Variance and Generalized Constraints for C$^\sharp$ Generics

Burak Emir[1], Andrew Kennedy[2], Claudio Russo[2], and Dachuan Yu[3]

[1] EPFL, Lausanne, Switzerland
[2] Microsoft Research, Cambridge, U.K.
[3] DoCoMo Communications Laboratories USA, San Jose, California

Abstract. Generic types in C$^\sharp$ behave invariantly with respect to subtyping. We propose a system of type-safe variance for C$^\sharp$ that supports the declaration of covariant and contravariant type parameters on generic types. To support more widespread application of variance we also generalize the existing constraint mechanism with arbitrary subtype assertions on classes and methods. This extension is useful even in the absence of variance, and subsumes equational constraints proposed for Generalized Algebraic Data Types (GADTs). We formalize the subtype relation in both declarative and syntax-directed style, and describe and prove the correctness of algorithms for constraint closure and subtyping. Finally, we formalize and prove a type safety theorem for a featherweight language with variant classes and generalized constraints.

1 Introduction

The Generics feature of C$^\sharp$ 2.0 introduced *parametric polymorphism* to the language, supporting type parameterization for types (classes, interfaces, structs, and delegates) and methods (static and instance). Being object-oriented, C$^\sharp$ already offers *subtype polymorphism*, namely the ability for a value of type T to be used in a context that expects type U, if T is a subtype of U.

As it stands, though, subtype and parametric polymorphism interact only through subclassing. In particular, there is no subtyping relationship between distinct instantiations of the same generic type – type parameters are said to behave *invariantly* with respect to subtyping. This leads to a certain inflexibility: a method whose parameter has type `IEnumerable<Control>` cannot be passed an argument of type `IEnumerable<Button>`, even though this is safe: since `Button` is a subclass of `Control`, something that enumerates `Button`s also enumerates `Control`s. Dually, a method expecting a parameter of type `IComparer<Button>` cannot be passed an argument of type `IComparer<Control>`, even though this is safe: something that can compare `Control`s can also compare `Button`s.

1.1 Variance

We can increase the flexibility of generic types by declaring *variance* properties on type parameters. For example, `IEnumerable` is declared covariant (`+`) in its type parameter, and `IComparer` is declared contravariant (`-`), meaning that if $T{<:}U$ ("T is a subtype of U") then `IEnumerable<`T`>`$<:$`IEnumerable<`U`>` and `IComparer<`U`>`$<:$`IComparer<`T`>`. In our extension, these interfaces are declared as follows:

```
interface IEnumerable<+T> { IEnumerator<T> GetEnumerator(); }
interface IEnumerator<+T> { T Current { get; } }
interface IComparer<-T>   { int Compare(T x, T y); }
interface IComparable<-T> { int CompareTo(T other); }
```

To be safe, covariant type parameters can be used only in 'producer' positions in signatures (e.g. as result types, as in the `GetEnumerator` method and `Current` property above), and contravariant type parameters can be used only in 'consumer' positions (e.g. as argument types, as in the `Compare` and `CompareTo` methods above). These stringent requirements can make it hard to apply variance where it is desired. For example, a `List<+T>` type representing functional-style lists cannot even declare an append operation (`T` occurs in argument position):

```
class List<+T> {
  public List<T> Append(T other); // illegal
  public List<T> Append(List<T> other); // also illegal
```

Without such restrictions, there would be nothing to stop an *implementation* of the `Append` method updating the receiver list with its argument:

```
class List<+T> { private T head; private List<T> tail;
  public List(T head, List<T> tail){ this.head=head; this.tail=tail; }
  public T Hd(){ return head;} public List<T> Tl(){ return tail;}
  public List<T> Append(T other){ this.head=other; return this; }
  public List<T> Append(List<T> other){ this.tail=other; return this; }
}
```

This is unsafe: a `List<Button>` object could be updated with a `VScrollBar` value by first coercing it to `List<Control>`. As `VScrollBar` subtypes `Control`, but not `Button`, this violates safety:

```
List<Button> lb = new List<Button>(new Button(),null);
((List<Control>) lb).Append(new VScrollBar());
Button b = lb.Hd(); // we just read a scrollbar as a button
```

