Principles of Programming Languages

Thomas Wies

Class 22 - Objects: Subtyping

In this class, we will extend the type system of JakartaScript to account for objects and provide typing rules that formalize our intuition of the substitution principle in object-oriented programs.

Typing Objects

First, we extend our type language with object types:

\[ \tau \in \text{Typ} ::= \ldots \mid \{ \text{mut} f_1: \tau_1; \ldots; \text{mut} f_n: \tau_n \} \]

The rules that extend our typing relation \( \Gamma \vdash e : \tau \) to account for objects are given in Figure 1. The rule \( \text{TypeObj} \) infers the object type of an object literal \( \{ \text{mut} f : e \} \) by recursively inferring the types of the initialization expressions \( e \) of the fields \( f \). The rules for field dereference, \( \text{TypeGetField} \), and assignment to a field, \( \text{TypeAssignFld} \), are straightforward. Note that \( \text{TypeAssignFld} \) requires that the type of the assigned expression is equal to the type of the field that is being assigned to. Moreover, the updated field must have mutability \( \text{var} \). This requirement is analogous to the rule \( \text{TypeAssignVar} \) for typing assignments to variables.

Substitution Principle Revisited

Consider the following simple program:

\begin{verbatim}
const fun = function(x: { f: number }) {
  return 2 * x.f
};
fun({ f: 1 })
\end{verbatim}

In this program it should be safe to replace the argument \( \{ f: 1 \} \) of the call to function \( \text{fun} \) by, e.g., \( \{ f: 1, g: 2 \} \). The new program can still be safely evaluated. In fact, it will reduce to exactly the same value. However, the call \( \text{fun}({ f: 1, g: 2 }) \) is not well-typed according to our current typing rules. The problem is the rule \( \text{TypeCall} \), which requires that the type of the argument
in a call expression precisely matches the type of the function parameter that it is passed to. This condition is violated in the call `fun({f:1,g:2})`, since the argument has type `{f: number; g: number}` whereas the parameter of function `fun` has type `{f: number}`. In the following, we extend our typing rules so that the modified call to `fun` is well-typed.

Structural Subtyping

In order to extend our type system to account for the substitution principle, we identify the situations when we are allowed to replace a value of one type with a value of another type without violating type safety. We formalize these situations by the *subtype relation*. The subtype relation is a binary relation on types, denoted $\tau <: \tau'$. The judgment form $\tau <: \tau'$ states that values of type $\tau'$ can be safely substituted by values of type $\tau$. Informally, this means that all the things that can be done with values of type $\tau'$ can also be done with values of type $\tau$. If $\tau <: \tau'$ holds, we say that $\tau$ is a *subtype* of $\tau'$ or, conversely, that $\tau'$ is a *supertype* of $\tau$.

We incorporate the subtype relation into our type system by extending our typing rules with the rule `TypeSub` shown in Figure 2. The rule states that if an expression $e$ has the inferred type $\tau'$ and $\tau$ is a subtype of $\tau'$, then $e$ also has type $\tau$. The rule `TypeSub` thus captures the essence of the substitution principle.

It remains to define the actual subtype relation. We here use *structural subtyping* to define this relation. This means that we define subtyping by comparing the structure of the type expressions. For instance, TypeScript, the typed version of JavaScript, uses structural subtyping in its type checker. An alternative to structural subtyping is *nominal subtyping*. In nominal subtyping, the subtype relation must be defined explicitly in the program using class inheritance.
The question whether one type is a subtype of another can then be checked by looking up the relationship of the corresponding class names in the inheritance hierarchy. For example, the type system of Java is based on nominal subtyping. Scala uses nominal subtyping but also supports a restricted form of structural subtyping.

Our subtype relation is defined by the rules in Figure 3. We discuss the individual rules in more detail.

**General properties of subtyping.** First, we state some basic properties of the subtype relation. Clearly, every type should be a subtype of itself, i.e., the subtype relation is reflexive. This is captured by the rule SubRefl. Similarly, the subtype relation should be transitive, i.e., if $\tau_1$ is a subtype of $\tau_2$ and $\tau_2$ a subtype of $\tau_3$, then $\tau_1$ should also be a subtype of $\tau_3$.

