From Types to Contracts: Supporting by Light-Weight Specifications the Liskov Substitution Principle*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

February 24, 2010

Abstract

In this paper we review the main theoretical elements of behavioral subtyping in object-oriented programming languages in a semi-formal style that should allow software developers to understand better in which situations the Liskov substitution principle (objects of subclasses may stand for objects of superclasses) is violated. We then shortly discuss the specification of class contracts in behavioral specification languages that allow to ensure that the substitution principle is preserved. Since many software developers may shy away form these languages because the learning curve is esteemed as too steep, we propose a language of light-weight specifications that provides by a hierarchy of gradually more expressive specification forms a more lenient path towards the use of behavioral specification languages. The specifications do not demand the use of predicate logic; by automatic checking certain violations of the substitution principle may be detected.

*Supported by the Austrian Academic Exchange Service (ÖAD) under the contract HU 14/2009.

Contents

1 Introduction

Subclasses and Subtypes One of the most important features of the type systems of statically typed object oriented languages is that subclasses are intended to denote subtypes [3] (which is not self-evident [5]): if a class S inherits from a class T,

```
class S extends T { ... }
```

then an assignment

```
T x = new S(...);
```

is legal, i.e., while the variable x has type T, it actually refers to an object of type S. Nevertheless, the type system of the language ensures that without further checks any client may use x as if it were an object of type T without the danger of triggering a runtime error by accessing some attribute of x which might be declared in T but not available in the actual object referenced by x. In this sense, the type systems of object-oriented languages are *safe* [3].

Many software developers encounter the subtleties of such type systems for the first time when they notice that in some object-oriented languages, even if class S is a subtype of class T, then the type $S[]$ (array of S) is *not*[1] a subtype of class $T[]$ (array of T). This constraint rules out programs like

```
S[] x = new S[...];
T[] y = x;
y[0] = new U(...); // U is another subclass of T
```

where both variables x and y refer to the same array but with different views on their types. Since y views the base type of the array as T, it may store in the array a value of another subclass U of T such that the view of x (all elements of x are of type S) is violated.

Similar problems arise with the use of parameterized classes ("generics"), where for a generic G, even if S is a subclass of T, the instantiation $G\langle S\rangle$ is not considered as a subtype of $G\langle T\rangle$. The reason is that we may write a container class

```
class G<X> { X v; void set(X x) { v = x; } }
```

that encapsulates a variable v of type X. Then the method *set* of an object referenced by a variable y of type $G\langle T\rangle$ may update v with a value of type T while the object is shared with another variable x of type $G\langle S\rangle$ assumes that it has value S.

Substitutability While a type system can ensure that substituting an object of a subtype S into a variable of supertype T is safe, it does not guarantee that a the more general principle of substitutability formulated by Liskov [11] (the "Liskov Substitution Principle" [13]) holds:

[1] Some languages, e.g. Java, consider $S[]$ as a subtype of $T[]$, at the price of checking at runtime in every update operation whether the base type of the array in memory is indeed a supertype of the value to be written. Some languages, e.g. C++, even omit the check; the type systems of such languages are therefore not safe.

> *What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_2 is substituted for o_1, then S is a subtype of T.*

A subtle violation of the principle can be demonstrated by the classical "square/rectangle" (also called "circle/eclipse") problem [15, 7]. Take a class

```
class Rectangle {
  private int width; private int height;
  Rectangle(int w, int h) { width = w; height = h; }
  int getWidth() { return width; }
  int getHeight() { return height; }
  void setWidth(int w) { width = w; }
  void setHeight(int h) { height = w; }
}
```

describing a rectangle by its width and its height. According to the mathematical definition

> *A square is a (special) rectangle, namely a rectangle with equal width and height.*

a programmer might derive a subclass

```
class Square extends Rectangle {
  // width and height are equal
  Square(int a) { super(a, a); }
}
```

describing a square as a special rectangle with equal width and height which is ensured by the constructor of `Square`. The definition is well-typed and no runtime error can be triggered by using `Square`. Still a program using `Square` may show unexpected behavior as in

```
Square s = new Square(a);
System.out.println(s.getWidth() + "x" + s.getHeight());
use(s);
System.out.println(s.getWidth() + "x" + s.getHeight());
```

which prints

```
2x2
4x2
```

Apparently the square has suddenly turned into a rectangle!

Investigating the problem, it turns out that the function

```
static void use(Rectangle r)
{
  int w = r.getWidth();
  r.setWidth(2*r);
}
```

modifies one dimension of a `Rectangle` by calling its `setWidth` function which invalidates the constraint that a square must have equal width and height.

The programmer realizes that it is necessary to override both methods `setWidth` and `setHeight` such that the constraint is preserved, which yields the following definition:

```
class Square extends Rectangle {
  Square(int a) { super(a, a); }
  void setWidth(int a)
  { super.setWidth(a); super.setHeight(a); }
  void setHeight(int a)
  { super.setWidth(a); super.setHeight(a); }
}
```

The resulting program is well-typed and no runtime error can be triggered by using `Rectangle` and `Square`. Still the use of these classes shows e.g. the following unexpected behavior.

```
static function use(Rectangle r, int h)
{
  System.out.println(getWidth() + "x" + getHeight());
  r.setHeight(h);
  System.out.println(getWidth() + "x" + getHeight());
}
```

which produces in one invocation output

```
3x3
3x5
```

and in another invocation output

```
3x3
5x5
```

So in both cases a rectangle of size 3×3 was passed to the function, however in one case only the height of the rectangle was changed, while in the other case both width and height were modified. Investigating the problem, it turns out that the outputs were produced by two invocations

```
...
Rectangle r1 = new Rectangle(w, h);
use(r1, h);
...
Rectangle r2 = new Square(a);
use(r2);
```

In one case, the function was called with a `Rectangle` of size 3×3, in another case with a `Square` of size 3×3. Apparently the program behaved different in both cases, because the method `setHeight` in `Rectangle` changed only the height, while the corresponding method in `Square` changed both width and height. So the program apparently contradicts the assertion "a square is a (special) rectangle" since apparently a rectangle and a square behave different.

Indeed, a *mutable* square is *not* a *mutable* rectangle, because a mutable rectangle has a particular capability (changing its height independently of its width) which a mutable square has not. Explicitly specifying the `setHeight` function in both classes shows that they have different *contracts*:

```
class Rectangle {
  ...

  // sets width to a but leaves its height unchanged
  void setHeight(int a) { ... }
}

class Square {
  ...

  // sets both width and height to a
  void setHeight(int a) { ... }
}
```

i.e. the overriding definition of `setHeight` has violated the contract of the original definition, causing the surprise.

Many solutions have been suggested to overcome this problem [7, 13, 15]; in essence they boil down to three possibilities [4]:

1. Give up the idea that a mutable square is a mutable rectangle. One may for instance write a class `ConstRectangle` without the mutator functions `setWidth` and `setHeight` and then construct the following inheritance hierarchy:

   ```
   ConstRectangle  ←  Rectangle
                   ←  ConstSquare  ←  Square
   ```

   Thus an immutable square is an immutable rectangle but a mutable square is not a mutable rectangle.

2. Weaken the contract of the superclass respectively its functions such that it is easier for a subclass to preserve it. We could for example redefine

   ```
   class Rectangle {
     ...

     // attempts to set height to a and
     // returns true iff the operation has succeeded
     boolean setHeight(int a) { height = a; return true; }
   }
   ```

   i.e. allow the possibility of failure. The subclass may then override the function as

   ```
   class Square {
     ...
   ```

```
  // leaves the height unchanged and returns false
  boolean setHeight(int a) { return false; }
}
```

The price of this solution is that all clients of `Rectangle` have to cope with the weaker contracts.

3. Strengthen the contract of the subclass respectively its functions to preserve the contract of the superclass. While this is the ultimate goal it may fail due to the conflicting requirements of the subclass and the superclass. In particular, there is no way how the conflicting requirements

```
class Rectangle {
  ...
  // sets width to a but leaves its height unchanged
  void setHeight(int a) { ... }
}
```

and

```
class Square extends Rectangle {
  // width and height are equal
  ...
}
```

can be reconciled by the implementor of the subclass.

Contracts and Behavioral Subtyping This formulation of the Liskov substitution principle has due to the phrase "the behavior of P is unchanged" given rise to various misunderstandings in that it seems to require that subtype S must behave *absolutely exactly* like supertype T. This misunderstanding can be clarified by using the concept of a *(behavioral) contract* (introduced by the closely related "design by contract" principle [14]), where a contract describes the set of possible behaviors of a program or object. Let us re-formulate the substitution principle as follows:

> Let type T fulfill a contract C, i.e. let every object o_1 of type T fulfill a contract C_{o_1}. If for every such object o_1 of type T there is an object o_2 of type S that also satisfies C_{o_1}, then the fulfillment of every contract by every program P that relies on the fulfillment of contract C is not affected when o_2 is substituted for o_1. S is thus a subtype of T *with respect to contract* C.

The question therefore is actually not whether a program behaves "identical" after an object substitution but whether it fulfills a certain contract. If this fulfillment depends on the fulfillment of a contract C by a type T, the details of the fulfillment are irrelevant to the program *as long as the objects of* T *fulfill their contracts.* Any object of another type S may also be plugged in, provided that it fulfills C, independently of the way how it does this. A contract serves thus as a "filter" that abstracts away from the details of a type that are irrelevant to the user of this type. A contract C' may also *refine* (strengthen) a contract C

in that it specifies more details of the fulfillment but still requires the fulfillment of C_T. If S is a subtype of T with respect to the strengthened contract C', it is thus also a subtype of T with respect to the weaker contract C.