1.2 Generalized Constraints

We can overcome these restrictions through the use of *type constraints*:

```
class List<+T> { ...
  List<U> Append<U>(U other) where T : U { ... }
  List<U> Append<U>(List<U> other) where T : U { ... } }
```

Here `Append` is parameterized on an additional type `U`, constrained to be a supertype of the element type `T`. So an implementation of `Append` cannot place its argument in the list, as U is not a subtype of T, but it can create a new list cell of type List<U> with tail of type List<T>, as List<T> is a subtype of List<U>. It is easy to check that these refined signatures for Append rule out the unsafe implementations above, while still allowing the intended, benign ones:

```
class List<+T> { ...
  public List<U> Append<U>(U other) where T : U
    { return new List<U>(head,
          tail==null? new List<U>(other,null) : tail.Append(other)); }
  public List<U> Append<U>(List<U> other) where T : U
    { return new List<U>(head, tail==null? other: tail.Append(other));}
```

Notice how this is actually *less* constraining for the client code – it can always instantiate Append with T, but also with any supertype of T. For example, given a List<Button> it can append a Control to produce a result of type List<Control>. The designers of Scala [12] identified this useful pattern.

The type constraint above is not expressible in C$^\sharp$ 2.0, which supports only 'upper bounds' on method type parameters. Here we have a lower bound on U. Alternatively, it can be seen as an additional upper bound of the type parameter T of the enclosing class, a feature that is useful in its own right, as the following example demonstrates:

```
interface ICollection<T> { ...
  void Sort() where T : IComparable<T>;
  bool Contains(T item) where T : IEquatable<T>; ...
```

Here, constraints on T are localized to the methods that take advantage of them.

We therefore propose a generalization of the existing constraint mechanism to support arbitrary subtype constraints at both class and method level. This neatly subsumes both the existing mechanism, which has an unnatural asymmetry, and the *equational* constraint mechanism proposed previously [10]: any equation T=U between types can be expressed as a pair of subtype constraints T:U and U:T.

1.3 Contribution and Related Work

Adding variance to parametric types has a long history [1], nicely summarized by [8]. More recently, others have proposed variance for Java. NextGen, one of the original designs for generics in Java, incorporated definition-site variance annotations, but details are sketchy [2]. Viroli and Igarashi described a system of *use-site* variance for Java [8], which is appealing in its flexibility. It was since adopted in Java 5.0 through its *wildcard* mechanism [16]. However, use-site variance places a great burden on the user of generic types: annotations can become complex, and the user must maintain them on every use of a generic type. We follow the designers of Scala [11] and place the burden on the library designer, who must annotate generic definitions, and if necessary, factor them into covariant and contravariant components.

Our type constraints generalize the 'F-bounded polymorphism' of Java [7] and C$^\sharp$ and the bounded method type parameters of Scala [11], and also subsume previous work on equational constraints [10]. The treatment of constraint *closure* was inspired by previous work on constrained polymorphism for functional programming languages [14, 17] but has been adapted to handle Java-style inheritance.

We present the first formalization and proof of type safety for an object system featuring definition-site variance and inheritance in the style of Java or C$^\sharp$. Independently, variance for generics in the .NET Common Language Runtime has recently been formalized and proved sound [5].

We present an algorithm to decide subtyping in the presence of contravariance and generic inheritance. Previous systems have been presented in a non-algorithmic fashion [8, 16]. This is sufficient for showing soundness, but as we demonstrate, a naïve reading even of syntax-directed subtyping rules as a procedure leads to non-termination.

2 Design Issues

In this section, we study variance and generalized constraints in more detail, considering issues of type safety and language expressivity informally in code fragments. Sections 3 and 4 provide a formal foundation, proving the correctness of a subtyping algorithm and the soundness of the type system.