**Subtyping of objects.** The crucial part of the subtype relation are the rules that govern when an object type is a subtype of another object type. These rules formalize our intuition of what subtyping in object-oriented languages is really about. There are three notions of object subtyping that we can distinguish. First, we can obtain a subtype of an object type by adding more fields to it. This is captured by the rule SubObjWidth. Second, we can replace the type $\tau$ of one of the fields of an object type by a subtype $\tau'$ of $\tau$. This is captured by the rule SubObjDepth. Observe that the depth rule is restricted to const fields only. We explain the rationale for this restriction in more detail below. Finally, we observe that for the semantics of objects the order in which we list the fields in an object does not matter for evaluation. Hence, we can permute the fields in an object type. This is captured by the rule SubObjPerm. The three rules can be stitched together using the transitivity rule. For example, the following derivation shows that the object type $\{g: \text{bool}; f: \text{number}\}$ is a subtype of $\{f: \text{number}\}$:

$\frac{\{\text{const } g: \text{bool}; \text{var } f: \text{bool}\} <: \{\text{var } f: \text{bool}; \text{const } g: \text{bool}\}}{\text{SubObjPerm}}$

$\frac{\{\text{var } f: \text{bool}; \text{var } g: \text{bool}\} <: \{\text{var } f: \text{bool}\}}{\text{SubObjWidth}}$

$\frac{\{\text{const } g: \text{bool}; \text{var } f: \text{bool}\} <: \{\text{var } f: \text{bool}\}}{\text{SubTrans}}$

**Subtyping of functions.** JAKARTA SCRIPT supports functions as first-class values. This raises the question whether we can formalize a subtyping rule for function types that captures the substitution principle for function values. That is, we are looking for a rule that places two function types in the subtype relation:

$((x: \tau_1) \Rightarrow \tau_2) <: ((x: \tau'_1) \Rightarrow \tau'_2)$

The question is then how the parameter types $\tau_1$ and $\tau'_1$, respectively, the result types $\tau_2$ and $\tau'_2$ should be related so that this subtype relationship is safe. To this end, consider the following example program:
const f1 = function (x: { f: number }) {
  return { g: x.f + x.f }
};

const f2 = function (x: { f: number }) {
  return { g: x.f + x.f, h: true }
};

const x = f1({f: 2});

The type of f1’s return value is { g: number } and the type of f2’s return value is { g: number; h: bool }. Clearly, it is safe to replace the call to f1 on line 7 by a call to f2 because all the capabilities of the object returned by f1 (reading and updating field g) are also provided by the object returned by f2. This observation generalizes: it is safe to substitute a function f1 by a function f2 if the f2 provides stronger guarantees about its return type than f1. That is, the first requirement for the subtype relationship

\[(x: \tau_1) \Rightarrow \tau_2 \prec (x: \tau'_1) \Rightarrow \tau'_2)\]

to hold is that \(\tau_2 \prec \tau'_2\) holds. We say that function types are covariant in their result types to indicate that the subtype relation keeps the same direction when we recurse into the result types.

What about the types of the function parameters \(\tau_1\) and \(\tau'_1\)? It may seem tempting to also require \(\tau_1 \prec \tau'_1\). However, this would be unsafe as the following program demonstrates:

class f1 = function (x: { f: number }) {
  return x.f + x.f
};

const f2 = function (x: { f: number; g: number }) {
  return x.f + x.g
};

f1({f: 2})
f2({f: 2, g: 2})

The parameter type of f2 is a subtype of f1’s parameter type. Observe that it is not safe to replace the call to f1 on line 7 with a call to f2, since we pass an object that only provides a field f but function f2 accesses fields f and g of its argument. However, we can safely replace the call to f2 on line 8 with a call to f1. The object passed object to f2 has fields f and g but f1 only accesses field f. Again this observation generalizes: it is always safe to replace a function by another function that makes fewer demands on the type of its argument. That is, for

\[(x: \tau_1) \Rightarrow \tau_2 \prec (x: \tau'_1) \Rightarrow \tau'_2\]

to hold we must require \(\tau'_1 \prec \tau_1\). We say that function types are contravariant in their parameter type to indicate that the subtype relation is inverted when we recurse into the parameter types. The resulting subtyping rule for function types is given by the rule SubFun.
Subtyping and Assignments

The purpose of a type system is to detect problems in programs that may lead to run-time errors during evaluation. So far, we have been extra careful in the design of our static type systems so that we obtain strong correctness guarantees for well-typed programs. Namely, we want to guarantee that the evaluation of a well-typed program will never get stuck. It turns out that with the introduction of subtyping we have introduced a flaw in our type system and this safety guarantee no longer holds. This flaw results from the interaction between the subtyping rule and the typing rule for assignment expressions. To understand this problem, consider the following program:

\[
\begin{align*}
\text{const } x &= \{f: \{g: \text{true}\}\}; \\
x.f &= \{\} \\
x.f.g
\end{align*}
\]

