Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

Chapter 7

History-based reasoning about behavioral subtyping

Behavioral subtyping [28], a concept predicated on the behavior of objects, is a key principle in object-oriented programming. This principle ensures that a subtype should seamlessly substitute its supertype without affecting the desirable properties or expected behavior of a program. The importance of this concept is particularly evident in the development of type hierarchies and type systems of programming languages that enable polymorphism and inheritance.

In this chapter, we introduce a new history-based proof-theory for reasoning about behavioral subtyping in class and interface hierarchies. Our approach is based on a semantic definition of types in terms of sets of sequences of method calls and returns, so-called histories. Behavioral subtyping is then naturally defined semantically as a set-theoretic subset relation between sets of histories, modulo a *projection relation* that captures the syntactic subtype relation. The main contribution is a Hoare-style proof theory for the specification and verification of the behavioral subtyping relation in terms of histories, abstracting from the underlying implementation. Through the use of a banking example we show the practical applicability of our approach.

This chapter is based on the following publication and artifact:

- **Bian, J.**, Hiep, H.A., de Boer, F.S. History-based Reasoning about Behavioral Subtyping. (Submitted for publication.)
- **Bian, J.**, Hiep, H. A., de Boer, F. S. (2024). History-Based Reasoning about Behavioral Subtyping: Proof files.
  https://doi.org/10.5281/zenodo.10998227

7.1 Introduction

The *programming to interfaces* discipline is one of the most important principles in software engineering. This methodology allows the developer of client code to abstract away from internal implementation details, such as object state, thereby aiding modular program development. Type hierarchies support this principle in object-oriented design by allowing the declaration of new subtypes that inherit properties and behaviors from their supertypes, while also providing the flexibility to add or override specific features as needed. The concept of *behavioral subtyping* (which refers to subtyping based on behavior, in contrast to nominal subtyping and structural subtyping [77]) ensures that in clients one should be able to replace the use of a supertype by a subtype without causing unexpected behavior [27, 78]. This concept is employed in object-oriented programming to ensure software maintainability and robustness.

Histories, as defined in our previous work [53], are sequences of method calls performed on the object. We define the semantics of a type as a set of histories, thus abstracting from the underlying state/implementation. This allows to define the behavioral subtype relation semantically as subset relation between sets of histories, modulo a projection relation between histories that corresponds with the syntactic definition of the subtype relation. For the specification and verification of the behavioral subtype relation we introduce method contracts using Hoare triples that involve suitable user-defined abstractions over histories, called *attributes*. We discuss behavioral subtyping in three settings: class-class inheritance, class-interface inheritance, as well as interface-interface inheritance.

There has been numerous research on behavioral subtyping [79, 80, 81, 82], starting from the seminal work by Liskov and Wing [28], who point out that a subtype must adhere to the behavioral contracts of its supertype. To define the subtype relation, they introduced an *abstraction function* that maps the state of each subtype to a state of its corresponding supertype. The soundness of the substitution principle follows from two conditions: the precondition of the supertype implies the precondition of the subtype, and the postcondition of the subtype implies the postcondition of the supertype. The pre/postconditions of the subtype speak of the state of the subtype, whereas the pre/postconditions of the supertype speak of a different state: so the abstraction function takes a state of the subtype and maps it to a state of the supertype in such a way that these conditions hold. Is worth mentioning that in Liskov and Wing's work, they introduce a notion of *history constraint*, which is different from our notion of a history. Their history refers to temporal properties of objects, which are used to declare a relationship between pre-states and post-states preserved by any method of a type [83]. Leavens and Weih [84, 85, 86] present a technique for the modular reasoning about object-oriented programs, called *supertype abstraction*, which allows adding behavioral subtypes without reverification. However, their method is based on the assumption that each specified subtype relation constitutes a behavioral subtype. Demonstrating such behavioral subtyping requires again the use of an abstraction function. Although there have been several logics for the reasoning about object-oriented programs including a notion of behavioral subtyping, such as [87, 88, 89], they are all based on the abstraction function.

In the field of refinement calculus [90, 91, 92], which focuses on the stepwise transformation of an abstract specification into an executable implementation, one also uses the abstraction functions. These functions help in mapping implementation to specification, ensuring that each refinement step is correct.

In contrast to the above related work the history-based reasoning approach in this paper avoids formulating ad hoc abstraction functions between different state-based implementations. Instead it is based on a general semantic definition of the behavioral sub-type relation as a subset relation between sets of histories modulo a projection relation. Further, our proof method is based on the use of suitable user-defined history abstractions which allows for a modular verification of the proof obligations. Finally, our approach is applicable to both interfaces and classes, and allows reasoning about behavioral subtyping in settings that are typically absent in most related studies [84, 93, 94, 95].

The paper is intentionally written to introduce and motivate a new idea rather than to work out all the formal details. We discuss the methodology of history-based behavioral subtyping in Sect. 7.2. Our specification methods are presented in the context of history and attributes. In Sect. 7.3, we use a banking example to illustrate our approach. We provide only informal proofs for three particular subtype relations: interface-interface, interface-class, and class-class. The part of this example is proven using the KeY theorem prover [23] and Isabelle/HOL [33]. The verification workflow is based on our previous work [96].

7.2 Methodology

In object-oriented programming, a method signature consists of a list of parameter types and a return type. An interface contains a set of method signatures. A class consists of a set of field declarations and a set of method declarations.