2.1 Variant Interfaces

Use-site variance, as first proposed by Viroli and Igarashi [8] and recast as *wildcards* in Java 5.0 [16], requires no annotations on type parameters at type definitions. Instead, an annotation on the *use* of a generic type determines (a) its properties with respect to subtyping, and (b) the members of the type that are 'visible' for that variance annotation. For example, a mutable `List<X>` class can be used covariantly, permitting values of type `List<+String>` to be passed at type `List<+Object>` (in Java, `List<? extends String>` passed at type `List<? extends Object>`), and restricting invocation to 'reader' methods such as `Get`. Conversely, the class can be used contravariantly, permitting values of type `List<-Object>` to be passed at type `List<-String>` (in Java, `List<? super Object>` passed at type `List<? super String>`), and restricting invocation to 'writer' methods such as `Add`.

With definition-site variance, the library designer must prepare for such uses ahead of time. One natural way to achieve this is to expose covariant and contravariant behaviour through the implementation of covariant and contravariant *interfaces*. For example, a non-variant mutable list class could implement two interfaces, one containing 'reader' methods, and the other 'writer' methods.

```
interface IListReader<+X> {
  X Get(int index);
  void Sort(IComparer<X> comparer); ...
}
interface IListWriter<-Y> {
  void Add(Y item);
  void AddRange(IEnumerable<Y> items); ...
}
class List<Z> : IListReader<Z>, IListWriter<Z> {
  private Z[] arr; public List() { ... }
  public Z Get(int index) { ... } ...
}
```

To be safe, covariant type parameters must not appear in argument positions, and contravariant parameters must not appear in result positions. To see why, consider the following counterexample:

```
interface IReader<+X> {
  X Get();              // this is legal
  void BadSet(X x)      // this is illegal
}
interface IWriter<-Y> {
  void Set(Y y);        // this is legal
  Y BadGet();           // this is illegal
}
class Bad<T> : IReader<T>, IWriter<T> {
  private T item; public Bad(T item) { this.item = item; }
  public void T Get() { return this.item ; }
  public void BadSet(T t) { this.item = t; }
  public void Set(T t) { this.item = t ; }
  public T BadGet() { return this.item; }
}
  IReader<object> ro = new Bad<string>("abc");
  ro.BadSet(new Button()); // we just wrote a button as a string
  ...
  IWriter<string> ws = new Bad<object>(new Button());
  string s = ws.BadGet(); // we just read a button as a string
```

This might give the impression that type safety violations necessarily involve reading and writing of object fields. This is not so: the toy subset of C$^\sharp$ studied in Section 4 is purely functional, but nevertheless it is worthwhile proving soundness, as the following example illustrates:

```
interface IComparer<+T> { int Compare(T x, T y); } // this is illegal
class LengthComparer : IComparer<string> {
  int Compare(string x, string y)
    { return Int32.Compare(x.Length, y.Length); }
}
... IComparer<object> oc = new LengthComparer();
    int n = oc.Compare(3,new Button()); // takes Length of int & button
```

2.2 Variant Delegates

Variance on interfaces has a very simple design: interfaces represent a pure contract, with no code or data, so there are no interactions with mutability, access qualifiers, or implementation inheritance. C$^\sharp$'s *delegates* have a similar feel – they can be considered as degenerate interfaces with a single `Invoke` method. It is natural to support variance on generic delegates too. Here are some examples, taken from the .NET base class library:

```
delegate void Action<-T>(T obj);
delegate int Comparison<-T>(T x, T y);
delegate bool Predicate<-T>(T obj)
delegate TOutput Converter<-TInput,+TOutput>(TInput input);
```

Variance on interfaces and delegates is already supported by the .NET Common Language Runtime (and was recently proved sound [5]). Although no CLR languages currently expose variance in their type system, it is expected that Eiffel's (unsafe) covariant generic classes will be represented by covariant generic interfaces, making use of the CLR's support for exact runtime types to catch type errors at runtime.