When we evaluate this program starting with some memory state \(M\), then after evaluating the defining expression of \(x\), we get a memory state like this

\[
M' = M[a_1 \mapsto \{g: \text{true}\}][a_2 \mapsto \{f: a_1\}]
\]

and we are left to evaluate the expression:

\[
\begin{align*}
\text{const } x &= \{f: \{g: \text{true}\}\}; \\
x.f &= \{\} \\
x.f.g
\end{align*}
\]

After evaluating the assignment expression, we obtain the memory state

\[
M'' = M'[a_3 \mapsto \{\}][a_2 \mapsto \{f: a_3\}]
\]

Note that we have allocated a new object at address \(a_3\) and have updated the value of the field \(f\) of the object at \(a_2\) to point to \(a_3\). We are now left with the expression

\[
\frac{\tau <: \tau'}{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{ SubTrans}
\]

\[
\frac{\{\text{mut } f:\tau; \text{mut'} g:\tau'\} <: \{f:\tau:\}}{\tau <: \tau'} \text{ SubObjWidth}
\]

\[
\frac{\{\ldots \text{const } f:\tau \ldots\} <: \{\ldots \text{const } f':\tau' \ldots\}}{\tau <: \tau'} \text{ SubObjDepth}
\]

\[
\frac{\{\ldots \text{mut } f:\tau; \text{mut'} g:\tau' \ldots\} <: \{\ldots \text{mut'} g':\tau'; \text{mut } f:\tau' \ldots\}}{\text{SubObjPerm}}
\]

\[
\frac{(x: \tau_1) \Rightarrow \tau_2 <: (y: \tau'_1) \Rightarrow \tau'_2}{\text{SubFun}}
\]
Evaluating the first field dereference operation \(a_2.f\) in \(M''\) will yield the address \(a_3\). The final field dereference operation that is left to be evaluated will now be \(a_3.g\). However, at this point we are stuck because the object stored at \(a_3\) in \(M''\) is the empty object \(\{\}\). Thus, the DoGetField rule for reducing field dereference operations does not apply because \(\{\}\) has no field \(g\).

Unfortunately, the above program is well-typed according to our current typing relation. This means that our typing relation does not have the progress property, which says that the evaluation of well-typed programs never gets stuck. Consequently, our current type system is not safe.

To see that the program is indeed well-typed. Let us look more closely at the critical step in the type inference. After typing the defining expression of \(x\), the typing environment \(\Gamma\) stores the inferred type of \(x\), which is:

\[
\Gamma(x) = (\text{const}, \{f:\{g:\text{bool}\}\})
\]

The types of the expressions \(x.f = \{\}\\) and \(x.f.g\) are inferred independently using the binding of \(x\) in \(\Gamma\). It is easy to see that \(x.f.g\) is well-typed under this binding of \(x\). To see that the assignment \(x.f = \{\}\\) is also well-typed we look at the complete subderivation for that expression:

\[
\frac{
\Gamma(x) = (\text{const}, \{f:\{g:\text{bool}\}\})}{\Gamma \vdash x : \{f:\{g:\text{bool}\}\}} \text{ TypeVar}
\]

\[
\frac{
\Gamma \vdash x : \{f:\{g:\text{bool}\}\}} \quad \Gamma \vdash x.f : \{g:\text{bool}\} \quad \text{TypeGetField}
\]

\[
\frac{
\{g:\text{bool}\} <: \{\}\} \quad \text{SubObjWidth}
\]

\[
\frac{
\Gamma \vdash x.f : \{\} \quad \Gamma \vdash \{\} : \{} \quad \text{TypeObj}
\]

\[
\frac{
\Gamma \vdash x.f = \{\} : \{} \quad \text{TypeAssignFld}
\]

We can see that the TypeSub rule provides the “glue” between the TypeAssignFld rule and the TypeGetField rule. It allows us to relax the type \(\{g : \text{bool}\}\) inferred for \(x.f\), the left side of the assignment, to its supertype \(\{\}\), which is the type that we infer for the right side of the assignment.

The problem with our current type system is that it allows the rule TypeSub to be used too liberally. The typing rules do not account for the fact that location expressions \(e.f\) can play two different roles during program evaluation. An \(e.f\) expression that is evaluated by the DoGetField rule serves as a source of a value that is read from memory. On the other hand, an expression \(e.f\) that occurs on the left side of an assignment and is evaluated by the DoAssignFld rule serves as a sink of a value that is stored into memory. In a source position, it is always safe to replace a location expression \(e.f\) of type \(\tau\) by an expression of a subtype \(\tau'\). That is, source positions are typed covariantly. However, for a sink position, it is only safe to replace \(e.f\) by an expression that is a supertype of \(\tau\), i.e., sink positions must be typed contravariantly. In other words, when we write a value into memory, then that value must at least provide all the
capabilities that are guaranteed by the type of the memory location to which we write. For example, when the type of \( e.f \) is that of a reset counter object, then we should not be allowed to reassign it to a counter object. Otherwise, we may later read \( e.f \) and call the \texttt{reset} method on the retrieved object, since we expect a reset counter object. We would then get stuck because the counter object does not have a reset method. Our current typing rules allow \( e.f \) expressions in sink positions to be typed covariantly, which violates type safety.