The type hierarchy for classes and interfaces in languages with a nominal type system can be declared as below:

$$\textbf{interface } I \text{ [} \textbf{extends } I_1, I_2, \ldots, I_n \text{]}$$
$$\textbf{class } C' \text{ [} \textbf{extends } C \text{] [} \textbf{implements } I_1, I_2, \ldots, I_n \text{]}$$

An interface can extend zero or more interfaces, which is known as interface inheritance. When one interface extends another, it inherits all of the methods defined in its super interfaces, but it can also add new methods of its own. A class can inherit from multiple interfaces, by providing implementations for all methods defined in the interfaces. However, a class can only inherit from a *single* class. This is due to the fact that class inheritance is typically used for defining the (memory) structure of a class. Allowing multiple inheritance of classes can potentially lead to conflicts among class invariants [97] and ambiguity, as exemplified by the so-called diamond problem [98].

The basic behavioral notion of subtyping discussed in [28] is shown as in Fig. 7.1. A Hoare triple specification, denoted as $\{p\}\, m\, \{q\}$, consists of a method m, a pre-

condition p that describes the object state before the method is executed, and a postcondition q that describes the expected object state after the method is executed. In Fig. 7.1, we have $\{p\}\, m\, \{q\}$ on top and $\{p'\}\, m\, \{q'\}$ below, which represent the supertype and subtype specification of m, respectively, where m is a method inherited by the subtype from the supertype.

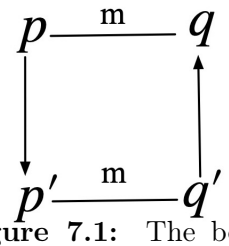

Figure 7.1: The behavioral notion of subtyping.

From the perspective of a client, we ensure before a method call that the precondition of the supertype holds, so that after dynamic dispatch where we jump to the implementation of the subtype, we also need that the precondition of the corresponding method in the subtype holds. After the execution of the subtype method finishes, it reaches the postcondition of the subtype's method. This postcondition should also imply the postcondition of the method in the supertype, since the client assumes that the postcondition of the supertype holds after the method returns. Moreover, if both types are classes then the invariant of the supertype must be preserved in the subtype.

However, typically the precondition and postcondition, given in some specification language, are intrinsically state-based and as such are not directly suitable for the specification of a state-hiding interface. In history-based reasoning, we introduce the concept of a history that can be seen as the most general abstraction of the state space of an interface. There are two approaches: the executable history-based (EHB) approach [53], and the logical history-based (LHB) approach [96]. In the former approach, histories become part of the run-time environment and are encoded as objects. In the latter approach, histories do not exist at run-time and are only introduced as bookkeeping devices for reasoning, similar to ghost variables. We proceed with the latter approach. In the LHB approach, histories are modeled as elements of an abstract data type (ADT). This means histories are immutable and inaccessible: no program can modify or even inspect a history value.

A history is a sequence of events. Every method is represented by a corresponding event type, that records the types of the parameters and the type of the return value. For technical convenience, we only regard normal returns from method calls as events. For each class and interface, we introduce a history type by defining it as an inductive data type of sequences of events.

Following the information hiding principle, we assume an object encapsulates its own state. Consequently, each object can enforce invariants over its own fields and its state can be completely determined by the sequence of method calls invoked on the object. Attributes are user-defined abstractions of histories that are in general defined inductively over the history. These attributes are used in method specifications to specify the intended behavior of implementations, and by using attributes the method specifications do not depend on the (hidden) state of an object.

The overall approach in history-based reasoning can be summarized by the following diagram, see Fig. 7.2. We will now provide more details on each of the components in Fig. 7.2.

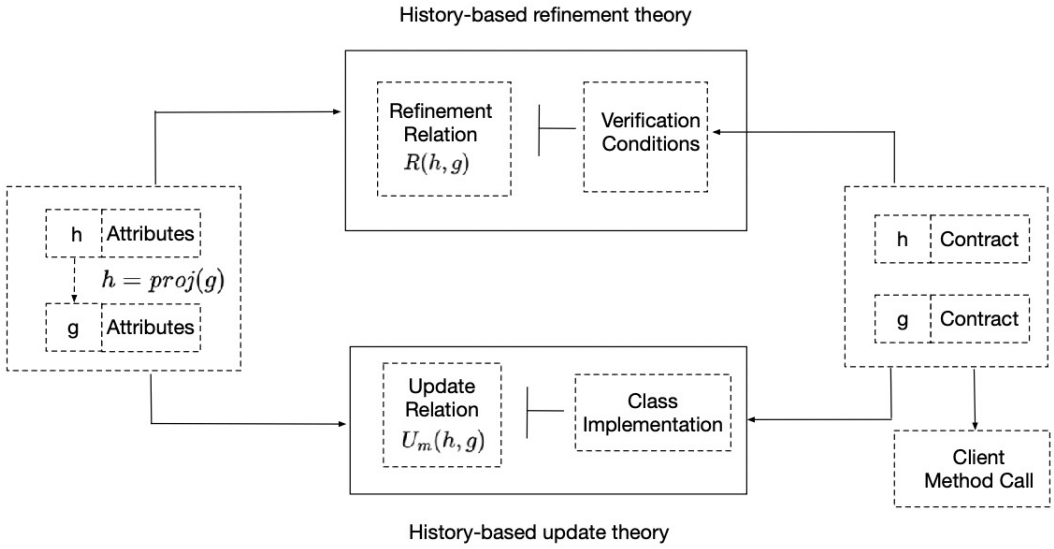

Figure 7.2: History-based reasoning about behavioral subtyping. In $R(h, g)$, h represents the history of the supertype and g represents the history of the subtype. In $U_m(h, g)$, h represents the history in the post-state, g represents the history in the pre-state and m is the corresponding method.