2.3 Variant Classes

The Adaptor pattern provides another means of factoring variant behaviour. Here, rather than implement variant interfaces directly, *adaptor* methods in a non-variant class provide alternative views of data by returning an object that implements a variant interface – or, if supported, a variant abstract class:

```
abstract class ListReader<+X> {
  abstract X Get(int index);
  abstract void Sort(IComparer<X> comparer); ...
}
abstract class ListWriter<-Y> {
  abstract void Add(Y item);
  abstract void AddRange(IEnumerable<Y> items); ...
}
class List<Z> {
  public ListReader<Z> AsReader() { ... }
  public ListWriter<Z> AsWriter() { ... } ...
}
```

Concrete variant classes are also useful. For example, here is a covariant class `Set` that implements immutable sets:

```
class Set<+X> : IEnumerable<X> where X : IComparable<X> {
  private RedBlackTree<X> items;
  public Set() { ... }
  public Set(X item) { ... }
  public bool All(Predicate<X> p) { ... }
  public bool Exists(Predicate<X> p) { ... }
  public IEnumerator<X> GetEnumerator() { ... } ...
}
```

When are covariant and contravariant parameters on classes safe? First, note that no restrictions need be placed on the signatures of constructors or static members, as the class type parameters cannot vary. The second constructor above has `X` appearing in a contravariant position (the argument), but this is safe: once an object is created, the constructor cannot be invoked at a supertype. For the same reason, constraints declared on a class may make unrestricted use of variant type parameters, as in the example above.

In general, fields behave invariantly and so their types must not contain any covariant or contravariant parameters. Fields marked `readonly`, however, can be treated covariantly – as we do in our formalization in Section 4.

No restrictions need be placed on `private` members, which is handy in practice when re-factoring code into private helpers. It is also useful on fields, as above, where a field can be mutated from within the class – for example, to re-balance the `RedBlackTree` representing the `Set` above. However, we must take care: if `private` is interpreted as a simple lexical restriction – "accessible from code lexically in this class" – then a type hole is the result:

```
class Bad<+X> { private X item;
  public void BadAccess(Bad<string> bs) {
    Bad<object> bo = bs;
    bo.item = new Button(); } // we just wrote a button as a string
}
```

A suitable safe interpretation of `private` is "accessible only through type-of-`this`". Here, that means access only through objects of type `Bad<X>`; `bo.item` would be inaccessible as `bo` has type `Bad<object>`.

The base types of a generic type must, of course, behave covariantly, otherwise we could circumvent our restrictions through inheritance.

2.4 Generalized Constraints

As we saw in the introduction, restrictions on the appearance of variant type parameters in signatures can be very limiting. For example, we cannot define a set-union operation for the class above because its argument has type `Set<X>`. But using generalized constraints, we *can* define it, as follows:

```
class Set<+X> : IEnumerable<X> where X : IComparable<X> { ...
  public Set<Y> Union<Y>(Set<Y> that) where X : Y { ... }
}
```

Note that Java cannot express such a signature, because it does not support lower bounds on method type parameters; though the use of bounded wildcards can achieve the same effect for type parameters used for a single argument.

What restrictions, if any, should we apply to occurrences of class type parameters within method level constraints? The answer is that a constraint on a method behaves covariantly on the left of the constraint, and contravariantly on the right. To see why this must be the case, consider the following pair of interfaces, which attempt to avoid occurrences of covariant (contravariant) parameters in argument (result) positions, by introducing illegal bounds:

```
interface IReader<+X> {
  X Get();                          // this is legal
  void BadSet<Z>(Z z) where Z : X; // this is illegal
}
interface IWriter<-Y> {
  void Set(Y y);                    // this is legal
  Z BadGet<Z>() where Y : Z;        // this is illegal
}
class Bad<T> : IReader<T>, IWriter<T> {
  private T item; public Bad(T item) { this.item = item; }
  public void T Get() { return this.item ; }
  public void BadSet<Z>(Z z) where Z : T { this.item = z; }
  public void Set(T t) { this.item = t ; }
  public Z BadGet<Z>() where T : Z { return this.item; }
}
... IReader<object> ro = new Bad<string>("abc");
    ro.BadSet<Button>(new Button()); // we wrote a button as a string
... IWriter<string> ws = new Bad<object>(new Button());
    string s = ws.BadGet<string>(); // we read a button as a string
```

2.5 Deconstructing Constraints

In earlier work [10], we made the observation that the interesting class of Generalized (rather than Parametric) Algebraic Datatypes, currently a hot topic in Functional Programming, are already definable using Generics in C$^\sharp$. However, capturing the full range of programs over such GADTs requires the addition of both equational constraints on methods and some equational reasoning on types.