Thus the notions of types and behavioral contracts, subtypes and contract refinement, cannot be clearly separated any more; the notion *behavioral subtyping* [12, 9] has been coined to denote types that describe object behaviors with subtypes describing refined/restricted behaviors.

Related Work The theory of types and object, subtypes and inheritance, the Liskov substitution principle, contracts, refinement, and behavioral subtyping has been very well elaborated since the 1990s [1, 3, 9, 12]. Nevertheless, various resources in the web demonstrate that the principles are often still not very well understood [4] and even remain controversial [6]. We do not want to claim that the contribution of the present paper is to shed new light on the theoretical principles of the field; we rather aim to present a concise overview that condenses the various bits and pieces found in the literature to a short summary (which is definitely colored by the personal views of the author); this summary may help to clarify the views of some readers interested in the topic (indeed every object-oriented software developer should be interested in this topic). For more details, see e.g. the cited literature.

Organization of this Paper Sections 2–5 present a simplified but essentially (under the chosen level of abstraction) correct sketch of the main theoretical elements underlying the substitution principle. The presentation is given in a semi-formal style that allows to discuss crucial questions with a certain level of exactness without digging too deep into the theoretical underpinnings. We hope that this allows to internalize the general principles with which every object-oriented programmer should be familiar in order to avoid violations of the substitution principle.

Section 6 discusses the main elements of the specification of contracts in behavioral specification languages like the Java Modeling Language [8] (JML). Section 7 contains the main original part of this paper: a proposal for "light-weight" specifications that may help to better understand the problems related to the substitution principle and detect certain cases of its violations; we start with a tiny core contract language that allows with small effort to characterize inheritance structures in such a way that certain violations of the substitution principle (as those presented by the "rectangle/square" problem can be avoided. The language is gradually extended such that with more effort inheritance hierarchies can be described in a more refined way and other violations may be detected. In this way a smooth transition to full-fledged behavioral specification languages like JML is provided. Section 8 concludes the paper.

2 Types and Subtypes

Types Let there be a domain of syntactic phrases whose elements we call "types". Each type denotes a set of values i.e. there is a function $[\![\, . \,]\!]$ ("the meaning of") such that $[\![\, T \,]\!]$ is the set denoted by type T. We call a value x to be "of type T" (or to be "a T-value"), if $x \in [\![\, T \,]\!]$.

Subtypes Our core notion of interest is the relationship

$$S <: T$$

to be read as "S is a subtype of T". This relationship holds, if there is some "interpretation function" $S \triangleright T : [\![S]\!] \to [\![T]\!]$, i.e. for every value x of type S, $(S \triangleright T)(x)$ is of type T. By the application of this function, every S-value can be interpreted as a T-value.

Apparently the subtype relationship is reflexive ($S <: S$), because we can define the interpretation $(S \triangleright S)(x) := x$ as the identity function. Furthermore, the subtype relationship is transitive (if $S <: T$ and $T <: U$, then $S <: U$), because we can define the interpretation $S \triangleright U := (S \triangleright T) \circ (T \triangleright U)$ as the composition of the respective interpretation functions.

Atomic Types As an example take a type Nat of "unsigned integers", i.e.

$$[\![\mathsf{Nat}]\!] = \{0, 1, 2, \ldots\}$$

and a type Int of "signed integers", i.e.

$$[\![\mathsf{Int}]\!] = \{0, +1, -1, +2, -2, \ldots\}.$$

Please note that the unsigned integer 1 is different from the signed integer $+1$, i.e. $[\![Nat]\!] \subseteq [\![Int]\!]$ does *not* hold. Nevertheless, $\mathsf{Nat} <: \mathsf{Int}$ does hold, because we have the conversion function

$$\mathsf{Nat} \triangleright \mathsf{Int}(x) := \begin{cases} 0 & \text{if x=0} \\ +x & \text{else} \end{cases}$$

i.e. every unsigned integer different from 0 can be interpreted as a signed integer with a "+" sign.

Compound Types In the following we investigate compound types, i.e. non-atomic types T that depend on some base types T_i. The main interesting question is, if we have two compound types $S <: T$ of the same kind, whether this implies

- **Covariance:** $S_i <: T_i$,
- **Contravariance:** $T_i <: S_i$,
- **Invariance:** none of above.

In this section we are going to investigate two compound types for which above questions will be answered as follows:

Tuple Types **A tuple type $\mathbf{T_1 \times T_2 \times \ldots \times T_n}$ is *covariant* in its component types $\mathbf{T_i}$.**

Function Types **A function type $\mathbf{T_1 \times T_2 \times \ldots \times T_n \to T}$ is *covariant* in its result type T but *contravariant* in its argument types $\mathbf{T_i}$.**

In the following section, we will investigate another type $*\mathbf{T}$ (of shared mutable variables) which is *invariant* in its base type T.

Tuple Types A *tuple type* has form $T_1 \times T_2 \times \ldots \times T_n$ with n types $T_1, \ldots, T_n$. Its meaning is defined as

$$\begin{array}{l}[\![\, T_1 \times T_2 \times \ldots \times T_n \,]\!] := \\ \quad \{x \mid x.1 \in [\![\, T_1 \,]\!] \wedge x.2 \in [\![\, T_2 \,]\!] \wedge \ldots \wedge x.n \in [\![\, T_n \,]\!]\}\end{array}$$

i.e. for every record x, the selector expression $x.i$ yields a value of type T_i (for $1 \leq i \leq n$).

We have the following subtype relationship on tuple types

$$S_1 \times S_2 \times \ldots \times S_n, S_{n+1}, \ldots, S_{n+m} <: T_1 \times T_2 \times \ldots \times T_n$$

provided that

$$S_1 <: T_1 \wedge S_2 <: T_2 \wedge \ldots \wedge S_n <: T_n$$

i.e. for tuple types the subtype relationship is *covariant* in the component types ($S_i <: T_i$, for all $1 \leq i \leq n$) and the subtype may have more components than the supertype. For instance, we have

$$\mathsf{Nat} \times \mathsf{Int} \times \mathsf{Int} <: \mathsf{Int} \times \mathsf{Int}$$

because $\mathsf{Nat} <: \mathsf{Int}$ and the first tuple type is not shorter than the second one.

The interpretation function corresponding to this relationship is

$$\begin{array}{l}(S_1 \times S_2 \times \ldots \times S_n, S_{n+1}, \ldots, S_{n+m} \triangleright T_1 \times T_2 \times \ldots \times T_n)(x) := y \\ \quad \text{where} \\ \quad\quad y.1 = S_1 \triangleright T_1(x.1), \\ \quad\quad y.2 = S_2 \triangleright T_2(x.2), \\ \quad\quad \ldots \\ \quad\quad y.n = S_n \triangleright T_n(x.n)\end{array}$$

i.e. from a value x of type $S_1 \times S_2 \times \ldots \times S_n, S_{n+1}, \ldots, S_{n+m}$ a value y of type $T_1 \times T_2 \times \ldots \times T_n$ can be constructed by taking as each component $y.i$ the component $x.1$ interpreted as a value of type T_i. For instance, we can define

$$\begin{array}{l}(\mathsf{Nat} \times \mathsf{Int} \times \mathsf{Int} \triangleright \mathsf{Int} \times \mathsf{Int})(x) := y \\ \quad \text{where} \\ \quad\quad y.1 = \mathsf{Nat} \triangleright \mathsf{Int}(x.1), \\ \quad\quad y.2 = \mathsf{Int} \triangleright \mathsf{Int}(x.2)\end{array}$$

Functions A *function type* has form $T_1 \times T_2 \times \ldots \times T_n \to T$ (with $n \geq 0$) whose meaning is given as

$$\begin{array}{l}[\![\, T_1 \times T_2 \times \ldots \times T_n \to T \,]\!] := \\ \quad \{f \mid \forall x_1 \in [\![\, T_1 \,]\!], x_2 \in [\![\, T_2 \,]\!], \ldots, x_n \in [\![\, T_n \,]\!] : f(x_1, x_2, \ldots, x_n) \in [\![\, T \,]\!]\}\end{array}$$

i.e. a function f of type $T_1 \times T_2 \times \ldots \times T_n \to T$ can be applied to n values $x1, x_2, \ldots, x_n$ of types $T_1, T_2, \ldots, T_n$ which yields a result of type T (for the special case $n = 0$, f itself is identified with a value of type T).

We have the following subtype relation among function types

$$(S_1 \times S_2 \times \ldots \times S_n \to S) <: (T_1 \times T_2 \times \ldots \times T_n \to T)$$

provided that

$$S <: T \wedge T_1 <: S_1 \wedge T_2 <: S_2 \wedge \ldots \wedge T_n <: S_n$$

i.e. for function types the subtype relation is *covariant* in the result type ($S <: T$) but *contravariant* in all argument types ($T_i <: S_i$, for all $1 \leq i < n$). For instance, we have

$$\mathsf{Int} \to \mathsf{Nat} <: \mathsf{Nat} \to \mathsf{Int}$$

i.e. every function of type $\mathsf{Int} \to \mathsf{Nat}$ can be interpreted as a function of type $\mathsf{Nat} \to \mathsf{Int}$ because $\mathsf{Nat} <: \mathsf{Int}$.