7.2.1 History-based refinement theory

To establish that the behavioral subtype relation holds between two types, we define a set of proof obligations between the preconditions and postconditions of methods inherited by a subtype. For a modular verification of these proof obligations we introduce a methodology that consists of two parts: verification of the refinement relation and, separately, verification of the proof obligations generated from method specifications, assuming the refinement relation. The method specifications refer to the attributes of the associated history, and abstract from the inductive definition of the history and its attributes. The *refinement relation* on the other hand captures logically the relationship between attributes of *different* histories, namely the histories of the supertype and the subtype. The axioms of the refinement relation itself, as a logical theory, should be established as logical consequences of the inductive definitions of the attributes and the projection relation.

This assumption can be justified as follows. When a method from a subtype that inherits from a supertype in the hierarchy is called, updates are made to both the histories of supertype and subtype. However, for methods only present in the subtype, updates are made only to the history of the subtype, while that of the supertype remains unchanged. This design choice is intentional to avoid the potential issues that may occur if the subtype is cast to the supertype. More general approaches, where the history of a subtype can be simulated by a history of the supertype, are out of scope in this paper. For any given histories h and g, where h is a projection of g, the user-defined refinement relation $R(h, g)$ describes a logical relation between the attributes of h and g, *abstracting* from their inductive definitions. Note that attributes in general may have different meanings when interpreted by the history of a supertype or by the history of a subtype.

The proof obligations, also called verification conditions, for interface-interface refinement, interface-class refinement and class-class refinement are shown below. For a supertype, h represents the history of the supertype, p represents the precondition and q represents the postcondition. For a subtype, g represents the history of the subtype, p' represents the precondition and q' represents the postcondition. For classes, I and I' denote the superclass and the subclass invariant, respectively. Class invariants in general describe a (logical) relation between the fields of a class and the attributes of the assoctated class. In proving the verification conditions for the pre- and postconditions of the inherited methods, the refinement relation is assumed. It should be noted that, by using the logical consequence relation $\vdash$, the history variables h and g are implicitly universally quantified (on both sides of $\vdash$). The refinement relation itself involves a separate proof obligation which is formulated by

$$h = proj(g) \rightarrow R(h, g)$$

That is, the logical relation between the attributes of the histories of the supertype and the subtype should follow from the projection relation and their inductive definitions.

Verification Condition IIR (Interface-Interface Refinement)**.**

$$R(h, g) \vdash (p \rightarrow p')$$
$$R(h, g) \vdash (q' \rightarrow q)$$

Verification Condition ICR (Interface-Class Refinement)**.**

$$R(h, g) \vdash (p \wedge Inv' \rightarrow p')$$
$$R(h, g) \vdash (q' \wedge Inv' \rightarrow q)$$

Verification Condition CCR (Class-Class Refinement)**.**

$$R(h, g) \vdash (Inv' \rightarrow Inv)$$
$$R(h, g) \vdash (p \wedge Inv' \rightarrow p')$$
$$R(h, g) \vdash (q' \wedge Inv' \rightarrow q)$$

7.2.2 Verifying method call and method implementation

In the usual manner, method calls are verified in terms of the corresponding method specification (as determined by the *static* type of the callee expression). This involves the usual substitution of the formal parameter by the actual parameter. More specifically, a method specification $\{p\}\, m(\bar{u})\, \{q\}$ can be instantiated to a method call $x = y.m(\bar{e})$ by substituting **this** with the calling object y and the method parameters $\bar{u}$ with the actual arguments $\bar{e}$ in the preconditions and postconditions.

To validate the postcondition of a method body, which specifies the corresponding update of the associated history, we assume in the following method implementation rule a logical *update relation* $U_m(h, h')$ between the attributes of the updated history h and the 'old' history h'.

Rule 1 (Method implementation Rule)**.** Given the method definition $\{p\}\, m\, \{q\}$, the

body S of m, and the class invariant I, we have the rule

$$\frac{\{p \wedge I\}\, S\, \{r\} \quad U_m(h, h') \vdash r \rightarrow (q \wedge I)}{\{p\}\, m\, \{q\}}$$

This rule thus allows to abstract from the inductive definitions of the history attributes in the validation of the method body. The logical *update relation* $U_m(h, h')$ between the attributes of the updated history h and the 'old' history h', then can be established separately as a logical consequence of $h = Cons(m(\bar{x}, \textbf{result}), h')$ which directly describes the relation between the updated history and the old one in terms of their sequence structure. Here $\bar{x}$ are the actual parameters and **result** is the return value. The **result** variable here may either be *null* (indicating no return value) or contain a return value.

7.3 Case study

In this section, we introduce a banking example to illustrate the methodology discussed in Section 2. This example features a type hierarchy, which allows us to demonstrate our ideas effectively. It also presents some interesting challenges that occur in real-world programs, such as how to enforce protocol at the interface level where we have no access to the underlying state. We also consider some real-world scenarios, like how to extend functionality in existing programs. We implement the case study by defining the ADTs in Isabelle/HOL, and used ADTs in specification and KeY proof for Java programs. The artifact accompanying this paper [99] includes the full Isabelle theory of banking example and the proof files for the example discussed later.

In the banking example, we have two interfaces: the `Saving` interface and the `Payment` interface.

```
interface Saving {
   void deposit(int i);
   int getbalance();
}
```

Listing 7.1: The `Saving` interface.

```
interface Payment extends Saving {
   boolean query(int i);
   void withdraw();
}
```

Listing 7.2: The `Payment` interface.

The `Saving` interface, as shown in Listing 7.1, specifies methods for depositing an integer amount into the account and for retrieving the current balance. The `Payment` interface (Listing 7.2) defines two methods: the `query` method and the `withdraw` method. The `query` method is used to check whether there are sufficient funds in the account before each withdrawal.

