{E}}_{\texttt{Store}}$, such that for each $\vec{u} \in \mathit{VALS}^{\mathbf{E}}_{\vec{U}}$ and $\vec{v} \in \mathit{VALS}^{\mathbf{E}}_{\vec{V}}$, $\vec{u}\ \mathrm{S}^w{}_{\vec{V}}\ \vec{v}$ if and only if

$$(|\ \vec{u}\ |=|\ \vec{v}\ |) \wedge \bigwedge_{(1 \leq i \leq |\vec{u}|)} \mathrm{S}^w{}_{V_i}(\sigma_1, \sigma_2)(u_i, v_i) \tag{3.2}$$

For all $S, T \in \{$ `MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount` $\}$ and for each $v_1 \in \widehat{\mathit{VALS}}^{\mathbf{E}}_S$ and $v_2 \in \widehat{\mathit{VALS}}^{\mathbf{E}}_T$ such that $S \neq \texttt{PlusAccount} \vee T \neq \texttt{PlusAccount}$:

$$\mathrm{S}^w{}_T(\sigma_1, \sigma_2)(v_1, v_2) \overset{\mathrm{def}}{=} A_T(v_1, \sigma_1) = A_T(v_2, \sigma_2) \tag{3.3}$$

For $T \in \{\texttt{PlusAccount}\}$, for each $v_1, v_2 \in \widehat{\mathit{VALS}}^{\mathbf{E}}_T$, if $(\sigma_1\ v_1) = (v_1^s, v_1^c)$ and $(\sigma_2\ v_2) = (v_2^s, v_2^c)$ then:

$$\begin{aligned}\mathrm{S}^w{}_T(\sigma_1, \sigma_2)(v_1, v_2) &= (A_{\texttt{MoneyObj}}(v_1^s, \sigma_1) = A_{\texttt{MoneyObj}}(v_2^s, \sigma_2)) \\ &\quad \wedge (A_{\texttt{MoneyObj}}(v_1^c, \sigma_1) = A_{\texttt{MoneyObj}}(v_2^c, \sigma_2))\end{aligned} \tag{3.4}$$

A coercion relation on abstract values is not enough because it does not preserve aliasing between related environments. We define an *environment homomorphism* that preserves the aliasing between related environments. While coercing environments we also need a coercion between subtype values that are locations to supertype values. Since coercion of locations require knowledge of the particular carrier sets of an algebra, we define environment homomorphism specific to a particular algebra, which in our case is $\mathbf{E}$. The aliasing between locations is preserved by a graph homomorphism between the alias graphs of the corresponding stores. We require variables also to have the same aliasing in the environments.

Definition 3.7.1 (environment-homomorphism) *Let* $\mathbf{E}$ *be a* $(E\Sigma, I\Sigma')$*-mutation algebra given in Figure 2.5 and Figure 2.6. Let* H *be a* $E\Sigma$ *type environment and*

$\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$ be valid H-environments over $\mathbf{E}$. Suppose that ($\eta_{\mathbf{E}}$ `store`) $= \sigma_{\mathbf{E}}$ and ($\eta'_{\mathbf{E}}$ `store`) $= \sigma'_{\mathbf{E}}$.

Then (h_v, h_n) is an environment homomorphism on $\mathbf{E}$ *between $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$ if and only if (h_v, h_n) is a graph homomorphism between* `aliasG`$(\sigma_{\mathbf{E}})$ *and* `aliasG`$(\sigma'_{\mathbf{E}})$ *and the following conditions are satisfied*

- *for each $T \in \mathit{ObjectTypes}$, if $x : T \in \mathit{Domain}(H)$, $h_v(\eta_{\mathbf{E}}\ x) = (\eta'_{\mathbf{E}}\ x)$,*
- *if $\mathit{stAliasOk}(H, \mathbf{A}, \eta_{\mathbf{E}})$ is false, then (h_v, h_n) is an injective graph homomorphism.*
- *if $\mathit{stAliasOk}(H, \mathbf{A}, \eta_{\mathbf{E}})$ is true, then for each $l_1, l_2 \in \mathit{Domain}(\sigma_{\mathbf{E}})$, $h_v(l_1) = h_v(l_2)$ if and only if $l_1 = l_2$ or there exists a l such that $(\sigma_{\mathbf{E}}\ l) = (l_1, l_2)$ and $h(l) \in \mathit{VALS}^{\mathbf{A}}_T$, where $T \in \{$ `BankAccount`, `FrozenAccount` $\}$.*

The first condition ensures that the denotations of identifiers are preserved in the graph homomorphism. The second and third conditions ensure that aliasing is preserved across related environments and that coercing of locations can be done only if the environment, $\eta_{\mathbf{E}}$ satisfies alias restrictions.

Since the coercion of locations depends on algebras, this notion of environment homomorphism, the last condition, is different for different algebras. But, one can define a environment homomorphism in similar lines by defining the coercion between locations as part of the last condition.

The following relation, R^w, between $\mathbf{E}$ and itself ties together the abstraction function and the environment homomorphism on $\mathbf{E}$.

Definition 3.7.2 ($\mathbf{R}^w$) *The relation R^w from $\mathbf{E}$ to $\mathbf{E}$ is a family of binary relations on environments, $\langle {R^w}_H : H \in \mathit{TENV}(E\Sigma^{\mathbf{E}})\rangle$, such that for each any type environment $H \in \mathit{TENV}(E\Sigma^{\mathbf{E}})$, ${R^w}_H \subseteq \mathit{Env}_H[\mathbf{E}]_\perp \times \mathit{Env}_H[\mathbf{E}]_\perp$, $\perp {R^w}_H \perp$ and for proper $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$, $\eta_{\mathbf{E}}\ {R^w}_H\ \eta'_{\mathbf{E}}$ if and only if the following conditions all hold:*

- *$\eta_{\mathbf{E}}, \eta'_{\mathbf{E}}$ are valid H-environments over $\mathbf{E}$,*
- *for each type $T \in (\mathit{ETYPES} \Leftrightarrow \{$`Store`$\})$, for each x such that $H(x) = T$, ${S^w}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ x), (\eta'_{\mathbf{E}}\ x))$,*
- *there exists an environment homomorphism on $\mathbf{E}$ between $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$.*

We claim that R^w is a $\mathrm{E}\Sigma^{\mathbf{E}}$-simulation relation between $\mathbf{E}$ and itself.

Proposition 3.7.3 *R^w is a $E\Sigma^{\mathbf{E}}$-simulation relation between $\mathbf{E}$ and itself.*

Proof: Let H be a type environment such that `store` $\in \mathit{Domain}(H)$. Let $\eta_{\mathbf{E}}, \eta'_{\mathbf{E}}$ be valid H-environments over $\mathbf{E}$.

substitution: To show that R^w satisfies the substitution property we must show that all the external operations of $(\mathrm{I}\Sigma^{\mathbf{E}}, \mathrm{E}\Sigma^{\mathbf{E}})$ satisfy the substitution property. The following example shows how the proof goes for one operation, `withdraw`, and the rest of the operations can be shown in a similar way.

To show the substitution property for `withdraw`, let

$$\begin{aligned} \vec{S} &= (\texttt{BankAccount}, \texttt{MoneyObj}, \texttt{Store}) \\ T &= \texttt{Void} \\ g &= \texttt{withdraw}. \end{aligned}$$

Suppose that $\vec{x} = (x_1, x_2, \texttt{store})$ and $H(x_1,\, x_2,\, \texttt{Store}) = \vec{S}$.

Suppose the result of $\texttt{withdraw}^{\mathbf{E}}((\eta_{\mathbf{E}}\ \vec{x}))$ is $\perp$. This can happen only in two cases. The first case is when $(\eta_{\mathbf{E}}\ x_i)$ is not in the domain of $\sigma_{\mathbf{E}}$. This cannot happen because $\eta_{\mathbf{E}}$ is a valid H-environment.

The other case is when the amount in $(\eta_{\mathbf{E}}\ x_1)$ is less than the amount in $(\eta_{\mathbf{E}}\ x_2)$. In such a case, because of $SW_{\texttt{BankAccount}}(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ x_1), (\eta'_{\mathbf{E}}\ x_1))$ and $SW_{\texttt{MoneyObj}}(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ x_2), (\eta'_{\mathbf{E}}\ x_2))$, the amount in $(\eta'_{\mathbf{E}}\ x_1)$ would be less than the amount in $(\eta'_{\mathbf{E}}\ x_2)$. Hence the result of $\texttt{withdraw}^{\mathbf{E}}((\eta_{\mathbf{E}}\ \vec{x}))$ would be $\perp$.

Similarly, we can show that when the result of $\texttt{withdraw}^{\mathbf{E}}((\eta'_{\mathbf{E}}\ \vec{x}))$ is $\perp$, so is the result of $\texttt{withdraw}^{\mathbf{E}}((\eta_{\mathbf{E}}\ \vec{x}))$.

If the result is not $\perp$ then, we need to show that for each identifier y, if $(*, \sigma_r) = \texttt{withdraw}^{\mathbf{E}}(\eta_{\mathbf{E}}\ \vec{x})$ and $(*, \sigma'_r) = \texttt{withdraw}^{\mathbf{E}}(\eta'_{\mathbf{E}}\ \vec{x})$ then

$$\begin{aligned} &\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta'_{\mathbf{E}} \\ &\Rightarrow \\ &([y \mapsto *][\texttt{store} \mapsto \sigma_r]\eta_{\mathbf{E}}\ \mathrm{R}^w{}_{[y \mapsto \texttt{Void}][\texttt{store} \mapsto \texttt{Store}]H}\ [y \mapsto *][\texttt{store} \mapsto \sigma'_r]\eta'_{\mathbf{E}}) \end{aligned}$$

The trivial case is when both $g^{\mathbf{E}}(\eta_{\mathbf{E}}\ \vec{x})$ and $g^{\mathbf{E}}(\eta'_{\mathbf{E}}\ \vec{x})$ have the same interpretation, that is they execute the same branch in the meaning of $g^{\mathbf{E}}$. The tricky case is when the two operations have different interpretations. This happens when $\eta_{\mathbf{E}}$ satisfies *stAliasOk* and x_1 : `BankAccount` denotes a location in the carrier set of `PlusAccount` in $\eta_{\mathbf{E}}$ and a location in the carrier set of `BankAccount` in $\eta'_{\mathbf{E}}$. We consider only this case in detail.

Let $(\eta_{\mathbf{E}}\ x) = l_1$, $(\eta'_{\mathbf{E}}\ x) = l_2$, $(\eta_{\mathbf{E}}\ y) = m_1$, and $(\eta'_{\mathbf{E}}\ y) = m_2$, where $l_1 \in \mathit{VALS}^{\mathbf{E}}_{\texttt{PlusAccount}}$, $l_2 \in \mathit{VALS}^{\mathbf{E}}_{\texttt{BankAccount}}$, and $m_1, m_2 \in \mathit{VALS}^{\mathbf{E}}_{\texttt{MoneyObj}}$.

Let $\eta^r_{\mathbf{E}}$ and $\eta'^r_{\mathbf{E}}$ be the resulting environments. That is,

$$\begin{aligned} \eta^r_{\mathbf{E}} &= [y \mapsto *][\texttt{store} \mapsto \sigma_r]\eta_{\mathbf{E}} \\ \eta'^r_{\mathbf{E}} &= [y \mapsto *][\texttt{store} \mapsto \sigma'_r]\eta'_{\mathbf{E}}. \end{aligned}$$

Further, suppose that

$$\eta_{\mathbf{E}} \; \mathrm{R}^w{}_H \; \eta'_{\mathbf{E}} \tag{3.5}$$

Then we need to show that

$$(\eta^r_{\mathbf{E}} \; \mathrm{R}^w{}_{[y \mapsto \mathtt{Void}][\mathtt{store} \mapsto \mathtt{Store}]H} \; \eta'^r_{\mathbf{E}}) \tag{3.6}$$

Consider the results of calling `withdraw` with $(l_1, m_1, \sigma_{\mathbf{E}})$ and with $(l_2, m_2, \sigma'_{\mathbf{E}})$ in $\mathbf{E}$. The corresponding operation interpretations of `withdraw` are given in Figure 2.6. Recall from the discussion in Section 2.4, that the two meanings of `withdraw` given in Figure 2.6 correspond to the different cases depending on the first argument. Hence, depending on the type of l_1 and l_2 we take the corresponding meaning for `withdraw`.

The result of $\mathtt{withdraw}^{\mathbf{E}}(l_1, m_1, \sigma_{\mathbf{E}})$ can be either $\perp$, or a pair of $*$ and a store. The case when the result is $\perp$ is already discussed.

The remaining case is when both the results are proper. We prove Equation 3.6 by showing that $\eta^r_{\mathbf{E}}$ and $\eta'^r_{\mathbf{E}}$ satisfy all the properties of R^w. For convenience, let $\sigma^r_{\mathbf{E}} = [m_s \mapsto (v_s \Leftrightarrow v_1)]\sigma_{\mathbf{E}}$ and $\sigma'^r_{\mathbf{E}} = [m' \mapsto (v_b \Leftrightarrow v_2)]\sigma_{\mathbf{E}}$ be the resulting stores corresponding to the operation interpretations in Figure 2.6. The values v_s corresponds to the amount in the savings component of l_1 in $\eta_{\mathbf{E}}$ and v_b corresponds to the balance of l_2 in $\eta'_{\mathbf{E}}$. The values v_1 and v_2 denote amount in m_1 and m_2 in $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$ respectively.

- From the hypothesis and the construction, the environments $\eta^r_{\mathbf{E}}$ and $\eta'^r_{\mathbf{E}}$ are valid $[r \mapsto \mathtt{Void}][\mathtt{store} \mapsto \mathtt{Store}]H$-environments over $\mathbf{E}$.
- To show that the abstract values for each identifier are similar in the final states, let T be a type and let $z : T$ be an identifier in $[r \mapsto \mathtt{Void}][\mathtt{store} \mapsto \mathtt{Store}]H$. We do this by cases.
  - Suppose that there is no path from $(\eta_{\mathbf{E}}\ z)$ to m_s in $\mathtt{aliasG}(\sigma_{\mathbf{E}})$, then the abstract value of z is unchanged in $\eta_{\mathbf{E}}$.
    From the hypothesis, we can conclude that

    $$\mathrm{S}^w{}_T(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})((\eta^r_{\mathbf{E}}\ z), (\eta'^r_{\mathbf{E}}\ z))$$

  - If $z = x$, we need to show $\mathrm{S}^w{}_{\mathtt{BankAccount}}(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})((\eta^r_{\mathbf{E}}\ x), (\eta'^r_{\mathbf{E}}\ x))$. We calculate from the desired formula backwards as follows.

    $$\begin{array}{ll} & \mathrm{S}^w{}_{\mathtt{BankAccount}}(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})((\eta^r_{\mathbf{E}}\ x), (\eta'^r_{\mathbf{E}}\ x)) \\ = & \langle \text{by dereferencing } x \text{ in } \eta^r_{\mathbf{E}} \text{ and } \eta'^r_{\mathbf{E}} \rangle \\ & \mathrm{S}^w{}_{\mathtt{BankAccount}}(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})(l_1, l_2) \end{array}$$

$=$ $\langle$by the definition of $\mathrm{S}^w{}_{\mathtt{BankAccount}}\rangle$
$\mathrm{S}^w{}_{\mathtt{Integer}}(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})((\sigma^r_{\mathbf{E}}\ m_s) + (\sigma^r_{\mathbf{E}}\ m_c), (\sigma'^r_{\mathbf{E}}\ m'))$

But from the hypothesis, we have $S_{\mathtt{BankAccount}}(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})(l_1, l_2)$. That is, $(\sigma_{\mathbf{E}}\ m_s) + (\sigma_{\mathbf{E}}\ m_c) = (\sigma'_{\mathbf{E}}\ m')$, where (m_s, m_c) and m' are the values denoted by l_1 and l_2 in respective stores. From the semantics of **E**, we know that the amount $(\sigma_{\mathbf{E}}\ m_1)$ is subtracted from m_c and the amount $(\sigma'_{\mathbf{E}}\ m_2)$ is subtracted from m' in the new stores σ_r and σ'_r. But from equation 3.5, $(\sigma_{\mathbf{E}}\ m_1) = (\sigma'_{\mathbf{E}}\ m_2)$. Hence, we can conclude that the relation $\mathrm{S}^w{}_{\mathtt{BankAccount}}(\sigma^r_{\mathbf{E}}, \sigma'^r_{\mathbf{E}})((\eta^r_{\mathbf{E}}\ x), (\eta'^r_{\mathbf{E}}\ x))$ is true.

– Otherwise z is not x but one of the contained objects of x, say m_s, is reachable along some path of `aliasG`$(\sigma_{\mathbf{E}})$ from $(\eta_{\mathbf{E}}\ z)$. From *stAliasOk*$(H, \mathbf{A}, \eta_{\mathbf{E}})$, the alias should be a complete one, that is,
$(\sigma_{\mathbf{E}}\ (\eta_{\mathbf{E}}\ z)) = (\sigma_{\mathbf{E}}\ (\eta_{\mathbf{E}}\ x)) = (m_s, m_c)$.
From the hypothesis, we have a (h_v, h_e)-environment homomorphism such that $h_v(\eta_{\mathbf{E}}\ z) = (\eta'_{\mathbf{E}}\ z)$ and $h_v(\eta_{\mathbf{E}}\ x) = (\eta'_{\mathbf{E}}\ x)$. Since (h_v, h_e) is a graph homomorphism, $h_e((\eta_{\mathbf{E}}\ z), m_s) = (h_v(\eta_{\mathbf{E}}\ z), h_v(m_s))$ and $h_e((\eta_{\mathbf{E}}\ z), m_c) = (h_v(\eta_{\mathbf{E}}\ z), h_v(m_c))$.
Since $(\eta_{\mathbf{E}}\ x) \in \mathit{VALS}^{\mathbf{E}}_{\mathtt{PlusAccount}}$ and $(\eta'_{\mathbf{E}}\ x) \in \mathit{VALS}^{\mathbf{E}}_{\mathtt{BankAccount}}$, from the definition of environment homomorphism on **E** we have, $h_v(m_s) = m'$ and $h_v(m_c) = m'$.
But $h_v(\eta_{\mathbf{E}}\ z) = (\eta'_{\mathbf{E}}\ z)$ and $(h_v(\eta_{\mathbf{E}}\ z), h_v(m_s)), (h_v(\eta_{\mathbf{E}}\ z), h_v(m_c))$ are edges of `aliasG`$(\sigma'_{\mathbf{E}})$, so we have $h_v(m_s) = m'$ and $h_v(m_c) = m'$. That is $(\sigma'_{\mathbf{E}}\ (\eta'_{\mathbf{E}}\ z)) = (\sigma'_{\mathbf{E}}\ (\eta'_{\mathbf{E}}\ x)) = m'$.
Since z denotes the same abstract value as x, as shown in the previous case, the abstract values of z are similar.

- We define a new environment homomorphism on **E**, (h'_v, h'_e) between $\eta^r_{\mathbf{E}}$ and $\eta'^r_{\mathbf{E}}$ such that $h'_v = h_v$ and $h'_e = h_e$. Because only the mapping of a `MoneyObj` object, which is not a part of the alias graph of the corresponding stores, is changed between σ_r and $\sigma^r_{\mathbf{E}}$, we can conclude that (h_v, h_e) is the required environment homomorphism on **E**.

The other case when `withdraw` has a proper value and $v_s < v'$ can be shown similarly.

coercion: If *stAliasOk* is false, then we can define $\eta'_{\mathbf{E}} = \eta_{\mathbf{E}}$. Because the relation R^w is reflexive with an environment homomorphism that is the identity on *Domain*(h_v), we can conclude $\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta'_{\mathbf{E}}$.

If *stAliasOk* is true, then we construct a nominal H-environment $\eta^n_{\mathbf{E}}$ such that $\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta^n_{\mathbf{E}}$.

We construct an environment $\eta^n_{\mathbf{E}}$ that is nominal and is related to $\eta_{\mathbf{E}}$. We first construct a nominal store, $\sigma^n_{\mathbf{E}}$ and then map the corresponding locations to identifiers for a nominal environment. Let $\sigma_{\mathbf{E}} = (\eta_{\mathbf{E}}\ \texttt{store})$.

For all $l \in \mathit{VALS}^{\mathbf{A}}_T$, if l is nominal for T in $\sigma_{\mathbf{E}}$ then $v \in \mathit{Domain}(\sigma^n_{\mathbf{E}})$, $(\sigma^n_{\mathbf{E}}\ v) = (\sigma_{\mathbf{E}}\ v)$ and for each $v_i \in \texttt{containedObjs}[\mathrm{T}](v, \sigma_{\mathbf{E}})$ $h_v(v_i) = v_i$, and $h_e(v, v_i) = (v, v_i)$.

For all $l \in \mathit{VALS}^{\mathbf{A}}_T$, if l is not nominal for T in $\sigma_{\mathbf{E}}$, then $T \in \{\texttt{FrozenAccount}, \texttt{BankAccount}\}$ and $(\sigma_{\mathbf{E}}\ l) \in \mathit{VALS}^{\mathbf{E}}_{\texttt{PlusAccount}}$. Let $(\sigma_{\mathbf{E}}\ l) = (m_s, m_c)$. We allocate a new `MoneyObj`, m, such that $(\sigma^n_{\mathbf{E}}\ m) = (\sigma_{\mathbf{E}}\ m_s) + (\sigma_{\mathbf{E}}\ m_c)$, and $(\sigma^n_{\mathbf{E}}\ l) = m$. Also, $h_v(l) = l$, $h_v(m_s) = h_v(m_c) = m$, and $h_e(l, m_s) = h_e(l, m_c) = (l, m)$. According to this construction, if l' is a direct alias of l in $\sigma_{\mathbf{E}}$, then $h_v(l')$ will be a direct alias of $h(l)$ in $\sigma^n_{\mathbf{E}}$.

We define the store of $\eta^n_{\mathbf{E}}$ to be the nominal store, $\sigma^n_{\mathbf{E}}$, constructed above, that is, $(\eta^n_{\mathbf{E}}\ \texttt{store}) = \sigma^n_{\mathbf{E}}$.

For all $T \in \mathit{TYPES} \Leftrightarrow \{\texttt{Store}\}$, $x : T \in H$, if $(\eta_{\mathbf{E}}\ x)$ is nominal for T in $(\sigma_{\mathbf{E}})$ then $(\eta^n_{\mathbf{E}}\ x) = (\eta_{\mathbf{E}}\ x)$ and $h_v(\eta_{\mathbf{E}}\ x) = (\eta^n_{\mathbf{E}}\ x)$.

For all $T \in \{\texttt{BankAccount}, \texttt{FrozenAccount}\}$, and for all $x : T \in H$, if $(\eta_{\mathbf{E}}\ x) \in \mathit{VALS}^{\mathbf{E}}_{\texttt{PlusAccount}}$ and $(\sigma_{\mathbf{E}}\ (\eta_{\mathbf{E}}\ x)) = (m_s, m_c)$ then $(\eta^n_{\mathbf{E}}\ x) = l$ such that $(\sigma^n_{\mathbf{E}}\ l) = m$ and $(\sigma^n_{\mathbf{E}}\ m) = (\sigma_{\mathbf{E}}\ m_s) + (\sigma_{\mathbf{E}}\ m_c)$. Further, $h_v(\eta_{\mathbf{E}}\ x) = l$, $h_v(m_s) = m$, $h_v(m_c) = m$, $h_e((\eta_{\mathbf{E}}\ x), m_s) = (l, m)$, and $h_e((\eta_{\mathbf{E}}\ x), m_c) = (l, m)$.

From the construction, $\eta^n_{\mathbf{E}}$ is nominal. It remains to be shown that $\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta^n_{\mathbf{E}}$.

- To show that $\eta^n_{\mathbf{E}}$ is valid, we consider only the cases when an identifier or a location is not nominal in $\eta_{\mathbf{E}}$. This is because all the other values in $\eta^n_{\mathbf{E}}$ are the same as in $\eta_{\mathbf{E}}$, which is a valid environment. In our construction, since we explicitly create a `MoneyObj` location and assign it to a identifier or a location of type `FrozenAccount` or `BankAccount`, we can conclude that $\eta^n_{\mathbf{E}}$ is a valid H-environment.
- for each type $T \in (\mathrm{E}\mathit{TYPES} \Leftrightarrow \{\texttt{Store}\})$, for each x such that $H(x) = T$, $\mathrm{S}^w{}_T(\sigma, \sigma^n)((\eta_{\mathbf{E}}\ x), (\eta^n_{\mathbf{E}}\ x))$ is true from the construction.
- The pair of mappings (h_v, h_e) is an environment homomorphism on $\mathbf{E}$ from the construction.