For function types, we can define the interpretation function as

$$\begin{array}{l}((S_1 \times S_2 \times \ldots \times S_n \to S) \triangleright (T_1 \times T_2 \times \ldots \times T_n \to T))(f) := g \\ \quad \text{where} \\ \quad\quad g : [\![T_1]\!] \times [\![T_2]\!] \times \ldots \times [\![T_n]\!] \to [\![T]\!] \\ \quad\quad g(x_1, x_2, \ldots, x_n) := \\ \quad\quad\quad S \triangleright T \\ \quad\quad\quad\quad (f(T_1 \triangleright S_1(x_1), T_2 \triangleright S_2(x_2), \ldots, T_n \triangleright S_n(x_n))).\end{array}$$

i.e. we can from a function f of type $S_1 \times S_2 \times \ldots \times S_n \to S$ construct another function g of type $T_1 \times T_2 \times \ldots \times T_n \to T$ as follows:

- g takes n arguments $x_1, x_2, \ldots, x_n$ of types $T_1, \ldots, T_n$,
- it interprets these as values of types $S_1, \ldots, S_n$ and applies f to these arguments,
- it receives from f a result of type S and interprets it as a value of type T which it returns as a result.

For instance, we can define

$$\begin{array}{l}((\mathsf{Int} \to \mathsf{Nat}) \triangleright (\mathsf{Nat} \to \mathsf{Int}))(f) := g \\ \quad \text{where} \\ \quad\quad g : [\![\mathsf{Nat}]\!] \to [\![\mathsf{Int}]\!] \\ \quad\quad g(x) := \\ \quad\quad\quad \mathsf{Nat} \triangleright \mathsf{Int} \\ \quad\quad\quad\quad (f(\mathsf{Nat} \triangleright \mathsf{Int}(x)))\end{array}$$

3 Shared Mutable Variables

A (shared mutable) variable x of type $*T$ refers to a location in the computer store that holds a value of type T; this value may be read as $*x$ and replaced by another value a of type T by an assignment operation $*x := a$. Furthermore, we assume that, if x is copied to another variable y, both copies share the same store location such that after an update $*y := b$ also $*x$ denotes the value b.

The question arises whether y may be of type $*S$ for some value type S that is (significantly) different from the value type T. As we will see, the answer is negative, i.e. there is no general subtype relationship $*S <: *T$ or $*T <: *S$ that allows both x and y to share a mutable store location in a safe way. In particular, we have the following result:

Even if S $<:$ T holds (i.e. every S-value can be interpreted as a corresponding T-value), $*$S $<:$ $*$T does in general *not* hold (i.e. a shared mutable S-variable can not be interpreted as a shared mutable T-variable).

Consequently, $*$S $<:$ $*$T can be in general only ensured if S = T, i.e. a variable type is *invariant* in its base type.

The core reason is that a T-variable has a particular capability (storing a general T-value) which an S-variable (which may only store an S-value) has not. Since T and S share the same location, if the T-variable exercises its capability, it may invalidate the further use of the S-variable.

Example Let variable x be of type $*$Nat and variable y be of type $*$Int and remember that Nat $<:$ Int i.e. every Nat-value can be interpreted as an Int-value (but not vice versa).

First let us investigate whether the (rather unlikely) relationship $*$Int $<:$ $*$Nat might hold i.e. whether an Int-variable could be interpreted as a Nat-variable. If yes, then a variable assignment $x := y$ were legal and both x and y would refer to the same location. However, then we might use y in an assignment $*y := -1$ to write the value -1 into the memory cell which cannot be interpreted as a Nat-value. Consequently, reading this memory cell as $*x$ cannot return any meaningful result of type Nat.

Next, let us investigate whether the (at first glance more likely) relationship $*$Nat $<:$ $*$Int might hold i.e. whether a Nat-variable could be interpreted as an Int-variable. If yes, then a variable assignment $y := x$ were legal and again both x and y would refer to the same location. However, then we again might use y in an assignment $*y := -1$ such that $*x$ could not return any meaningful result of type Nat.

So while a Nat-value can be interpreted as an Int-value, a (shared mutable) Nat-variable can*not* be interpreted as a (shared mutable) Int-variable.

Formalism[2] To model references, we need to introduce the domains *Variable*, *Value*, and *Store* where a store is intuitively a mapping of variables of values. We assume that there exist operations

$$*^{\text{-}}\_ : \mathit{Store} \times \mathit{Variable} \to \mathit{Value}$$
$$*^{\text{-}}\_ := \_ : \mathit{Store} \times \mathit{Variable} \times \mathit{Value} \to \mathit{Store}$$

with the central property

$$\begin{array}{l} \forall s, s' in Store, x, y \in \mathit{Variable}, v \in \mathit{Value} : \\ \quad s' = (*^s x := v) \Rightarrow \\ \qquad *^{s'} y = \begin{cases} v & \text{if } y = x \\ *^s y & \text{else} \end{cases} \end{array}$$

i.e. if we write into a store s for variable x a value v, then reading from s' the value of a variable y will either give x (if y and x are identical) or the value of y in the original store s (if y is different from x).

[2]The remainder of this section presents a formal justification of above statements. The reader who is more interested in the general picture than the technical details may skip it.

We assume that the value of every type T can be converted to a storable value, i.e. that there exists a function

$$T \triangleright : [\![T]\!] \to \mathit{Value}$$

Some storable values may be interpreted as values of type T, i.e. there exists a *partial* function

$$\triangleright T : \mathit{Value} \overset{\text{partial}}{\longrightarrow} [\![T]\!]$$

where $\mathit{domain}(\triangleright T)$ denotes those values that may be interpreted as values of type T. We demand

$$\begin{array}{l} \forall t \in [\![T]\!] : T \triangleright (t) \in \mathit{domain}(\triangleright T) \\ \forall t \in [\![T]\!] : \triangleright T(T \triangleright (t)) = t \\ \forall v \in \mathit{domain}(\triangleright T) : T \triangleright (\triangleright T(v)) = v \end{array}$$

Furthermore, we assume that for all types S and T the interpretation $S \triangleright T$ coincides (or is defined by) the value conversions i.e.

$$(S \triangleright T)(a) = \triangleright T(S \triangleright (a))$$

With these preliminaries we define the meaning of a variable type $*T$ as

$$[\![*T]\!] = \{x \in \mathit{Variable} \mid \forall s \in \mathit{Store} : *^s x \in \mathit{domain}(\triangleright T)\}$$

i.e. as the set of all variables whose values can in any store be interpreted as T-values. We define

$$(*S \triangleright *T)(x) := x$$

i.e. we assume that variables preserve their meanings (storage locations) under different type interpretations (this is the common assumption in all programming languages).

We are now going to investigate the subtype relationship on variable types, more specifically we show that two variable types $*S$ and $*T$ with $*S <: *T$ are invariant in their base types S and T.

Contravariance We first show that $*S$ and $*T$ with $*S <: *T$ are in general not contravariant in their base types S and T. Take two types S and T with $T <: S$ where S is a "true" supertype of T, i.e. S has some value b that is not the interpretation of any T-value:

$$b \in [\![S]\!] \wedge \forall a \in [\![T]\!] : (T \triangleright S)(a) \neq b$$

Take a variable x of type $*S$. We can then write into some store s at x the value b and thus construct a new store s':

$$\begin{array}{l} v := (S \triangleright)(b) \\ s' := (*^s x := v) \end{array}$$

We then know $*^{s'} x = v$. Since $*S <: *T$, $x = (*S \triangleright *T)(x)$ is also of type $*T$ such that we can interpret v as a T-value, i.e. $v \in \mathit{domain}(\triangleright T)$. We can thus compute $a \in T$ as

$$a := (\triangleright T)(v)$$

Since $T <: S$, we can compute $b' \in S$ as

$$b' := (T \triangleright S)(a)$$

Expanding the definitions and applying above laws, we thus have

$$\begin{array}{rcl} b' & = & \triangleright S(T \triangleright (\triangleright T(S \triangleright (b)))) \\ & = & \triangleright S(S \triangleright (b)) \\ & = & b \end{array}$$

i.e. $b = (T \triangleright S)(a)$ which contradicts our assumption that no T-value can be interpreted as b.

Covariance Now we show that $*S$ and $*T$ with $*S <: *T$ are in general not covariant in their base types S and T. Take two types S and T with $S <: T$ where T is a "true" supertype of S, i.e. T has some value b that is not the interpretation of any S-value:

$$b \in [\![T]\!] \wedge \forall a \in [\![S]\!] : (S \triangleright T)(a) \neq b$$

Take a variable x of type $*S$. Since $*S <: *T$, $x = (*S \triangleright *T)(x)$ is also of type $*T$ such that we can write into some store s at x a T-value; in particular, we can write the value b and thus construct a new store s':

$$\begin{array}{l} v := (T \triangleright)(b) \\ s' := (*^s x := v) \end{array}$$

We then know $*^{s'} x = v$ and, since x is of type $*S$, we can interpret v as an S-value i.e. $v \in \mathit{domain}(\triangleright S)$. We thus can compute $a \in S$ as

$$a := (\triangleright S)(v)$$

We then can compute $b' \in S$ as

$$b' := \triangleright T(S \triangleright (a))$$

Expanding all definitions and applying the laws given above

$$\begin{array}{rcl} b' & = & \triangleright T(S \triangleright (\triangleright S(T \triangleright (b)))) \\ & = & \triangleright T(T \triangleright (b)) \\ & = & b \end{array}$$

i.e. $b = S \triangleright T(a)$ which contradicts our assumption that no S-value can be interpreted as b. Thus $*S$ and $*T$ can not be covariant in their base types.

Invariance We have shown that the types of (shared mutable) variables are neither covariant nor contravariant in their base types, i.e. they are *invariant*.