In the next class, we will repair our type system by fusing the \texttt{TypeSub} rule with those typing rules where subtyping is actually needed to capture the substitution principle. In particular, we will restrict the use of subtyping in the \texttt{TypeAssignFld} rule so that the above program is no longer well-typed.

### Depth-Subtyping of Mutable Fields.

The reason for restricting the depth-subtyping rule to \texttt{const} fields is related to the issue that \texttt{var} fields can serve as sinks of values. If we allowed depth subtyping of \texttt{var} fields, we would introduce another loop hole in our type system. The following modified version of the problematic program above demonstrates this issue:

```javascript
const y = { f: { g: true } };
const fun = function(x: { f: {} }) { x.f = {}; }; 
fun(y);
y.f.g
```

The type of \( y \) is \{\texttt{var} f: \{\texttt{var} g: bool\}\}. Evidently, the function \texttt{fun} is well-typed since its parameter \( x \) has type \{\texttt{var} f: {}\} and we are assigning \( x.f \) to \{\}. If we allow depth-subtyping of \texttt{var} fields, the call \texttt{fun(y)} is also well-typed since the type of \( y \) is then a subtype of \{\texttt{var} f: {}\}. Finally, the sequence of field dereferences \( y.f.g \) is again well-typed. Thus, if we relaxed the depth-subtyping rule, this program would be accepted by the type checker. However, during evaluation of the program, we will again get stuck when we try to evaluate the final dereference operation on field \( g \).

Although the problem with depth subtyping of objects is similar to the problem with the combination of the rules \texttt{TypeAssignFld} and \texttt{TypeSub} that we discussed earlier, the two issues are orthogonal. Even if we restrict the usage of the \texttt{TypeSub} with the \texttt{TypeAssignFld} rule, allowing depth subtyping of \texttt{var} fields will still be unsafe. We therefore disallow depth-subtyping of \texttt{var} fields. Consequently, the above program will be rejected by the type checker because the call \texttt{fun(y)} is not well-typed.

### Array Subtyping in Java.

Many existing typed object-oriented languages have weaknesses in their type systems that stem from the incorrect handling of subtyping and assignments. Sometimes these issues are oversights in the language design but other times they are deliberate design decisions. For example, TypeScript deliberately allows covariant subtyping of location expressions in sink positions. While this may be desirable for a scripting language that is not
used for writing critical programs, it can be problematic for other programming languages. A prime example of this is the handling of array subtyping in Java.

In Java, arrays are typed covariantly, i.e., if a Java type $S$ is a subtype of a type $T$, then $S[]$ is a subtype of $T[]$. This treatment of arrays is unsafe because arrays can serve both as sources and sinks of values, as demonstrated by the following program:

```java
1. String[] a = new String[1];
2. Object[] b = a; // a and b are now aliased
3. b[0] = new Object(); // a[0] now stores an Object
4. a[0].indexOf('c'); // indexOf not a method of Object
```

This Java program is accepted by the Java type checker. However, the call to `indexOf` on line 4 would fail when the program is executed because instances of class `Object` do not have an `indexOf` method. To prevent this problem, Java checks at run-time whether the type of a value that is stored into an array is a subtype of the array’s element type. This check is performed before each array store operation is executed. The array store check fails on line 3, since the element type of the array referenced by `b` is that of `a`, which is `String`, and the type of the stored value, which is `Object`, is not a subtype of `String`. At this point, the run-time environment raises an `ArrayStoreException` and aborts the execution of the program.

The covariant treatment of arrays in Java was introduced as a convenience for programmers in the initial version of the language. The rationale for this decision was that Java 1.0 did not yet support generic types. Covariant arrays provided programmers with more flexibility to write polymorphic code, even without generics. However, in retrospect, this decision is considered a mistake in the language’s design. The overhead of the additional dynamic type checks for array store operations make the usage of arrays in Java quite expensive.

The correct treatment of arrays in a type system with subtyping is to consider them invariant in their element type. This means that, if $S$ is a subtype of $T$, then it does not follow that $S[]$ is also a subtype of $T[]$ nor that $T[]$ is a subtype of $S[]$. In this case, the program above is rejected by the type checker because the assignment of $a$ to $b$ on line 2 is ill-typed. Additional run-time checks for array store operation are then no longer needed.