EXTERNALS-**identical:** From the definition of R^w, for all $T \in \mathit{VIS}$, and for all $H(x) = T$, $\mathrm{S}^w{}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ x), (\eta'_{\mathbf{E}}\ x))$ should be hold. That is, $(\eta_{\mathbf{E}}\ x) = (\eta'_{\mathbf{E}}\ x)$.

bistrict: $\bot \mathrm{R}^w{}_H \bot$ from the definition. And suppose $\eta_{\mathbf{E}} \mathrm{R}^w{}_H \eta'_{\mathbf{E}}$, then by construction, if only one of η_H and $\eta'_{\mathbf{E}}$ is $\bot$ then S^w is not satisfied. Hence for $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$ to be related, $\eta'_{\mathbf{E}}$ has to be $\bot$.

bindable: Let $\eta_{\mathbf{E}} \mathrm{R}^w{}_H \eta'_{\mathbf{E}}$. For each type T, for each identifier y such that $H(y) = T$, for each identifier $x : T$, we need to show that $[x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}\ \mathrm{R}^w{}_{[x \mapsto T]H}\ [x \mapsto (\eta'_{\mathbf{E}}\ y)]\eta'_{\mathbf{E}}$. We show this by proving that the new environments satisfy all the properties of R^w.

- $[x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}$ and $[x \mapsto (\eta'_{\mathbf{E}}\ y)]\eta'_{\mathbf{E}}$ are valid $[x \mapsto T]H$-environments over $\mathbf{E}$ by construction.
- for each type T, for each z such that $[x \mapsto T]H(z) = T$, from the assumption that $\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta'_{\mathbf{E}}$, we can conclude that $\mathrm{S}^w{}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ z), (\eta'_{\mathbf{E}}\ z))$ is true. The case when $z = x$ $\mathrm{S}^w{}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ z), (\eta'_{\mathbf{E}}\ z))$ will be true because $\mathrm{S}^w{}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ y), (\eta'_{\mathbf{E}}\ y))$ is true from hypothesis.
- since the only difference between $\eta_{\mathbf{E}}$ and $[x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}$ is the alias between x and y, and since x and y have the same type, *stAliasOk*$([x \mapsto T]H, \mathbf{E}, [x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}})$, and
- Let (h_v, h_e) be an environment homomorphism on $\mathbf{E}$ between $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$. Based on (h_v, h_e), we define an environment homomorphism on $\mathbf{E}$, (h'_v, h'_e), between $[x \mapsto (\eta_{\mathbf{E}}\ y)]\eta_{\mathbf{E}}$ and $[x \mapsto (\eta'_{\mathbf{E}}\ y)]\eta'_{\mathbf{E}}$ as follows. $h'_v = [(\eta_{\mathbf{E}}\ x) \mapsto (\eta_{\mathbf{E}}\ y)]h_v$ and $h'_e = h_e$. From our assumption that (h_v, h_e) is an environment homomorphism on $\mathbf{E}$ and from the construction, we can conclude that (h'_v, h'_e) is an environment homomorphism on $\mathbf{E}$.

shrinkable: Let $H' \subseteq H$, $\eta_{\mathbf{E},H'} \subseteq \eta_{\mathbf{E}}$, $\eta'_{\mathbf{E},H'} \subseteq \eta'_{\mathbf{E}}$, where $\eta_{\mathbf{E},H'}$ and $\eta'_{\mathbf{E},H'}$ are valid H'-environments. Let us suppose that $\eta_{\mathbf{E}}\ \mathrm{R}^w{}_H\ \eta'_{\mathbf{E}}$. From these assumptions it is easy to see that the first three conditions of R^w are trivially satisfied. So, to conclude that $\eta_{\mathbf{E},H'} \mathrm{R}^w{}_{H'} \eta'_{\mathbf{E},H'}$ we need to show that there exists an environment homomorphism between $\eta_{\mathbf{E},H'}$ and $\eta'_{\mathbf{E},H'}$.

Let (h_v, h_e) be the environment homomorphism between $\eta_{\mathbf{E}}$ and $\eta'_{\mathbf{E}}$. A subgraph homomorphism of (h_v, h_e) between $(\eta_{\mathbf{E},H'}$ `store`$)$ and $(\eta'_{\mathbf{E},H'}$ `store`$)$ is the required environment homomorphism between $\eta_{\mathbf{E},H'}$ and $\eta'_{\mathbf{E},H'}$. The subgraph homomorphism is defined by restricting the domains of h_v to nodes and h_e to edges of `aliasG`$(\eta'_H$ `store`$)$. ∎

Corollary 3.7.4 *The presumed subtype relation, $\leq_W$ of Figure 2.3 is a weak behavioral subtype relation for* $\{\mathbf{E}\}$.

3.8 Strong behavioral subtyping

Recall that strong behavioral subtyping permits all forms of aliasing but preserves certain properties of types across states of a program. These properties are specified as *history constraints*, which are introduced by Liskov and Wing [LW94]. These history constraints can be thought of as invariants over ordered pairs of stores produced in the extension of a program. The first component of such an ordered pair is created earlier than the second component in a program.

Definition 3.8.1 (history constraint, satisfaction) *Let $(I\Sigma, E\Sigma)$ be a mutation signature and let* $\mathbf{A}$ *be a $(I\Sigma, E\Sigma)$-mutation algebra. A binary relation,* $\P_{\mathbf{A}}$*, is a* history constraint over $\mathbf{A}$ *if and only if it is reflexive and transitive.*

Then $\mathbf{A}$ satisfies history constraint $\P_{\mathbf{A}}$ *if and only if for each type S, for each type T, for all $g \in EOPS$, such that $g : \vec{S} \to (T, \mathtt{Store})$, for all $\vec{v} : \vec{S}$, if $(r_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = g^{\mathbf{A}}(\vec{v})$ then $\sigma_{\mathbf{A}} \P_{\mathbf{A}} \sigma'_{\mathbf{A}}$.*

In contrast to invariants that require properties to be preserved in a every store, history constraints require properties to be preserved across different stores. A few examples of the kinds of properties that can be captured by history constraints are that the value of an object does not change or the integer value of an object never decreases. A reasoning technique based on such history constraints can make conclusions based on the immutability or nondecreasing value of such an object.

To see a detailed example of a history constraint, consider a variation of $(\mathrm{I\Sigma}, \mathrm{E\Sigma})$, $(\mathrm{I\Sigma}^S, \mathrm{E\Sigma}^S)$, given in Figure 3.2. This signature is similar to $(\mathrm{I\Sigma}, \mathrm{E\Sigma})^{\mathbf{E}}$, except that we remove the subtype relationship between the `PlusAccount` and the `FrozenAccount` types. Let $\mathbf{D}$ be a $(\mathrm{I\Sigma}^S, \mathrm{E\Sigma}^S)$-algebra such that the only difference between $\mathbf{D}$ and $\mathbf{E}$ is in the signatures. Note that, in $\mathbf{D}$ an operation with `FrozenAccount` values can never invoke the code corresponding to `PlusAccount`'s methods because of the subtype relation in the signature $(\mathrm{I\Sigma}^S, \mathrm{E\Sigma}^S)$.

Example 3.8.2 *Let* $\mathbf{D}$ *be a $(I\Sigma^S, E\Sigma^S)$ mutation-algebra. Let $\mathcal{P}' \subseteq (VALS^{\mathbf{D}}_{\mathtt{Store}} \times VALS^{\mathbf{D}}_{\mathtt{Store}})$. For each $\sigma_1, \sigma_2 \in VALS^{\mathbf{D}}_{\mathtt{Store}}$, $\mathcal{P}'(\sigma_1, \sigma_2)$ holds if and only if: for each $l \in VALS^{\mathbf{D}}_{\mathtt{FrozenAccount}}$ such that $l \in Domain(\sigma_1)$ $(\sigma_1\ l) = (\sigma_2\ l)$, $(\sigma_1\ (\sigma_1\ l)) = (\sigma_2\ (\sigma_2\ l))$.*

The constraint $\mathcal{P}'$ holds if the amount of a `FrozenAccount` is constant over two environments. If this constraint property holds across all environments, one can deduce that `FrozenAccount` objects are immutable irrespective of any aliasing or behavioral subtyping.

Recall, that *SPEC* represents the set of all implementations of a given set of type specifications. Similarly, for a model-theoretic equivalent of history constraints,

$$\begin{array}{lcl}
\mathrm{E}\Sigma^{\S}.\mathrm{E}\mathit{TYPES} & = & \mathrm{E}\Sigma.\mathit{TYPES} \\
\mathrm{E}\Sigma^{\S}.\mathrm{E}\mathit{OPS} & = & \mathrm{E}\Sigma.\mathit{OPS} \\
\mathrm{E}\Sigma^{\S}.\mathrm{E}\mathit{ResType} & = & \mathrm{E}\Sigma.\mathit{ResType} \\
\mathrm{I}\Sigma^{\S}.\mathit{ITYPES} & = & \mathrm{I}\Sigma.\mathit{TYPES} \\
\mathrm{I}\Sigma^{\S}.\mathrm{I}\mathit{OPS} & = & \mathrm{I}\Sigma.\mathit{OPS} \\
\mathrm{I}\Sigma^{\S}.\mathrm{I}\mathit{ResType} & = & \mathrm{I}\Sigma.\mathit{ResType}
\end{array}$$

$$\begin{array}{lcl}
\leq_{\mathrm{S}} & \stackrel{\mathrm{def}}{=} & \{(\mathtt{PlusAccount}, \mathtt{BankAccount})\} \\
& & \quad \cup \{(T,T) \mid T \in \mathrm{E}\Sigma^{\S}.\mathit{TYPES}\}
\end{array}$$

Figure 3.2: A mutation signature, $(\mathrm{I}\Sigma^{S}, \mathrm{E}\Sigma^{S})$, where $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)^{\mathbf{E}}$ is given in Figure 2.4.

for each algebra that implements the given set of type specifications, we require an interpretation of the history constraint for that algebra. That is, if p is the (binary) predicate given in the type specification for the history constraint in the type specifications then for every $\mathbf{A}$ that implements the given type specifications, we require a history constraint over $\mathbf{A}$. We refer to the set of all such interpretations as HCONST. *HCONST is a function from algebras to history constraint over SPEC* such that for every $\mathbf{A} \in \mathit{SPEC}$, HCONST($\mathbf{A}$) is a history constraint over $\mathbf{A}$. That is, HCONST($\mathbf{A}$) represents the history constraint over $\mathbf{A}$ that corresponds to the denotation, in $\mathbf{A}$, of the constraint given in the set of type specifications.

Since our goal in defining strong behavioral subtypes is to permit all forms of aliasing and use static types to make conclusions about OO programs, we define strong behavioral subtyping to satisfy history constraints.

Definition 3.8.3 (strong behavioral subtyping) *Let $E\Sigma$ be a signature and let SPEC be a set of $E\Sigma$-algebras, such that each $\mathbf{A}$ in SPEC is a $(I\Sigma, E\Sigma)$-mutation algebra for some $I\Sigma$. Let HCONST be a history constraint over SPEC. The presumed subtype relation $\leq$ of $E\Sigma$ is a* strong behavioral subtype relation for *SPEC* with respect to HCONST *if and only if for each $\mathbf{B} \in$ SPEC there is some $\mathbf{A} \in$ SPEC such that there is a $E\Sigma$-simulation relation from $\mathbf{B}$ to $\mathbf{A}$, $\mathbf{B}$ satisfies history constraint HCONST($\mathbf{B}$) and $\mathbf{A}$ satisfies history constraint HCONST($\mathbf{A}$).*

The simulation relation ensures that strong behavioral subtypes “behave like” their supertype objects and the constraint property ensures that history properties are preserved by operations even when all forms of aliasing is allowed.

3.9 Example of a strong behavioral subtype relation

In this section we show that the presumed subtype relation, $\leq_{\mathrm{S}}$, given in Figure 3.2 is a strong behavioral subtype relation for $\{\mathbf{D}\}$ with respect to $\mathcal{P}'$. To show this, we first define a relation, R^s between algebra $\mathbf{D}$ (given in Figures 2.5 and 2.6) and itself. Then we prove that R^s is a simulation relation and $\mathbf{D}$ satisfies $\mathcal{P}'$.

Th proof of simulation is similar to Section 3.7. The only difference is in dealing with `FrozenAccount` values. Unlike in $\mathbf{E}$, `FrozenAccount` values in $\mathbf{D}$ cannot denote any `PlusAccount` values because according to $(\mathrm{E}\Sigma, \mathrm{I}\Sigma)^S$, `PlusAccount` is not a behavioral subtype of `FrozenAccount`.

We first define an abstraction function A_T for each $T \in \mathit{ObjectTypes}^{\mathrm{E}\Sigma^{\mathbf{E}}}$ that maps the abstract values of object types to an integer. The type of each A_T is:

$$A_T : (\widehat{VALS}^{\mathbf{E}}_T \times VALS^E_{\mathtt{Store}}) \rightarrow \mathtt{Integer}_\perp$$

and the particular versions are defined as follows:

$$\begin{array}{lcl}
A'_{\mathtt{MoneyObj}}(v, \sigma) & = & (\sigma\ v) \\
A'_{\mathtt{FrozenAccount}}(v, \sigma) & = & A'_{\mathtt{MoneyObj}}(\sigma\ v) \\
A'_{\mathtt{BankAccount}}(v, \sigma) & = & \textbf{if } (\sigma\ v) \in VALS^{\mathbf{E}}_{\mathrm{sortFor}(\mathtt{BankAccount})} \textbf{ then } A'_{\mathtt{MoneyObj}}(\sigma\ v) \\
 & & \textbf{else let } (m_s, m_c) = (\sigma\ v) \textbf{ in} \\
 & & \quad A'_{\mathtt{MoneyObj}}(\sigma\ m_s) + A'_{\mathtt{MoneyObj}}(\sigma\ m_c) \\
A'_{\mathtt{PlusAccount}}(v, \sigma) & = & \textbf{let } (m_s, m_c) = (\sigma\ v) \textbf{ in} \\
 & & (A'_{\mathtt{MoneyObj}}(\sigma\ m_s) + A'_{\mathtt{MoneyObj}}(\sigma\ m_c))
\end{array}$$

We define the notion of similarity, S^s, between abstract values of two types of $\mathbf{D}$ and use that notion in defining simulation relations. For each $T \in (\mathrm{E}\mathit{TYPES} \Leftrightarrow \{\mathtt{Store}\})$,

$$\mathrm{S}^s{}_T : (VALS^{\mathbf{D}}_{\mathtt{Store}} \times VALS^{\mathbf{D}}_{\mathtt{Store}}) \rightarrow (\widehat{VALS}^{\mathbf{D}} \times \widehat{VALS}^{\mathbf{D}}) \rightarrow \mathtt{Bool}$$

is a predicate that tests the similarity of two values in their respective stores. It is defined as follows. Let $\sigma_1, \sigma_2 \in VALS^{\mathbf{D}}_{\mathtt{Store}}$.

For all $T \in (\mathit{VIS} \cup \{\mathtt{Void}\})$ and for each $v_1, v_2 \in VALS^{\mathbf{D}}_{\mathrm{sortFor}(T)}$:

$$\mathrm{S}^s{}_T(\sigma_1, \sigma_2)(v_1, v_2) \stackrel{\mathrm{def}}{=} (v_1 = v_2) \tag{3.7}$$

We extend S^s pointwise to tuples. That is, $\vec{U}, \vec{V} \in \mathit{TYPES}$ such that $\vec{U} \leq_{\mathrm{W}} \vec{V}$, for each pair $\sigma_1, \sigma_2 \in VALS^{\mathbf{D}}_{\mathtt{Store}}$, such that for each $\vec{u} \in VALS^{\mathbf{D}}_{\vec{U}}$ and $\vec{v} \in VALS^{\mathbf{D}}_{\vec{V}}$, $\vec{u}\ \mathrm{S}^s{}_{\vec{V}}\ \vec{v}$ if and only if

$$(\mid \vec{u} \mid = \mid \vec{v} \mid) \wedge \bigwedge_{(1 \leq i \leq |\vec{u}|)} \mathrm{S}^s{}_{V_i}(\sigma_1, \sigma_2)(u_i, v_i) \tag{3.8}$$

For all $S, T \in \{$ `MoneyObj`, `FrozenAccount`, `BankAccount`, `PlusAccount` $\}$ and for each $v_1 \in \widehat{VALS}^{\mathbf{D}}_S$ and $v_2 \in \widehat{VALS}^{\mathbf{D}}_T$ such that $S \neq \texttt{PlusAccount} \vee T \neq \texttt{PlusAccount}$:

$$\mathrm{S}^s{}_T(\sigma_1, \sigma_2)(v_1, v_2) \overset{\text{def}}{=} A'_T(v_1, \sigma_1) = A'_T(v_2, \sigma_2) \tag{3.9}$$

For $T \in \{\texttt{PlusAccount}\}$, for each $v_1, v_2 \in \widehat{VALS}^{\mathbf{D}}_T$, if $(\sigma_1\ v_1) = (v_1^s, v_1^c)$ and $(\sigma_2\ v_2) = (v_2^s, v_2^c)$ then:

$$\begin{aligned}\mathrm{S}^s{}_T(\sigma_1, \sigma_2)(v_1, v_2) &= (A'_{\texttt{MoneyObj}}(v_1^s, \sigma_1) = A'_{\texttt{MoneyObj}}(v_2^s, \sigma_2)) \\ &\quad \wedge (A'_{\texttt{MoneyObj}}(v_1^c, \sigma_1) = A'_{\texttt{MoneyObj}}(v_2^c, \sigma_2))\end{aligned} \tag{3.10}$$

Like in construction of an example simulation for weak behavioral subtyping, we define an environment homomorphism for $\mathbf{D}$.

Definition 3.9.1 (environment-homomorphism) *Let* $\mathbf{D}$ *be the* $(E\Sigma, I\Sigma)^S$*-mutation algebra given in Figure 2.6 and Figure 2.5. Let* H *be a* $E\Sigma$ *type environment and* $\eta_{\mathbf{D}}$ *and* $\eta'_{\mathbf{D}}$ *be valid* H*-environments over* $\mathbf{D}$*. Suppose that* $(\eta_{\mathbf{D}}\ \texttt{store}) = \sigma_{\mathbf{D}}$ *and* $(\eta'_{\mathbf{D}}\ \texttt{store}) = \sigma'_{\mathbf{D}}$.

Then (h_v, h_n) *is an* environment homomorphism on $\mathbf{D}$ *between* $\eta_{\mathbf{D}}$ *and* $\eta'_{\mathbf{D}}$ *if and only if* (h_v, h_n) *is a graph homomorphism between* $\texttt{aliasG}(\sigma_{\mathbf{D}})$ *and* $\texttt{aliasG}(\sigma'_{\mathbf{D}})$ *and the following conditions are satisfied*

- *for each* $T \in ObjectTypes$*, if* $x : T \in Domain(H)$*,* $h_v(\eta_{\mathbf{D}}\ x) = (\eta'_{\mathbf{D}}\ x)$*,*
- *if* $stAliasOk(H, \mathbf{A}, \eta_{\mathbf{D}})$ *is false, then* (h_v, h_n) *is an injective graph homomorphism.*
- *if* $stAliasOk(H, \mathbf{A}, \eta_{\mathbf{D}})$ *is true, then for each* $l_1, l_2 \in Domain(\sigma_{\mathbf{D}})$*,* $h_v(l_1) = h_v(l_2)$ *if and only if* $l_1 = l_2$ *or there exists a* l *such that* $(\sigma_{\mathbf{D}}\ l) = (l_1, l_2)$ *and* $h(l) \in VALS^{\mathbf{A}}_T$*, where* $T \in \{\texttt{BankAccount}\}$.

The following relation, R^s, between $\mathbf{D}$ and itself ties together the abstraction function and the environment homomorphism on $\mathbf{D}$. We claim that R^s is a $\mathrm{E}\Sigma^S$-simulation relation between $\mathbf{D}$ and itself.

Definition 3.9.2 ($\mathbf{R}^s$) *The relation* R^s *from* $\mathbf{D}$ *to* $\mathbf{D}$ *is a family of binary relations on environments,* $\langle R^s{}_H : H \in TENV(E\Sigma^{\S})\rangle$*, such that* $R^s{}_H \subseteq Env_H[\mathbf{D}]_\perp \times Env_H[\mathbf{D}]_\perp$*,* $\perp R^s{}_H \perp$ *and* $\eta_{\mathbf{D}}\ R^s{}_H\ \eta'_{\mathbf{D}}$ *if and only if the following conditions all hold:*

- $\eta_{\mathbf{D}}, \eta'_{\mathbf{D}}$ *are valid* H*-environments over* $\mathbf{E}$*,*
- *for each type* $T \in (ETYPES \Leftrightarrow \{\texttt{Store}\})$*, for each* x *such that* $H(x) = T$*,* $S^s{}_T(\sigma_{\mathbf{E}}, \sigma'_{\mathbf{E}})((\eta_{\mathbf{E}}\ x), (\eta'_{\mathbf{E}}\ x))$*,*

- *there exists an environment homomorphism on* $\mathbf{E}$ *between* $\eta_{\mathbf{E}}$ *and* $\eta'_{\mathbf{E}}$.

Proposition 3.9.3 *R^s is a $E\Sigma^{\mathbf{D}}$-simulation relation between* $\mathbf{D}$ *and itself.*

Proof: (sketch) Since R^s is similar to R^w and $\mathbf{D}$ is similar to $\mathbf{E}$, the proof that R^s is a simulation relation from $\mathbf{D}$ to $\mathbf{D}$ is similar to the proof that R^w is a simulation relation from $\mathbf{E}$ to $\mathbf{E}$. In the construction of nominal environments, unlike before, we do not coerce values of `FrozenAccount` to `PlusAccount` because there is not subtype relation between `PlusAccount` and `FrozenAccount` in $(\mathrm{I}\Sigma^S, \mathrm{E}\Sigma^S)$. ∎

Proposition 3.9.4 $\mathbf{D}$ *satisfies history constraint* $\mathcal{P}'$.

Proof: None of operations of $\mathbf{D}$ mutate `FrozenAccount` values. This is true even for operations that take `FrozenAccount` values as arguments, such as `balance` and `get_interest`. From this, we can conclude that $\mathcal{P}'$ is satisfied. ∎

Corollary 3.9.5 *The presumed subtype relation,* $\leq_S$ *of Figure 3.2 is a strong behavioral subtype relation for* $\{\mathbf{D}\}$ *with respect to* $\mathcal{P}'$.

3.10 Discussion

In this section we first discuss the relationship between weak and strong behavioral subtyping and then give examples of various behavioral subtypes, which are used in our discussion of the related work.

3.10.1 Weak behavioral subtyping versus strong behavioral subtyping

From the definition of weak and strong behavioral subtyping it is clear that every strong behavioral subtype relation is a weak behavioral subtype relation.

Theorem 3.10.1 *Let SPEC be a set of $E\Sigma$-algebras such that each* $\mathbf{A}$ *in SPEC is a $(I\Sigma, E\Sigma)$-mutation algebra for some $I\Sigma$. Let HCONST be a history constraint over SPEC.*

If $\leq$ is a strong behavioral subtype relation for SPEC with respect to HCONST then $\leq$ is a weak behavioral subtype for SPEC. ∎

In our example signature given in Figure 2.4, we have `BankAccount` as both a strong and a weak behavioral subtype of `PlusAccount`.

However, the converse is not true. The following proposition shows that the weak behavioral subtype relation $\leq_W$ in Figure 2.3 is not a strong behavioral subtype relation. This is because of a mutable type, `PlusAccount`, is declared to be a subtype of an immutable type, `FrozenAccount` in our example signature $\mathrm{E}\Sigma^{\mathbf{E}}$.

Proposition 3.10.2 *The presumed subtype relation $\leq_W$ of $(I\Sigma, E\Sigma)$ in Figure 2.4 is not a strong behavioral subtype for $\{\mathbf{E}\}$ with respect to $\mathcal{P}'$.*

Proof: We prove this by contradiction. Consider a state over $\mathbf{E}$ such that a x : `FrozenAccount` denotes a `PlusAccount` value and y : `PlusAccount` denotes is aliased to x. This is possible from our assumption that `PlusAccount` $\leq_S$ `FrozenAccount`. But invoking `withdraw` on y, changes the value of x, and hence violates the constraint, $\mathcal{P}'$.

Hence the presumed subtype relation between `PlusAccount` and `FrozenAccount` is not a strong behavioral subtype. ∎

3.10.2 Weak and strong behavioral subtype hierarchies

In this section we discuss several weak and strong behavioral subtype relations among commonly used types. We treat these examples informally. In most cases the names of the types indicate the operations and their behavior.

3.10.2.1 More mutable subtypes Weak behavioral subtyping provides more interesting subtype hierarchies where the subtypes can have varying degrees of mutability. As shown earlier, `PlusAccount`, which is a mutable type, is a weak behavioral subtype of `FrozenAccount`, which is immutable. In the following discussion, by an immutable type we mean the history constraint states that objects of that type do not change over in different environments.

Other examples include tuples of varying degrees of mutability. Figure 3.3 gives weak and strong behavioral subtype relationship between tuples.

The type `ImmutablePair` consists of two components and its value does not change over time. A `MutablePair` can allow updates on one or more of its components. `MutablePair` is a weak behavioral subtype of `ImmutablePair` because the common operations of `MutablePair` act like that of `ImmutablePair` and a value of `MutablePair` can be coerced to a value of `ImmutablePair`. But `MutablePair` cannot be a strong behavioral subtype with respect to a constraint that values of `ImmutablePair` cannot change over time.