4 Classes and Objects

A *class* is a type $\mathsf{class}\ C$ where C is a collection of declarations of variables and definitions of functions of the following form[3]:

$$\begin{array}{l}\mathsf{var}\ x_1 : T_1\\ \mathsf{var}\ x_2 : T_2\\ \ldots\\ \mathsf{var}\ x_n : T_n\\ \\ \mathsf{function}\ f_1 : A_{11} \times A_{12} \times \ldots \times A_{1a_1} \to B_1 := \ldots\\ \mathsf{function}\ f_2 : A_{21} \times A_{22} \times \ldots \times A_{2a_2} \to B_2 := \ldots\\ \ldots\\ \mathsf{function}\ f_m : A_{m1} \times A_{m2} \times \ldots \times A_{ma_m} \to B_m := \ldots\end{array}$$

An *object* is a value of a class which consists of a collection of the (shared mutable) variables and the functions declared in the class. Given an object o of class C, every variable $x_i : T_i$ declared in C may be read as $o.x_i$ which yields a value of type T_i; for this variable a new value v_i of type T_i may be written as $o.x_i := v_1$. Likewise, every function $f_i : A_{i1} \times A_{i2} \times \ldots A_{ia_i} \to B_i$ declared in C may be called with n arguments $v_1, v_2, \ldots, v_{a_i}$ of types $A_{i1}, A_{i2}, \ldots, A_{ia_i}$ as $o.f_i(v_1, v_2, \ldots, v_{a_i})$; this may change the values of the variables in o and return a value of type B_i.

The meaning of $\mathsf{class}\ C$ can be defined in terms of the types introduced in the previous sections and with the help of a new type Store with $[\![\,\mathsf{Store}\,]\!] = \mathit{Store}$:

$$\begin{array}{l}[\![\,\mathsf{class}\ C\,]\!] :=\\ \quad [\![\,*T_1\,]\!] \times [\![\,*T_2\,]\!] \times \ldots \times [\![\,*T_n\,]\!] \times\\ \quad [\![\,\mathsf{Store} \times A_{11} \times A_{12} \times \ldots \times A_{1a_1} \to \mathsf{Store} \times B_1\,]\!] \times\\ \quad [\![\,\mathsf{Store} \times A_{21} \times A_{22} \times \ldots \times A_{2a_2} \to \mathsf{Store} \times B_2\,]\!] \times\\ \quad \ldots \times\\ \quad [\![\,\mathsf{Store} \times A_{m1} \times A_{m2} \times \ldots \times A_{ma_m} \to \mathsf{Store} \times B_m\,]\!]\end{array}$$

We see that the meaning of a class is a set of tuples (representing the objects of the class) where each tuple contains representations of all (actually all of the non-private[4]) variables and functions declared in the class:

- Each variable is represented by a location in the store,
- Each function receives as an extra argument the pre-store of the function application and returns as an extra result the post-store (which differs from the pre-store if the function is not "pure" i.e. causes side-effects on the store)[5]. We assume that the application of a function does not change

[3] We assume that the component types of $\mathsf{class}\ C$ do not refer to $\mathsf{class}\ C$; in reality the definition of $\mathsf{class}\ C$ may be *recursive*. While this causes considerable technical complications, the main messages of this section still hold in the general scenario.

[4] The semantic model of an object needs only represent its *non-private* variables/functions, all references to private entities can be considered as "compiled into" the non-private entities. Thus all private entities do *not* contribute to the object type.

[5] A reader familiar with the implementation of object-oriented languages might wonder why there is no extra argument for the "this" pointer referring to the object on which the function is applied. The answer is that this pointer is only necessary in implementations where all objects of the same class share the same function f_i; in our model, every object o has its own version of f_i which is already specialized with respect to the variable locations contained in o.

the meaning (location) of a variable (as in object-oriented languages where the layout of an object in memory remains fixed), thus the function needs not return new variable locations.

Given a current store s, the operation $o.x_i$ looks up o for the value (store location) of x_i and returns the value of s at that location; likewise the operation $o.x_i := v_1$ updates s at that location with v_i. A function application $o.f_i(v_1, v_2, \ldots, v_{a_i})$ looks up o for the value (function) of f_i and passes to it $s, v_1, v_2, \ldots, v_{a_i}$. It receives as a result a store s' which replaces s and a value b which is returned as the result of the application.

Now, from the definition of $[\![\,\mathsf{class}\ \mathbf{C}\,]\!]$ given above, all questions about the properties of the subtype relation of classes (object types) can be immediately reduced to questions about the already known subtype relations on atomic types, tuple types, shared mutable variables, and function types.

It is therefore *not* necessary any more to introduce a separate definition of the class subtype relationship

$$\mathsf{class}\ S <: \mathsf{class}\ T$$

but it suffices to investigate the consequences of the model introduced above:

1. Since class types are tuple types, if $\mathsf{class}\ T$ has n non-private fields (variables and functions), $\mathsf{class}\ S$ may have $n + m$ non-private fields. In other words, $\mathsf{class}\ S$ may introduce extra variables and functions.

2. For every $1 \leq i \leq n$ with corresponding non-private fields of type S_i in $\mathsf{class}\ S$ and type T_i in $\mathsf{class}\ T$, we must have $S_i <: T_i$. In other words, corresponding field types must covariantly preserve the subtype relationship.

3. If $S_i = *V$ for some type T, then also $T_i = *V$, because variable types are invariant. In other words, $\mathsf{class}\ S$ and $\mathsf{class}\ T$ must have the *same* types for corresponding variables.

4. If $S_i = \mathsf{Store} \times A_1 \times A_2 \times \ldots \times A_a \to \mathsf{Store} \times B$, then $T_i = \mathsf{Store} \times A'_1 \times A'_2 \times \ldots \times A'_a \to \mathsf{Store} \times B'$ with $B <: B'$ and $A'_j <: A_j$. In other words, $\mathsf{class}\ S$ and $\mathsf{class}\ T$ must have the same number a of arguments for corresponding functions and, while the result types are *covariant*, the argument types are *contravariant*.

Above rules constrain the way how *inheritance* may be used to derive a subclass S from a superclass T by a construction like $\mathsf{class}\ S\ \mathsf{extends}\ \mathsf{T}\ \ldots$ such that the subtype relationship is preserved:

- **The subclass S may introduce extra variables and functions that are not among the (non-private) variables and functions of the superclass S.**
- **The subclass S must not override the declarations of non-private variables of superclass T by declarations with new types.**
- **The subclass S must only override the declarations/definitions of non-private functions of superclass S in such a way that the number of arguments remains the same, result types are overridden covariantly, argument types are overridden covariantly.**

The substitution principle only holds, if the inheritance mechanism of the language obeys these constraints.

5 Contracts

By using the more expressive language of logic, a domain of admissible values may be further constrained than by using the simple language of types introduced so far. The idea is to generalize a plain type T to a *contract* T^L (where L is a specification based on formal logic) such that

$$[\![T^L]\!] \subseteq [\![T]\!]$$

In other words, T^L denotes a *subtype* of T that describes a domain of admissible values more precisely than T alone can.

In particular, contracts may be described by the following elements:

- The admissible applications of every function may be constrained by a *precondition* and (possibly exceptional) *postcondition*; furthermore, the function's effect on the store may be constrained by a *frame condition*.
- The admissible values of the variables of a class may be constrained by *invariants* and *history constraints*.

In the following, we discuss these elements in more detail. Please remember that a function declaration

$$\mathsf{function}\ f_i : A_{i1} \times A_{i2} \times \ldots \times A_{ia_i} \to B_i$$

denotes a proposition

$$f_i \in [\![\mathsf{Store} \times A_{11} \times A_{12} \times \ldots \times A_{1a_1} \to \mathsf{Store} \times B_1]\!]$$

i.e.

$$f_i \in \mathit{Store} \times [\![A_{11}]\!] \times [\![A_{12}]\!] \times \ldots \times [\![A_{1a_1}]\!] \to \mathit{Store} \times [\![B_1]\!]$$

Preconditions A *precondition* of a function

$$\mathsf{function}\ f_i : A_{i1} \times A_{i2} \times \ldots \times A_{ia_i} \to B_i$$

is a predicate P_i that constrains the store and the arguments with which the function is called i.e.

$$[\![P_i]\!] \subseteq \mathit{Store} \times [\![A_{i1}]\!] \times [\![A_{i2}]\!] \times \ldots \times [\![A_{ia_i}]\!]$$

The type of the function thus effectively becomes

$$[\![P_i]\!] \to \mathit{Store} \times [\![B_i]\!]$$

This type is apparently *contravariant* in P_i, i.e.

$$([\![P_i]\!] \to \mathit{Store} \times [\![B_i]\!]) <: ([\![P_i']\!] \to \mathit{Store} \times [\![B_i']\!])$$

holds only if $[\![P_i']\!] \subseteq [\![P_i]\!]$ i.e. if

$$P_i' \Rightarrow P_i$$

This implies that, if f_i is an element of some $\mathsf{class}\ T$, then every $\mathsf{class}\ S$ with $\mathsf{class}\ S <: \mathsf{class}\ T$ may only weaken the precondition of f. With respect to class inheritance we thus get the following constraint:

If a class S inherits from a class T and overrides a function f, the overriding may only *weaken* (*not* strengthen) the precondition of f.