Perhaps the smallest example requiring equational constraints and reasoning is implementing strongly-typed equality over type-indexed expressions. The special case for tuple expressions highlights the issues (see [10] for the full example):

```
abstract class Exp<T> {
  public abstract T Eval();
  public abstract bool Eq(Exp<T> that);
  public abstract bool EqTuple<C,D>(Tuple<C,D> that)
    where Tuple<C,D> : Exp<T>;
}
class Tuple<A,B>: Exp<Pair<A,B>> { public Exp<A> e1; public Exp<B> e2;
  public Tuple(Exp<A> e1,Exp<B> e2) { this.e1 = e1; this.e2 = e2; }
  public override Pair<A,B> Eval(){
    return new Pair<A,B>(e1.Eval(),e2.Eval()); }
  public override bool Eq(Exp<Pair<A,B>> that) {
    return that.EqTuple<A,B>(this);} // NB: Tuple<A,B><:Exp<Pair<A,B>>
  public override bool EqTuple<C,D>(Tuple<C,D> that) {
    // where Tuple<C,D><:Exp<Pair<A,B>>
    return e1.Eq(that.e1) && e2.Eq(that.e2); }
}
```

In [10], we add the equational constraint `where Pair<C,D> = T` to the abstract `EqTuple` method to allow the override in the specialized `Tuple` subclass to typecheck. In the override, the constraint specializes to the assumption `Pair<A,B>=Pair<C,D>` which the type system can deconstruct (since all generic type constructors are both injective and invariant) to deduce the equations `A=C` and `B=D`. From this it follows that `Exp<C><:Exp<A>` and `Exp<D><:Exp<B>`, justifying, respectively, the calls to methods `e1.Eq(that.e1)` and `e2.Eq(that.e2)`.

With subtype constraints we can employ the more natural pre-condition `Tuple<C,D><:Exp<T>`, shown here, which directly relates the type of `that` to the type of `this` using a bound rather than an oblique equation on `T`. In the override, the inherited bound yields the assumption `Tuple<C,D><:Exp<Pair<A,B>>`. From the class hierarchy, it is evident that `Exp<Pair<C,D>><:Exp<Pair<A,B>>`, since the only way `Tuple<C,D>` can subtype `Exp<Pair<A,B>>` is if its declared superclass, `Exp<Pair<C,D>>`, does so too. Since `Exp<T>` is invariant, we can deconstruct this constraint to conclude that `Pair<C,D><:Pair<A,B>` and, symmetrically, `Pair<A,B><:Pair<C,D>`. Deconstructing yet again, assuming that `Pair` is covariant, we obtain `C<:A,D<:B` and `A<:C,B<:D`. Shuffling these inequalities we can derive `Exp<C><:Exp<A>` and `Exp<D><:Exp<B>` which, finally, justify the recursive calls to `e1.Eq(that.e1)` and `e2.Eq(that.e2)`. To accommodate this sort of reasoning in general, our subtype judgement must be able to both deconstruct the inheritance relation, to obtain lower bounds on superclass instantiations, and deconstruct subtype relationships betweens different instantiations of the same generic class, to deduce relationships between corresponding type arguments, oriented by the variance properties of the class.

3 Types and Subtyping

We begin our formal investigation of variance and constraints with a description of the subtype relation, presented in both declarative and syntax-directed styles. Types, ranged over by T, U and V, are of two forms:

– *type variables*, ranged over by X, Y and Z, and
– *constructed types*, ranged over by K, of the form $C\texttt{<}\overline{T}\texttt{>}$ where C is a class or interface name, and $\overline{T}$ is a sequence of zero or more type arguments.

(As is common, we write vectors such as $\overline{T}$ as shorthand for $T_1, \ldots, T_n$).