The method signatures of the interface are designed to allow for the expression of the intended protocol, similar to how interfaces in Java Collection Framework are explained in informal Javadoc documentation [71]. The protocol for the `Payment` interface stipulates that the `withdraw` method can only be invoked when the return value of the `query` is true. This protocol is designed to protect the interface from

executing invalid withdrawal operations that could potentially lead to errors in the system (e.g. increasing the balance by calling `withdraw` multiple times).

We have three classes for our example, the `Account` class, the `Credit` class and the `Debit` class. The `Account` class (Listing 7.3) implements the methods defined in the `Saving` interface.

```
class Account implements Saving {
   int balance; // field
   void deposit(int i) { balance=balance + i; }
   int getbalance() { return balance; } }
```

Listing 7.3: The `Account` class.

The `Credit` class (Listing 7.4) permits withdrawals even if the balance is insufficient, similar to a real-world credit card. The `query` method, which is designed to check whether there are sufficient funds in the account before each `withdraw` method: in the case of the credit card, the caller always receives an affirmative response.

```
class Credit implements Payment {
   int request = −1; int balance; // fields
   void deposit(int i) { balance=balance + i; }
   int getbalance() { return balance; } }
   boolean query(int i) { request = i; return true; }
   void withdraw() { balance = balance − request; request = −1; }}
```

Listing 7.4: The `Credit` class.

In contrast, the `Debit` class (Listing 7.5) allows withdrawals only if the account has sufficient funds. This condition is determined by the return value of the `query` method. Specifically, it is true if and only if the balance is greater than or equal the argument.

```
class Debit extends Account implements Payment {
   int request = −1; // field
   boolean query(int i) {
        if (balance ≥ i) { request = i; return true; } else return false; }
   void withdraw() { balance = balance − request; request = −1; }}
```

Listing 7.5: The `Debit` class.

The hierarchical structure for our running example is depicted in Fig. 7.3.

7.3.1 History-based reasoning

In this subsection, we illustrate how we formalize ADTs for banking examples. We define data types and functions to logically model domain-specific knowledge of the Java program that we want to verify. Although these definitions cannot directly reference Java types, they can instead be defined using polymorphic type parameters. One defines data types and recursive functions using the **datatype** and **fun** commands.

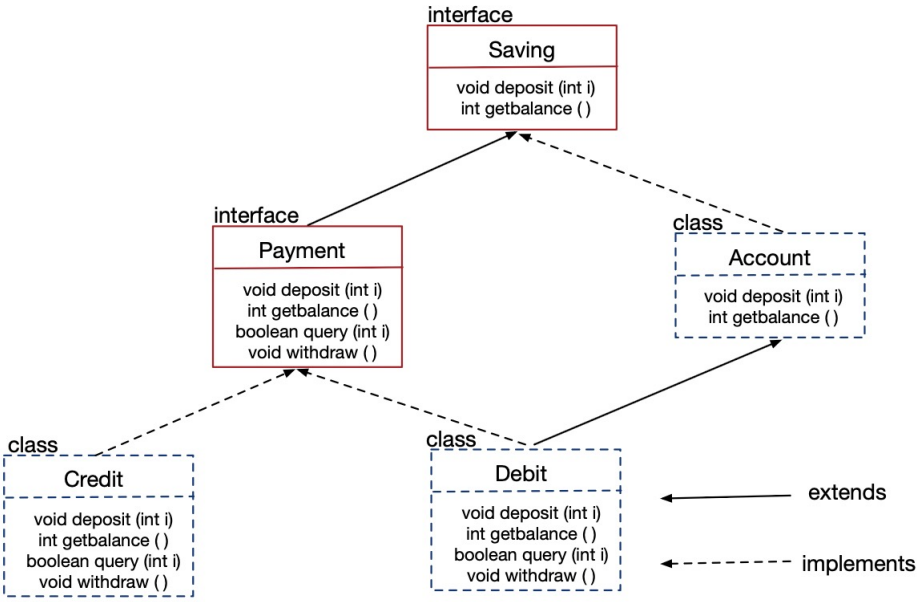

Figure 7.3: The type hierarchy of our running example.

The data types *events* contain the method name, the actual parameter values, and the output value (which is the final argument of the event) of a method call. The events are designed to be generic and do not contain information about the caller. This design choice makes events useful for a general history-based theory that is related to the caller. With regard to methods, there are methods like `deposit` that take an integer as a parameter but have no return value, whereas the `getbalance` method takes no input parameters but returns an integer. To distinguish between the input type and output type, we use a unit type **void** to represent the absence of a meaningful value. As discussed above, each subtype includes the events that are inherited from its supertype. For example, the definitions of the events for the interfaces in our running example are as follows:

$$\begin{aligned}\textbf{datatype}\ \mathit{saveEvent} &= \mathit{deposit}(\textbf{int}, \textbf{void}) \mid\ \mathit{balance}(\textbf{void}, \textbf{int})\\ \textbf{datatype}\ \mathit{payEvent} &= \mathit{deposit}(\textbf{int}, \textbf{void}) \mid\ \mathit{balance}(\textbf{void}, \textbf{int}) \mid\\ &\quad\ \mathit{query}(\textbf{int}, \textbf{bool}) \mid \mathit{withdraw}(\textbf{void}, \textbf{void})\end{aligned}$$

The concept of *history* is formally defined as an inductive data type of a sequence of events. Rather than employing temporal logic or formalizing history as an indexed set of events, we find that inductive data types offer a more convenient approach for defining attributes by induction and are easier to integrate with theorem provers in general. Thus, we introduce the history as a parameterized inductive datatype:

$$\textbf{datatype}\ \mathit{history}(\alpha) = \mathit{Empty} \mid\ \mathit{Cons}(\alpha, \mathit{history}(\alpha))$$

The type parameter α corresponds to the type of event occurring in the history, such as *saveEvent* and *payEvent*. For example, we can instantiate the parameter by the datatype *saveEvent* to obtain the histories for the `Saving` interface, which is represented as $\mathit{history}(\mathit{saveEvent})$. The history data type uses the constructors *Empty* and *Cons*, indicating that the history is either empty or composed of an event as its head and another history as its tail. When a new event is added, the new

event, along with its argument and return type, becomes the head of the history, while the old history turns to become the tail. It is worth mentioning that a history is generated in reverse order, which means that the last generated event appears at the start of the sequence.