Postconditions A *postcondition* of a function

$$\mathsf{function}\ f_i : A_{i1} \times A_{i2} \times \ldots \times A_{ia_i} \to B_i$$

is a predicate Q_i that relates the store and the arguments with which the function is called to the store and the result returned by the function, i.e.

$$[\![\, Q_i \,]\!] \subseteq \mathit{Store} \times [\![\, A_{i1} \,]\!] \times [\![\, A_{i2} \,]\!] \times \ldots \times [\![\, A_{ia_i} \,]\!] \times \mathit{Store} \times [\![\, B_i \,]\!]$$

The type of the function effectively becomes a subset of

$$[\![\, Q_i \,]\!]$$

i.e. it is *covariant* in Q_i. The type of a function with precondition Q_i is therefore only a subtype of a function with precondition Q'_i if $[\![\, Q_i \,]\!] \subseteq [\![\, Q'_i \,]\!]$, i.e. if

$$Q_i \Rightarrow Q'_i$$

This implies that, if f_i is an element of some $\mathsf{class}\ T$, then every $\mathsf{class}\ S$ with $\mathsf{class}\ S <: \mathsf{class}\ T$ may only strengthen the postcondition of f. With respect to class inheritance we thus get the following constraint:

If a class S inherits from a class T and overrides a function f, the overriding may only *strengthen* (*not* weaken) the postcondition of f.

Exceptional Postconditions In most programming languages, a function may not only return in a normal way but also in the form of throwing an exception. The postcondition of a function

$$\mathsf{function}\ f_i : A_{i1} \times A_{i2} \times \ldots \times A_{ia_i} \to B_i$$

may thus contain the specification of a set E_i of exceptions that may be thrown:

$$[\![\, Q_i \,]\!] \subseteq \mathit{Store} \times [\![\, A_{i1} \,]\!] \times \ldots \times [\![\, A_{ia_i} \,]\!] \times \mathit{Store} \times (\{\mathsf{Normal}\} \cup E_i) \times [\![\, B_i \,]\!]$$

The outcome of the function is now tagged either as Normal or with one of the exceptions in E. Given two exceptional postconditions Q_i and Q'_i with exceptions E_i and E'_i, $[\![\, Q_i \,]\!] \subseteq [\![\, Q'_i \,]\!]$ can only hold if

$$E_i \subseteq E'_i$$

i.e. postconditions are *covariant* in their sets of exceptions. This implies that, if f_i is an element of some $\mathsf{class}\ T$, then every $\mathsf{class}\ S$ with $\mathsf{class}\ S <: \mathsf{class}\ T$ may only shrink the set of exceptions of f. With respect to class inheritance we thus get the following constraint:

If a class S inherits from a class T and overrides a function f, the overriding may only *shrink* (*not* increase) the set of exceptions that may be thrown by f.

Frame Conditions To simplify reasoning about the effect of a function on the store, the specification of a function may include the definition of a set L of those locations in the store that may be altered by the execution of the function. The type of a function

$$\mathsf{function}\ f_i : A_{i1} \times A_{i2} \times \ldots \times A_{ia_i} \to B_i$$

with modifiable location set L_i thus effectively becomes the value of a *frame condition* F_i with

$$\begin{array}{l}[\![F_i]\!] := \{f \in \mathit{Store} \times [\![A_{i1}]\!] \times [\![A_{i2}]\!] \times \ldots \times [\![A_{ia_i}]\!] \to \mathit{Store} \times [\![B_i]\!] \mid \\ \quad \forall s, s' \in \mathit{Store}, a_1 \in [\![A_1]\!], a_2 \in [\![A_2]\!], \ldots, a_n \in [\![A_n]\!], b \in [\![B]\!] : \\ \quad\quad f(s, a_1, a_2, \ldots, a_n) = \langle s', b \rangle \Rightarrow \forall x \notin L_i : s(x) = s'(x)\}\end{array}$$

i.e. only those pairs s, s' of pre- and poststate are allowed where the values of all the locations not in L_i remain unchanged. A frame condition is essentially a postcondition; one can show that given two frame conditions F_i and F_i' with location sets L_i and L_i', we have $[\![F_i]\!] \subseteq [\![F_i']\!]$ only if

$$L_i \subseteq L_i'$$

i.e. frame conditions are *covariant* in their sets of modifiable locations. This implies that, if f_i is an element of some $\mathsf{class}\ T$, then every $\mathsf{class}\ S$ with $\mathsf{class}\ S <: \mathsf{class}\ T$ may only shrink the set of modifiable locations of f. With respect to class inheritance we thus get the following constraint:

> **If a class S inherits from a class T and overrides a function f, the overriding may only *shrink* (*not* increase) the set of locations that may be modified by f.**

Invariants An *invariant* of an object is a predicate I that constrains the values of the object variables in the store, i.e.

$$[\![I]\!] \subseteq \mathit{Store}$$

For the further discussion, we need to generalize our model a bit by introducing the notion of a current "context" $P \subseteq \mathit{Store}$ which is a predicate on stores (respectively a set of stores satisfying the predicate). We parameterize our semantic function on types over the current context, i.e. for a type T and a context P, $[\![T]\!]_P$ denotes the set of values denoted by T in context P. By this generalization, the original semantics described above will become a special case for $P = \mathit{Store}$. The semantics of all previously introduced kinds of types are naturally generalized by appropriately forwarding the context, e.g. $[\![T1 \times T_2 \times \ldots \times T_n]\!]_P = [\![T_1]\!]_P \times [\![T_2]\!]_P \times \ldots \times [\![T_n]\!]_P$.

The meaning of $\mathsf{class}\ C^I$, a class C annotated by invariant I, now is

$$\begin{array}{l}[\![\mathsf{class}\ C^I]\!]_P := \\ \quad [\![*T_1]\!]_P \times [\![*T_2]\!]_P \times \ldots \times [\![*T_n]\!]_P \times \\ \quad (P \times [\![A_{11}]\!]_P \times [\![A_{12}]\!]_P \times \ldots \times [\![A_{1a_1}]\!]_P) \to (I \times [\![B_1]\!]_P) \times \\ \quad (P \times [\![A_{21}]\!]_P \times [\![A_{22}]\!]_P \times \ldots \times [\![A_{2a_2}]\!]_P) \to (I \times [\![B_2]\!]_P) \times \\ \quad \ldots \times \\ \quad (P \times [\![A_{m1}]\!]_P \times [\![A_{m2}]\!]_P \times \ldots \times [\![A_{ma_m}]\!]_P) \to (I \times [\![B_m]\!]_P)\end{array}$$

i.e. every function must ensure that under the assumption that the pre-store satisfies P that also its returned post-store satisfies I. If I initially (after the creation of the object) holds, it is therefore guaranteed that it is preserved by every application of an object function. Furthermore, if it can be ensured (by other means) that no function outside of the object invalidates the invariant, the invariant thus always holds before and after every invocation of the object function. This is in particular the case, if the invariant only refers to the store locations of variables that are kept private to the object.

To discuss the subtype properties of $\mathsf{class}\ C^I$ we generalize the subtype relationship to $S <:^P T$ which denotes that S is a subtype of T in context P. It holds if there exists an interpretation $S \triangleright^P T : [\![S]\!]_P \to [\![T]\!]_P$ i.e., in context I, every value of type S may be interpreted as a value of type T. The properties of subtypes described in the previous sections still hold by appropriately forwarding the context, e.g. $(S_1 \times S_2 \times \ldots \times S_n) <:^P (T_1 \times T_2 \times \ldots \times T_n)$ if $S_i <:^P T_i$ for $1 \leq i \leq n$. We now investigate the subtype relationship

$$\mathsf{class}\ S^I <:^P \mathsf{class}\ T^{I'}$$

where $\mathsf{class}\ S$ has invariant I and $\mathsf{class}\ S$ has invariant I'. By expanding the definitions (and induction on the structure of C^I), one can show that,

- if $\mathsf{class}\ S <: \mathsf{class}\ T$, and
- if $I \Rightarrow I'$ (i.e. $[\![I]\!] \subseteq [\![I']\!]$),
- then $\mathsf{class}\ S^I <:^I \mathsf{class}\ T^{I'}$.

This is easy to see if we assume that none of the T_i, A_i, j, and B_i involve a class. In that case, the only differences between the original class semantics and the new one is the occurrence of $P = I$ as an argument type both in S^I and in $T^{I'}$ and the occurrence of I as a result type in S^I and the the corresponding occurrence of I' as a result type in $T^{I'}$; the later is allowed, since function types are covariant in their result types and $[\![I]\!] \subseteq [\![I']\!]$. The general claim can then be shown by induction on the structure of the class type.

Still we must not forget that the subtype relationship is qualified in that it only holds in a context $P = I$, i.e. we may substitute an object of type $\mathsf{class}\ S^I$ for any object of type $\mathsf{class}\ T^{I'}$ only if the invariant I holds. However, if we can guarantee that I initially (i.e. immediately after the construction of the object) holds, then, since all functions in P preserve I, the context $P = I$ is established at all times when also I' is expected to hold. Therefore classes are *covariant* in their invariants.

As for the consequences on the semantics on inheritance see the corresponding paragraph given below.