The subtype relation $<:$ is determined by a class hierarchy (subclassing *is* subtyping in C$^\sharp$), and by variance properties of generic types. We therefore assume a set of declarations which specify for each class C its formal type parameters $\overline{X}$, variance annotations on those parameters $\overline{v}$, and base class and interfaces $\overline{K}$. We write $C\texttt{<}\overline{vX}\texttt{>:}\overline{K}$ for such a declaration. A variance annotation v is one of $\circ$ (invariant), `+` (covariant), and `-` (contravariant). In our examples, omitted annotations are implicitly $\circ$ (for backwards compatibility with C$^\sharp$).

For type soundness it is necessary to impose restrictions on how variant type parameters appear in signatures. Formally, we define a judgment $\overline{vX} \vdash T\ \textit{mono}$ which states that a type T behaves 'monotonically' with respect to its type variables $\overline{X}$ whose variance annotations are $\overline{v}$. This predicate on types is presented in Figure 1, with extension to subtype assertions. It makes use of a negation operation on variance annotations, with the obvious definition.

$$\frac{v_i \in \{\circ, +\}}{\overline{v}\overline{X} \vdash X_i\ \mathit{mono}}\ \text{V-VVAR} \qquad \frac{X \notin \overline{X}}{\overline{v}\overline{X} \vdash X\ \mathit{mono}}\ \text{V-VAR}$$

$$\frac{C\text{<}\overline{w}\overline{Y}\text{>} : \overline{K} \quad \begin{array}{c}\forall i\ w_i \in \{\circ, +\} \Rightarrow \overline{v}\overline{X} \vdash T_i\ \mathit{mono} \\ \forall i\ w_i \in \{\circ, -\} \Rightarrow \neg\overline{v}\overline{X} \vdash T_i\ \mathit{mono}\end{array}}{\overline{v}\overline{X} \vdash C\text{<}\overline{T}\text{>}\ \mathit{mono}}\ \text{V-CON} \qquad \neg v = \begin{cases} -, & \text{if } v = +, \\ \circ, & \text{if } v = \circ, \\ +, & \text{if } v = - \end{cases}$$

$$\frac{\neg\overline{v}\overline{X} \vdash T\ \mathit{mono} \quad \overline{v}\overline{X} \vdash U\ \mathit{mono}}{\overline{v}\overline{X} \vdash T <: U\ \mathit{mono}}\ \text{V-SUB}$$

Fig. 1. Variance validity of types and subtypes

$$\frac{\Delta \vdash T <: U \quad \Delta \vdash U <: V}{\Delta \vdash T <: V}\ \text{TRAN} \qquad \frac{}{\Delta \vdash X <: X}\ \text{VAR} \qquad \frac{T <: U \in \Delta}{\Delta \vdash T <: U}\ \text{HYP}$$

$$\frac{C\text{<}\overline{v}\overline{X}\text{>} : \overline{K} \quad \forall i, v_i \in \{\circ, +\} \Rightarrow \Delta \vdash T_i <: U_i \text{ and } v_i \in \{\circ, -\} \Rightarrow \Delta \vdash U_i <: T_i}{\Delta \vdash C\text{<}\overline{T}\text{>} <: C\text{<}\overline{U}\text{>}}\ \text{CON}$$

$$\frac{C\text{<}\overline{v}\overline{X}\text{>} : \overline{K}}{\Delta \vdash C\text{<}\overline{T}\text{>} <: [\overline{T}/\overline{X}]K_i}\ \text{BASE} \qquad \frac{C\text{<}\overline{T}\text{>} \lhd D\text{<}\overline{U}\text{>} \quad \Delta \vdash C\text{<}\overline{T}\text{>} <: D\text{<}\overline{V}\text{>}}{\Delta \vdash D\text{<}\overline{U}\text{>} <: D\text{<}\overline{V}\text{>}}\ \text{DEBASE}$$

$$\frac{C\text{<}\overline{v}\overline{X}\text{>} : \overline{K} \quad \Delta \vdash C\text{<}\overline{T}\text{>} <: C\text{<}\overline{U}\text{>} \quad v_i \in \{\circ, +\}}{\Delta \vdash T_i <: U_i}\ \text{DECON}^{+}$$

$$\frac{C\text{<}\overline{v}\overline{X}\text{>} : \overline{K} \quad \Delta \vdash C\text{<}\overline{T}\text{>} <: C\text{<}\overline{U}\text{>} \quad v_i \in \{\circ, -\}}{\Delta \vdash U_i <: T_i}\ \text{DECON}^{-}$$