The *amount* attribute, defined below, denotes the total amount of money in a given account. Intuitively, it serves as a snapshot representation of the interface's 'contents' at a particular instant. In the `Saving` interface, *amount* is defined inductively over the structure of the saving history, as shown below: a successful deposit increases the amount of money according to the value of the provided argument. We use *null* to represent a constant of type **void**.

$$
\begin{aligned}
&\textbf{fun}\ amount : history(saveEvent) \Rightarrow \textbf{int} \\
&\quad amount(Empty) = 0 \\
&\quad amount(Cons(deposit(i, null), h)) = amount(h) + i \\
&\quad amount(Cons(balance(null, i), h)) = amount(h)
\end{aligned}
$$

Attributes of history are treated similarly as "fields" in a class. To be specific, attributes defined by a supertype can be freely used and reinterpreted by its subtype. In our case, we redefine the *amount* attribute based on the payment history, taking the new methods `query` and `withdraw` into account.

$$
\begin{aligned}
&\textbf{fun}\ amount : history(payEvent) \Rightarrow \textbf{int} \\
&\quad amount(Empty) = 0 \\
&\quad amount(Cons(deposit(i, null), h)) = amount(h) + i \\
&\quad amount(Cons(balance(null, i), h)) = amount(h) \\
&\quad amount(Cons(query(i, b), h)) = amount(h) \\
&\quad amount(Cons(withdraw(null, null), h)) = \\
&\qquad\quad amount(h) - (ready(h)\ ?\ take(h) : 0)
\end{aligned}
$$

We define attributes *ready* and *take* as follows: given a history, *ready* checks whether the previous query event has returned **true**, and if so, *take* returns the parameter of `query` method.

$$
\begin{aligned}
&\textbf{fun}\ ready : history(payEvent) \Rightarrow \textbf{boolean} \\
&\quad ready(Empty) = \textbf{false} \\
&\quad ready(Cons(query(i, b), h)) = b \\
&\quad ready(Cons(withdraw(null, null), h)) = \textbf{false} \\
&\quad ready(Cons(e, h)) = ready(h) \\
&\textbf{fun}\ take : history(payEvent) \Rightarrow \textbf{int} \\
&\quad take(Empty) = -1 \\
&\quad take(Cons(query(i, b), h)) = (b\ ?\ i : -1) \\
&\quad take(Cons(withdraw(null, null), h)) = -1 \\
&\quad take(Cons(e, h)) = take(h)
\end{aligned}
$$

In the last clause, e is any event of *payEvent* not specified in the clauses above.

7.3.2 History-based specification

We now formulate method contracts of the methods of interface and class, making use of histories and its attributes. By overloading field access notation, we can treat the attributes of a history associated with **this** like how we would treat an unqualified field. For example, when considering the *amount* defined in the `Saving` interface, we can use syntactic sugar to simplify $amount(h)$ which h is of type $history(saveEvent)$ to just *amount* within the `Saving` interface. For external objects, we can explicitly indicate the object of which the corresponding history is taken in an attribute. Listing 7.6 shows a concrete example: suppose we would add a default method `transfer` to the `Payment` interface, that performs a withdrawal and immediately transfers the amount to the given `Saving` instance, then the postcondition illustrates that the amount of (the history of) **this** decreases, while the amount of (the history of) the receiver increases. Moreover, this example illustrates why hiding the concrete structure of a history from specifications is useful: while the default implementation does not record *transfer* as an event in the history of `Payment` and instead records the events which are used by the default implementation (in our case *withdraw*), non-default implementations do record a *transfer* event, thus have a different structure of the history.

```
// Transfer money from this Payment account to the given Saving account
{ready = true ∧ take = i}
default void transfer(int i, Saving s) { withdraw(); s.deposit(i); }
{(s ≠ this → amount = old(amount)−i ∧ s.amount = old(s.amount)+i) ∧
 (s = this → amount = old(amount))}
```

Listing 7.6: An example to specify the object of a history explicitly.

To avoid introducing a logical *freeze* variable, which would capture the history as it were in the pre-state, we use the notation **old** as a logical operation on terms to denote the attribute value evaluated in the pre-state of the method call, where **old** distributes over pure operations such as arithmetical functions. In the postcondition, *amount* in our example refers to the amount after the method call, while **old**(*amount*) represents the amount before the method call.

An interface specification includes the name of the interface being specified and the method signatures that the interface provides along with their respective precondition and postcondition. Listing 7.7 illustrates the use of the history attribute in the specification of the `Saving` interface.

```
Specification(Saving) =
({i ≥ 0} void deposit(int i) {amount = old(amount)+i},
 {true} int getbalance() {amount = old(amount) ∧ result = amount})
```

Listing 7.7: The specification of the `Saving` interface in terms of attribute *amount*.

The special variable **result** in the postcondition captures the return value of a method.

The specification of the `Payment` interface in terms of the attribute *amount* is shown in Listing 7.8.

Specification(`Payment`) =
($\{\texttt{i} \geq 0\}$ **void** `deposit`(**int** `i`) $\{amount = \textbf{old}(amount)+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{amount = \textbf{old}(amount) \wedge \textbf{result} = amount\}$,
$\{\texttt{i} \geq 0\}$ **boolean** `query`(**int** `i`) $\{amount = \textbf{old}(amount) \wedge$
$\textbf{result} = ready \wedge (ready \rightarrow take = \texttt{i})\}$,
$\{ready\}$ **void** `withdraw`() $\{amount = \textbf{old}(amount-take) \wedge \neg ready\}$)

Listing 7.8: The specification of the `Payment` interface in terms of attributes *amount*.