History Constraints A *history constraint* is a generalization of an invariant in that it relates the current values of the variables in the object store to their initial values (i.e. the values they had at the time when the object was created). For a history constraint H we thus have

$$[\![H]\!] \subseteq \mathit{Store} \times \mathit{Store}$$

We denote by $\mathsf{class}\ C^{I,H}$ a class C annotated by invariant I and history constraint H. Its meaning is given as:

$$
\begin{array}{l}
[\![\, \mathsf{class}\ C^{I,H} \,]\!]_P := \\
\quad [\![\, *T_1 \,]\!]_P \times [\![\, *T_2 \,]\!]_P \times \ldots \times [\![\, *T_n \,]\!]_P \times \\
\quad (P \times\ [\![\, A_{11} \,]\!]_P \times [\![\, A_{12} \,]\!]_P \times \ldots \times [\![\, A_{1a_1} \,]\!]_P) \stackrel{[\![H]\!]}{\rightarrow} (I \times [\![\, B_1 \,]\!]_P) \times \\
\quad (P \times\ [\![\, A_{21} \,]\!]_P \times [\![\, A_{22} \,]\!]_P \times \ldots \times [\![\, A_{2a_2} \,]\!]_P) \stackrel{[\![H]\!]}{\rightarrow} (I \times [\![\, B_2 \,]\!]_P) \times \\
\quad \ldots \times \\
\quad (P \times\ [\![\, A_{m1} \,]\!]_P \times [\![\, A_{m2} \,]\!]_P \times \ldots \times [\![\, A_{ma_m} \,]\!]_P) \stackrel{[\![H]\!]}{\rightarrow} (I \times [\![\, B_m \,]\!]_P)
\end{array}
$$

where S represents the set of possible initial states and

$$
\begin{array}{l}
P \times A_1 \times A_2 \times \ldots \times A_n \stackrel{H}{\rightarrow} I \times B := \\
\quad \{f \in P \times A_1 \times A_2 \times \ldots \times A_n \rightarrow I \times B \mid \\
\qquad \forall t \in P, s, s' \in I, a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n, b \in B : \\
\qquad\quad \langle t, s \rangle \in H \wedge f(s, a_1, a_2, \ldots, a_n) = \langle s', b \rangle \Rightarrow \langle t, s' \rangle \in H\}
\end{array}
$$

i.e. for all applications of object functions the only legal pairs $\langle t, s' \rangle$ of initial state t and poststate s' are those allowed by H.

We now investigate the subtype relationship

$$\mathsf{class}\ S^{I,H} <:^P \mathsf{class}\ T^{I',H'}$$

which essentially boils down to the relationship

$$(I \times A_1 \times \ldots \times A_n \stackrel{H}{\rightarrow} I \times B) <:^P (I' \times A'_1 \times \ldots \times A'_n \stackrel{H}{\rightarrow} I' \times B')$$

Let $I \sqcap H$ be the condition with meaning

$$[\![\, I \sqcap H \,]\!] := [\![\, I \,]\!] \cap \{s' \in \mathit{Store} \mid \exists s \in \mathit{Store} : \langle s, s' \rangle \in H\}$$

i.e. it denotes all states in which the invariant I holds and which are allowed by the history constraint H. Then, by generalization of the argument in the previous subsection, one can show that

- if $\mathsf{class}\ S <: \mathsf{class}\ T$, and
- if $I \Rightarrow I'$ (i.e. $[\![\, I \,]\!] \subseteq [\![\, I' \,]\!]$),
- if $H \Rightarrow H'$ (i.e. $[\![\, H \,]\!] \subseteq [\![\, H' \,]\!]$),
- then $\mathsf{class}\ S^{I,H} <:^{I \sqcap H} \mathsf{class}\ T^{I',H'}$.

i.e. I and H are preserved by all contexts in which the invariant holds and that are allowed by the history constraint H. If we now can guarantee that the initial state s (immediately after the creation of the object) satisfies $s \in I$ and $\langle s, s \rangle \in H$, then the context $P = I \sqcap H$ is established at all times when also I' respectively H' are expected to hold. Therefore classes are also *covariant* in their history constraints.

In the following we investigate the consequence of the subtype relationship of classes with invariants and history constraints with respect to inheritance.

Inheritance and Invariants respectively History Constraints The results above show that $\mathsf{class}\ S^{I,H}$ can be a subtype of class $\mathsf{class}\ T^{I',H'}$ if $I \Rightarrow I'$ and $H \Rightarrow H'$, i.e. class S may strengthen the invariant and history constraint of class T. However, if class S inherits from class T, this requires that not only the new functions of S (possibly overriding some functions of T) but also the inherited (non-overridden) functions of T must satisfy I respectively S.

> **If a class S inherits from a class T, and strengthens the invariant respectively history constraint of T, then also the non-overridden inherited functions of T must preserve the strengthened invariant respectively history constraint of S.**

However, it is very unlikely that a (previously defined) class $\mathsf{class}\ T^{I',H'}$ satisfies the additional constraints of a (still unknown) class $\mathsf{class}\ S^{I,H}$, except for one particular situation: if we define

$$I :\Leftrightarrow I' \wedge I''$$
$$H :\Leftrightarrow H' \wedge H''$$

where I'' and H'' are relations that cannot be violated by the execution of any function of T, because I'' and H'' refer to storage locations that cannot be modified by T. This can be shown, if all functions f_i of T are equipped with frame conditions with location sets L_i and the truth values of I'' and H'' do not depend on any store location in $\bigcup L_i$. The most typical case however is that I'' and H'' only refer to variables that were newly introduced by class S and that are kept secret by T. We thus may state:

> **If a class S inherits from a class T, it may strengthen the invariant $\mathbf{I'}$ respectively history constraint $\mathbf{H'}$ of T to $\mathbf{I' \wedge I''}$ respectively $\mathbf{H' \wedge H''}$, provided that the truth values of $\mathbf{I''}$ and $\mathbf{H''}$ only depend on store locations that cannot be modified by any function of T (e.g. store locations that are kept secret by T).**

6 Specifying Contracts

The type systems of most statically typed object-oriented programming languages are designed to obey the constraints outlined in Section 4. Consequently, they ensure that

- on the one side any object of type $\mathsf{class}\ S$ may be considered as an object of type $\mathsf{class}\ T$ if $\mathsf{class}\ S$ is a subtype of $\mathsf{class}\ T$, but
- on the other side the execution of a program may never assume that an object has some type $\mathsf{class}\ T$ while actually its type is some type $\mathsf{class}\ T'$ which is *not* a subtype of $\mathsf{class}\ T$.

The goal of these type systems is to guarantee that programs are *safe* in the sense that no unchecked errors of this kind occur. However, these type systems are not sufficiently strong to express true *contracts*, i.e. they cannot ensure that

also the constraints outlined in Section 5 are obeyed; consequently even in well-typed programs, the semantic *substitutability* of objects may be violated and unexpected behaviors may occur.

In order to express true contracts, we need more than a simple type system, we need a *behavioral specification language* which allows to precisely describe the behavior of functions and the constraints of values. Such a language is the Java Modeling Language (JML) [8] whose core is a version of first order predicate logic that is embedded into the syntactic and semantic framework of Java. JML is based on semantic foundations of "behavioral subtyping" [12] and thus supports the development of programs that obey the substitution principle. In particular, it allows to express the semantic constraints formulated in Section 5 as it is sketched below.

Method Behaviors Each Java function ("method") can be annotated by a "behavior" which integrates a pre-condition, a frame condition, a normal postcondition and possibly exceptional post-conditions of the form

```
requires precondition;
assignable framecondition;
ensures postcondition;
signals (exception_1 e) excondition_1;
...
signals (exception_n e) excondition_n;
```

with the following interpretation:

- The behavior is applicable, if *precondition* holds.
- The method only modifies store locations denoted by *framecondition*.
- The method terminates normally or throws an exception of one of the types *exception_1*,...,*exception_n*.
- If the method terminates normally, then *postcondition* holds.
- If the method throws an exception of type *exception_i*, then *excondition_i* holds.

If a class S inherits from a class T and overrides a method, the overridden method inherits the behavior of the overridden method and possibly extends it by an additional behavior. The combination of two behaviors is defined as follows:

- Preconditions are combined by logical *disjunction* to

  $$precondition_S \vee precondition_T$$

  In other words, the precondition is *weakened*.
- Frame conditions are combined such that the value of a store location may be only modified, if this is admissible according to all behaviors that is applicable for the method's prestate; i.e. if both behaviors with modifiable

location sets L_S and L_T are applicable, the method may modify only values at locations in the *intersection*

$$L_S \cap L_T$$

In other words, the frame condition is *strengthened*.

- Normal postconditions are combined such that every postcondition must hold in every behavior that is applicable to the current state; i.e. if both behaviors are applicable, the postcondition is combined by logical *conjunction* to

  $$\mathit{postcondition}_S \wedge \mathit{postcondition}_T$$

  In other words, the normal postcondition is *strengthened*.

- The postconditions of the same exception type $\mathit{exception}_i$ are combined such that every postcondition must hold in every behavior that is applicable to the current state; i.e. if both behaviors are applicable, the postcondition is combined by logical *conjunction* to

  $$\mathit{excondition}_{i,S} \wedge \mathit{excondition}_{i,T}$$

  An exception of type $\mathit{exception}_i$ may be only raised, if it may be raised according to all behaviors that are applicable in the current state. If both behaviors with exception sets E_S and E_T are applicable, then only exceptions in the *intersection*

  $$E_S \cap E_T$$

  may be raised. Exceptional postcondition are thus *strengthened*.

We see that the constraints described in Section 5 are obeyed and the overriding method can be semantically substituted for the overridden method.

Invariants and History Constraints Every Java class can be annotated by class invariants and history constraints as

```
invariant invariant;
constraint constraint;
```

These declarations essentially constrain the states of the objects at all times when a function of the object is called and when a function of the object returns.

If a class S inherits from a class T, it inherits the invariant and constraint of T and combines it with its own to

$$\mathit{invariants}_S \wedge \mathit{invariant}_T$$
$$\mathit{constraints}_S \wedge \mathit{constraint}_T$$

i.e. invariants and constraints are always *strengthened* and the obligations stated in Section 5 are met.

As a side remark, proving the correctness of invariants and constraints in JML is more complex than it might look at first glance. The JML specification essentially states that every function

- may, when it is called, assume that the invariants of all objects (of *any* type) hold,
- must, when it returns, ensure that the invariants of all objects (of *any* type) is preserved

The reason for this is described in the JML reference manual [10]:

> *The semantics given above is highly non-modular, but is in general necessary for the enforcement of invariance when no mechanisms are available to prevent aliasing problems, or when constructs like (concrete) public fields are used.*

This highlights a problem we have deliberately side-stepped in our previous discussion: in the presence of shared mutable variables, every function in every object may invalidate the invariant of *all objects in all classes*.