But an `ImmutableTriple` can be a strong behavioral subtype of `ImmutablePair` with respect to the constraint that values of `ImmutablePair` do not change over time.

The type `SemiMutableTriple` consists of an immutable pair and a mutable third component. `SemiMutableTriple` is both a weak and strong behavioral subtype of `ImmutablePair`.

To see the practical uses of allowing varying degrees of mutable types as subtypes to immutable types consider a model of various departments of a university. Each department mutates only a part of student records. For example, the payroll

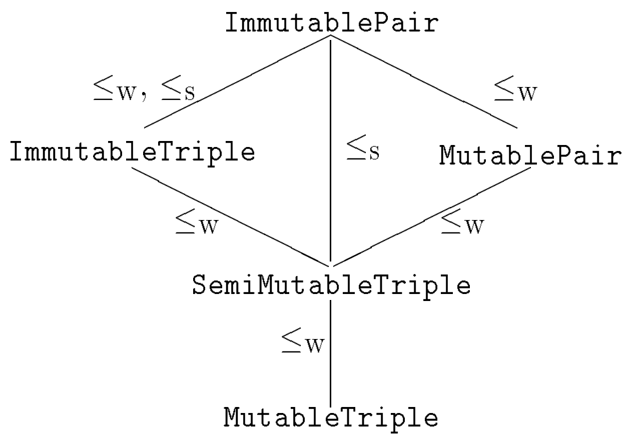

Figure 3.3: Behavioral subtype relationships between tuple types

department mutates only those components of a student record that are pertinent to the student's finances while the admissions office mutates components related to the student's grades. In order to protect from accidental updates we can make a series of weak behavioral subtypes of student records and use client functions that access or mutate the fields. Figure 3.4 gives such weak behavioral subtypes.

Note that the aliasing restrictions required for weak behavioral subtyping are not really restrictive in this example. Further, the `StudentRecord` type can have weak behavioral subtypes that contain more fields and/or more mutators.

This kind of hierarchy, based on mutability, can be visualized for several commonly used types like arrays, trees, and other similar types (only) for weak behavioral subtyping.

3.10.2.2 Virtual supertypes Often to capture common properties of different types, we need virtual supertypes. These types do not have any instances but only capture some common behavior of their subtypes. Both weak and strong behavioral subtypes allow such virtual supertypes. For example, the `ImmutableStudentRecord` discussed in Figure 3.4 can be a virtual type.

Another canonical example of a virtual type is the geometrical hierarchy of shapes. This hierarchy consists of a virtual type `Shape` with several concrete shapes such as `Rectangle`, `Square`, and `Triangle` as behavioral subtypes. Whether they are weak or strong behavioral subtypes depends on the mutability of these shapes.

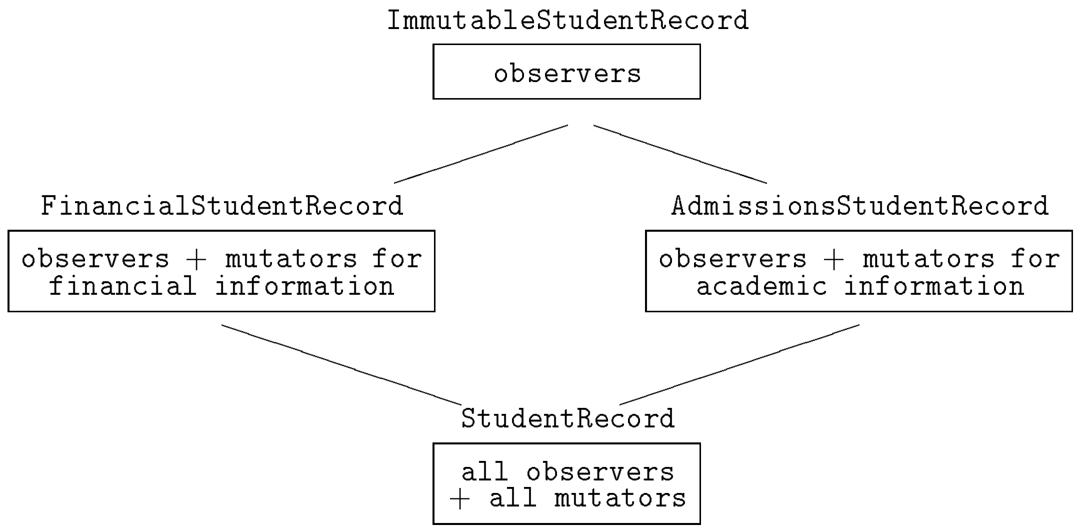
ImmutableStudentRecord
observers
FinancialStudentRecord
observers + mutators for financial information
AdmissionsStudentRecord
observers + mutators for academic information
StudentRecord
all observers + all mutators

Figure 3.4: Weak behavioral subtype, $\leq_W$, relations among different (views of) student records.

3.11 Related work

In this section we present a brief description of some related work. In Chapter 7, we present our conclusions on how useful these subtype notions are based on the kinds of reasoning they permit. (This discussion is in Chapter 7 because we prove the relevant properties of our notions of behavioral subtyping in Chapter 5.) In this section, we limit our discussion to the kinds of subtyping allowed by different notions of behavioral subtyping.

Subtyping based on signature [Car84, Car91] does not imply behavioral subtyping [Sny86]. To see this, consider two types `Stack` and a `Queue` with `get`, `put`, and `isIn` operations. If subtyping is just based on the signature then a `Stack` is a subtype of a (FIFO) `Queue` and vice versa. But it is quite clear that objects of `Stack` do not behave like objects of `Queue`. Allowing a `Stack` to be a subtype of a `Queue` would permit `Stack` objects to be sent to functions that expect `Queue` object. This would lead to surprising results because of the difference in behavior. So we believe that subtype relations should be based on the behavior, not just on the structure of types. We do not discuss these structural notions [Car84, Car91] of subtyping further.

Studies of subtyping that are based on the behavior of types can be broadly categorized into model-theoretic [BW90, Lea89, LW95, LP94] and proof-theoretic approaches [Ame87, Ame91, LW93a, LW94, DL96]. While the former approaches study models of types and define behavior in terms of these models, the latter approaches define behavioral subtyping based on the specification of individual types.

3.11.1 Model theoretic definitions

There are several model-theoretic studies of behavioral subtyping [BW90, Lea89, LW95, LP94] that use simulation relations or simulation functions to define behavioral subtyping. However, all these works are restricted to types with immutable objects, and hence do not deal with mutation and aliasing.

Even if one were to eliminate aliasing between identifiers/objects of different types, the definitions of behavioral subtyping for immutable types cannot be directly applied to the behavioral subtyping when mutable types are allowed.

3.11.1.1 Leavens and Pigozzi's behavioral subtyping Like Leavens and Pigozzi [LP94], we also define behavioral subtyping based on simulations that preserve substitution, coercion, *VIS*-identical, and other properties. Our substitution property, as in [LP94], captures the effects of operations by relating environments before and after the invocation of an operation. But since we deal with mutation, we require a special identifier `store` to capture the effects of operation on the internal states of different objects. Because of mutation, unlike in their work, two values that related in an environment might not be related in a new environment. Whenever the `store` changes several values in the environment are affect though there is no change in that value. This notion along with aliasing presents a different set of problems and hence, different notions of behavioral subtyping.

Further, our notion of nominality is not straightforward as it depends on contained objects and the store, which can have locations mapping to their subtype values. Because of nominality and aliasing we obtain two different notions of behavioral subtyping. In essence, the difference between our work and Leavens and Pigozzi's work is the study of the effects of mutation and aliasing in defining behavioral subtyping. In this dissertation we limit our work to proving the soundness of our behavioral subtype notions. Since we use a similar framework as Leavens and Pigozzi, it is interesting to see if we could adopt their techniques to show that our notions are complete. We leave this as future work.

3.11.1.2 Abadi and Leino's subsumption Abadi and Leino [AL97] extend Cardelli's [Car91] structural subtyping rules on records to include behavior. One of the conditions for their notion of subsumption, which is their notion of behavioral subtyping, is that two types should be structurally subtypes. Hence, their notion of subsumption cannot relate arbitrary abstract data types.

Abadi and Leino present a sound verification logic based on their notion of subsumption. It is not clear if this verification logic is modular and if one can conclude properties of types from specification using their verification logic. For example, in the case of the bankaccounts, it is not clear if one can conclude from their verification logic that `FrozenAccount` objects are immutable. This property is important for modular reasoning because it allows conclusions about existing procedures without looking at programs that use these procedures. Introducing history constraints in their subsumption relations would be allow such reasoning and we believe that their notion of subsumption would then be equivalent to strong behavioral subtyping that is restricted to relations between structural subtypes.

3.11.1.3 Cusack's Specialization

Cusack [Cus91] defines a notion of specialization over schemas of `Z` [Spi88] that is like behavioral subtyping. Since she uses schemas to define the specialization both the subtypes and the supertypes should be specified in the same schema. Further, she does not deal with the subtyping in the extra arguments of methods and does not have any notion of history constraints. So it is not clear on what are the properties of types that are preserved in the context of mutation and aliasing.

3.11.1.4 Lewerentz et al.'s refinement calculus

In their study [LLRS95], Lewerentz and his colleagues present a theory of objects and present a refinement calculus based on observations on types. Their goal is to use refinement calculus in OO modeling. Like us, they define refinement using simulations. Unlike our simulations, their simulations use programs to define a "behaves like" notion. It is defined using the attributes of subtypes and supertypes. Simulations, with the help of a coercion program, relate the effect of constructors and methods on the states of the subtype and the supertype objects.

The difference between our approach and their approach is both in our objective and in our technique. We treat types as abstractly, that is our simulation relations are not defined on the attributes or contained objects of the types. They define simulations based on the attributes of types and their simulations are dependent on language constructs. Defining simulation relations independent of the language allows a wider application of our results. They do not consider aliasing or interference between identifiers in their programs, which is a main component in our study of behavioral subtyping.

3.11.2 Proof-theoretic approaches

In contrast to our model-theoretic approach, there are several other works on proof-theoretic notions of behavioral subtyping. These include America [Ame87], Liskov and Wing [LW93a, LW94] and our proof-theoretical notions of behavioral subtyping in [DL96].

Other than the approach, there are some fundamental differences between this dissertation and these proof-theoretic notions. Model-theoretic approaches look at (abstractions of) different implementations of types and determine whether the subtyping in the programs is a behavioral subtype or not. Looking at a set of types, together with a set of operations, allows model-theoretic approaches to draw conclusions about programs that use these types. However, proof-theoretic approaches look at individual type specifications and define a relationship between such types. Our goal in studying behavioral subtyping is to provide a sound notion of behavioral subtyping that can be used for supertype abstraction. None of these proof-theoretic approaches [Ame87, LW93a, LW94, DL96] prove soundness of behavioral subtyping. But proving behavioral subtype relations using proof-theoretic notions is easier than using model-theoretic notions.

We also identify two notions of behavioral subtyping of which the strong behavioral subtype notion is closer to the notions proposed by several proof-theoretic approaches. The notion of weak behavioral subtyping is new to our work [DL96].

Since Liskov and Wing's work is an extension of America's work, we omit discussion of America's work.

3.11.2.1 Liskov and Wing's definition In [LW93a, LW94], Liskov and Wing define behavioral subtyping based on the specification of types. They require the invariant, the constraint, and the postcondition of each method of the subtype to imply that of the supertype's and the precondition of each method of the supertype to imply that of the subtype. In order to ensure that the vocabulary of the pre and postcondition is same they use a coercion function from subtype values to supertype values. Their notions, both the constraint and extension based behavioral subtyping, allow all forms of aliasing in states.

If one were to specify a type as immutable in the history constraint, then the constraint-based behavioral subtyping of Liskov and Wing, does not allow any mutable subtypes of immutable types. This is because the subtype's history constraint should imply its supertype's history constraint, which for a mutable subtype does not hold.

Even the behavioral subtype relation with the extension rule of Liskov and Wing does not allow a behavioral subtype relation between a mutable type and an immutable type. This is because, according to the "extension" rule, all the extra methods in the subtype should be explained by operations of the supertype. But the method `withdraw` for `PlusAccount` cannot be explained in terms of the methods of `FrozenAccount`. Hence, `PlusAccount` is not a behavioral subtype of `FrozenAccount` according to the behavioral subtype relation based on the extension rule. Weak behavioral subtyping allows such relationships where the subtype's extra operations have more mutability.

Comparing our notion of strong behavioral subtyping to Liskov and Wing's definitions is interesting. Like Liskov and Wing's notions, our notion of strong behavioral subtyping allows all forms of aliasing and disallows mutable subtypes of immutable types. One of our goals in defining strong behavioral subtyping is to obtain a sound model-theoretic equivalent of Liskov and Wing's notions of behavioral subtyping. However, we leave a relation between strong behavioral subtyping and their definition as a future work. Since it is not clear what is the adequate notion of soundness for proof-theoretic notions of behavioral subtyping, a formal relation between strong behavioral subtyping that is shown to be sound for reasoning and the proof-theoretic notions that are easy to prove is useful. Such a relation would have the advantages of both these approaches.

3.11.2.2 Dhara and Leavens' proof-theoretic definition In [DL96], we present our proof-theoretic notion of behavioral subtyping. We weaken Liskov and Wing's "constraint" based behavioral subtyping rule slightly and define a proof-theoretic equivalent of our weak behavioral subtyping.

By weakening the post condition rule, we allow subtype objects to operate in a bigger domain than supertype objects. For example [DL96], consider a type T with a method `foo(int x)` with a specification that requires the $x > 0$ as a precondition. Unlike, Liskov and Wing, we [DL96] allow a behavioral subtype with a `foo` method that weakens the precondition of T. That is, we allow a strong behavioral subtype that has a pre-condition $x < 0 \wedge x > 0$.

Our model-theoretic notion of strong behavioral subtype is closer to this notion of behavioral subtyping than Liskov and Wing's. We leave the proof that the strong and weak notion of behavioral subtyping of [DL96] are equivalent to those defined here as future work.

4. $OBS^{\leq}$ – A MULTI-METHOD LANGUAGE

The soundness of the behavioral subtype notions defined in the last chapter can be shown by comparing the results of a function over a state with subtyping and over a state without subtyping. Our criteria for the soundness of the behavioral subtype notions is that the results over a state with subtyping should be a subset of the results over a state without subtyping. To do this comparison, we require a language that allows a (client) function to observe states with or without subtyping. Hence the name of our language, $OBS^{\leq}$.

In this chapter, we define $OBS^{\leq}$, and refer to the function and the observation as a main program of $OBS^{\leq}$. Since the soundness depends on the features of $OBS^{\leq}$, it is important to see the features offered by main programs of $OBS^{\leq}$. They are as follows.

- $OBS^{\leq}$ allows mutation and aliasing.
- Programs in $OBS^{\leq}$ allow subtyping in their states. That is, identifiers and locations can denote subtype values of their static (nominal) types.
- Programs in $OBS^{\leq}$ can construct a state (with or without subtyping) and pass this state to a (client) function.
- $OBS^{\leq}$ allows dynamic message passing.
- $OBS^{\leq}$ has a sound type and alias checking rules that demonstrate that the constraints defined for weak behavioral subtyping are not too restrictive.

The method invocations used in the main programs of $OBS^{\leq}$ are interpreted using mutation algebras. These mutation algebras are obtained by a "compilation" of the semantics of type and method declarations of $OBS^{\leq}$. We present the syntax of type and method declarations of $OBS^{\leq}$, but for simplicity, we do not give the details of its "compilation" into algebras. We leave this as a future work, which would allow a blend of denotational and algebraic semantics [LD94].

In the next section we give an overview of $OBS^{\leq}$ and present its syntax (including the type and method declarations) with an example. In later sections, we present the semantics of main programs of $OBS^{\leq}$ with appropriate alias constraints and we

Abstract syntax:

ALG ∈ Algebra	TD ∈ Type-Declarations	IV ∈ Instance-Variables
T ∈ Type	I ∈ Identifiers	MD ∈ Method-Declarations
F ∈ Formal-Parameters	A ∈ Alias-Declaration	B ∈ Body
D ∈ Const-Declaration	C ∈ Command	E ∈ Expression
N ∈ Number		

ALG ::= TD* MD*
TD ::= `type` I `subtype of` {T*} `instance variables` IV* `end`
T ::= I
IV ::= I `:` T

MD ::= `method` I `(` F* `)` `:` T A `is` B
F ::= I `:` T
A ::= | `may alias` `{` T* `}`

B ::= D `do` C `return` E
D ::= | `const` I `:` T `:=` E | D_1 `;` D_2
E ::= N | `nothing` | `true` | `false` | I | I `(` E* `)` | `new` I (E*) | I_1 `.` I_2
C ::= E | `if` E_1 `then` C_1 `else` C_2 `fi` | C_1 `;` C_2 | I_1 `.` I_2 `:=` E

Figure 4.1: Abstract syntax for type and method implementations of $\mathit{OBS}^{\leq}$. "TD*" is a sequence of zero or more "TD"s (with separators in concrete examples).

conclude this chapter by presenting the soundness of our alias rules for the main programs.

4.1 Syntax and overview of $\mathit{OBS}^{\leq}$

4.1.1 Split in $\mathit{OBS}^{\leq}$

Figure 4.1 shows abstract syntax of the type and method implementation of $\mathit{OBS}^{\leq}$ and Figure 4.2 shows the abstract syntax of main programs of $\mathit{OBS}^{\leq}$. The language for type and method implementations could be completely different from the main program. But for clear presentation, we use a similar language for both these components.

This split in the language (refer to Figure 4.3) allows the study of the properties of types independent of the properties of main programs that use those types. Another interesting aspect of our language is the form of the (client) programs, that

Abstract Syntax:

P ∈ Main-Program	FN ∈ Client-Function	D ∈ Const-Declaration
C ∈ Command	E ∈ Expression	I ∈ Identifier
T ∈ Type	F ∈ Formal-Parameters	B ∈ Body
N ∈ Number		

```
P ::= observe  FN by D do C; call I ( E*)

FN ::= client function I ( F* ) : T is B
F ::= I : T

B ::= D do C return E
D ::= | const I : T := E | D1 ; D2

E ::= N | nothing | true | false | I | I ( E* )
C ::= E | if E1 then C1 else C2 fi | C1 ; C2
```

Figure 4.2: Abstract syntax of main programs of $OBS^{\leq}$.

is the second component. We refer to this component as the main program. A main program consists of a function, a set of declarations, and commands to set up a state for observing the function. This division in the main program allows us to study the results of an existing function while changing the state in which the function is called. That is, by setting up a state where the call to the client function contains subtype objects, we can observe the behavior of the client function in the context of subtyping. Such a notion simulates the effects of calling existing functions with new subtype objects, which is essential for our study of behavioral subtyping.

The corresponding split in the semantics compiles type declarations and method declarations into a mutation algebra and then runs the main program M using that algebra. Figure 4.3 illustrates the idea of split semantics. Split semantics is the technique in which the denotations of types and method declarations are captured in an algebra and the meanings of the main programs are defined using these algebras as parameters. The signature of the algebra acts as an interface for split semantics. That is, the denotations of the constructs in the main program are defined with respect to any algebra that satisfies the signature at the split in Figure 4.3.

Though the commands and expressions used in the body of methods and in main programs are similar, there are a few differences; these differences provide a basic form

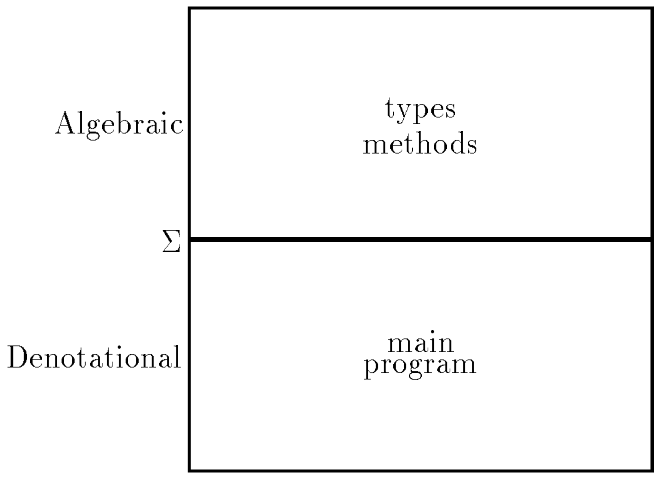

Figure 4.3: A picture that illustrates the idea of the split semantics.

of encapsulation. This encapsulation prevents main programs from accessing instance variables and enforces the split between the main programs and implementations of types. Method bodies can access instance variables ($\texttt{I}_1\texttt{.I}_2$), create new objects (`new`), and also assign instance variables ($\texttt{I}_1\texttt{.I}_2$ `:= E`). These expressions and the instance variable assignment command cannot be used in the main procedure (M). Since subtype objects may be implemented in different ways than the supertype objects, we do not allow subtypes to use instance variables of their supertypes. In $OBS^{\leq}$, for any method this simplification would force the programmers to write a set of methods such that for each combination of the type of the permitted arguments there is a unique method that applies. Chambers and Leavens [CL94] use inheritance and a form of specialization [Cha92] to reduce the number of methods that need to be written in such cases. For more flexibility (with respect to the above restriction), inheritance could be added to $OBS^{\leq}$, but that would complicate the semantics and would not be of any help to our study of behavioral subtyping. Hence, there is no inheritance in $OBS^{\leq}$.

In the next sections we present semantics of main programs of $OBS^{\leq}$.

4.1.2 A sample program in $OBS^{\leq}$

```
type MoneyObj subtype of {}
  instance variables money: Integer end;
type FrozenAccount subtype of {}
  instance variables acct: MoneyObj end;
type BankAccount subtype of {}
  instance variables acct: MoneyObj end;
type PlusAccount subtype of {FrozenAccount, BankAccount}
  instance variables svgs: MoneyObj, chkg: MoneyObj end;

method mkFrozenAccount(m: Money):FrozenAccount
  is const money: MoneyObj := new MoneyObj (value(m));
     const result: FrozenAccount := new FrozenAccount (money)
  do
     nothing
  return result;

method withdraw(p: PlusAccount; m: MoneyObj):Void
  is const pSvgs: Integer := value(p.svgs);
     const pChkg: Integer := value(p.chkg);
     const mValu: Integer := value(m)
  do
     if (pSvgs > mValu) then update(p.svgs, (pSvgs - mValu))
       else if ((pSvgs + pChkg) > mValu) then update(p.svgs, 0);
                        update(p.chkg, (pSvgs + pChkg) - mValu);
            else nothing
  return nothing;

method check_balance(p: PlusAccount): MoneyObj
                    may alias {MoneyObj}
   is
   do
      nothing
   return p.chkg;
...
...
```

Figure 4.4: Type and method implementation for types and methods of $E\Sigma^{\mathbf{E}}$. Omitted details can be found in Appendix B.

```
observe
 client function verify_withdraw(b: BankAccount): Bool
  is const m: MoneyObj := mkMoneyObj(3);
     const bal: Int := value(balance(b))
  do
     withdraw(b, m)
  return (equal(value(balance(b)), minus(bal,3)));
by
     const b: BankAccount := mkPlusAccount(mkMoney(20), mkMoney(10))
  do
     deposit(b, mkMoney(10));
  call verify_withdraw(b);
```

Figure 4.5: A sample main program, which observes the types `BankAccount` and `PlusAccount` of $E\Sigma^{\mathbf{E}}$. The result of this program is `true`.

Figure 4.4 gives the type and method implementations of the $E\Sigma^{\mathbf{E}}$-algebra $\mathbf{E}$ (given in Figure 2.5 and Figure 2.6). The type declaration for `MoneyObj` indicates that it has an instance variable `money` of type `Integer`. The type declaration for `PlusAccount` states that it is a subtype of both `FrozenAccount` and `BankAccount` and it has instance variables `svgs` and `chkg` of type `MoneyObj`.

The `mkFrozenAccount` method returns a new object of type `FrozenAccount` with a `MoneyObj` whose initial value is that of money object `m` passed as a parameter. The method `check_balance` illustrates the "may alias" construct. The alias component of this method states that the result of this method may be aliased to a location of type `MoneyObj`. This aliasing information is used in statically restricting aliases between objects of different types. Such restrictions help in creating stores that satisfy the aliasing constraints that are required for sound reasoning when weak behavioral subtyping is used.

The client function in the main program in Figure 4.5, `verify_withdraw` compares the amount in b before invoking `withdraw(b, m)` and after the invocation. To reason about `verify_withdraw` statically, we can check the code of the `withdraw` operation of `BankAccount` and can conclude that it returns `true`.

However, method dispatching in $OBS^{\leq}$ is dynamic in the sense that the method lookup does not depend on the static types of arguments, but depends on the types of the objects at runtime, that is, during the call. In the environment that is set up by the main program in Figure 4.5, the runtime type of `b` is a `PlusAccount`. Hence, in the `verify_withdraw` the call `withdraw(b, m)` invokes the `withdraw` method on `PlusAccount` objects, because `b` denotes a `PlusAccount` object. If behavioral subtyping is sound, that is if reasoning at the static types of objects is sound, then the result of `verify_withdraw`, when b denotes a `PlusAccount` object, should be `true`.