One should observe that the return value of the `query` method remains unspecified, thereby leaving design decisions open for subtypes and implementors. The intended meaning of the `query` method is to check whether money can be withdrawn from the account. Two implementations can be considered: a credit account (see Listing 7.10) and a debit account (see Listing 7.11). It is not possible to further specify the result in the `Payment` interface in a way that is compatible with both subtypes.

In addition to the type name and method specifications, a class specification may also contain the *class invariant*. The class invariant is an essential component of the class specification that should hold in the pre- and post-state of each method execution [97]. The method specifications of a class are described in terms of both fields and history attributes. The specification of the `Account` class is shown in Listing 7.9.

Specification(`Account`) =
(`balance` $= amount \wedge$ `balance` ≥ 0, // *class invariant*
$\{\texttt{i} \geq 0\}$ **void** `deposit`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance})+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge \textbf{result} = \texttt{balance}\}$)

Listing 7.9: The specification of the `Account` class.

The specification of the `Credit` class and the `Debit` class are present in Listing 7.10 and Listing 7.11, respectively.

Specification(`Credit`) =
(`balance` $= amount \wedge$ `request` $= take$ // *class invariant*
$\{\texttt{i}\geq 0\}$ **void** `deposit`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance})+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge \textbf{result} = \texttt{balance}\}$,
$\{\texttt{i}\geq 0\}$ **boolean** `query`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge$
$\textbf{result} = \textbf{true} \wedge \texttt{request} = \texttt{i}\}$,
$\{\texttt{request} \neq -1\}$ **void** `withdraw`() $\{\texttt{request} = -1 \wedge$
$\texttt{balance} = \textbf{old}(\texttt{balance}-\texttt{request})\}$)

Listing 7.10: The specification of the `Credit` class.

The value of **result** for the `query` method is explicitly specified in the classes `Debit` and `Credit` that implement the interface. Specifically, for the `Credit` class, **result** is unconditionally true. Conversely, for the `Debit` class, **result** is true when and

only when the *balance* $\geq i$. Note that these two conditions are not compatible: no debit object can be considered as a credit object.

```
Specification(Debit) =
(balance = amount ∧ request = take ∧ balance ≥ 0 ∧ balance ≥ request,
 {i ≥ 0} void deposit(i) {balance = old(balance)+i},
 {true} int getbalance() {balance = old(balance) ∧ result = balance},
 {i≥0} boolean query(i) {balance = old(balance) ∧ request = i ∧
                         result=(balance≥i)},
 {balance ≥ request ∧ request ≠ −1} void withdraw()
                    {request = −1 ∧ balance = old(balance−request)})
```

Listing 7.11: The specification of the `Debit` class.

The preconditions and postconditions effectively specify protocols for methods. For instance, suppose that the `withdraw` method should only be invoked following a valid query. In both the `withdraw` method in the `Debit` and `Credit` class, we impose a precondition constraint `request` $\neq -1$. When dealing with an interface method where we have no access to the underlying state, such as in the `Payment` interface (Listing 7.8), the protocol can describe using the attribute of history, specifically the *ready* attribute in this case. This allows us to capture the return value of a previous query, abstracting from the underlying implementation.

7.3.3 Behavioral subtyping

In this subsection, we discuss the refinement of interface-interface, interface-class, and class-class in the context of the banking example. Let us start with interface-interface refinement. The example we provide is the `deposit` method in the `Saving` interface (Listing 7.7) and its subtype, the `Payment` interface (Listing 7.8). First, we consider formulating the refinement relation. For user-defined refinement relation $R(h, g)$, h is of type $history(saveEvent)$, and g is of type $history(payEvent)$, we can formulate $R(h, g)$ according to the definition of *amount* for both the `Saving` and `Payment` interfaces, as provided below: every time the `withdraw` is called within the `Payment` interface and returns a *true* value, the *amount* attribute defined on the `Payment` history will decrease. To reflect this behavior, we introduce a new attribute, *withdrawamount*, to accumulate the total amount successfully withdrawn:

$$
\begin{aligned}
&\textbf{fun } withdrawamount : history(payEvent) \Rightarrow \textbf{int} \\
&\quad withdrawamount(Empty) = 0 \\
&\quad withdrawamount(Cons(withdraw(null, null), h)) = \\
&\qquad\qquad withdrawamount(h) + (ready(h)\ ?\ take(h) : 0) \\
&\quad withdrawamount(Cons(e, h)) = withdrawamount(h)
\end{aligned}
$$

again e is any *payEvent* not mentioned above.

Due to the introduction of the new attributes, we need to modify the method specification to capture the behavior of the interface. In this example, the method within the `Payment` interface requires modification for the introduction of attribute *withdrawamount*, as illustrated in Listing 7.12.

Spec(`Payment`) =
({`i`≥0} **void** `deposit`(`i`) {*amount*=**old**(*amount*)+`i` ∧
withdrawamount=**old**(*withdrawamount*)}, ...)

Listing 7.12: The `deposit` method specification for the `Payment` interface. The specifications for other methods within the `Payment` interface have also been revised.

We can then define the refinement relation as follows:

$$R(h, g) \stackrel{\text{def}}{\equiv} amount(h) = amount(g) + withdrawamount(g)$$

For $h : history(saveEvent)$ and $g : history(payEvent)$, we can formulate $R(h, g)$ by unfolding the attribute definition of h and g.