The reasoning about JML invariants and history constraints does therefore not scale to complex programs; for this reason, many JML verification tools are *not sound* (in the sense of logic) because they only verify that a function preserves the invariant/history constraint of the *current object*, not even of all objects of the current class, much less of all objects of all classes.

7 Light-Weight Specifications

Behavioral interface specification languages like JML are rich enough to constrain class inheritance such that the substitution principle is obeyed; furthermore automatic checking and semi-automatic verification tools are available that allow to detect certain violations of this principle [2]. Still programmers not familiar with logic may not find the learning curve of the formalism too steep and thus shy away from its use.

The situation may be possibly improved by a more gradual introduction to class specification such that with little effort some benefits can be gained; if then one is willing to invest more effort, also the gain may be increased. In particular, we would like to support the programmer by helping to avoid the the violation of the substitution principle.

In the further discussion let the term *constraint* denote both invariants and history constraints and let the term *mutator* denote a function that changes the state of an object respectively the global program state (in contrast, a *pure* function does not change any state). Then a subclass can only violate the substitution principle by one of the following actions:

1. **Adding a Mutator:** In a subclass, a mutator is introduced that violates a (possibly implicit) constraint of the superclass.

   As an example, a class `HorizontalRectangle` maintains the length and the width of a rectangle with the constraint that the length must not be less than the width. A subclass `MutableHorizontalRectangle` introduces a method `setLength` which forgets to check whether the requested new length is greater than or equal the current width.

2. **Adding a Constraint:** In a subclass, a (possibly implicit) constraint is introduced which is violated by a mutator in the superclass.

As an example, a class `Rectangle` maintains the length and width of a rectangle with a method `setLength` to change the length of the rectangle. A subclass `Square` introduces the constraint that the length equals the width; this constraint is violated if the method `setLength` of the superclass is invoked.

3. **Overriding a Function:** In a subclass, a function of the superclass is overridden in such a way that violates the (possibly implicit) specification of the function.

   As an example, a class `Square` maintains the length of a square with a method `expand` that multiplies the length with some same given factor, provided that the factor is positive (such that the square cannot be shrunk to a point). A subclass `MinSquare` maintains a minimal size and overrides the function `expand` such that if the resulting size would be too small, no change is performed. Using a `MinSquare` object for a `Square` object thus gives unexpected results.

To gradually tackle this problem, we suggest the following language of class annotations

class ::= [`nosubtype` | `subtype`] [*kind* `contract`] `class` *C* ...
kind ::= `core` | `simple` | `extended` | `full[`*lang*`]`
lang ::= `"JML"` | ...

which is further explained below.

7.1 `nosubtype` | `subtype`

The annotation `nosubtype` indicates that the corresponding class C is not intended as a subtype of another class i.e. that it is a root node in the subtype hierarchy of classes. Class C may still inherit from another class D but the type of C is not considered as a subtype of the type of D. The rationale for this annotation is that frequently inheritance is just used for code sharing respectively code reuse such that a deeper semantic relationship between the superclass and the subclass is intended; in that case also no obligation for the specification of such a relationship should be imposed. An extended type checker might subsequently refuse the use of an object of type C in a place where an object of type D is expected. On the other hand, the annotation `subtype` explicitly indicates that the corresponding class C is intended as the subtype of the class D from which it inherits.

Annotating a class as `nosubtype` or `subtype` also indicates that the objects of the class may be constrained by the contract language described below; if a class is annotated in such a way, also its subclasses must be annotated in one of the two forms. An extended type-checker should also ensure that every annotated class only contains declarations of variables whose visibility is restricted to the class and its subclasses (`private` and `protected`). For variables with more visibility, object constraints could be invalidated by any client through plain variable assignments.

7.2 `core contract`

The annotation `core contract` indicates the use of a tiny annotation language that helps to void fundamental violations of the substitution principle.

- Each class must indicate by a declaration

  ```
  mutable mvars;
  ```

  that only the values of the non-private variables *mvars* are modified by applications of the functions that are declared in the current class; optionally, also private variables may be included in the declaration. A declaration `mutable \nothing` indicates that a class has no mutators.

  In the presence of pointer structures, above declaration should be more generally read as "only properties that depend on the values of *mvars* may become invalidated by the application of functions of the current class". By this statement e.g. a non-private variable x should be included in *mvars*, if it contains a pointer that (directly or indirectly) refers to some object whose content is changed.

  As an alternative format, each variable itself might be declared as `mutable` for the object functions of the current class and its subclasses; this format however makes it easy to overlook missing `mutable` declarations. In the following, we stick to the primary form of declaration stated above.

- Every class may introduce *predicates*: a declaration of the form

  ```
  [ public | protected | private ]
  predicate pname(params) "text";
  ```

  introduces a new predicate named *pname* with the denoted visibility level and the number and types of arguments denoted by *params*. The meaning of the predicate is informally described by *text*.

- Every class may introduce *constraints*: a declaration of the form

  ```
  constrains cname: pname(cvars);
  ```

  introduces a constraint (invariant or a history constraint) *cname* on the non-private object variables *cvars* by predicate *pname*; optionally, also private variables may be included in the declaration.

  The use of explicitly declared predicates may look a bit heavy-weight for the core contract language; however, it comes handy for the extensions of the contract language described in the following sections. If this extension is not considered, we can also just have constraint declarations of form

  ```
  constrains cname(cvars): "text";
  ```

  which states that the variables *cvars* are constrained and what is the informal interpretation of the constraint.

- (Optional) For each constraint *cname* introduced in a (direct or indirect) superclass, the class must contain a declaration

```
constrains cname inherited;
```

such that all constraints for the current class (also the inherited ones) are textually visible within the class.

This rule is not crucial for the following discussion; its main purpose is a pragmatic one, namely to make programmers explicitly aware of all the constraints they have inherited from superclasses. A checker may easily detect and report its violation.

No constraint may refer to a variable whose visibility is not restricted to the class and its subclasses; every such occurrence should be reported as an error because the corresponding constraint could be violated by any client of the class through plain variable assignments.

Checking the Annotations A checker may investigate whether the declarations indicate a possible violation of the substitution principle. The checking proceeds by processing a class hierarchy top down such that for each class T two variable sets $T.cvars$ and $T.mvars$ are maintained:

- $T.cvars$ is the set of non-private variables that are constrained by C or an ancestor class of C.
- $T.mvars$ is the set of non-private variables that are modified by C or an ancestor class of C.

In detail, class T is processed as follows:

1. If T is a root class, then $C := \emptyset$, else $C := \text{parent}(T).cvars$.
2. If T is a root class, then $M := \emptyset$, else $M := \text{parent}(T).mvars$.
3. If T has constrained non-private variables $cvars$ such that $M \cap cvars \neq \emptyset$, then report "constraint prohibited".
4. If T has mutable non-private variables $mvars$ such that $C \cap mvars \neq \emptyset$, then report "mutation prohibited".
5. $T.cvars := C \cup \{x \in cvars$: x is not private$\}$
6. $T.mvars := M \cup \{x \in mvars$: x is not private$\}$

No variable in $T.mvars$ may be further constrained by a subclass of T because this constraint is potentially violated by a mutator of M. Likewise, no variable in $T.cvars$ may be further modified by a subclass of T because this would possibly violate a constraint of this variable (but see the paragraph "Overriding Mutators" below).

If correctly applied, this mechanism rules out the previously stated violations "Adding a Mutator" and "Adding a Constraint" of the substitution principle. For example, the declaration of `MutableHorizontalRectangle` in

```
core contract class HorizontalRectangle {
  protected int length; protected int width;
  mutable \nothing;
  predicate LE(int l, int w) "length is not below width";
```

```
  constrains sides: LE(length, width);
  ...
}

subtype core contract class MutableHorizontalRectangle
  extends HorizontalLength {
  mutable length;
  constrains sides inherited;
  void setLength(int l) { length = l; }
}
```

is reported as erroneous, because `MutableHorizontalRectangle` mutates a variable that was constrained by constraint `sides` in `HorizontalRectangle`. Likewise, the declaration of `Square` in

```
core contract class Rectangle {
 protected int length; protected int width;
 mutable length;
 void setLength(int l) { length = l; }
 ...
}

subtype core contract class Square extends Rectangle {
  predicate EQ(int l, int w) "length equals width";
  constrains isSquare: EQ(length, width);
  ...
}
```

is reported as erroneous because it constrains by the new constraint `isSquare` a variable that was mutated in `Rectangle`.

Overriding Mutators While above calculus rules out possible errors, it is also very stringent in that it does not allow to override in a subclass the mutator of a superclass in order to maintain additional constraints that the subclass might introduce.

In a refined version of the calculus, also each function is annotated by a clause

```
assignable avars;
```

where *avars* is the subset of *mvars* mutated by the function (an annotation `assignable \nothing` indicates that the function is a pure function rather than a mutator). Then Step 3 of the processing of class T can be refined as follows:

3 If T has among its constrained non-private variables *cvars* a variable that appears among the assigned variables *avars* of an inherited function f that is not overridden by T, then report

> "constraint prohibited because of inherited mutator f (you may override the definition of f if you can simultaneously ensure the constraint and also preserve the contract of f)".

The error message reminds the programmer of the possibility to override the mutator but also states the requirement that the original specification of the mutator must be preserved.