From the discussion in the previous chapter (Section 3.4 we saw the aliasing requirements for the soundness of weak behavioral subtyping. In the following sections, we discuss these aliasing restrictions with respect to $OBS^{\leq}$ and present the semantics of the main programs of $OBS^{\leq}$.

4.2 Alias restrictions for weak behavioral subtyping

Our notion of weak behavioral subtyping is adequate for supertype abstraction only if we restrict direct aliasing between variables or locations of different types in an environment. In Section 4.2.1 we present the conditions on algebras that are used (in Section 4.3) to conservatively restrict aliasing between variables and locations of different types in $OBS^{\leq}$.

4.2.1 Alias legality

Since the semantics (shown later) of main programs of $OBS^{\leq}$ is parameterized by a mutation algebra, that provides interpretations for operations, we need develop aliasing constraints on both the main programs and the mutation algebras. Because mutation algebras are independent of particular implementation details, we develop a conservative notion, which is referred as *alias legality*, to constrain certain kinds of aliasing for weak behavioral subtyping. All the mutation algebras need to satisfy alias legality for the soundness of our type and alias checking rules that guarantee that the environments generated by $OBS^{\leq}$ programs satsify *stAliasOk*. The following discussion formalizes the notion of alias legality for mutation algebras.

Semantically, for the soundness of weak behavioral subtyping, every environment in the program should satisfy *stAliasOk*. Since we do not work with the compilation of algebras, and since the main programs are parametrized with algebras that can be obtained from other forms of compilation, we require all the external operations of mutation algebras to satisfy the alias restrictions.

Mutation signatures, defined in Chapter 2, give only the type information, but we need the alias information for each operation. That is the alias type set of the return value of operations. In case of methods of $OBS^{\leq}$, this information is obtained from the "may alias" component of method declarations. This information could be added to a mutation signature while compiling the type and method declarations. But that would make the signatures non-standard. Hence we define our notion with

$\forall g \in \mathrm{E}\mathit{OPS}^{\mathrm{E}\Sigma^{\mathbf{E}}}, \forall \vec{S}, T \in \mathrm{E}\mathit{TYPES}$ such that $\mathrm{E}\mathit{ResType}(g, \vec{S}) = T$,

$$L^{\mathbf{E}}(g, \vec{S}) = \{\}.$$

Figure 4.6: Alias context $L^{\mathbf{E}}$ for operations of algebra $\mathbf{E}$

respect to an alias context, which gives the upper bound on the alias type set of the return value of external operations in a mutation algebra. We refer to this notion as the *alias context* of a signature and denote it by L. An alias context is a family of partial functions such that $L : \mathit{OPS} \times \mathit{TYPES} \rightarrow \mathit{PowerSet}(\mathit{TYPES})$. The alias context of algebra $\mathbf{E}$, given in Figures 2.5 and 2.6, is shown in Figure 4.6. Since, the alias context for all operations returns an empty set, operations of algebras that satisfy the alias context should return a new object.

To illustrate more about alias context, we could have a method `return_savings` with an alias context, $L(\texttt{return\_savings}, (\texttt{PlusAccount}, \texttt{Store})) = \{\texttt{MoneyObj}\}$. A method `return_savings` that returns the savings component of its `PlusAccount` object satisfies the above alias context for `return_savings`. As `return_savings` is used only for illustration, we do not include it in our signature for $\mathrm{E}\Sigma^{\mathbf{E}}$.

If an environment that satisfies the alias constraints for weak behavioral subtyping is passed to an operation, then the resulting environment should satisfy the constraints and the alias type set of the return value of the operation should be a subset of the expected set of aliases given by its alias context. The following definition formalizes these notions as a condition on mutation algebras.

Definition 4.2.1 (preserves alias legality) *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra and let* L *be an alias context for* $(I\Sigma, E\Sigma)$*. Then* $\mathbf{A}$ preserves alias legality with respect to L *if and only if for each type environment* H*, for each* H*-environment,* η*, over* $\mathbf{A}$ *such that* $\mathit{stAliasOk}(H, \mathbf{A}, \eta)$*, for each operation* $g \in \mathit{EOPS}$*, for each type* $\vec{S}$*, if* $\mathit{ResType}(g, \vec{S}) = (T, \texttt{Store}), \vec{v} \in \mathit{VALS}^{\mathbf{A}}_{\vec{S}}$*, and* $(v, \sigma') = g^{\mathbf{A}}(\vec{v})$ *then:*

$$\begin{aligned}&\mathit{stAliasOk}(H, \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta)\\ &\quad \wedge (T \in \mathit{ObjectTypes} \Rightarrow \mathit{aliasTypeSet}(H, \mathbf{A}, v, [\texttt{store} \mapsto \sigma']\eta) \subseteq L(g, \vec{S}))\end{aligned}$$

The first conjunct states that the resulting environment, which includes the updated store, should be *stAliasOk*. The set inclusion in the second clause ensures that the alias type set of the returned value is a subset of the expected alias type set given by the alias context. The following subsection discusses the implications of these requirements on mutation algebras.

Claim 4.2.2 *Algebra* **E** *given Figures 2.5 and 2.6 satisfies alias legality with respect to the alias context* $L^{\mathbf{E}}$ *given in Figure 4.6.* ∎

The proof of this claim is trivial because all operation interpretations of **E** in Figure 2.6 return new objects, and hence these objects are not aliased to any of their arguments.

4.2.2 Restrictions on type and method declarations of $OBS^{\leq}$

This section discusses the practical implications of enforcing the required aliasing conditions. That is, it answers the question: what does it mean for algebras to satisfy alias legality from the point of view of implementing the corresponding type and method declarations?

4.2.2.1 Aliasing in method arguments Aliasing between arguments in method calls may result in identifiers of distinct types to be direct aliases in method bodies. This could happen to a method with formals of two different types. One way of preventing this is to prevent aliasing between actuals. This is more restrictive than necessary because aliasing between formals of the same type does not violate the required aliasing restrictions. Instead, we require that the programmer implement enough methods that would cover all kinds of method calls and would prevent aliasing between formals of different types. For example, a method `foo` that takes two arguments of unrelated types S and T. If at a later point, a common subtype of S and T, U is added then the user has to provide a method for `foo` that takes two U objects. A call to `foo` with the two actuals aliases would invoke the new method and hence do not violate any alias restrictions.

4.2.2.2 Aliasing in method bodies The instance variable assignment in method bodies can make direct aliases that result in denotations of type and method implementations, that is algebras, that do not satisfy alias legality. To see this, consider an operation, `alias_savings`, that takes two `PlusAccount` objects and make an alias between the savings component of the two `PlusAccount` objects. Let x : `BankAccount` and y : `BankAccount` and let l_1 : `BankAccount` and l_2 : `BankAccount` be the denotations of x and y. Further, let l_1 and l_2 denote two different sortFor(`PlusAccount`) values. This state itself satisfies *stAliasOk*, but the resulting state of a call to `alias_savings` with l_1 and l_2 as arguments makes an illegal alias that violates *stAliasOk*. That is, in the resulting state l_1 and l_2 are direct aliases, and the values of l_1 and l_2 denote subtype values and are different. Hence, to ensure that algebras preserve alias legality, we disallow aliasing of instance variables. This is again more restrictive than necessary but we leave the investigation of techniques that are less restrictive than this and still permit sound type and alias checking rules to future work.

4.3 Semantics of main programs

In this section we present our alias and type checking rules for the main programs of $OBS^{\leq}$. Our goal is a conservative alias analysis of main programs such that each environment sequence in the sequence of environments given by the semantics of main programs satisfies *stAliasOk*.

4.3.1 Type and alias checking rules for $OBS^{\leq}$

Figure 4.7 gives the alias and type checking rules for the main programs of $OBS^{\leq}$. The notation $\Sigma; H; L \vdash \mathrm{E} : T :: r$ means that an expression E has a static type T and r is the upper bound on the alias type set of the result of E, when evaluated in an algebra with an external signature of Σ, a type context H, and an alias context, L. For integers and booleans, which are immutable, the upper bound on the alias type set is $\{\}$ because there cannot be any observable aliasing for these types. For a variable, the alias type set is the singleton set consisting of the type of the variable given by the type-context H. The alias type set of a method invocation, g, with static argument types $\vec{S}$, is given by $L(g, (\vec{S}, \texttt{Store}))$. This set can be obtained from method declarations' "may alias" construct.

The [decl] allows the creation of states with subtyping in the main programs of $OBS^{\leq}$. To avoid the improper kinds of aliasing for weak behavioral subtyping, we require the alias type set of E to be the same type as the identifier it is assigned to or to be empty. The empty case allows assigning subtype objects to identifiers of supertype.

The [Main] rule types the client function, checks the declarations, commands, and the call to the client function. To ensure that there is no aliasing among arguments in the client function call, for any call with more than one argument, we require that its alias type set be empty. This ensures that the values or objects that are bound to the formals of the client function do not have any other references, thus avoiding aliasing between them.

4.3.2 Valuation functions

In this section we present the valuation functions, which define the meaning of the various constructs in the main programs of $OBS^{\leq}$. We use standard denotation semantics [Sch94]. These valuation functions correspond to the typing rules and are used in proving the soundness of the type and alias rules given in the previous section.

[Num] $\Sigma; H; L \vdash \mathrm{N} : \texttt{Integer} :: \{\}$ [nothing] $\Sigma; H; L \vdash \texttt{nothing} : \texttt{Void} :: \{\}$

[true] $\Sigma; H; L \vdash \texttt{true} : \texttt{Bool} :: \{\}$ [false] $\Sigma; H; L \vdash \texttt{false} : \texttt{Bool} :: \{\}$

[ident] $\Sigma; H; L \vdash \mathrm{I} : H(\mathrm{I}) :: \{H(\mathrm{I})\}$ if $\mathrm{I} \in \mathit{Domain}(H)$

[call] $$\frac{\begin{array}{c}\Sigma; H; L \vdash \vec{\mathrm{E}} : \vec{S} :: \vec{r}, \\ L(\mathrm{I}, (\vec{S}, \texttt{Store})) = r\end{array}}{\Sigma; H; L \vdash \mathrm{I}(\vec{\mathrm{E}}) : T :: r} \qquad \text{if } \Sigma.\mathit{ResType}(\mathrm{I}, (\vec{S}, \texttt{Store})) = (T, \texttt{Store})$$

[NList] $\Sigma; H; L \vdash () : () :: ()$

[EList] $$\frac{\Sigma; H; L \vdash \vec{\mathrm{E}} : \vec{T} :: \vec{r}, \quad \Sigma; H; L \vdash \mathrm{E} : T :: r}{\Sigma; H; L \vdash \vec{\mathrm{E}}\mathrm{E} : (\vec{T}, T) :: (\vec{r}, r)}$$

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

[FN] $$\frac{\Sigma; [\vec{I} \mapsto \vec{T}]H; L \vdash B : T :: r}{\Sigma; H; L \vdash \texttt{client function}\ \mathrm{I}(\vec{I} : \vec{T})\text{:T is B} : (\vec{T} \rightarrow T)} \qquad \text{if } T \in \mathit{VIS}$$

[EDecl] $\Sigma; H; L \vdash \quad \Longrightarrow \{\}$

[SDecl] $$\frac{\Sigma; H; L \vdash E : S :: r}{\Sigma; H; L \vdash \texttt{const}\ \ I\texttt{:}T\texttt{:=}E \Longrightarrow \{(I, T), (\texttt{store}, \texttt{Store})\}} \quad \text{if } S \leq T, r \subseteq \{T\}$$

[MDecl] $$\frac{\Sigma; H; L \vdash D_1 \Longrightarrow H', \quad \Sigma; H \cup H'; L \vdash D_2 \Longrightarrow H''}{\Sigma; H; L \vdash D_1 ;\ \ D_2 \Longrightarrow H' \cup H''} \quad \text{if } \texttt{uniqueIDS}(H', H'')$$

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

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

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

[Main] $$\frac{\begin{array}{c}\Sigma; \{\}; L \vdash FN : (\vec{T} \rightarrow T), \ \Sigma; \{\}; L \vdash D \Longrightarrow H \\ \Sigma; H; L \vdash C \surd, \quad \Sigma; H; L \vdash E* : \vec{S} :: \vec{r}\end{array}}{\begin{array}{l}\Sigma; L \vdash \texttt{observe client function} \\ \quad \mathrm{I}(\vec{I}\texttt{:}\vec{T})\texttt{:}T\ \ \texttt{is}\ \ B\ \ \texttt{by}\ \mathrm{D}\ \texttt{do}\ \mathrm{C};\ \texttt{call}\ \mathrm{I}(\mathrm{E}^*) \surd\end{array}} \quad \begin{array}{l}\text{if } \vec{S} \leq \vec{T}, \text{ and} \\ (\mid \vec{S} \mid > 1 \wedge (1 \leq i \leq \mid \vec{S} \mid) \\ \quad \Rightarrow \text{each } r_i = \{\})\end{array}$$

Figure 4.7: Type and alias checking rules for the main procedure part of $OBS^{\leq}$. The auxiliary function $\texttt{uniqueIDs}(H', H'')$ is $\texttt{true}$ if $\mathit{Domain}(H') \Leftrightarrow \{\texttt{Store}\}$ and $\mathit{Domain}(H'') \Leftrightarrow \{\texttt{Store}\}$ are disjoint, otherwise it is $\texttt{false}$.

Let *SIGS* denote the class of all mutation signatures; thus the notation $\Sigma : \mathit{SIGS}$ means that Σ is a mutation signature. Let $\mathit{Alg}(\Sigma)$ denote the class of all Σ-algebras.

$$\mathit{Alg}(\Sigma) \overset{\text{def}}{=} \{\mathbf{A} \mid \mathbf{A} \text{ is a } \Sigma\text{-mutation algebra}\} \tag{4.1}$$

We use $\mathit{STATE}_H^{\mathbf{A}}$ to represent the set of valid H-environments. That is,

$$\mathit{STATE}_H^{\mathbf{A}} = \{\eta \mid \eta \in \mathit{Env}_H^{\mathbf{A}} \text{ and } \eta \text{ is a valid } H\text{-environment}\}. \tag{4.2}$$

The subscript H is dropped whenever the type context is clear. All the following valuation functions, take a signature and a mutation algebra over that signature as parameters. Hence, the type of valuation functions is a dependent type. For example, the type of valuation function for expressions is:

$$\begin{aligned}\mathcal{E} : (\Sigma : \mathit{SIGS}) \to (H : \text{TENV}) \to \text{Expression} \to (\mathbf{A} : \mathit{Alg}(\Sigma))\\ \to \mathit{STATE}_H^{\mathbf{A}} \to \mathit{VALS}^{\mathbf{A}}{}_{\perp}\end{aligned}$$

The type of this $\mathcal{E}$ can be thought of as a function, which given a mutation signature, Σ, a type context H, an expression, a Σ-mutation algebra $\mathbf{A}$ returns a function from a state over $\mathbf{A}$ to the lifted domain of values of $\mathbf{A}$.

Though this type is more complicated than the usual valuation function for denotational semantics, to ensure a proper blend between the algebraic and denotational semantics this kind of parameterization is required. The notation $\mathcal{E}_\Sigma^H$ is used to denote this function applied to a signature, Σ, and a type context H. A similar convention is used for the other valuation functions.

For brevity, we avoid any distinction between lists and vectors in the following semantic functions. The function $\mathit{addToEnd}(\vec{v}, v)$ returns a vector after adding v to $\vec{v}$ at the end. We use $(\vec{v}, v)$ for $\mathit{addToEnd}(\vec{v}, v)$ as if they are equivalent.

4.3.2.1 Expressions and expression lists

As described above, the meaning of an expression is found by either looking up an identifier in the environment, or by using the algebra and store to evaluate an operation. The store is needed by algebras because all external operations take a store.

The semantic function for numerals, $\mathcal{N}$, has the following type.

$$\mathcal{N}_\Sigma : \text{Numeral} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to \mathit{VALS}_{\texttt{Store}}^{\mathbf{A}} \to (\mathit{VALS}^{\mathbf{A}})_\perp$$

The denotations for numerals should be dependent on the algebra. Details of $\mathcal{N}$ are trivial because one can use a series of operations calls to denote various integer numerals. That is, $\mathcal{N}_\Sigma\ [\![N]\!]\ \mathbf{A}\ \sigma = N^{\mathbf{A}}(\sigma)$.

The denotations for `true`, `false`, and `nothing` use the appropriate method in the algebra; hence these semantic equations are also independent of the particular algebra. Similarly, the meaning of a method call uses an operation in an algebra.

$\mathcal{E}^H_\Sigma$: Expression $\rightarrow (\mathbf{A} : \mathit{Alg}(\Sigma)) \rightarrow \mathit{STATE}^{\mathbf{A}}_H \rightarrow \mathit{VALS}^{\mathbf{A}} \times \mathit{VALS}^{\mathbf{A}}_{\mathtt{Store}\bot}$
$\mathcal{E}^H_\Sigma$ $[\![\mathrm{I}]\!]$ $\mathbf{A}$ $\eta =$ **let** $v = \eta[\![\mathrm{I}]\!]$ **in** $(v, (\eta\ \mathtt{store}))$
$\mathcal{E}^H_\Sigma$ $[\![\mathrm{N}]\!]$ $\mathbf{A}$ $\eta = \mathcal{N}_\Sigma\ [\![\mathrm{N}]\!]\ \mathbf{A}\ (\eta\ \mathtt{store})$
$\mathcal{E}^H_\Sigma$ $[\![\mathtt{true}]\!]$ $\mathbf{A}$ $\eta = \mathtt{true}^{\mathbf{A}}(\eta\ \mathtt{store})$
$\mathcal{E}^H_\Sigma$ $[\![\mathtt{false}]\!]$ $\mathbf{A}$ $\eta = \mathtt{false}^{\mathbf{A}}(\eta\ \mathtt{store})$
$\mathcal{E}^H_\Sigma$ $[\![\mathtt{nothing}]\!]$ $\mathbf{A}$ $\eta = \mathtt{nothing}^{\mathbf{A}}(\eta\ \mathtt{store})$
$\mathcal{E}^H_\Sigma$ $[\![\mathrm{g}(\vec{\mathrm{E}})]\!]$ $\mathbf{A}$ $\eta =$
    let $(\vec{v}, \sigma') = \mathcal{E}{*}^H_\Sigma\ [\![\vec{\mathrm{E}}]\!]\ \mathbf{A}\ \eta$ **in** $\mathrm{g}^{\mathbf{A}}(\vec{v}, \sigma')$

The `let` expressions used in the semantics are strict [Sch86]. That is, for a let expression `let` v `=` E `in` $\mathtt{E}_1$, if E is $\bot$, then the result of the let expression is $\bot$. For example, if $\vec{E}$ is $\bot$ then $g(\vec{E})$ is $\bot$.

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

$\mathcal{E}{*}^H_\Sigma$: Expression-List $\rightarrow (\mathbf{A} : \mathit{Alg}(\Sigma)) \rightarrow \mathit{STATE}^{\mathbf{A}}_H \rightarrow (\mathit{List}(\mathit{VALS}^{\mathbf{A}}) \times \mathit{VALS}^{\mathbf{A}}_{\mathtt{Store}})_\bot$
$\mathcal{E}{*}^H_\Sigma$ $[\![]\!]$ $\mathbf{A}$ $\eta = ((), (\eta\ \mathtt{Store}))$
$\mathcal{E}{*}^H_\Sigma$ $[\![\vec{\mathrm{E}}\mathrm{E}]\!]$ $\mathbf{A}$ $\eta =$
    let $(\vec{v}, \sigma_n) = \mathcal{E}{*}^H_\Sigma\ [\![\vec{\mathrm{E}}]\!]\ \mathbf{A}\ \eta$ **in**
    let $(v, \sigma') = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma_n]\eta$ **in**
    $((\vec{v}, v),\ \sigma')$

Since `let` is strict, if one of the expressions is $\bot$ then the result of the entire expression list, $\vec{E}$, is $\bot$.

4.3.2.2 Commands

The semantic functions for commands are straightforward. In an `if`-command, though the value of the test is used, since it is of a visible type, the semantics for `if`-command is independent of the algebra.

$\mathcal{C}^H_\Sigma$: Command $\rightarrow (\mathbf{A} : \mathit{Alg}(\Sigma)) \rightarrow \mathit{STATE}^{\mathbf{A}}_H \rightarrow (\mathit{VALS}^{\mathbf{A}}_{\mathtt{Store}})_\bot$
$\mathcal{C}^H_\Sigma$ $[\![\mathrm{E}]\!]$ $\mathbf{A}$ $\eta =$ **let** $(v, \sigma') = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ \eta$ **in** σ'
$\mathcal{C}^H_\Sigma$ $[\![\mathrm{C}_1;\ \mathrm{C}_2]\!]$ $\mathbf{A}$ $\eta =$ **let** $\sigma_1 = \mathcal{C}^H_\Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ \eta$ **in** $\mathcal{C}^H_\Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma_1]\eta$
$\mathcal{C}^H_\Sigma$ $[\![\mathtt{if}\ \mathrm{E}_1\ \mathtt{then}\ \mathrm{C}_1\ \mathtt{else}\ \mathrm{C}_2\ \mathtt{fi}]\!]$ $\mathbf{A}$ $\eta =$
    let $(v, \sigma') = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}_1]\!]\ \mathbf{A}\ \eta$ **in**
    if $(v \notin \mathit{VALS}^{\mathbf{A}}_{\mathtt{Bool}})$ **then** $\bot$
    else if v **then** $(\mathcal{C}^H_\Sigma\ [\![\mathrm{C}_1]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma']\eta)$ **else** $(\mathcal{C}^H_\Sigma\ [\![\mathrm{C}_2]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma']\eta)$

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

$\mathcal{D}^H_\Sigma$: Declaration $\rightarrow (\mathbf{A} : \mathit{Alg}(\Sigma)) \rightarrow (\mathit{STATE}^{\mathbf{A}}_H) \rightarrow \bigcup_{H':\mathrm{TENV}} \mathit{STATE}^{\mathbf{A}}_{H'\bot}$
$\mathcal{D}^H_\Sigma$ $[\![\;]\!]$ $\mathbf{A}$ $\eta = [\texttt{store} \mapsto (\eta\ \texttt{store})]\mathit{emptyEnviron}$
$\mathcal{D}^H_\Sigma$ $[\![\texttt{const}\ \mathrm{I}\!:\!\mathrm{T} = \mathrm{E}]\!]$ $\mathbf{A}$ $\eta =$
    let $T' = \mathcal{T}_\Sigma\ [\![\mathrm{T}]\!]$ **in**
    let $\widehat{(v, \sigma')} = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}]\!]\ \mathbf{A}\ \eta$ **in**
    if $v \notin \mathit{VALS}^{\mathbf{A}}_{T'}$ **then** $\bot$ **else** $[I \mapsto v][\texttt{store} \mapsto \sigma']\mathit{emptyEnviron}$
$\mathcal{D}^H_\Sigma$ $[\![D_1 ; D_2]\!]$ $\mathbf{A}$ $\eta =$
    let $\eta_1 = \mathcal{D}^H_\Sigma\ [\![D_1]\!]\ \mathbf{A}\ \eta$ **in**
    let $\eta_2 = \mathcal{D}^H_\Sigma\ [\![D_2]\!]\ \mathbf{A}\ (\eta \uplus \eta_1)$ **in**
    $\eta_1 \uplus \eta_2$

The auxiliary function $\mathcal{T}_\Sigma\ [\![\mathrm{T}]\!]$ is used to check if the declared type is in Σ and the (infix) auxiliary function $\uplus$ is used to combine environments. Note that the subscript $\mathbf{A}$ is omitted for $\uplus$.

$\mathcal{T} : (\Sigma : \mathit{SIGS}) \rightarrow$ Type-Name $\rightarrow$ Identifier$_\bot$
$\mathcal{T}_\Sigma\ [\![\mathrm{T}]\!] =$ **if** $T \in \mathrm{E}\mathit{TYPES}$ **then** T **else** $\bot$
$\uplus_{\mathbf{A}} : (\mathit{STATE}_H[\mathbf{A}] \times \mathit{STATE}_{H'}[\mathbf{A}] \rightarrow \mathit{STATE}_{H \cup H'}[\mathbf{A}])$
$\eta_1 \uplus \eta_2 = [\texttt{store} \mapsto (\eta_2\ \texttt{Store})](\eta_1 \cup \eta_2)$

4.3.2.4 Observation function Our observation functions are unique in the sense that they return both a value and a state. This is because we would like to check some properties of the returned state. These observations are obtained from the functions in the main programs of $\mathit{OBS}^{\leq}$.