Now we apply Liskov and Wing's method rules to supertype and subtype in order to generate the verification conditions for behavioral subtyping. We first consider the implication between the preconditions of both types, where `i` serves as the actual parameter of the `deposit` method. The proof seems straightforward: $\texttt{i} \geq 0 \rightarrow \texttt{i} \geq 0$ is trivial. But what about postcondition $amount = \mathbf{old}(amount) + \texttt{i} \rightarrow amount = \mathbf{old}(amount) + \texttt{i}$? We cannot directly prove this due to the attribute *amount* of the `Saving` history is different from the attribute *amount* of the `Payment` history, as *amount* definition given in below. Even though attribute names may be identical, their definitions are specific to and can differ between different histories.

Instead, by de-sugaring and renaming the attribute, we explicitly get the *amount* of h, which is of type $history(saveEvent)$, and g, which is of type $history(payEvent)$. Since **old** distributes over pure operations, the designation of an attribute with the keyword **old** means to take the attribute of the old history, that is, the history prior to the method call. Thus, the expression $\mathbf{old}(amount(g))$ is equivalent to $amount(\mathbf{old}(g))$. We now have to show the following verification condition:

$$\begin{array}{c} amount(g) = amount(\mathbf{old}(g)) + \texttt{i} \\ \downarrow \\ amount(h) = amount(\mathbf{old}(h)) + \texttt{i} \end{array} \qquad \text{(VC1)}$$

However, the condition (VC1) remains unproven because there is a lack knowledge about the internal structure of the history and the definition of the attributes. To solve this issue, we use the *refinement relation* which allows us to relate the histories of supertype and subtype, and then we can further relate predicates about the supertype to those about the subtype and vice versa. By assuming $R(h, g)$, h represents the history of the supertype and g represents the history of the subtype, we can get the refinement relation for the history in the state where the method call started for free, that is $R(old(h), old(g))$. We then can simply prove the VC1.

$$\begin{array}{c} amount(g) = amount(\mathbf{old}(g)) + \texttt{i} \wedge \\ withdrawamount(g) = withdrawamount(\mathbf{old}(g)) \\ amount(h) = amount(g) + withdrawamount(g) \wedge \\ amount(\mathbf{old}(h)) = amount(\mathbf{old}(g)) + withdrawamount(\mathbf{old}(g)) \\ \downarrow \\ amount(h) = amount(\mathbf{old}(h)) + \texttt{i} \end{array}$$

The refinement for interface and class needs to take into account the class invariants. The specific example of interface-class refinement involves the precondition of the `withdraw` method of the `Payment` interface and its subclass, the `Debit` class. We can derive the following from their specifications (Listing 7.8 and Listing 7.11) based on the precondition rule:

$$\mathit{ready}(h) \rightarrow (\texttt{balance} \geq \texttt{request} \land \texttt{request} \neq -1) \tag{VC2}$$

In the attribute declaration $\mathit{ready}(h)$, the type of variable h is $\mathit{history}(\mathit{payEvent})$.

In this case, the `Debit` class fully inherits from the `Payment` interface. Thus, the refinement relation between them is as follows:

$$R(h, g) \stackrel{\text{def}}{\equiv} \mathit{ready}(h) = \mathit{ready}(g) \land \mathit{amount}(h) = \mathit{amount}(g)$$

The parameter h is an instance of the datatype $\mathit{history}(\mathit{payEvent})$, while g is an instance of the datatype $\mathit{history}(\mathit{debitEvent})$.

One can see that only the refinement relation is not sufficient to solve the VC2. The class invariant of the `Debit` class contains `balance` = *amount* and `request` = *take* which relates the attributes *amount* and *take* to the fields `balance` and `request`. By assuming the refinement relation, alongside the class invariants shown in Listing 7.11, we can prove the VC2: according to the definition of the attribute, if *ready* returns true, then *take* $\neq -1$.

$$\begin{gathered}
\mathit{ready}(h) \land \\
\texttt{balance} = \mathit{amount}(g) \ \land \ \texttt{request} = \mathit{take}(g) \\
\texttt{balance} \geq 0 \land \texttt{balance} \geq \texttt{request} \\
\mathit{ready}(h) = \mathit{ready}(g) \land \mathit{amount}(h) = \mathit{amount}(g) \\
\downarrow \\
\texttt{balance} \geq \texttt{request} \land \texttt{request} \neq -1
\end{gathered}$$

Now we turn to focus on class-class refinement. The relation between two classes needs to consider the use of class invariants in both the superclass and the subclass. If invariants can be employed in supertypes for reasoning, subtypes must also obey these invariants. To be specific, the complete invariant of a subclass specification is formed as a conjunction of both the invariant in the supertype and the unique invariant specific to the subtype itself. The class invariant, which typically connects the fields and attributes of history, can leverage refinement relation to prove the verification conditions. To be specific, given a refinement relation between a subclass and a superclass, if the class invariants for the subclass hold, it would logically follow that the class invariants for the superclass also hold. In the banking example, an invariant property of the `Account` class is that its balance is always greater or equal to zero. The `Credit` class allows one to withdraw money even if the balance is negative, so the `Credit` class cannot be a subclass of the `Account` class. Conversely, the `Debit` class inherits invariants from the specification of the `Account` class and also has its own invariants, as outlined in Listing 7.11.

We now delve deeper into the design problem of method placement within the type hierarchy, in order to emphasize the importance of adherence to the behavioral

subtyping rule. In most cases, developers have the flexibility to decide at what level a method should be placed. For instance, the `Payment` interface introduces two new methods: `query` and `withdraw`. One potential approach is to define the `withdraw` method within the `Payment` interface and introduce the `query` method exclusively as an addition to the `Debit` implementation. However, this approach clashes with behavioral subtyping, which requires the implication *ready* $\rightarrow$ `balance` $\geq$ `request` $\wedge$ `request` $\neq -1$ to hold, but *ready* cannot be given a sensible meaning at the level of the `Payment` interface without the `query` method. Thus, in our running example, we define the `query` method in the supertype. This allows the subclasses to implement the method, thereby ensuring compliance with behavioral subtyping.