This would allow a definition of `Square` as in

```
core contract class Rectangle {
 protected int length; protected int width;
 mutable length;
 // contract: ...
 void setLength(int l) { length = l; }
 ...
}

subtype core contract class Square extends Rectangle {
  predicate EQ(int l, int w) "length equals width";
  constrains isSquare: EQ(length, width);
  // preserves isSquare and the original contract of setLength
  void setLength(int l) { ... }
  ...
}
```

where it is now the task of the programmer to make sure that the overriding definition of `setLength` preserves the constraint `isSquare` (if this is not possible, then the subclass is ill-defined i.e. violates the substitution principle).

The problem with the extension is that the contract of a mutator itself is just implicitly stated (e.g. as a comment) but not part of the calculus itself. The next section is going to overcome this limitation.

7.3 `simple contract`

The annotation `simple contract` indicates that the corresponding class C is (in addition to `core contract` annotations) annotated by a simple form of a behavioral contract:

- Class constraints are given in the general form

  `constrains` *cname*`:` *pname1*`(`*cvars1*`)` `&&...&&` *pnamen*`(`*cvarsn*`);`

  i.e. as *conjunctions* of atomic formulas.

- Methods are specified by behaviors in the following form

  ```
  requires pname1(pvars1) || ... || pnamen(pvarsn);
  assignable avars;
  ensures qname1(qvars1) && ... && qnamen(qvarsn);
  signals (ex1 e1) ename11(ev11) &&...&& ename1e1(ev1n1);
  ...
  ```

  with preconditions specified as *disjunctions* of atomic formulas and (normal and exceptional) postconditions as *conjunctions* of atomic formulas.

If a class C overrides a function of a subclass, the following rules apply for the specification of an overriding function:

- The list of atomic formulas in a precondition may be extended to

```
requires pname1(pvars1) || ... || pnamen(pvarsn)
      || pname_n+1(pvars_n+1) || ...;
```

- The set of assignable variables may be a subset of the original set.
- The list of atomic formulas in a postcondition may be extended to

```
ensures qname1(qvars1) && ... && qnamen(qvarsn)
      && qname_n+1(qvars_n+1) && ...;
```

- The set of `signals` declarations may be a subset of the original set. Each declaration may be extended to

```
signals (exi ei) enamei1(evi1) &&...&& enamin_i(evin_i)
               && enamein_i+1(evin_i+1) && ...;
```

The specification of the overriding definition is thus at least as strong as the specification of the original function definition.

The virtue of this format is that there are simple syntactic rules the programmer must follow rather than semantic arguments. Unlike JML (where behaviors are implicitly inherited), the specification format makes also the total set of obligations on the function definition explicit, i.e. no obligation can remain "hidden". A checker may easily verify that the obligations on the specification format are met and report an error if a violation is detected.

With this format, it is thus possible to specify

```
simple contract class Rectangle {
 protected int length; protected int width;
 mutable length;

 predicate PSL(int l) "l is not negative";
 predicate QSL(int l, int length, int width)
   "length and width...";

 requires PSL(l);
 assignable length, width;
 ensures QSL(l, length, width);
 void setLength(int l) { length = l; }
 ...
}

subtype simple contract class Square extends Rectangle {
  predicate EQ(int l, int w) "length equals width";
  constrains isSquare: EQ(length, width);

  requires PSL(l);
```

```
  assignable length, width;
  ensures QSL(l, length, width);
  void setLength(int l) { ... }
  ...
}
```

which indicates that the overriding definition of `setLength` must satisfy both the constraint `isSquare` and the postcondition `QSL` (whatever this condition may look like). Furthermore, if the `setLength` in `Rectangle` would have specified

```
assignable length;
```

then the overriding definition in `Square` must also specify this clause and is thus not allowed to modify *width* (which may, depending on the interpretation of `QSL`, indicate that *setLength* is infeasible).

Propositions In order to support a core form of logical reasoning, declarations of the form

```
proposition(vars) pname1(...) ==> pname2(...);
```

may be introduced where as predicate arguments "..." any subset of *vars* may be used. The logical interpretation of such a statement is that of a universally quantified implication $\forall vars : pname1(\ldots) \Rightarrow pname2(\ldots)$. It implies that

- any occurrence of predicate *pname2* in a constraint or postcondition may be replaced by an occurrence of the stronger predicate *pname1* (after appropriate substitutions of the formal parameters by the concrete arguments)
- any occurrence of predicate *pname1* in a precondition may be replaced by an occurrence of the weaker predicate *pname2* (after appropriate substitutions of the formal parameters by the concrete arguments)

In this way the predicate introduced in a superclass may be implied by a stronger predicate introduced in a subclass or it may imply a weaker predicate introduced in a subclass. As an example, take the definitions:

```
simple contract class Rectangle {
 protected int length; protected int width;
 mutable length;

 predicate PSL(int l) "l is positive"
 predicate QSL(int l, int length, int width)
   "length and width..."

 requires PSL(l);
 assignable length, width;
 ensures QSL(l, length, width);
 void setLength(int l) { length = l; }
 ...
}
```

```
subtype simple contract class Square extends Rectangle {
  predicate EQ(int l, int w) "length equals width";
  constrains isSquare: EQ(length, width);

  predicate PSL2(int l) "l is not negative"
  predicate QSL2(int l, int length, int width)
    "length and width..."

  proposition forall(int l) PSL(l) ==> PSL2(l);
  proposition forall(int l, int length, int width)
    QSL2(l, length, width) ==> QSL(l, length, width);

  requires PSL2(l);
  assignable length, width;
  ensures QSL2(l, length, width);
  void setLength(int l) { ... }
  ...
}
```

Here the overriding definition of `setLength` in `Square` uses a weaker precondition and a stronger postcondition.

7.4 `extended contract`

The annotation `extended contract` uses the same language elements as `simple` contract but allows the use of arbitrary combinations of logical connectives in the specifications of constraints, preconditions, and (normal and exceptional) postconditions as well as the specification of arbitrary propositional logic formulas in propositions.

For verifying the wellformedness of contracts in sub-classes, simple syntactic considerations do not suffice any more; rather it has to be argued from the semantic point of view whether a precondition is not erroneously strengthened and a postcondition is not erroneously weakened.

However, the validity of these arguments can be fully automatically checked with the help of propositional satisfiability solvers; such solvers are freely available and can be easily integrated in a corresponding checker.

7.5 `full contract`

The annotation `full contract` indicates the use of a full-fledged behavioral specification language like "JML". Semantically, the core difference is that instead of undefined atomic predicates, the full power of first-order predicate logic with quantifiers may be used to give formal definitions of such predicates. Such specifications can in general not any more be automatically checked; while extended static checkers may automatically detect certain violations, their validity can only be verified with semi-automatic theorem proving tools.

For truly expressive specifications, the use of such a specification language is required. However, as the previous sections have strived to show, also simpler specification forms may be useful, at least as first steps in the use of formal specifications in object-oriented languages.

8 Conclusions

We have presented in this paper a sketch of the main theoretical elements that underly the Liskov substitution principle which should help to understand better under which circumstances class inheritance may violate this principle. While full-fledged behavioral specification languages like JML are designed in such a way that they help to obey substitutability, they are quite heavy-weight, partially because they depend on formal specifications expressed in predicate logic. In this paper, we have attempted to indicate a more lenient path by proposing a light-weight specification format which already with little effort may help to detect certain violations of the substitution principle. The specifications can be gradually refined to more precise descriptions that characterize inheritance hierarchies better and may detect more errors. Certain violations of these specifications are statically checkable with existing technology. Software developers that proceed along this path may thus have a considerably more lenient learning curve towards the use of behavioral specification languages.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, Secaucus, NJ, USA, 1996.

[2] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[3] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[4] Marshall Cline. Inheritance — Proper Inheritance and Substitutability, 2010. `http://www.parashift.com/c++-faq-lite/proper-inheritance.html` [Online; accessed 22-February-2010].

[5] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance Is Not Subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[6] Cunningham. Liskov Substitution Principle, 2010. `http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple`, [Online; accessed 22-February-2010].

[7] Kelvin Henney. From Mechanism to Method: Total Ellipse. *Dr.Dobb's*, March 2001. `http://www.drdobbs.com/cpp/184403771`, [Online; accessed 22-February-2010].

[8] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, September 2006. `http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf`, [Online; accessed 24-February-2010].

[9] Gary T. Leavens and Krishna Kishore Dhara. Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems. In *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, New York, NY, USA, 2000.

[10] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual (DRAFT), September 2009. `http://www.jmlspecs.org/OldReleases/jmlrefman.pdf`, [Online; accessed 24-February-2010].

[11] Barbara Liskov. Data Abstraction and Hierarchy (Keynote Address). In *OOPSLA '87: Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[12] Barbara H. Liskov and Jeannette M. Wing. Behavioural Subtyping Using Invariants and Constraints. In *Formal Methods for Distributed Processing: a Survey of Object-Oriented Approaches*, pages 254–280. Cambridge University Press, New York, NY, USA, 2001.

[13] R. C. Martin. The Liskov Substitution Principle. *C++ Report*, 8(3):14, 16–17, 20–23, March 1996.

[14] Bertrand Meyer. Design by Contract: Making Object-Oriented Programs that Work. In *TOOLS 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems, 24-28 November 1997, Melbourne, Australia*, page 360. IEEE Computer Society, 1997.

[15] Wikipedia. Circle-Ellipse Problem — Wikipedia, The Free Encyclopedia, 2010. `http://en.wikipedia.org/w/index.php?title=Circle-ellipse_problem&oldid=340530233`, [Online; accessed 22-February-2010].