Since the primary purpose of $\mathit{OBS}^{\leq}$ is to provide observations for client functions, before presenting the semantics of client functions we first define observations. A (Σ, H)-observation, of type $\mathit{OBSERVATION}^{\Sigma}_H$, is a function that takes a Σ-algebra, a H-state, and returns a value of visible type and the result state.

$$\mathit{OBSERVATION}^{\Sigma}_H = (\mathbf{A} : \mathit{Alg}(\Sigma)) \rightarrow \mathit{STATE}^{\mathbf{A}}_H \rightarrow (\mathit{VALS}^{\mathbf{A}} \times \mathit{STATE}^{\mathbf{A}}_H)$$

Given a Σ-mutation algebra and a state, an observation returns a value and a state. We refer to the value in the result of $\mathit{OBSERVATION}^{\Sigma}_H$ as the result of the observation and the state as the resulting state of the observation. An observation can be thought of as a frozen computation with respect to a signature Σ and a type context, H, over that signature, which given a Σ-mutation algebra and a state returns a result and possibly a new state.

The denotation of a client function returns a type context, H and an observation. The type context determines the kinds of states that can be passed to the observation.

```
$\mathcal{FN} : (\Sigma : \mathit{SIGS}) \to \text{Client-Function} \to ((\text{H:TENV}) \times \mathit{OBSERVATION}^{\Sigma}_{H})_{\perp}$
$\mathcal{FN}_{\Sigma}$ $[\![\texttt{client function}\ \text{FN}(\vec{F})\text{:T}\ \texttt{is}\ \ \text{D}\ \texttt{do}\ \ \text{C}\ \texttt{return}\ \ \text{E}]\!] =$
        $\textbf{let}\ H' = \mathcal{F}*_{\Sigma}\ [\![\text{F}]\!]\ \textbf{in}$
        $\textbf{let}\ H = H' \cup \{(\texttt{store}, \texttt{Store})\}\ \textbf{in}$
        $\textbf{let}\ f\ = (\lambda\mathbf{C}\,.\,(\lambda s\,.$
          $\textbf{if}\ \eta \notin \mathit{STATE}^{\mathbf{C}}_{H}\ \textbf{then}\ \perp$
          $\textbf{else let}\ \eta' = \mathcal{D}^{H}_{\Sigma}\ [\![\text{D}]\!]\ \mathbf{C}\ \eta\ \textbf{in}$
            $\textbf{let}\ \sigma' = \mathcal{C}^{H}_{\Sigma}\ [\![\text{C}]\!]\ \mathbf{C}\ (\eta \uplus \eta')\ \textbf{in}$
            $\textbf{let}\ (v, \sigma'') = \mathcal{E}^{H}_{\Sigma}\ [\![\text{E}]\!]\ \mathbf{C}\ [\texttt{store} \mapsto \sigma'](\eta \uplus \eta')\ \textbf{in}$
            $(v, [\texttt{store} \mapsto \sigma''](\eta \uplus \eta'))$
        $\textbf{in}\ (H, f)$
```

The auxiliary function $\mathcal{F}*_{\Sigma}\ [\![\text{F}]\!]$ returns a list of identifier and type pairs. The function *map* takes a function, f, and a list, l, and returns a list that contains the result of applying f to each element of l in order.

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

4.3.2.5 Main programs

The meaning of a main program is a tuple consisting of a type environment H, an H-observation function, and a H-state. The type declaration and observation function is obtained from the denotation of the client function as described above. A tuple of values that are obtained from the call expression is bound to the actuals of the client function. This environment along with the store produced by the first part of the main program consists of the third component of the denotation of a main program. This environment can be non-nominal, that is can contain subtyping.

```
$\mathcal{M}_{\Sigma} : \text{Main} \to (\mathbf{A} : \mathit{Alg}(\Sigma)) \to ((\text{H:TENV}) \times \mathit{OBSERVATION}^{\Sigma}_{H} \times \mathit{STATE}^{\mathbf{A}}_{H})$
$\mathcal{M}^{H}_{\Sigma}\ [\![\texttt{observe}\ \text{FN}\ \texttt{by}\ \text{D}\ \texttt{do}\ \text{C};\ \texttt{call}\ \text{I}(\vec{E})]\!]\ \mathbf{A} =$
        $\textbf{let}\ (H, f) = \mathcal{FN}_{\Sigma}\ [\![FN]\!]\ \textbf{in}$
        $\textbf{let}\ \eta = \mathcal{D}^{H}_{\Sigma}\ [\![D*]\!]\ \mathbf{A}\ [\texttt{store} \mapsto \texttt{emptyStore}]\mathit{emptyEnviron}\ \textbf{in}$
        $\textbf{let}\ \sigma' = \mathcal{C}^{H}_{\Sigma}\ [\![C]\!]\ \mathbf{A}\ \eta\ \textbf{in}$
        $\textbf{let}\ (\vec{v}, \sigma'') = \mathcal{E}*^{H}_{\Sigma}\ [\![\vec{\text{E}}]\!]\ \mathbf{A}\ [\texttt{store} \mapsto \sigma']\eta\ \textbf{in}$
```

```
    **let** $\eta'' = \mathit{bindActuals}[\mathbf{A}]\ (\vec{v}, \sigma'')\ H$ **in**
    $(H, f, \eta'')$
```

Note that the environment returned by a main program does not contain the identifiers declared $[\![D]\!]$. This means the identifiers used in observations, that is in the body of the client functions, are either the formal identifiers or the locally defined identifiers.

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

```
$\mathit{bindActuals}[\mathbf{A}] : (\mathit{VALS}^{\mathbf{A}})^* \to \mathit{List}(\text{Identifier} \times \text{Type-Name}) \to \mathit{Env}[\mathbf{A}]_\perp$
$\mathit{bindActuals}[\mathbf{A}]\ \vec{v}\ \hat{F} =$
    **let** $\vec{S} = (\mathit{formalTypes}\ \hat{F})$ **in**
    **if** $\vec{v} \notin \widehat{\mathit{VALS}}^{\mathbf{A}}_{\vec{S}}$ **then** $\perp$
    **else let** $(\eta', n) =$
            $(\mathit{foldright}$
                $(\lambda((I_i, S_i), (\eta, i)) \,.\, ([I_i \mapsto (\vec{v} \downarrow i)]\eta,\ i \Leftrightarrow 1))$
                $(\mathit{emptyEnviron},\ \mathit{length}\ \hat{F})$
                $\hat{F})$
        **in** $\eta'$
$\mathit{formalTypes} : \mathit{List}(\text{Identifier} \times \text{Type-Name}) \to \mathit{List}(\text{Type-Name})$
$\mathit{formalTypes} F = \mathit{map}(\lambda(I, T).T)F$
```

4.4 Soundness of alias and type checking rules

In this section, we prove a theorem (4.4.6), which states that the alias and type checking rules are sound. We first present a series of lemmas and then present the theorem and its proof. The significance of this theorem is that if a mutation algebra preserves alias legality with respect to an alias constraint, L, then main programs in $\mathit{OBS}^{\leq}$ satisfy the alias constraints required for sound reasoning with weak behavioral subtyping.

If the result of the main program is a $\perp$, then requiring that the result needs to satisfy *stAliasOk* becomes moot. Hence, the lemmas and the theorem in this section do not consider the case when $\perp$ is a possible result.

In all the following lemmas $\mathbf{A}$ refers to an $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra, L refers to an alias context for $\mathrm{E}\Sigma$, and H refers to a type environment. For clarity, we use $[I \mapsto v]\eta$ for its equivalent $[I \mapsto v][I \mapsto v']\eta$, especially while mapping `store` in the environments.

Lemma 4.4.1 *Let E be an expression in the main program of* OBS$^{\leq}$ *(as in Figure 4.2.) Let η be a valid H-environment over* $\mathbf{A}$*. If* $\mathbf{A}$ *preserves alias legality with respect to L, $stAliasOk(H, \mathbf{A}, \eta)$, and $(v, \sigma') = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{A}\ \eta$ for $v \in VALS^{\mathbf{A}}$, then*

$$\begin{array}{l} \mathrm{E}\Sigma; H; L \vdash E : T :: r \\ \quad \Rightarrow \\ \quad (stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta) \wedge (aliasTypeSet(H, \mathbf{A}, v, [\texttt{store} \mapsto \sigma']\eta) \subseteq r)) \end{array}$$

Proof: This is shown by induction on the structure of E.
Basis: For the expressions N, `nothing`, `true`, and `false`, since $\sigma' = \sigma$ and since the alias type set for non-object types is $\{\}$, the conclusion is satisfied trivially.
Inductive step: Let E be $g(\vec{E})$. Assume the conclusion holds true for all subexpressions of E. Suppose $\mathrm{E}\Sigma; H; L \vdash \mathrm{g}(\vec{E}) : T :: r$.

Let $(\vec{v}, \sigma_n) = \mathcal{E}*^H_\Sigma\ [\![\vec{\mathrm{E}}]\!]\ \mathbf{A}\ \eta$ and let $(v, \sigma') = g^{\mathbf{A}}(\vec{v}, \sigma_n)$.

We need to show that

$$stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta)\ \wedge\ (aliasTypeSet(H, \mathbf{A}, v, [\texttt{store} \mapsto \sigma']\eta) \subseteq r).$$

It suffices to show that $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_n]\eta)$ because of the following reason. Since $\mathrm{E}\Sigma; H; L \vdash \mathrm{g}(\vec{E}) : T :: r$, and since $\mathbf{A}$ satisfies alias legality with respect to L, if $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_n]\eta)$ holds then from the alias legality condition the conclusion of the lemma holds.

$$stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta) \wedge (aliasTypeSet(H, \mathbf{A}, v, [\texttt{store} \mapsto \sigma']\eta) \subseteq r)$$

So we need to show that $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_n]\eta)$. We show this by induction on the length of $\vec{E}$.

When the length is 0, then $\sigma_n = \sigma$ and $[\texttt{store} \mapsto \sigma_n]\eta = \eta$, then the condition $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_n]\eta)$ holds trivially.

Assume that $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_i]\eta)$ is true for any i less than n. Let $\vec{E} = (\vec{E'}, E)$, where $\mid \vec{E'} \mid = i$. Let $(v_i, \sigma_i) = \mathcal{E}*^H_\Sigma\ [\![\vec{E'}]\!]\ \mathbf{A}\ \eta$ and $\Sigma; H; L \vdash \vec{E'} : \vec{S'} :: \vec{r'}$. Let $(v_{i+1}, \sigma_{i+1}) = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{A}\ [\texttt{store} \mapsto \sigma_i]\eta$ and $\Sigma; H; L \vdash E : S_{i+1} :: r_{i+1}$.

The following calculation, based on the denotations of $\vec{E'}$ and E of the expression $(\vec{E'}\ E)$, shows that $stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_{i+1}]\eta)$. The first clause in the calculation is true from the hypothesis of the lemma.

$$\begin{array}{ll} & stAliasOk(H, \mathbf{A}, \eta) \\ \Rightarrow & \langle \text{by applying the induction hypothesis for } i < n \rangle \\ & stAliasOk(H, \mathbf{A}, [\texttt{store} \mapsto \sigma_i]\eta)\ \wedge\ aliasTypeSet(H, \mathbf{A}, v_m, [\texttt{store} \mapsto \sigma_i]\eta) \subseteq r_i \\ \Rightarrow & \langle \text{by applying induction hypothesis on } E \rangle \end{array}$$

$$\begin{array}{ll} & \mathit{stAliasOk}(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma_{i+1}]\eta) \\ & \qquad \wedge\ (\mathit{aliasTypeSet}(H, \mathbf{A}, v, [\mathtt{store} \mapsto \sigma_{i+1}]\eta) \subseteq r_{i+1}) \\ \Rightarrow & \langle \text{by predicate calculus} \rangle \\ & \mathit{stAliasOk}(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma_{i+1}]\eta) \end{array}$$

∎

Lemma 4.4.2 *Let D be declarations of main program of* OBS$^{\leq}$. *Let η be a H-state over $\mathbf{A}$. If $\mathbf{A}$ preserves alias legality, $\mathit{stAliasOk}(H, \mathbf{A}, \eta)$, and $\eta' = \mathcal{D}^H_\Sigma\ [\![D]\!]\ \mathbf{A}\ \eta$ then*

$$(\Sigma; H; L \vdash D \Longrightarrow H') \Rightarrow \mathit{stAliasOk}((H \cup H'), \mathbf{A}, (\eta \uplus \eta')) \tag{4.3}$$

Proof: This is shown by induction on the structure of D.

- If D is empty, then Equation 4.3 is trivially true because from the semantics of declarations, $(\eta \uplus \eta') = \eta$.

- If $D = \mathtt{const}\ I{:}T\ \mathtt{:=}\ E$. Let $(v, \sigma') = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{A}\ \eta$ and $\Sigma; H; L \vdash E : S :: r$ for some $S \leq T$. Then $H' = \{(I, T), (\mathtt{store}, \mathtt{Store})\}$ and the resulting environment $\eta' = [I \mapsto v][\mathtt{store} \mapsto \sigma']\mathit{emptyEnviron}$. Hence, $(\eta \uplus \eta') = (\eta \uplus [I \mapsto v][\mathtt{store} \mapsto \sigma']\mathit{emptyEnviron})$.

  Let's assume that Equation 4.3 does not hold in this case. But from the Lemma 4.4.1, we know that

  $$\mathit{stAliasOk}(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma']\eta) \wedge \mathit{aliasTypeSet}(H, \mathbf{A}, v, [\mathtt{store} \mapsto \sigma']\eta) \subseteq r \tag{4.4}$$

  From Equation 4.4 and from the definition of *stAliasOk*, we can conclude that $\mathit{storeAliasOk}^W(\mathbf{A}, \sigma')$ is true. The only case when $\mathit{stAliasOk}((H \cup H'), \mathbf{A}, (\eta \uplus \eta'))$ can be false is if $\mathit{aliasTypeSet}(H, \mathbf{A}, v, [\mathtt{store} \mapsto \sigma]\eta)$ contains a U such that $U \neq T$.

  But from Equation 4.4, $\mathit{aliasTypeSet}(H, \mathbf{A}, v, [\mathtt{store} \mapsto \sigma']\eta) \subseteq r$. Hence $\{U\} \subseteq r$. Also, from the type rule [SDECL], we have $r \subseteq \{T\}$. That is, $U = T$. This contradicts our assumption that $U \neq T$, hence Equation 4.3 holds.

- If $D = D_1 ; D_2$. The induction hypothesis is that Equation 4.3 holds for all substructures of D.

  Let $\Sigma; H; L \vdash D_1 \Longrightarrow H_1$ and $\Sigma; (H \cup H_1); L \vdash D_2 \Longrightarrow H_2$. Let $\eta_1 = \mathcal{D}^H_\Sigma\ [\![D_1]\!]\ \mathbf{A}\ \eta$ and $\eta_2 = \mathcal{D}^H_\Sigma\ [\![D_2]\!]\ \mathbf{A}\ (\eta \uplus \eta_1)$. Then from the semantics of declarations, $H' = H_1 \cup H_2$ and $\eta' = (\eta_1 \uplus \eta_2)$. The following calculation shows the desired conclusion.

$$\begin{array}{ll}
 & \mathit{stAliasOk}(H, \mathbf{A}, \eta) \\
\Rightarrow & \langle \text{by the induction hypothesis on } D_1 \rangle \\
 & \mathit{stAliasOk}((H \cup H_1), \mathbf{A}, (\eta \uplus \eta_1)) \\
\Rightarrow & \langle \text{by the induction hypothesis on } D_2 \rangle \\
 & \mathit{stAliasOk}(((H \cup H_1) \cup H_2), \mathbf{A}, ((\eta \uplus \eta_1) \uplus \eta_2)) \\
= & \langle \text{by associativity of } \uplus \text{ and } \cup \rangle \\
 & \mathit{stAliasOk}((H \cup (H_1 \cup H_2)), \mathbf{A}, (\eta \uplus (\eta_1 \uplus \eta_2))) \\
= & \langle \text{by semantics of declarations} \rangle \\
 & \mathit{stAliasOk}((H \cup H'), \mathbf{A}, (\eta \uplus \eta'))
\end{array}$$

∎

Lemma 4.4.3 *Let D be declarations of main program of* OBS$^{\leq}$. *Let η be a H-state over* $\mathbf{A}$. *If* $\mathbf{A}$ *preserves alias legality, $\mathit{stAliasOk}(H, \mathbf{A}, \eta)$, and $\eta' = \mathcal{D}^H_\Sigma \; [\![D]\!] \; \mathbf{A} \; \eta$ then*

$$(\Sigma; H; L \vdash D \Longrightarrow H') \Rightarrow \mathit{stAliasOk}(H', \mathbf{A}, \eta') \tag{4.5}$$

Proof: From Lemma 4.4.2, we have $\mathit{stAliasOk}((H \cup H'), \mathbf{A}, (\eta \uplus \eta'))$. From the definition of *stAliasOk*, we can conclude that $\mathit{stAliasOk}(H', \mathbf{A}, \eta')$.

If that is not the case, since the store is same in both cases, there exists a $x \in \mathit{Domain}(H')$ such that $\mathit{aliasTypeSet}(H', \mathbf{A}, (\eta' \; x), \eta')$ contains more than one type. But from the definition of $\uplus$ and since $H \Leftrightarrow \{\texttt{Store}\}$ and $H' \Leftrightarrow \{\texttt{Store}\}$ are disjoint, we have for any $y \in \mathit{Domain}(H')$ $(\eta' \; y) = ((\eta \uplus \eta') \; y)$. So, if $\mathit{aliasTypeSet}(H', \mathbf{A}, (\eta' \; x), \eta')$ contains more than one type then $\mathit{aliasTypeSet}((H \cup H'), \mathbf{A}, (\eta' \; x), (\eta \uplus \eta'))$ also contains more than one type, which contradicts our assumption. Hence, we can conclude that $\mathit{stAliasOk}(H', \mathbf{A}, \eta')$. ∎

Lemma 4.4.4 *Let C be a command that can be used in main program of* OBS$^{\leq}$. *Let η be a H-state over* $\mathbf{A}$. *If* $\mathbf{A}$ *preserves alias legality, $\mathit{stAliasOk}(H, \mathbf{A}, \eta)$, and $\sigma' = \mathcal{C}^H_\Sigma \; [\![C]\!] \; \mathbf{A} \; \eta$ then*

$$(\Sigma; H; L \vdash C \; \surd) \Rightarrow \mathit{stAliasOk}(H, \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta) \tag{4.6}$$

Proof: (sketch)(induction on the structure of C) The base case, when C is E follows from Lemma 4.4.1.

The inductive case, when C is C_1;C_2, follows by applying the inductive hypothesis on C_1 and then on C_2. The other inductive case, when C is `if` E `then` C_1 `else` C_2 follows by using the lemma on expressions and the induction hypothesis. The first when E is true, applying Lemma 4.4.1 on E, the resulting state is *stAliasOk*. Then by induction hypothesis applied to either C_1 or C_2 depending on the result of E, we can conclude that the resulting state is *stAliasOk*. ∎

The following lemma shows that the call to the client function in a main program satisfies the alias constraints.

Lemma 4.4.5 *Let* $P =$ `observe client function` $\mathtt{I}(\vec{I}{:}\vec{T}){:}T$ `is` B `by` D `do` C; `call` $I(\vec{E})$ *be a main program of* $\text{OBS}^{\leq}$ *such that* $\Sigma; L \vdash P \surd$.

Let η *be a valid* H*-environment over* $\mathbf{A}$. *If* $\mathbf{A}$ *preserves alias legality with respect to* L, *stAliasOk*$(H, \mathbf{A}, \eta)$, *and* $(\vec{v}, \sigma_n) = \mathcal{E}{*}^{H}_{\Sigma}\ [\![\vec{E}]\!]\ \mathbf{A}\ \eta$ *then,*

1. *stAliasOk*$(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma_n]\eta)$ *and*
2. *stAliasOk*$([\vec{I} \mapsto \vec{T}], \mathbf{A}, [\vec{I} \mapsto \vec{v}][\mathtt{store} \mapsto \sigma_n]\mathit{emptyEnviron})$

Proof: (1)(sketch) This is shown by induction on the structure of $\vec{E}$. The base case is when $\vec{E}$ is empty, then $\sigma_n = (\eta\ \mathtt{store})$, so the conclusion holds trivially.

The inductive case is when $\vec{E} = (\vec{E'}, E)$. This is shown by applying the inductive hypothesis on $\vec{E'}$ to obtain, *stAliasOk*$(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma_{n-1}]\eta)$, where σ_{n-1} is the resulting store of $\mathcal{E}{*}^{H}_{\Sigma}\ [\![\vec{E'}]\!]\ \mathbf{A}\ \eta$. Then applying Lemma 4.4.1 to E, we can conclude *stAliasOk*$(H, \mathbf{A}, [\mathtt{store} \mapsto \sigma_n]\eta)$.

(2) We need to show

$$\mathit{stAliasOk}([\vec{I} \mapsto \vec{T}], \mathbf{A}, [\vec{I} \mapsto \vec{v}][\mathtt{store} \mapsto \sigma_n]\mathit{emptyEnviron}) \tag{4.7}$$

From (1), we know that $\mathit{storeAliasOk}^{W}(\mathbf{A}, \sigma_n)$. So it suffices to show that there is no aliasing between the identifiers $\vec{I}$ that violate *stAliasOk*.

If $|\ \vec{I}\ | \leq\ 1$, then Equation 4.7 is trivially satisfied.

If $|\ \vec{I}\ | >\ 1$ then let us assume that Equation 4.7 does not hold. That means, for $T_i \neq T_j$ such that $T_i, T_j \in \mathit{ObjectTypes}$, there exists an $I_i : T_i$ and an $I_j : T_j$, such that the corresponding values denoted by them v_i and v_j are equal for some $i \neq j$.

Let E_i and E_j be the expressions such that $(v_i, \sigma_i) = \mathcal{E}^{H}_{\Sigma}\ [\![E_i]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma_{i-1}]\eta$ and $(v_j, \sigma_j) = \mathcal{E}^{H}_{\Sigma}\ [\![E_j]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma_{j-1}]\eta$.

Without loss of generality, assume that $i < j$. From the typing rule [Elist] for $(\vec{E_{i-1}}, E_i)$ we know that $\Sigma; H; L \vdash E_i : T_i :: r_i$. But from the typing rule [Main],

$r_i = \{\}$. Applying Lemma 4.4.1 on E_i, we have $aliasTypeSet(H, \mathbf{A}, v_i, [\texttt{store} \mapsto \sigma_i]\eta) = \{\}$. That is, v_i is not reachable in $[\texttt{store} \mapsto \sigma_i]\eta$.

Similarly applying the typing rules and the Lemma 4.4.1 for E_j, we can conclude that $aliasTypeSet(H, \mathbf{A}, v_j, [\texttt{store} \mapsto \sigma_j]\eta) = \{\}$. That is, v_j is not reachable in $[\texttt{store} \mapsto \sigma_j]\eta$.

Since $v_j = v_i$, v_i is reachable in $[\texttt{store} \mapsto \sigma_j]\eta$ through a path in the $\texttt{aliasG}(\sigma_j)$ as accessed by E_j. So before the denotation of E_i, v_i is not reachable, and after the denotation of E_i, at some point v_i is reachable. This can only happen if v_i is bound either by E_i or by some other expression after E_i. But in such a case the alias type set of v_j in $[\texttt{store} \mapsto \sigma_i]\eta$ will not be $\{\}$. This contradicts our earlier conclusion that v_j is not reachable in $[\texttt{store} \mapsto \sigma_i]\eta$. Hence our assumption that such a v_i and v_j exist is wrong.

So we can conclude $stAliasOk([\vec{x} \mapsto \vec{T}], \mathbf{A}, [\vec{x} \mapsto \vec{T}][\texttt{store} \mapsto \sigma_n]emptyEnviron)$.
∎

Theorem 4.4.6 *Let $(I\Sigma, E\Sigma)$ be a mutation signature and let L be an alias context for $(I\Sigma, E\Sigma)$. Let P be a main program of* $\text{OBS}^{\leq}$. *Let $\mathbf{A}$ be an $(I\Sigma, E\Sigma)$-mutation algebra. If $\mathbf{A}$ preserves alias legality with respect to L and $(H, f, \eta) = \mathcal{M}_{\Sigma}^{H}\ [\![P]\!]\ \mathbf{A}$, then $(E\Sigma; L \vdash P\ \surd) \Rightarrow stAliasOk(H, \mathbf{A}, \eta)$.*

Proof: Let P = `observe client function I(`$\vec{I}$`:`$\vec{T}$`):`T `is` B `by` D `do` C; `call I(`$\vec{E}$`)` such that $\Sigma; L \vdash P\ \surd$. Let $(H, f, \eta) = \mathcal{M}_{\Sigma}^{H}\ [\![P]\!]\ \mathbf{A}$.