From a developers' perspective, another important consideration is how to extend the functionality of a program. For example, with the increasing need for security, the system may require an additional step: entering the pin code to verify whether the person withdrawing money is legitimate. Instead of modifying the existing `Payment` interface, a new sub-interface, named `Security`, can be defined. This sub-interface would include a new attribute, *pin*, designed to capture the entered pin code. Thus, in the `query` method, the subtype need to add a new precondition to verify the correctness of the pin code. Benefiting from our approach, we do not need to reconstruct abstract functions for each state, but only formulate refinement relations between the new type and its supertypes and subtypes to ensure behavioral subtyping.

7.3.4 Example of method call and implementation

We exemplify the method call rule through a client-side example: the verification of the `mybalance` method, as shown in Listing 7.13.

```
class ClientExample{
  {true}
  int mybalance(Saving s){
  int i = s.getbalance(); return i;}
  {result = s.amount ∧ s.amount = old(s.amount)}

  {j≥0}
  int myAccount(Debit d, int j){
     d.deposit(j); int r = mybalance(a); return r;}
  {result = d.amount ∧ d.amount = old(d.amount)+j}
}
```

Listing 7.13: Client code that illustrate the method call rule.

At the beginning of the method, its precondition is assumed. To verify the call to `getbalance` method, we rely on the specification of the callee, in this case, the `Saving` interface (with specifications provided in Listing 7.7). By substituting the callee for the implicit receiver **this** in the specification, we can assume the postcondition.

$$\{\textbf{true}\}\,\textbf{result} = \mathit{this}.\texttt{mybalance}(\texttt{Saving s})\,\{\textbf{result} = \texttt{s}.\mathit{amount}\}$$

The technique for method call verification relies on the method specification, which uses the attributes and fields to describe the expected behavior of the method. The verification of clients based on those method specifications leaves histories uninterpreted, thereby eliminating the need to prove the correctness of the method's implementation each time the method is called. For `myAccount` method, the verification of the method call is independent of the implementation of the argument. This means that only the specification given in the `Account` is accessible. We can prove the postcondition of the `myAccount` method by referring to the history of the `Account` class and its corresponding redefinition of the attribute *amount*.

A specific example of method implementation we can consider the `deposit` method in the `Account` class, as shown in Listing 7.14.

```
class Account implements Saving {
  balance ≥ 0 ∧ balance = amount, //invariant
  {i≥0}
  void deposit(i){balance=balance+i;}
  {balance=old(balance)+i}
}
```

Listing 7.14: The `deposit` implementation in the `Account` class.

One verification condition involves reasoning about the class invariant `balance` = *amount* should hold before and after the method `deposit` call. We verify the `deposit` method is correct with respect to the contract. How can we show the attribute *amount* also changed without knowing the internal structure of the history and the attribute definition? Within the implementing class, the history is defined by the field **this**, which is updated during the method call with a newly created history that involves the new event: the `deposit` event. Attributes are used to map the history to a particular value, with the update of the history, the value of the attribute also changed.

For the example in Listing 7.14, we can establish the update relation in terms of the attributes as below. This can be verified by unfolding the *amount* definition.

$$amount(Cons(deposit(i, null), h')) = amount(h') + \texttt{i}$$

We provide a manual translation as $amount = \mathbf{old}(amount) + \texttt{i}$, so it can be used in the verification condition. One can prove the class invariant by showing that both the class field and attribute increase accordingly.

7.4 Summary

Programming to interfaces, a key principle in object-oriented programming, is fundamental to numerous popular frameworks that offer hierarchies of interfaces and classes. These interfaces abstract from state and implementation, enhancing modularity and maintainability in software systems. Behavioral subtyping complements the practice of programming to interfaces by ensuring that subtypes not only match

the method signatures defined by their supertypes but also adhere to the intended behaviors.

The main contribution of this chapter is to develop a general history-based refinement approach for verifying behavioral subtyping, allowing consistent program specification at different abstraction levels. Our methodology enables us to reason about interfaces, generating interfaces-based behavioral subtyping rules that are notably absent in the majority of studies. We showed how our refinement approach can be effectively employed to rationalize various kinds of refinements in terms of projection relation: from interface to interface, interface to class and class to class.

As logical properties, attributes serve several purposes. They can map a single history to a value, represent the relationship for different histories like the refinement relation, and reflect different states of the same history like the update relation. Moreover, we applied our approach to verifying method calls as well as classes that implement interfaces. Our running example served as a practical guide, showcasing the value of our approach in realistic scenarios.

Our history-based refinement theory is suitable for all three scenarios: method overriding, method inheritance, and methods explicitly defined within the subtype itself [27]. When a method is inherited, the subclass simply inherits the same precondition and postcondition as the method in its supertype. For method overriding, a subtype that overrides its supertype's method must adhere to the behavioral subtyping rule. In the case of method overloading, the method in the subtype may have a different signature compared to the methods of the same name in its supertype. We can interpret this distinctive scenario as the subtype defining a new method, which is independent of the supertype's method.

This work simplifies the workflow by clearly distinguishing between the role of the designer, who deals with attributes, and the role of the verifier, who handles verification conditions. Designers can not only define attributes but also provide the grounding to confirm the realizability of the theory. The verifier's assumptions are based on refinement relation provided by the designer, which can used to prove the verification conditions generated for behavioral subtyping.