Let $\Sigma; \{\}; L \vdash D \Longrightarrow H'$. Let

$$\eta_1 = \mathcal{D}_{\Sigma}^{H}\ [\![D]\!]\mathbf{A}[\texttt{store} \mapsto \texttt{emptyStore}]emptyEnviron,$$

$\sigma' = \mathcal{C}_{\Sigma}^{H}\ [\![C]\!]\ \mathbf{A}\ \eta_1$, and $(\vec{v}, \sigma'') = \mathcal{E}*_{\Sigma}^{H}\ [\![\vec{E}]\!]\ \mathbf{A}\ [\texttt{store} \mapsto \sigma']\eta_1$. From the denotation of P, $\eta = [\vec{I} \mapsto \vec{v}][\texttt{store} \mapsto \sigma'']emptyEnviron$.

$$\begin{array}{ll} & stAliasOk(\{\}, \mathbf{A}, [\texttt{store} \mapsto \texttt{emptyStore}]emptyEnviron) \\ \Rightarrow & \langle \text{by Lemma 4.4.3} \rangle \\ & stAliasOk(H', \mathbf{A}, \eta_1) \\ \Rightarrow & \langle \text{by Lemma 4.4.4} \rangle \\ & stAliasOk(H', \mathbf{A}, [\texttt{store} \mapsto \sigma']\eta_1) \\ \Rightarrow & \langle \text{by (2) of Lemma 4.4.5} \rangle \\ & stAliasOk([\vec{I} \mapsto \vec{T}], \mathbf{A}, [\vec{I} \mapsto \vec{v}][\texttt{store} \mapsto \sigma'']emptyEnviron). \end{array}$$

Hence, $stAliasOk(H, \mathbf{A}, \eta)$ is true. ∎

5. BEHAVIORAL SUBTYPING MEANS NO SURPRISES

In this chapter, we first prove some general properties of $\mathit{OBS}^{\leq}$ and simulations, define expected results for weak and strong behavioral subtyping, and prove the "no surprises" theorems for both strong and weak behavioral subtyping. Recall that proving "no surprises" ensures the soundness of supertype abstraction in reasoning, by showing that the set of results of programs when subtypes are used in place of supertypes is an expected one. The expected set is the set of results when only the nominal type, that is no subtyping, objects are used in programs. Since the "no surprises" result is shown for main programs of $\mathit{OBS}^{\leq}$, whenever we refer to $\mathit{OBS}^{\leq}$, we mean the main program parts of $\mathit{OBS}^{\leq}$.

5.1 Properties of simulation relations

In this section we show that simulation relations are preserved by expressions, declarations, and commands in $\mathit{OBS}^{\leq}$.

Since we did not present a separate language for weak and strong behavioral subtyping, the type and alias checking rules in Figure 4.7 are interpreted just as type checking rules by ignoring the alias checks, for reasoning in the context of strong behavioral subtyping. For example, the notation $\Sigma, H; L \vdash \mathrm{E} : T :: r$ is just interpreted as $\Sigma, H \vdash \mathrm{E} : T$. Such a type system would allow all forms of aliasing and does not require algebras to satisfy alias legality.

Since the following lemmas show that various constructs of $\mathit{OBS}^{\leq}$ preserve simulation relations, we do not use the alias context L in the type rules. That is, all the lemmas in this section can be applied to both weak and strong behavioral subtype relations.

The following lemma says that simulation relations are preserved by expressions of $\mathit{OBS}^{\leq}$. Recall that **let** is used as a strict binding mechanism in the semantics.

Lemma 5.1.1 *Let $(I\Sigma', E\Sigma)$ and $(I\Sigma'', E\Sigma)$ be mutation signatures. Let $\mathbf{C}$ be a $(I\Sigma', E\Sigma)$-mutation algebra and $\mathbf{A}$ be a $(I\Sigma'', \Sigma)$-mutation algebra. Let $\mathcal{R}$ be a $E\Sigma$-simulation relation from $\mathbf{C}$ to $\mathbf{A}$.*

Let H be a type environment and $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{A}}$ be valid H-states. Then for each expression E in $\mathrm{OBS}^{\leq}$ *such that $E\Sigma; H \vdash E : T$, for each $y : T \notin H$,*

$$\begin{array}{l} \eta_{\mathbf{C}} \, \mathcal{R}_H \, \eta_{\mathbf{A}} \\ \Rightarrow \\ \quad (\mathbf{let}\ (v_{\mathbf{C}}, \sigma'_{\mathbf{C}}) = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{C}\ \eta_{\mathbf{C}}\ \mathbf{in}\ [y \mapsto v_{\mathbf{C}}][\mathtt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C}}) \\ \quad \mathcal{R}_{[y \mapsto T][\mathtt{store} \mapsto \mathtt{Store}]H} \\ \quad (\mathbf{let}\ (v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{A}\ \eta_{\mathbf{A}}\ \mathbf{in}\ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}) \end{array}$$

Proof: (by induction on the structure of E.) Let $\eta_{\mathbf{C}}$, and $\eta_{\mathbf{A}}$ be valid H-states over $\mathbf{C}$ and $\mathbf{A}$ respectively. Let $T \in \mathit{TYPES}$ and $E : T$ such that $\Sigma; H; \vdash E : T$.

Suppose that

$$\eta_{\mathbf{C}} \, \mathcal{R}_H \, \eta_{\mathbf{A}} \tag{5.1}$$

Basis: Suppose that E is either a numeral `N`, `true`, or `false`. In that case, since $\sigma'_{\mathbf{C}} = \sigma_{\mathbf{C}}$ and $\sigma'_{\mathbf{A}} = \sigma_{\mathbf{A}}$. Further, from the semantics, we have $v_{\mathbf{C}} = v_{\mathbf{A}}$. If E is `nothing`, then from then the result follows from the substitution property. If E is an identifier, then the result follows from the hypothesis (Equation 5.1) and the bindable property.

Inductive Step: Suppose that E has the form $g(\vec{E})$. Since $g(\vec{E})$ has a type T, it must be that $g : (\vec{S}, \mathtt{Store}) \to (T, \mathtt{Store})$, for some $\vec{S} \in \mathit{TYPES}$.

If the result of executing $g\mathbf{C}(\vec{E})$ in $\eta_{\mathbf{C}}$ or $g(\vec{E})$ in $\eta_{\mathbf{A}}$ is $\perp$, then from the substitution property the conclusion of the lemma is true.

When the results are proper, let $(v_{\mathbf{C}}, \sigma'_{\mathbf{C}}) = \mathcal{E}^H_\Sigma\ [\![\mathrm{g}(\vec{E})]\!]\ \mathbf{C}\ \eta_{\mathbf{C}}$ and $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}^H_\Sigma\ [\![\mathrm{g}(\vec{E})]\!]\ \mathbf{A}\ \eta_{\mathbf{A}}$

The inductive hypothesis is that the lemma is true for each subexpression, E_i of type S_i of $\vec{E}$. So, for fresh identifiers $z_i : S_i$, if $H' = H \cup \{(z_0, S_0), \cdots, (z_{i-1}, S_{i-1})\}$ then for all H'-states, $\eta_{\mathbf{C},i-1}$ and $\eta_{\mathbf{A},i-1}$, over $\mathbf{C}$ and $\mathbf{A}$ such that for each type S_i and for each expression E_i of type S_i, if $(v_{\mathbf{C},i}, \sigma_{\mathbf{C},i}) = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}_i]\!]\ \mathbf{C}\ \eta_{\mathbf{C},i-1}$ and $(v_{\mathbf{A},i}, \sigma_{\mathbf{A},i}) = \mathcal{E}^H_\Sigma\ [\![\mathrm{E}_i]\!]\ \mathbf{A}\ \eta_{\mathbf{A},i-1}$ then

$$\eta_{\mathbf{C},i-1} \, \mathcal{R}_{H'} \, \eta_{\mathbf{A},i-1}$$

The plan is to use the induction hypothesis for each expression $\vec{E}$, for each i, binding the result of E_i to z_i. Using this we construct new states $\eta_{\mathbf{C},i}$ and $\eta_{\mathbf{A},i}$ corresponding to $\vec{E}$, such that the following hold for each $1 \leq i \leq n$:

$$(\vec{v}_{\mathbf{C},n}, \sigma_{\mathbf{C},n}) = \mathcal{E}{*}^H_\Sigma\ [\![\vec{\mathrm{E}}]\!]\ \mathbf{C}\ (\eta_{\mathbf{C}}, \sigma\mathbf{C}) \tag{5.2}$$

$$(\eta_{\mathbf{C},i}\ z_i) = v_{\mathbf{C},i} \tag{5.3}$$

$$(\vec{v}_{\mathbf{A},n}, \sigma_{\mathbf{A},n}) \quad = \quad \mathcal{E}{*}^{H}_{\Sigma}\ [\![\vec{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta_{\mathbf{A}}, \sigma\mathbf{A}) \tag{5.4}$$

$$(\eta_{\mathbf{A},i}\ z_i) \quad = \quad v_{\mathbf{A},i} \tag{5.5}$$

$$[\mathtt{store} \mapsto \sigma_{\mathbf{C},n}]\eta_{\mathbf{C},n} \quad \mathcal{R}_{[\vec{z_i} \mapsto \vec{S_i}][\mathtt{store} \mapsto \mathtt{Store}]H} \quad [\mathtt{store} \mapsto \sigma_{\mathbf{A},n}]\eta_{\mathbf{A},n} \tag{5.6}$$

If equation 5.6 holds, the following calculation shows the inductive step for $g(\vec{E})$. Let $(v_{\mathbf{C}}, \sigma'_{\mathbf{C}}) = \mathrm{g}^{\mathbf{C}}(\vec{v}, \sigma_{\mathbf{C},n})$ and $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathrm{g}^{\mathbf{A}}(\vec{v}, \sigma_{\mathbf{A},n})$. Let $H' = [\vec{z_1} \mapsto \vec{S_1}] \cdots [z_n \mapsto S_n][\mathtt{store} \mapsto \mathtt{Store}]H$.

$$\begin{array}{ll}
 & [\mathtt{store} \mapsto \sigma_{\mathbf{C},n}]\eta_{\mathbf{C},n}\ \mathcal{R}_{H'}\ [\mathtt{store} \mapsto \sigma_{\mathbf{A},n}]\eta_{\mathbf{A},n} \\
\Rightarrow & \langle \text{by the substitution property, for some } y : T \rangle \\
 & [y \mapsto v_{\mathbf{C}}][\mathtt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C},n}\ \mathcal{R}_{[y \mapsto T][\mathtt{store} \mapsto \mathtt{Store}]H'}\ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A},n} \\
\Rightarrow & \langle \text{by the shrinkable property} \rangle \\
 & [y \mapsto v_{\mathbf{C}}][\mathtt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C}}\ \mathcal{R}_{[y \mapsto T][\mathtt{store} \mapsto \mathtt{Store}]H}\ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}
\end{array}$$

It remains to construct the related states $\eta_{\mathbf{C},n}\ \mathcal{R}_{H''}\ \eta_{\mathbf{A},n}$, where $H'' = [z_i \mapsto S_i]H$ and the vectors $\vec{v_{\mathbf{C}}}$ and $\vec{v_{\mathbf{A}}}$. These states are constructed by induction on number of arguments n, that is the length of $\vec{\mathrm{E}}$. If any of the E evaluates to $\perp$ in $\mathbf{C}$, then by the substitution property of simulations it evaluates to $\perp$ in $\mathbf{A}$ and hence satisfies our condition.

For the basis, if $n = 0$, then $\vec{E}$ is empty and we let $\eta_{\mathbf{C},0} = \eta_{\mathbf{C}}$, $\eta_{\mathbf{A},0} = \eta_{\mathbf{A}}$, $\vec{v_{\mathbf{C}}} = ()$, and $\vec{v_{\mathbf{A}}} = ()$. The required properties hold trivially.

For the inductive step, suppose that $\vec{E}$ is $E_1, \cdots, E_{j-1}, E_j$, and assume inductively that for $j > 0$ $\eta_{\mathbf{C},j-1}$, $\eta_{\mathbf{A},j-1}$, $v_{\mathbf{C},\vec{j}-1}$, and $v_{\mathbf{A},\vec{j}-1}$ satisfy the required properties.

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

$$(v_{\mathbf{C},j}, \sigma_{\mathbf{C},j}) \stackrel{\mathrm{def}}{=} \mathcal{E}^{H}_{\Sigma}\ [\![E_j]\!]\ \mathbf{C}\ \eta_{\mathbf{C},j-1} \tag{5.7}$$

$$(v_{\mathbf{A},j}, \sigma_{\mathbf{A},j}) \stackrel{\mathrm{def}}{=} \mathcal{E}^{H}_{\Sigma}\ [\![E_j]\!]\ \mathbf{A}\ \eta_{\mathbf{A},j-1} \tag{5.8}$$

We define the required environments and lists as follows.

$$\vec{v}_{\mathbf{C},j} \stackrel{\mathrm{def}}{=} (\vec{v}_{\mathbf{C},j-1}, v_{\mathbf{C},j}) \tag{5.9}$$

$$\vec{v}_{\mathbf{A},j} \stackrel{\mathrm{def}}{=} (\vec{v}_{\mathbf{A},j-1}, v_{\mathbf{A},j}) \tag{5.10}$$

$$\eta_{\mathbf{C},j} \stackrel{\mathrm{def}}{=} [z_j \mapsto v_{\mathbf{C},j}][\mathtt{store} \mapsto \sigma_{\mathbf{C},j}]\eta_{\mathbf{C},j-1} \tag{5.11}$$

$$\eta_{\mathbf{A},j} \stackrel{\mathrm{def}}{=} [z_j \mapsto v_{\mathbf{A},j}][\mathtt{store} \mapsto \sigma_{\mathbf{A},j}]\eta_{\mathbf{A},j-1}. \tag{5.12}$$

To show that $\vec{v}_{\mathbf{C},j}$ and $\sigma_{\mathbf{C},j}$ have the required properties of Equation 5.2 we calculate as follows.

$$\mathcal{E}{*}^{H}_{\Sigma}\ [\![E_1 \cdots E_{j-1}\ E_j]\!]\ C\ \eta_{\mathbf{C}}$$

$=$ $\langle$by definition of $\mathcal{E}*\rangle$
  let $((v_1, \cdots, v_{j-1}), \sigma_{j-1}) = \mathcal{E}*^H_\Sigma\ [\![E_1, \cdots, E_{j-1}]\!]\ \mathbf{A}\ \eta_\mathbf{C}$ **in**
   $\mathcal{E}^H_\Sigma\ [\![E_j]\!]\ \mathbf{A}\ [\mathtt{store} \mapsto \sigma_{j-1}]\eta_\mathbf{C}$
$\Rightarrow$ $\langle$by inductive hypothesis$\rangle$
  let $(v_{\mathbf{C},j}, \sigma_{\mathbf{C},j}) = \mathcal{E}^H_\Sigma\ [\![E_j]\!]\ C\ [\mathtt{store} \mapsto \sigma_{\mathbf{C},j-1}]\eta_\mathbf{C}$ **in**
   $((\vec{v}_{\mathbf{C},j-1}, v_{\mathbf{C},j}), \sigma_{\mathbf{C},j})$
$=$ $\langle$by definition of $(\vec{v}_{\mathbf{C},j}, \sigma_{\mathbf{C},j})\rangle$
  $(\vec{v}_{\mathbf{C},j}, \sigma_{\mathbf{C},j})$

Similarly, $\vec{v}_{\mathbf{C},j}$ and $\sigma_{\mathbf{A},j}$ have the required properties of Equation 5.4.

From the construction of $\eta_{\mathbf{C},j}$ and from the distinctness of each z_i, $(\eta_{\mathbf{C},j}\ z_i) = v_i$ for any $i \leq j$. This verifies the required property 5.3. Similarly we can show that $\eta_{\mathbf{C},j}$ has the property given in Equation 5.5.

Equation 5.6 thus follows directly from the main inductive hypothesis, because of the inductive assumption that $\eta_{\mathbf{C},j-1}\ \mathcal{R}\ \eta_{\mathbf{A},j-1}$. ∎

The following lemma extends the above lemma to show that simulation relations are preserved by command in $OBS^{\leq}$.

Lemma 5.1.2 *Let $(I\Sigma', E\Sigma)$ and $(I\Sigma'', E\Sigma)$ be mutation signatures. Let* $\mathbf{B}$ *be a $(I\Sigma', E\Sigma)$-mutation algebra and* $\mathbf{A}$ *be a $(I\Sigma'', \Sigma)$-mutation algebra. Let $\mathcal{R}$ be a $E\Sigma$-simulation algebra from* $\mathbf{B}$ *to* $\mathbf{A}$.

Let H be a type environment and $\eta_\mathbf{B}$ and $\eta_\mathbf{A}$ be a valid H-states. Then for each command C such that $E\Sigma; H; \vdash C\ \surd$, if $\sigma'_\mathbf{B} = \mathcal{C}^H_\Sigma\ [\![C]\!]\ \mathbf{B}\ \eta_\mathbf{B}$ and $\sigma'_\mathbf{A} = \mathcal{C}^H_\Sigma\ [\![C]\!]\ \mathbf{A}\ \eta_\mathbf{A}$ then

$$\eta_\mathbf{B}\ \mathcal{R}_H\ \eta_\mathbf{A} \Rightarrow [\mathtt{store} \mapsto \sigma'_\mathbf{B}]\eta_\mathbf{B}\ \mathcal{R}_H\ [\mathtt{store} \mapsto \sigma'_\mathbf{A}]\eta_\mathbf{A} \tag{5.13}$$

Proof: (by induction on the structure of C)

Let $\eta_\mathbf{B}$, $\eta_\mathbf{A}$ be a valid H-states over $\mathbf{B}$ and $\mathbf{A}$ respectively. Let C be a command in $OBS^{\leq}$ such that the free variables of C are in H.

Let $\sigma'_\mathbf{B} = \mathcal{C}^H_\Sigma\ [\![\mathrm{C}]\!]\ \mathbf{B}\ \eta_\mathbf{B}$ and $\sigma'_\mathbf{A} = \mathcal{C}^H_\Sigma\ [\![\mathrm{C}]\!]\ \mathbf{A}\ \eta_\mathbf{A}$

Suppose that $\eta_\mathbf{B}\ \mathcal{R}_H\ \eta_\mathbf{A}$.

Basis: Suppose C is an expression E of type T.

If E evaluates to $\bot$ in $\eta_\mathbf{B}$, then from the previous lemma it evaluate to $\bot$ in $\eta_\mathbf{A}$. And from bistrict property $\bot\ \mathcal{R}_H\ \bot$.

Otherwise, let $(v_\mathbf{B}, \sigma'_\mathbf{B}) = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{B}\ \eta_\mathbf{B}$ and $(v_\mathbf{A}, \sigma'_\mathbf{A}) = \mathcal{E}^H_\Sigma\ [\![E]\!]\ \mathbf{A}\ \eta_\mathbf{A}$. Then we can show the result by the following calculation.

  $\eta_\mathbf{B}\ \mathcal{R}_H\ \eta_\mathbf{A}$
$\Rightarrow$ $\langle$by the previous lemma, for some fresh identifier $y : T\rangle$

$[y \mapsto v_{\mathbf{B}}][\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_{[y \mapsto T][\mathtt{store} \mapsto \mathtt{Store}]H} \ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}$

$\Rightarrow$ ⟨by the shrinkable property of simulations⟩

$[\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_{[\mathtt{store} \mapsto \mathtt{Store}]H} \ [\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}$

$=$ ⟨by the hypothesis that $H(\mathtt{store}) = \mathtt{Store}$⟩

$[\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_H \ [\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{B}}$

Inductive Step: Assume, inductively, that the result holds for all subcommands of C. There are two cases.

1. Suppose C is $C_1; C_2$. Let $\sigma_{\mathbf{B},1} = \mathcal{C}^H_\Sigma \ [\![C_1]\!] \ \mathbf{B} \ \eta_{\mathbf{B}}$, $\sigma_{\mathbf{A},1} = \mathcal{C}^H_\Sigma \ [\![C_1]\!] \ \mathbf{A} \ \eta_{\mathbf{A}}$, $\sigma_{\mathbf{B},2} = \mathcal{C}^H_\Sigma \ [\![C_2]\!] \ \mathbf{B} \ [\mathtt{store} \mapsto \sigma_{\mathbf{B},1}]\eta_{\mathbf{B}}$, and $\sigma_{\mathbf{A},2} = \mathcal{C}^H_\Sigma \ [\![C_2]\!] \ \mathbf{A} \ [\mathtt{store} \mapsto \sigma_{\mathbf{A},1}]\eta_{\mathbf{A}}$.

$\eta_{\mathbf{B}} \ \mathcal{R}_H \ \eta_{\mathbf{A}}$

$\Rightarrow$ ⟨by the inductive hypothesis⟩

$[\mathtt{store} \mapsto \sigma_{\mathbf{B},1}]\eta_{\mathbf{B}} \ \mathcal{R}_H \ [\mathtt{store} \mapsto \sigma_{\mathbf{A},1}]\eta_{\mathbf{A}}$

$\Rightarrow$ ⟨by the inductive hypothesis⟩

$[\mathtt{store} \mapsto \sigma_{\mathbf{B},2}]\eta_{\mathbf{B}} \ \mathcal{R}_H \ [\mathtt{store} \mapsto \sigma_{\mathbf{A},2}]\eta_{\mathbf{A}}$

2. Suppose C is "`if` E `then` C_1 `else` C_2".

If the result of evaluation E in $\eta_{\mathbf{B}}$ or in $\eta_{\mathbf{A}}$ is $\bot$, then by the previous lemma it evaluates in both cases. And from bistrict property $\bot \ \mathcal{R}_H \ \bot$.

Let $y : \mathtt{Bool}$ be a fresh variable. Let $(v_{\mathbf{B}}, \sigma'_{\mathbf{B}}) = \mathcal{E}^H_\Sigma \ [\![E]\!] \ \mathbf{B} \ \eta_{\mathbf{B}}$ and $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}^H_\Sigma \ [\![E]\!] \ \mathbf{A} \ \eta_{\mathbf{A}}$. Then by the previous lemma

$$[y \mapsto v_{\mathbf{B}}][\mathtt{store} \mapsto \mathit{store}'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_{[y \mapsto \mathtt{Bool}]H} \ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}} \tag{5.14}$$

Since `Bool` is a visible type, if $v_{\mathbf{B}} \in \mathit{VALS}^{\mathbf{B}}_{\mathtt{Bool}}$ then from $\mathcal{R}$ is VIS-identical,

$$v_{\mathbf{B}} = v_{\mathbf{A}} \tag{5.15}$$

Hence the result of the test is the same in both $\mathbf{B}$ and in $\mathbf{A}$.

$[y \mapsto v_{\mathbf{B}}][\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_{[y \mapsto \mathtt{Bool}]H} \ [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}$

$\Rightarrow$ ⟨by the shrinkable property of simulations⟩

$[\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}} \ \mathcal{R}_H \ [\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}$

$\Rightarrow$ ⟨by the inductive hypothesis and if $v_{\mathbf{B}}$ is `true`⟩

$[\mathtt{store} \mapsto (\mathcal{C}^H_\Sigma \ [\![C_1]\!] \ \mathbf{B} \ [\mathtt{store} \mapsto \sigma'_{\mathbf{B}}]\eta_{\mathbf{B}})]\eta_{\mathbf{B}}$
$\quad \mathcal{R}_H \ [\mathtt{store} \mapsto \mathcal{C}^H_\Sigma \ [\![C_1]\!] \ \mathbf{A} \ [\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}}]\eta_{\mathbf{A}}$

Similarly, if $v_{\mathbf{B}}$ is `false`, we can prove that

$$\begin{array}{l}[\mathtt{store} \mapsto (\mathcal{C}^H_\Sigma \ [\![C_2]\!] \mathbf{C} \ [\mathtt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C}})]\eta_{\mathbf{C}} \\ \qquad \mathcal{R}_H \ [\mathtt{store} \mapsto (\mathcal{C}^H_\Sigma \ [\![C_2]\!] \ \mathbf{A} \ [\mathtt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}})]\eta_{\mathbf{A}}.\end{array}$$

∎

We prove the following lemma that is useful in combining two environments.

Lemma 5.1.3 *Let $(I\Sigma', E\Sigma)$ and $(I\Sigma'', E\Sigma)$ be mutation signatures. Let* $\mathbf{C}$ *be a $(I\Sigma', E\Sigma)$-mutation algebra and* $\mathbf{A}$ *be a $(I\Sigma'', \Sigma)$-mutation algebra. Let $\mathcal{R}$ be a $E\Sigma$-simulation relation from* $\mathbf{C}$ *to* $\mathbf{A}$.

Let H be a type environment and let D be a declaration such that $E\Sigma; H \vdash D \implies H'$. If $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{A}}$ be valid H-states and $\eta'_{\mathbf{C}}$ and $\eta'_{\mathbf{A}}$ be valid H'-states such that $\eta'_{\mathbf{C}} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{C}$ $\eta_{\mathbf{C}}$ and $\eta'_{\mathbf{A}} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{A}$ $\eta_{\mathbf{A}}$. Then

$$(\eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}}) \wedge (\eta'_{\mathbf{C}}\, \mathcal{R}_{H'}\, \eta'_{\mathbf{A}}) \Rightarrow (\eta_{\mathbf{C}} \uplus \eta'_{\mathbf{C}})\, \mathcal{R}_{H\cup H'}\, (\eta_{\mathbf{A}} \uplus \eta'_{\mathbf{A}}) \tag{5.16}$$

Proof: This is shown by induction on the structure of D. If D is an empty declaration, then Equation 5.16 is trivially true.

If D is `const` $I\!:\!T$ `:=` E, then applying Lemma 5.1.1 on E when the identifier $y = I$, we can conclude that Equation 5.16 is true. The following calculation gives more detail. Let $(v_{\mathbf{C}}, \sigma'_{\mathbf{C}}) = \mathcal{E}^H_\Sigma$ $[\![E]\!]$ $\mathbf{C}$ $\eta_{\mathbf{C}}$ and $(v_{\mathbf{A}}, \sigma'_{\mathbf{A}}) = \mathcal{E}^H_\Sigma$ $[\![E]\!]$ $\mathbf{A}$ $\eta_{\mathbf{A}}$.

$$\begin{array}{ll}
 & \eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}} \\
\Rightarrow & \langle \text{by lemma 5.1.1 using } I \text{ for } y \rangle \\
 & [I \mapsto v_{\mathbf{C}}][\texttt{store} \mapsto \sigma'_{\mathbf{C}}]\eta_{\mathbf{C}}\, \mathcal{R}_{[I\mapsto T]H}\, [I \mapsto v_{\mathbf{A}}][\texttt{store} \mapsto \sigma'_{\mathbf{A}}]\eta_{\mathbf{A}} \\
\Rightarrow & \langle \text{by rearranging the environments using } \mapsto \rangle \\
 & [\texttt{store} \mapsto \sigma'_{\mathbf{C}}](\eta_{\mathbf{C}} \cup [I \mapsto v_{\mathbf{C}}][\texttt{store} \mapsto \sigma'_{\mathbf{C}}]\mathit{emptyEnviron}) \\
 & \quad \mathcal{R}_{H\cup\{(I,T),(\texttt{store},\texttt{Store})\}}\, [\texttt{store} \mapsto \sigma'_{\mathbf{A}}](\eta_{\mathbf{A}} \cup [I \mapsto v_{\mathbf{A}}][\texttt{store} \mapsto \sigma'_{\mathbf{A}}]\mathit{emptyEnviron}) \\
= & \langle \text{by definition of } \uplus \rangle \\
 & (\eta_{\mathbf{C}} \uplus [I \mapsto v_{\mathbf{C}}][\texttt{store} \mapsto \sigma'_{\mathbf{C}}]\mathit{emptyEnviron}) \\
 & \quad \mathcal{R}_{H\cup H'}\, (\eta_{\mathbf{A}} \uplus [I \mapsto v_{\mathbf{A}}][\texttt{store} \mapsto \sigma'_{\mathbf{A}}]\mathit{emptyEnviron})
\end{array}$$

The induction step is when $D = D_1\,; D_2$. Assume that the hypothesis is true for all subdeclarations of D. Let $\eta_{\mathbf{C},1} = \mathcal{D}^H_\Sigma$ $[\![D_1]\!]$ $\mathbf{C}$ $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{C},2} = \mathcal{D}^H_\Sigma$ $[\![D_2]\!]$ $\mathbf{C}$ $(\eta_{\mathbf{C},1} \uplus \eta_{\mathbf{C}})$. Let $\eta_{\mathbf{A},1} = \mathcal{D}^H_\Sigma$ $[\![D_1]\!]$ $\mathbf{A}$ $\eta_{\mathbf{A}}$ and $\eta_{\mathbf{A},2} = \mathcal{D}^H_\Sigma$ $[\![D_2]\!]$ $\mathbf{A}$ $(\eta_{\mathbf{A},1} \uplus \eta_{\mathbf{A}})$. Let H_1 and H_2 be the corresponding type contexts for D_1 and D_2 respectively.

$$\begin{array}{ll}
 & \eta_{\mathbf{C}}\, \mathcal{R}_H\, \eta_{\mathbf{A}} \\
\Rightarrow & \langle \text{by induction hypothesis on } D_1 \rangle \\
 & (\eta_{\mathbf{C},1} \uplus \eta_{\mathbf{C},1})\, \mathcal{R}_{H\cup H_1}\, (\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1}) \\
\Rightarrow & \langle \text{by induction hypothesis on } D_2 \rangle \\
 & ((\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1}) \cup \eta_{\mathbf{C},2})\, \mathcal{R}_{(H\cup H_1)\cup H_2}\, ((\eta_{\mathbf{A}} \cup \eta_{\mathbf{A},1}) \uplus \eta_{\mathbf{A},2}) \\
\Rightarrow & \langle \text{by associativity of } \uplus \rangle \\
 & (\eta_{\mathbf{C}} \uplus (\eta_{\mathbf{C},1} \uplus \eta_{\mathbf{C},2}))\, \mathcal{R}_{H\cup(H_1\cup H_2)}\, (\eta_{\mathbf{A}} \uplus (\eta_{\mathbf{A},1} \uplus \eta_{\mathbf{A},2})) \\
= & \langle \text{by semantics of declarations} \rangle \\
 & (\eta_{\mathbf{C}} \uplus \eta'_{\mathbf{C}})\, \mathcal{R}_{H\cup H'}\, (\eta_{\mathbf{A}} \uplus \eta'_{\mathbf{A}})
\end{array}$$

∎

Lemma 5.1.4 *Let $(I\Sigma', E\Sigma)$ and $(I\Sigma'', E\Sigma)$ be mutation signatures. Let* $\mathbf{C}$ *be a $(I\Sigma', E\Sigma)$-mutation algebra and* $\mathbf{A}$ *be a $(I\Sigma'', \Sigma)$-mutation algebra. Let $\mathcal{R}$ be a $E\Sigma$-simulation relation from* $\mathbf{C}$ *to* $\mathbf{A}$.

Let H be a type environment and $\eta_{\mathbf{C}}$ and $\eta_{\mathbf{A}}$ be a valid H-states. Then for each declaration D such that $E\Sigma; H \vdash D \implies H'$, if $\eta'_{\mathbf{C}} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{C}$ $\eta_{\mathbf{C}}$ and $\eta'_{\mathbf{A}} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{A}$ $\eta_{\mathbf{A}}$ then

$$\eta_{\mathbf{C}} \, \mathcal{R}_H \, \eta_{\mathbf{A}} \Rightarrow \eta'_{\mathbf{C}} \, \mathcal{R}_{H'} \, \eta'_{\mathbf{A}} \tag{5.17}$$

Proof: From Lemma 5.1.3, we have $(\eta_{\mathbf{C}} \uplus \eta'_{\mathbf{C}})$ $\mathcal{R}_{H \cup H'}$ $(\eta_{\mathbf{A}} \uplus \eta'_{\mathbf{C}})$. By shrinkable property of simulations, we have $\eta'_{\mathbf{C}} \, \mathcal{R}_H \, \eta'_{\mathbf{A}}$. ∎

The following lemma uses previous lemmas to show that bodies of the client functions of $OBS^{\leq}$ preserve simulations.

Lemma 5.1.5 *Let $(I\Sigma, E\Sigma)$ be a mutation signature. Let SPEC be the set of $E\Sigma$ mutation algebras and let $\mathbf{A} \in SPEC$. Let P =* `client function` $I(\vec{F}$`:`$\vec{T}$`):`T `is` D C `return` E `by` D `do` C`;` `call` $I(\vec{E})$ *such that $E\Sigma; \{\} \vdash P \surd$. Let $(H, f, s) = \mathcal{M}^H_\Sigma$ $[\![P]\!]$ $\mathbf{A}$. Let $E\Sigma; H \vdash D \Longrightarrow H'$ and $E\Sigma; H \cup H' \vdash E : T$.*

Let $\mathbf{C}, \mathbf{A} \in SPEC$. Let $\mathcal{R}$ be a $E\Sigma$-simulation relation from $\mathbf{C}$ *to* $\mathbf{A}$. *Let $\eta_{\mathbf{C}}$ be a valid H-state over* $\mathbf{C}$ *and $\eta_{\mathbf{A}}$ be a valid H-state over* $\mathbf{A}$. *If $(v_{\mathbf{C}}, \eta'_{\mathbf{C}}) = (f \ \mathbf{C} \ \eta_{\mathbf{C}})$ and $(v_{\mathbf{A}}, \eta'_{\mathbf{A}}) = (f \ \mathbf{A} \ \eta_{\mathbf{A}})$ then for any fresh identifier $y : T$,*

$$\eta_{\mathbf{C}} \, \mathcal{R}_H \, \eta_{\mathbf{A}} \Rightarrow [y \mapsto v_{\mathbf{C}}]\eta'_{\mathbf{C}} \, \mathcal{R}_{[y \mapsto T](H \cup H')} \, [y \mapsto v\mathbf{A}]\eta'_{\mathbf{A}} \tag{5.18}$$

Proof: Let $\mathrm{E}\Sigma; H \vdash D \Longrightarrow H'$, $\mathrm{E}\Sigma; H \cup H' \vdash C \surd$, and $\mathrm{E}\Sigma; H \cup H' \vdash E : T$.

Let $\eta_{\mathbf{C},1} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{C}$ $\eta_{\mathbf{C}}$, $\eta_{\mathbf{A},1} = \mathcal{D}^H_\Sigma$ $[\![D]\!]$ $\mathbf{A}$ $\eta_{\mathbf{A}}$, $\sigma_{\mathbf{C},2} = \mathcal{C}^H_\Sigma$ $[\![C]\!]$ $\mathbf{C}$ $(\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1})$, $\sigma_{\mathbf{A},2} = \mathcal{C}^H_\Sigma$ $[\![C]\!]$ $\mathbf{A}$ $(\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1})$, $(v_{\mathbf{C}}, \sigma_{\mathbf{C},3}) = \mathcal{E}^H_\Sigma$ $[\![E]\!]$ $\mathbf{C}$ $[\mathtt{store} \mapsto \sigma_{\mathbf{C},2}](\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1})$, and $(v_{\mathbf{A}}, \sigma_{\mathbf{A},3}) = \mathcal{E}^H_\Sigma$ $[\![E]\!]$ $\mathbf{A}$ $[\mathtt{store} \mapsto \sigma_{\mathbf{A},2}](\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1})$.

$\eta_{\mathbf{C}} \, \mathcal{R}_H \, \eta_{\mathbf{A}}$
$\Rightarrow$ ⟨by Lemma 5.1.3⟩
$(\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1}) \, \mathcal{R}_{H \cup H'} \, (\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1})$
$\Rightarrow$ ⟨by Lemma 5.1.2⟩
$[\mathtt{store} \mapsto \sigma_{\mathbf{C},2}](\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1}) \, \mathcal{R}_{H \cup H'} \, [\mathtt{store} \mapsto \sigma_{\mathbf{A},2}](\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1})$
$\Rightarrow$ ⟨by Lemma 5.1.1 for any $y : T$⟩
$[y \mapsto v_{\mathbf{C}}][\mathtt{store} \mapsto \sigma_{\mathbf{C},3}](\eta_{\mathbf{C}} \uplus \eta_{\mathbf{C},1})$
$\quad \mathcal{R}_{[y \mapsto T](H \cup H')} \, [y \mapsto v_{\mathbf{A}}][\mathtt{store} \mapsto \sigma_{\mathbf{A},3}](\eta_{\mathbf{A}} \uplus \eta_{\mathbf{A},1})$

∎

5.2 Expected results and "no surprises" for weak behavioral subtyping

In this section we develop the notion of the set of expected results for an observation, which is used to show the adequacy of our notions of behavioral subtyping. For modular reasoning, conclusions about programs in the context of supertypes should be valid in the context of executing the programs with subtypes. The kinds of conclusions one can make in the presence of aliasing will be different than the conclusions one can make with some restrictions on aliasing. Hence, we require different notions of expected results for weak and strong behavioral subtyping. In this section, we define two sets of expected results for (client) functions of $\mathit{OBS}^{\leq}$: one appropriate for weak behavioral subtyping, where aliasing is restrictive and one for strong behavioral subtyping, where aliasing is not restricted.

Ideally, we would like to define the expected set of results for behavioral subtyping in terms of stores without any subtyping. This form of nominal store is possible for weak behavioral subtyping, because of restriction of aliasing. But for strong behavioral subtyping, because we allow all forms of aliasing, it is not sensible to restrict our study to stores that do not contain subtyping, that is nominal stores. This is because we cannot, in general, construct nominal stores that preserve aliasing. Hence, the kinds of conclusions for programs with strong behavioral subtyping are different than for weak behavioral subtyping. So we have two different notions of the set of expected results for weak and strong behavioral subtyping.

The set of expected results for weak behavioral subtyping are the results of invoking the client function, f, over all stores without any subtyping and over all possible values such that they do not violate any aliasing constraints. Formally, it is defined as follows.

Definition 5.2.1 (weakly expected results) *Let SPEC be a set of $E\Sigma$-mutation algebras that preserve alias legality with respect to an alias context L. Let FN be a client function in* $\mathrm{OBS}^{\leq}$ *and let $(H, f) = \mathcal{FN}_{\Sigma}\ [\![FN]\!]$ Let (H, f) be the denotation of a client function in* $\mathrm{OBS}^{\leq}$*.*

Then the set of weakly expected results of f for SPEC *is the union over all $\mathbf{A} \in$ SPEC and all H-states, $s_{\mathbf{A}}$, over $\mathbf{A}$ such that $s_{\mathbf{A}}$ is nominal, of the results v, where $(v, s'_{\mathbf{A}}) = (f\ \mathbf{A}\ s_{\mathbf{A}})$.*

A result is *weakly surprising* for *SPEC* if it is not a weakly expected result. Weakly surprising results can occur when one uses a presumed subtype relation that is not a weak behavioral subtype relation and observes a state that is not nominal.

The "no surprises" theorem ensures the correctness of our weak behavioral subtyping. It shows that the set of actual results of an observation is subset of the set of weakly expected results. This theorem ensures using supertype abstraction for modular verification is sound.

Theorem 5.2.2 (Weak behavioral subtyping means "no surprises") *Let* $(I\Sigma,\ E\Sigma)$ *be a mutation signature and let* L *be an alias context for* $E\Sigma$*. Let SPEC be a set of* $E\Sigma$*-mutation algebras that preserve alias legality with respect to* L *and let* $\mathbf{A} \in$ *SPEC. Let* P *be a main program of* $\mathrm{OBS}^{\leq}$ *such that* $E\Sigma; \{\}; L \vdash P \surd$*. Let* $(H, f, s) = \mathcal{M}^{H}_{\Sigma}\ [\![P]\!]\ \mathbf{A}$*.*

Then for all $\mathbf{C} \in$ *SPEC, for all* H*-states,* $s_{\mathbf{C}}$ *such that* $stAliasOk(H, \mathbf{C}, s_{\mathbf{C}})$ *and* $(v, s'_{\mathbf{C}}) = (f\ \mathbf{C}\ s_{\mathbf{C}})$*, if* $E\Sigma.\leq$ *is a weak behavioral subtype relationship for SPEC, then the result* v *is a weakly expected result.*

Proof: Let $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$ be a mutation signature and let L be an alias context for the external signature $\mathrm{E}\Sigma$. Let *SPEC* be a set of $\mathrm{E}\Sigma$ mutation algebras that preserve alias legality with respect to L and let $\mathbf{A} \in$ SPEC. Let P be a main program of $OBS^{\leq}$ such that the body of the client function of P is D `do` C `return` E. Let $(H, f, s) = \mathcal{M}^{H}_{\Sigma}\ [\![P]\!]\ \mathbf{A}$, $\mathrm{E}\Sigma; H; L \vdash D \Longrightarrow H'$, and $\mathrm{E}\Sigma; H \cup H'; L \vdash E : T :: r$.

Let $\mathbf{C} \in$ *SPEC* and let $s_{\mathbf{C}}$ be a H-state over $\mathbf{C}$. Let $\mathrm{E}\Sigma.\leq$ be a weak behavioral subtype relation for *SPEC*.

Since $\mathrm{E}\Sigma.\leq$ is a weak behavioral subtype relation, from the definition, there is some $\mathbf{C}' \in$ *SPEC* and a $\mathrm{E}\Sigma$-simulation relation, $\mathcal{R}$, from $\mathbf{C}$ to $\mathbf{C}'$. From the coercion property there exists a nominal H-state, $s_{\mathbf{C}'}$, over $\mathbf{C}'$ such that $s_{\mathbf{C}}\ \mathcal{R}_H\ s_{\mathbf{C}'}$.

Let $(v_{\mathbf{C}}, \eta_{\mathbf{C}}) = (f\ \mathbf{C}\ s_{\mathbf{C}})$ and $(v_{\mathbf{C}'}, \eta_{\mathbf{C}'}) = (f\ \mathbf{C}'\ s_{\mathbf{C}'})$.

From Lemma 5.1.5 and from the meaning of f, for any fresh identifier $y : T$, we have $[y \mapsto v_{\mathbf{C}}]\eta_{\mathbf{C}}\ \mathcal{R}_{[y \mapsto T](H \cup H')}\ [y \mapsto v_{\mathbf{C}'}]\eta_{\mathbf{C}'}$.

Since, $T \in$ *VIS* and applying *EXTERNALS*-identical property, we have $v_{\mathbf{C}} = v_{\mathbf{C}'}$.

Since $s'_{\mathbf{C}}$ is a nominal state, we can conclude that the result of $(f\ \mathbf{C}\ s_{\mathbf{C}})$ is an expected result. ∎

5.3 Expected results and "no surprises" for strong behavioral subtyping

Since reasoning based on strong behavioral subtyping does not assume any restrictions on aliasing, the coercion property does not ensure that a related nominal environment exists. Hence, a new notion of expected result based on the history constraint is defined in this section.

Any result that is obtained in a state that along with the initial state satisfies the history constraint required by the strong behavioral subtyping is an expected result. This set of expected results is referred as the set of *strongly expected results.*

Note that, though we use "strongly" this set of expected results is usually larger than the set of weakly expected results. We use "strongly" to match the name of strong behavioral subtyping.

Definition 5.3.1 (strongly expected results) *Let SPEC be a set of $E\Sigma$-mutation algebras. Let HCONST be a history constraint over SPEC such that each* $\mathbf{A}$ *in SPEC satisfies the history constraint. Let FN be a client function and let* $(H, f) = \mathcal{FN}_\Sigma [\![FN]\!]$.

Then the set of strongly expected results of f for SPEC and HCONST is the union over all $\mathbf{A} \in SPEC$ *and all H-states,* $s_\mathbf{A}$*, over* $\mathbf{A}$ *of the results v, where* $(v, s'_\mathbf{A}) = (f\ \mathbf{A}\ s_\mathbf{A})$ *and* $(s_\mathbf{A}\ \texttt{store})\ HCONST(\mathbf{A})\ (s'_\mathbf{A}\ \texttt{store})$.

A result is *strongly surprising for SPEC and HCONST* if it is not strongly expected, that is if it is not a proper result (i.e., not $\perp$) or if the resulting state and the original state pair do not satisfy the history constraint.

Lemma 5.3.2 *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra. Let* $\P_\mathbf{A}$ *be a history constraint over* $\mathbf{A}$.

Let E be an expression in the main program of $\mathrm{OBS}^{\leq}$*. Let H be a type environment such that* $E\Sigma; H \vdash E : T$*. Let* η *be a H-state over* $\mathbf{A}$*. Let* $(v, \sigma') = \mathcal{E}^H_\Sigma [\![E]\!]\ \mathbf{A}\ \eta$.

If $\mathbf{A}$ *satisfies history constraint* $\P_\mathbf{A}$*, then* $(\eta\ \texttt{store})\ \P_\mathbf{A}\ \sigma'$.

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

Let $\mathbf{A}$ be a $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra and let $\P_\mathbf{A}$ be a history constraint over $\mathbf{A}$.

Let H be a type environment and let η be a valid H-environment. Let $\sigma = (\eta\ \texttt{store})$.

If E is a numeral, identifier, `true`, `false`, or `nothing`, $\sigma' = \sigma$ and hence by reflexivity of the history constraint, the conclusion of the lemma is true.

The inductive step is when E is $g(\vec{E})$. The inductive hypothesis is that the lemma is true for all subexpressions of $g(\vec{E})$.

Let $(v_n, \sigma_n) = \mathcal{E}^H_\Sigma [\![\vec{E}]\!]\ \mathbf{A}\ \eta$. If $\sigma\ \P_\mathbf{A}\ \sigma_n$, then we use the constraint property of strong behavioral subtyping to conclude the lemma.

So it suffices to show that

$$\sigma\ \P_\mathbf{A}\ \sigma_n. \tag{5.19}$$

We show this by induction on the length of $\vec{E}$. If $\vec{E}$ is empty, then Equation 5.19 follows from the reflexivity of the history constraint.

Let Equation 5.19 be true when n is $i \Leftrightarrow 1$. Let $(v_{i-1}, \sigma_{i-1}) = \mathcal{E}^H_\Sigma [\![E_1, \cdots, E_{i-1}]\!]\ \mathbf{A}\ \eta$ and let $(v_i, \sigma_i) = \mathcal{E}^H_\Sigma [\![E_i]\!]\ \mathbf{A}\ [\texttt{store} \mapsto \sigma_{i-1}]\eta$. So from the induction hypothesis we have, $\sigma\ \P_\mathbf{A}\ \sigma_{i-1}$. Applying main hypothesis on E_i, we can conclude that $(\sigma_{i-1}\ \P_\mathbf{A}\ \sigma_i)$. By the transitivity of history constraints we have $\sigma\ \P_\mathbf{A}\ \sigma_i$, which completes the induction. ∎

Lemma 5.3.3 *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra. Let* $\P_{\mathbf{A}}$ *be a history constraint over* $\mathbf{A}$*.*

Let C *be a command in the main program of* $\mathrm{OBS}^{\leq}$*. Let* H *be a type environment such that* $E\Sigma; H \vdash C \surd$*. Let* η *be a* H*-state over* $\mathbf{A}$*. Let* $\sigma' = \mathcal{E}^{H}_{\Sigma}\ [\![C]\!]\ \mathbf{A}\ \eta$*.*

If $\mathbf{A}$ *satisfies history constraint* $\P_{\mathbf{A}}$ *then* $(\eta\ \mathtt{store})\ \P_{\mathbf{A}}\ \sigma'$*.*

Proof: (sketch) This can be shown by induction on the structure of C.

If C is an expression, then the conclusion follows from the previous lemma. If C is $C_1; C_2$, applying the induction on C_1 and C_2 and using the transitivity of the history constraint, we can conclude that $(\eta\ \mathtt{store})\ \P_{\mathbf{A}}\ \sigma'$. For the case when C is an `if` command, we use the previous lemma for E, induction hypothesis for C_1 or C_2 depending on the value of E, and transitivity to conclude the lemma. ∎

Lemma 5.3.4 *Let* $\mathbf{A}$ *be a* $(I\Sigma, E\Sigma)$*-mutation algebra. Let* $\P_{\mathbf{A}}$ *be a history constraint over* $\mathbf{A}$*.*

Let D *be a command in the main program of* $\mathrm{OBS}^{\leq}$*. Let* H *be a type environment such that* $E\Sigma; H \vdash D \Longrightarrow H'$*. Let* η *be a* H*-state over* $\mathbf{A}$*. Let* $\eta' = \mathcal{E}^{H}_{\Sigma}\ [\![D]\!]\ \mathbf{A}\ \eta$*.*

If $\mathbf{A}$ *satisfies history constraint* $\P_{\mathbf{A}}$ *then* $(\eta\ \mathtt{store})\ \P_{\mathbf{A}}\ \sigma'$*.*

Proof: (sketch) (induction on the structure of D)

The empty case is trivially true. The case when $D = \mathtt{const}\ I{:}T\ \mathtt{:=}\ E$ follows directly from lemma 5.3.2. The induction $D = D_1; D_2$ follows by applying the induction hypothesis on D_1 and D_2 and transitivity of the history constraint. ∎

Theorem 5.3.5 (Strong behavioral subtyping means "no surprises") *Let* $(I\Sigma,\ E\Sigma)$ *be a mutation signature. Let SPEC be the set of* $E\Sigma$*-mutation algebras. Let HCONST be a history constraint over SPEC. Let* $\mathbf{A} \in$ *SPEC. Let* P *be a main program in* $\mathrm{OBS}^{\leq}$ *such that* $\Sigma : H \vdash P \surd$*. Let* $(H, f, s_{\mathbf{A}}) = \mathcal{M}^{H}_{\Sigma}\ [\![P]\!]\ \mathbf{A}$*.*

Then for all $\mathbf{C} \in$ *SPEC, for all* H*-states,* $s_{\mathbf{C}}$*, over* $\mathbf{C}$ *such that if* $E\Sigma.\leq$ *is a strong behavioral subtype relationship for SPEC with respect to HCONST and if* $(v, s'_{\mathbf{C}}) = (f\ \mathbf{C}\ s_{\mathbf{C}})$ *then* v *is a strongly expected result.*

Proof: Let $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$ be a mutation signature. Let $\mathbf{A}$ be a $(\mathrm{I}\Sigma, \mathrm{E}\Sigma)$-mutation algebra. Let P be a main program in $OBS^{\leq}$. Let $(H, f, s_{\mathbf{A}}) = \mathcal{M}^{H}_{\Sigma}\ [\![M]\!]\ \mathbf{A}$, where f takes a $\mathbf{A}$ a H-state.

Let $\mathbf{C} \in$ *SPEC* and let $s_{\mathbf{C}}$ be a H-state over $\mathbf{C}$. and let $\P_{\mathbf{C}} = \mathrm{HCONST}(\mathbf{C})$.

Since $\mathrm{E}\Sigma.\leq$ is a strong behavioral subtype relation for SPEC with respect to HCONST, $\mathbf{C}$ satisfies the history constraint with respect to $\P_{\mathbf{C}}$.

Let the body of the program f be D `do` C `return` E. Let $\eta_{\mathbf{C}} = \mathcal{D}^{H}_{\Sigma}\ [\![D]\!]\ \mathbf{C}\ s_{\mathbf{C}}$, let $\sigma'_{\mathbf{C}} = \mathcal{D}^{H}_{\Sigma}\ [\![C]\!]\ \mathbf{C}\ (s_{\mathbf{C}} \uplus \eta_{\mathbf{C}})$, and let $(v, \sigma'') = \mathcal{E}^{H}_{\Sigma}\ [\![E]\!]\ \mathbf{C}\ [\mathtt{store} \mapsto \sigma'_{\mathbf{C}}](s_{\mathbf{C}} \uplus \eta_{\mathbf{C}})$.

Let $(v, s'_{\mathbf{C}}) = (f\ \mathbf{C}\ s_{\mathbf{C}})$. Then from the semantics, we have $(s'_{\mathbf{C}}\ \texttt{store}) = \sigma''$.

From Lemma 5.3.4, we have $(s_{\mathbf{C}}\ \texttt{store})\, \P_{\mathbf{C}}\, (\eta_{\mathbf{C}}\ \texttt{store})$. From Lemma 5.3.3, we have $(\eta_{\mathbf{C}}\ \texttt{store})\, \P_{\mathbf{C}}\, \sigma'_{\mathbf{C}}$. From Lemma 5.3.2, we have $\sigma'_{\mathbf{C}}\, \P_{\mathbf{C}}\, \sigma''_{\mathbf{C}}$. From the transitivity of $\P$, we can conclude that $(s_{\mathbf{C}}\ \texttt{store})\, \P_{\mathbf{C}}\, \sigma''_{\mathbf{C}}$.

That is, $(s_{\mathbf{C}}\ \texttt{store})\, \P_{\mathbf{C}}\, (s'_{\mathbf{C}}\ \texttt{store})$. Hence, the result $(v_{\mathbf{C}}, s'_{\mathbf{C}})$ of $(f\ \mathbf{C}\ s'_{\mathbf{C}})$ is a strongly expected result. ∎

These theorems prove the soundness of weak and strong behavioral subtyping. In the next chapter, we discuss the adequacy of these notions for modular reasoning of OO programs.

6. DISCUSSION

In this chapter we discuss the significance of the “no surprises” results for weak and strong behavioral subtyping and place the context of our results in long term goal of understanding OO programs.

6.1 Modular reasoning of OO programs

Recall that in Chapter 1, we state that our motivation for the study of behavioral subtyping is modular verification of OO programs. Since a formal discussion on the verification of OO programs is beyond the scope of this dissertation, we informally discuss how the “no surprises” results aid in the modular verification of OO programs.

6.1.1 Adequacy of weak behavioral subtyping

The “no surprises” theorem for weak behavioral subtyping states that if aliasing is restricted, then the results of programs that use subtypes in place of supertypes are not surprising. That is, programs that use subtype objects do not produce any surprises.

Note that, for weak behavioral subtyping the set of expected results is calculated using only the nominal states. This set of expected results does not change with the addition of new behavioral subtypes. Further, the “no surprises” guarantees that the results of using these new behavioral subtypes in place of supertypes are not surprising. Hence, conclusions based on this set of expected results does not change with the addition of new behavioral subtypes. For sound modular reasoning using supertype abstraction this result is significant because it ensures that the addition of new behavioral subtypes does not require reverification.

6.1.2 Adequacy of strong behavioral subtyping

Every strong behavioral subtype relation is a weak behavioral subtype relation. Thus if the alias conditions are satisfied then the “no surprises” theorem of weak behavioral subtyping applies for strong behavioral subtype relations also. Hence, the supertype abstraction principle can be used as a modular reasoning technique in such a case.

But strong behavioral subtyping does not place any restrictions on aliasing. So a sound reasoning technique based on strong behavioral subtyping should not make any assumptions on aliasing. If the reasoning technique does not assume any alias restrictions, then we cannot conclude properties of programs based on method invocations directly. To see this, consider the following program, where `NewAccount0` and `NewAccount1` are two unrelated account types with no common behavioral subtype.

```
client function is_Changing(b:NewAccount0, p:NewAccount1): Bool
  is const m: MoneyObj = mkMoneyObj(5);
     const bal: Int = value(balance(b));
  do
     withdraw(p, m)
  return (equal(value(balance(b)), bal));
```

The set of actual results when there are no subtypes to `NewAccount0` and `NewAccount1` is {`true`}. However, with the addition of a new common behavioral subtype, unlike weak behavioral subtyping, b and p can be aliased to an object of the new subtype. So the set of expected results changes to {`true`, `false`}. If the addition of new types changes the set of expected results, then any conclusions based on the set of expected results might also change. So a verification technique that uses supertype abstraction should either reverify or should not use this set of expected results to make conclusions when new strong behavioral subtypes are added. But reverification of all existing functions when new types are added is not practical and nonmodular.

However, a different kind of modular reasoning technique based on supertype abstraction could be used for programs that use strong behavioral subtyping. Such a reasoning technique makes conclusions based on only the history constraints of the static types of identifiers in the program. If all the behavioral subtypes satisfy these constraints, then the conclusion do not change with the addition of new types. So this reasoning technique would be modular.

The "no surprises" result of strong behavioral subtyping guarantees that using subtype objects in place of supertype objects does not produce any surprises with respect to the constraint. Hence, for any sound modular reasoning technique based on supertype abstraction that makes conclusions based only on the constraint property, strong behavioral subtyping is an adequate notion.

A closer look at the kinds of conclusions one can derive out of the constraint property shows that only a few conclusions can be made using this approach. Since one cannot specify properties with specific values to variables over states, one cannot conclude anything but reflexive and transitive properties such as immutability, never increasing, never decreasing, and similar monotonic properties. However, if the reasoning technique assumes aliasing restrictions then we could conclude more about programs based on effects of method invocations.

6.2 Context of our results

The context of any study of behavioral subtyping is to understand OO programs that use subtype polymorphism. Whether one uses proof-theoretic techniques or model-theoretic techniques, the main goal is to capture a relationship that aids in understanding, formally or informally, OO programs. In this section, we look at this context and place the results of our work with respect to this context.

Most proof-theoretic approaches [Ame91, LW94, DL96] study relationships between specific types and conclude that a behavioral subtype relationship exists. Proving these subtype relationships is easier. Since the behavior of types is given by their specification, these studies do not need a different model and a different semantics for studying behavioral subtype relations. However, none of the proof-theoretic studies, so far, apply behavioral subtyping to the results of programs and show that their notion of behavioral subtyping do not produce any surprising results.

Model-theoretic studies [LW95], including this study, give models of type specifications, define behavioral subtype relations over those models, and prove that valid behavioral subtype relationships do not result any surprising behavior. These are shown over the context of a OO programming language. Compared to proof-theoretic approaches, it is tedious to prove behavioral subtype relationships using model-theoretic techniques. The results of these studies can be used for a sound modular reasoning technique [LW95].

An important future work is the study that bridges these two approaches. The results of such a study combine the ease of proving subtype relationships and the automatic application of "no surprises" results to modular verification. The work in this dissertation leads to a result that allows modular reasoning of OO programs. The next step is a modular verification logic, which is left as a future work.

In [LP94], Leavens and Pigozzi give a definition of behavioral subtyping for immutable types that is both sound and complete. Though we use a similar notion of simulations, it is not clear whether our notions of weak and strong behavioral subtyping are complete. We leave this as an open problem.

Work on split semantics, analyzing aliasing constraints in method bodies, and extending mutation algebras to model nondeterministic types, are also left as future work.

7. CONCLUSIONS AND SUMMARY

In this chapter we summarize our study of behavioral subtyping in the context of mutation and aliasing by highlighting our contributions and by presenting our conclusions.

7.1 Conclusions

In this section, we offer some conclusions from our study and provide informal guidelines for defining behavioral subtyping among different types.

Most OO languages allow users to declare subtype relationships. Often large OO software systems are constructed using subtype hierarchies and subtype polymorphism. If the subtype relation is not a valid one, then software systems based on such incorrect relationships produce unexpected results and hence, the whole software systems might fail. So understanding behavioral subtype relations is important for the correctness of OO software.

Behavioral subtyping is closely tied to aliasing allowed in a language. Aliasing allowed in programs determines the notion of behavioral subtyping and hence the kinds of reasoning technique that can be used to reason about the programs.

If subtype hierarchies are used in a context where there are very few aliases, then weak behavioral subtyping with appropriate alias restrictions should be used. This gives several practical type hierarchies as seen in Figure 3.4. It also gives a richer reasoning technique in the sense that it can make more conclusions about programs. Hence, we believe that these alias restrictions may actually be of some benefit.

However, if objects of these types are used in a context where one needs unrestricted aliasing then strong behavioral subtyping should be used. In such cases fewer conclusions can be made about programs. Programmers that use strong behavioral subtyping should be aware of the effects of behavioral subtyping on aliasing. It might be surprising to find out that identifiers of two different types can, in future, be aliased. This happens when one declares a common strong behavioral subtype to these two types.

The following are a set of guidelines that can be used as an informal checklist to define behavioral subtype relationships.

- What is the relationship between the abstract values of subtype objects and supertype objects? That is, how can one coerce a state that contains subtype objects to a state that contains supertype objects? This can be done with respect to a individual value without considering any aliasing.

- What are the effects of the common methods of the subtype? By common methods, we mean methods that are common to the subtypes and the supertypes. For valid behavioral subtype relationships, both weak and strong, these common methods in the subtypes should act like their corresponding methods in their supertypes. Not just the return values of the methods but the effects of these methods on the state should also be similar.

- What are the effects of the extra methods in the subtype? For weak behavioral subtypes, the behavior of the extra methods of the subtype need not be considered. That is, they can have extra mutability. However, in such cases the aliasing of identifiers of different types need to be restricted. For strong behavioral subtypes, the extra methods in the behavioral subtypes should satisfy the history constraint of the supertype.

- Are the alais restrictions needed are practical and can they be enforced in the system?

We believe that the answer to the last question is "yes". Though we show that the alias restrictions can be statically enforced in a language, we do not have any experience that it is actually the case in general.

In practice, if one does not use all the methods of a type, it is possible to identify a subset of the type specification that permits more behavioral subtype relations [Lea89]. However, any future uses that might use the excluded specification might produce surprising results. To prevent such surprising behavior, we recommend that the user define and use a new type that includes only the required subspecification.

7.2 Summary

The main contribution of this dissertation is the definition of a new notion of behavioral subtyping in the context of mutation and aliasing that allows subtype objects to be passed in place of supertype objects without any surprises. Other important contributions are a model-theoretic formulation of strong behavioral subtyping, a new technique to statically enforce alias restrictions on programs, and a new model for mutable types.

We presented a new, weaker notion of behavioral subtyping that allows interesting behavioral subtype relationships between mutable types and immutable types. We have shown that this notion is sound and argued that it allows supertype abstraction as a sound and modular reasoning principle.

A notion of nominality is defined in the context of mutable types and we formulate the necessary aliasing restrictions to construct these nominal states. Since supertype abstraction principle uses nominal states, these alias restrictions need to be enforced by any reasoning technique that uses supertype abstraction principle to reason about programs that use mutable types. We present a multi-method language, $OBS^{\leq}$, and show that these alias restrictions can be statically enforced. We show the soundness of the type and alias rules that enforce alias restrictions for $OBS^{\leq}$.

We also defined a strong behavioral subtype relation that allows all forms of aliasing. This notion uses history constraints that specifies invariant properties of objects across two different stores. We discuss the kinds of modular reasoning that can be done when strong behavioral subtyping is used in programs. We prove a soundness result for strong behavioral subtyping and argued that this notion is adequate for a modular reasoning technique that can make conclusions based only on the history constraints.

APPENDIX A. OMITTED DETAILS OF ALGEBRA E

We complete details of the algebra **E** omitted in Chapter 2 in this appendix. The following are the omitted operation interpretations of **E** from Figure 2.6.

$$
\begin{array}{lcl}
\texttt{nothing}^{\mathbf{E}}(\sigma) & \overset{\text{def}}{=} & (*, \sigma) \\
\texttt{change}^{\mathbf{E}}(m^m, i^i, \sigma) & \overset{\text{def}}{=} & (*, [m^m \mapsto i^i]\sigma) \\
\texttt{get\_interest}^{(}f^f, \sigma) & \overset{\text{def}}{=} & \textbf{let } m = (\sigma\ f^f) \textbf{ in} \\
 & & \textbf{if } (\texttt{value}^{\mathbf{E}}(m, \sigma) < 1000) \\
 & & \quad \textbf{then } \texttt{alloc[MoneyObj]}(0, \sigma) \\
 & & \textbf{else } \texttt{alloc[MoneyObj]}(50, \sigma) \\
\texttt{get\_interest}^{\mathbf{E}}(p^p, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ f^f) \textbf{ in} \\
 & & \textbf{if } ((\sigma\ m_s) + (\sigma\ m_c)) < 1000) \\
 & & \quad \textbf{then } \texttt{alloc[MoneyObj]}(0, \sigma) \\
 & & \textbf{else } \texttt{alloc[MoneyObj]}(50, \sigma) \\
\texttt{interest}^{\mathbf{E}}(b^b, \sigma) & \overset{\text{def}}{=} & \textbf{let } m = (\sigma\ b^b) \textbf{ in} \\
 & & \textbf{if } (\texttt{value}^{\mathbf{E}}(m, \sigma) < 1000) \textbf{ then } (*, \sigma) \\
 & & \textbf{else } (*, [m \mapsto ((\sigma\ m) + 50)]\sigma) \\
\texttt{interest}^{\mathbf{E}}(p^p, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ p^p) \textbf{ in} \\
 & & \textbf{if } ((\sigma\ m_s) + (\sigma\ m_c)) < 1000) \textbf{ then } (*, \sigma) \\
 & & \textbf{else } (*, [m_s \mapsto ((\sigma\ m_s) + 50)]\sigma)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{deposit}^{\mathbf{E}}(b^b, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } m = (\sigma\ b^b) \textbf{ in} \\
 & & (*, [m \mapsto ((\sigma\ m) + (\sigma\ m^m))]\sigma) \\
\texttt{deposit}^{\mathbf{E}}(p^p, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ p^p) \textbf{ in} \\
 & & (*, [m_s \mapsto ((\sigma\ m_s) + (\sigma\ m^m))]\sigma) \\
\texttt{check\_deposit}^{\mathbf{E}}(p^p, m^m, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ p^p) \textbf{ in} \\
 & & (*, [m_c \mapsto ((\sigma\ m_c) + (\sigma\ m^m))]\sigma) \\
\texttt{check\_balance}^{\mathbf{E}}(p^p, \sigma) & \overset{\text{def}}{=} & \textbf{let } (m_s, m_c) = (\sigma\ p^p) \textbf{ in } (m_c, \sigma)
\end{array}
$$

Omitted details of internal operations interpretations in Figure 2.7 are given below.

$$\begin{array}{lcl}
\texttt{emptyStore}^{\mathbf{E}}() & \stackrel{\text{def}}{=} & (\lambda(x).\bot) \\
\texttt{emptySet}^{\mathbf{E}}() & \stackrel{\text{def}}{=} & (\lambda(x).\texttt{false}) \\
\texttt{isIn}^{\mathbf{E}}(l, s) & \stackrel{\text{def}}{=} & (s\ l) \\
\texttt{addSet}^{\mathbf{E}}(l, s) & \stackrel{\text{def}}{=} & (\lambda(x).\mathbf{if}\ (x = l)\ \mathbf{then}\ \texttt{true} \\
 & & \mathbf{else}\ (s\ x)) \\
\texttt{isInDom}^{\mathbf{E}}(l, \sigma) & \stackrel{\text{def}}{=} & l \in \mathit{Domain}(\sigma) \\
\texttt{lookUp}^{\mathbf{E}}(l, \sigma) & \stackrel{\text{def}}{=} & (\sigma\ l) \\
\texttt{update}^{\mathbf{E}}(l, v, \sigma) & \stackrel{\text{def}}{=} & [l \mapsto v]\sigma
\end{array}$$

For each $T \in \mathit{ObjectTypes}(\mathbf{E})$,

$$\begin{array}{lcl}
\texttt{alloc}[T](v, \sigma) & \stackrel{\text{def}}{=} & \mathbf{let}\ l = l^{T}_{1+lub\{i | l^{T}_{i} \in Domain(\sigma)\}}\ \mathbf{in} \\
 & & (l, [l \mapsto v]\sigma)
\end{array}$$

APPENDIX B. OMITTED DETAILS OF FIGURE 4.4

Method implementation for operations given in algebra **E**.

```
method mkMoneyObj(i: Integer):MoneyObj
  is
  do nothing
  return new MoneyObj (i);
method value(m: MoneyObj):Integer
  is const result: Integer := m.money;
  do nothing
  return result;
method change(m: MoneyObj, i: Integer): Void
  is
  do m.money = m.money + i;
  return nothing;
method mkBankAccount(m: Money):BankAccount
  is const money: MoneyObj := new MoneyObj (value(m));
     const result: BankAccount := new BankAccount (money)
  do
     nothing
  return result;
method mkPlusAccount(m1: Money, m2:Money):PlusAccount
  is const mny1: MoneyObj := new MoneyObj (value(m1));
     const mny2: MoneyObj := new MoneyObj (value(m2));
     const result: FrozenAccount := new PlusAccount (mny1, mny2)
  do
     nothing
  return result;
method balance(f: FrozenAccount): MoneyObj
  is const m: MoneyObj := new MoneyObj (value(f.acct))
  do nothing
  return m;
method get_interest(f: FrozenAccount):MoneyObj
```

```
  is const m: MoneyObj := new MoneyObj (0)
  do if (f.acct < 1000) then nothing
      else change(m, 50)
  return m;
method get_interest(p: PlusAccount):MoneyObj
  is const m: MoneyObj := new MoneyObj (0)
  do if ((p.svgs + p.chkg)  < 1000) then nothing
      else change(m, 50)
  return m;
method balance(b: BankAccount): MoneyObj
  is const m: MoneyObj := new MoneyObj (value(b.acct))
  do nothing
  return m;
method balance(p: PlusAccount): MoneyObj
  is const m: MoneyObj := new MoneyObj (value(p.svgs + p.chkg))
  do nothing
  return m;
method withdraw(b: BankAccount, m: Money):Void
  is const i: Integer := value(b.acct);
  is const i2: Integer := value(m)
  do if (i > i2) then change(b.acct, (i - i2))
      else withdraw(b,m)
  return nothing;
method withdraw(p: PlusAccount, m: Money):Void
  is const i_s: Integer := value(p.svgs);
  is const i_c: Integer := value(p.chkg);
  is const i2: Integer := value(m)
  do if (i_s > i2) then change(p.svgs, (i_s - i2))
      else if ((i_s + i_c) > i2)
              then change(p.svgs, 0);
                   change(p.chkg, (i_c - (i_s -i2)))
              else withdraw(p, m)
  return nothing;
method interest(b: BankAccount):Void
  is
  do if (value(b.acct) < 1000) then nothing
       else change(b.acct, (value(b.acct) + 50))
  return nothing;
method interest(p: PlusAccount):Void
  is const i_s: Integer := value(p.svgs);
  is const i_c: Integer := value(p.chkg)
  do if ((i_s + i_c) < 1000) then nothing
       else change(p.svgs, (value(p.svgs) + 50))
  return nothing;
method deposit(b: BankAccount, m: MoneyObj):Void
  is const i0: Integer := value(b.acct);
     const i1: Integer := value(m)
  do change(b.acct, (i0 + i1));
method deposit(p: PlusAccount, m: MoneyObj):Void
  is const i0: Integer := value(p.svgs);
     const i1: Integer := value(m)
  do change(p.svgs, (i0 + i1));
method check_deposit(p: PlusAccount, m: MoneyObj):Void
  is const i0: Integer := value(p.chkg);
     const i1: Integer := value(m)
  do change(p.chkg, (i0 + i1));
```

BIBLIOGRAPHY

[AL97] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, Proceedings of Theaory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, pages 682-696, *Lecture Notes in Computer Science, Springer-Verlag*, volume 1214, New York, N.Y., 1997.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, ECOOP '87, European Conference on Object-Oriented Programming, Paris, France, pages 234–242, *Lecture Notes in Computer Science*, volume 276, Springer-Verlag, New York, N.Y., 1987.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990, pages 60-90, *Lecture Notes in Computer Science*, volume 489, Springer-Verlag, New York, N.Y., 1991.

[BW90] 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.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In D. B. MacQueen G. Kahn and G. Plotkin, editors, Semantics of Data Types: International Symposium, Sophia-Antipolis, France, *Lecture Notes in Computer Science*, pages 51–66. volume 173, Springer-Verlag, New York, N.Y., June 1984. A revised version of this paper appears in *Information and Computation*, volume 76, numbers 2/3, pages 138–164, February/March 1988.

[Car91] 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, N.Y., 1991.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole L. Madsen, editor, ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands, *Lecture Notes in Computer Science*, pages 33–56, volume 615, Springer-Verlag, New York, N.Y., 1992.

[CL94] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In OOPSLA '94 Conference Proceedings, volume 29(10) of *ACM SIGPLAN Notices*, pages 1–15, October 1994.

[Coo92] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In OOPSLA '92 Proceedings, Andreas Paepcke (editor), volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15, October 1992.

[Cus91] E. Cusack. Inheritance in object oriented Z. In Pierre America, editor, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Geneva Switzerland, *Lecture Notes in Computer Science*, pages 167–179, volume 512, Springer-Verlag, New York, N.Y., 1991.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Los Alamitos, CA, March 1996.

[EM85] Hartmut Ehrig and Bernd Mahr. Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, N.Y., 1985.

[GD94] Joseph A. Goguen and Razvan Diaconescu. Towards an algebraic semantics of the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, Recent Trends in Data Type Specification, pages 1–29, *Lecture Notes in Computer Science*, Volume 785, Springer-Verlag, New York, N.Y., 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[GM87] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce

Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. The MIT Press, Cambridge, Mass., 1987.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishing Co., Reading, Mass., 1984.

[Gur91] Yuri Gurevich. Evolving algebras: A tutorial introduction. *Bulletin of the EATCS*, 43:264–284, February 1991.

[LD94] 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.

[Lea89] Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989.

[Lis88] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[LLRS95] C. Lewerentz, Th. Lindner, A. Rüping, and E. Sekerinski. On object-oriented design and verification. volume 1009 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, New York, N.Y., 1995.

[LP92] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. In Stephen Brookes, editor, Mathematical Foundations of Programming Semantics '91, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[LP94] Gary T. Leavens and Don Pigozzi. The behavior-realization adjunction and generalized homomorphic relations. Technical Report 94-18a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 1994. To appear in *Theoretical Computer Science*. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[LW90] 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.

[LW93a] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In Oscar M. Nierstrasz, editor, ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, N.Y., July 1993.

[LW93b] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. OOPSLA '93 Proceedings, Andreas Paepcke (editor), volume 28(10), *ACM SIGPLAN Notices*, pages 16–28, October 1993.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mas86] I. A. Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[Mos92] Peter D. Mosses. Action Semantics, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, N.Y., 1992.

[MT91] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–328, July 1991.

[MT92] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on Logic in Computer Science.* IEEE, 1992.

[Nip86] Tobias Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[Rey80] 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.

[Rey85] John C. Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85), volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[Sch91] Oliver Schoett. An observational subset of first-order logic cannot specify the behavior of a counter. In C. Choffrut and M. Jantzen, editors, STACS 91 8th Annual Symposium on Theoretical Aspects of Computer Science Hamburg, Germany, February 1991 Proceedings, volume 480 of *Lecture Notes in Computer Science*, pages 499–510. Springer-Verlag, New York, N.Y., 1991.

[Sch94] David A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon, volume 21(11) of *ACM SIGPLAN Notices*, pages 38–45, November 1986.

[Spi88] J. M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics.* Cambridge University Press, New York, N.Y., 1988.

[Wag92] Eric G. Wagner. Some mathematical thoughts on languages for data directed design. In Rattray & Clark, editor, *The Unified Computation Laboratory*, pages 3–26. Oxford University Press, Cambridge, U.K., 1992.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice-Hall, New York, N.Y., 1991.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76. ACM, January 1989.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, N.Y., 1990.

[AZ93] Egidio Astesiano & Elena Zucca. D-oids: a model for dynamic data-types. In *Mathematical Structures in Computer Science* Vol. 11, Cambridge University Press, New York, NY, 1993.