Fig. 2. Subtyping rules

Before defining the subtyping relation proper, we introduce an auxiliary relation $\lhd$ over constructed types, denoting the reflexive transitive closure of the 'is an immediate base class of' relation. It is defined as follows.

$$\frac{}{K \lhd K} \qquad \frac{C\text{<}\overline{v}\overline{X}\text{>} : \overline{K} \quad [\overline{T}/\overline{X}]K_i \lhd K}{C\text{<}\overline{T}\text{>} \lhd K}$$

We impose three restrictions on the class hierarchy. First, that it is *acyclic*: if $C\text{<}\overline{v}\overline{X}\text{>} : \overline{K}$ and $K_i \lhd D\text{<}\overline{T}\text{>}$ then $C \neq D$. Second, that generic instantiations are *uniquely determined*: if $C\text{<}\overline{X}\text{>} \lhd D\text{<}\overline{T}\text{>}$ and $C\text{<}\overline{X}\text{>} \lhd D\text{<}\overline{U}\text{>}$ then $\overline{T} = \overline{U}$. Third, that it *respects variance*: if $C\text{<}\overline{v}\overline{X}\text{>} : \overline{K}$ then $\overline{v}\overline{X} \vdash \overline{K}\ \mathit{mono}$. It is easy to show that this extends transitively: under the same definition of C, if $C\text{<}\overline{X}\text{>} \lhd K$ then $\overline{v}\overline{X} \vdash K\ \mathit{mono}$.

We are now ready to specify subtyping. Let Δ range over lists of subtype assumptions of the form $T <: U$. Our subtyping relation is defined by a judgment $\Delta \vdash T <: U$ which should be read "under assumptions Δ we can deduce that T is a subtype of U". A declarative presentation of this relation is given in Figure 2.

Ignoring Δ for the moment, ground subtyping requires just three rules: we assert that subtyping is transitive (TRAN), that instantiations of the same class vary according to the annotations on the type parameters (CON), and that subclassing induces subtyping (BASE). Observe that reflexivity is admissible (by repeated use of CON), and that the induced equivalence relation for ground types is just syntactic equality.

Now suppose that subtyping judgments are open and we make use of assumptions in Δ. We add reflexivity on type variables (VAR), and hypothesis (HYP). This lets us deduce, for example, for contravariant I that $X<:C \vdash I\texttt{<}C\texttt{>}<:I\texttt{<}X\texttt{>}$.

These rules alone are insufficient to check code such as in Section 2.5. Suppose our subtype assumptions include $C\texttt{<}X\texttt{>}<:C\texttt{<}Y\texttt{>}$. Take any ground instantiation of X and Y, say $[T/X, U/Y]$. If C is invariant or covariant then $\vdash C\texttt{<}T\texttt{>}<:C\texttt{<}U\texttt{>}$ can hold only if $\vdash T<:U$. Dually, if C is invariant or contravariant then $\vdash C\texttt{<}T\texttt{>}<:C\texttt{<}U\texttt{>}$ can hold only if $\vdash U<:T$. This justifies inverting rule CON to obtain rules DECON$^+$ and DECON$^-$ that 'deconstruct' a type according to its variance.

In a similar vein, suppose our subtype assumptions include $C\texttt{<}X\texttt{>}<:D\texttt{<}Y\texttt{>}$, for class definitions $C\texttt{<}\circ Z\texttt{>}\texttt{:}D\texttt{<}Z\texttt{>}$ and $D\texttt{<-}Z\texttt{>}\texttt{:object}$. Consider any ground instantiation of X and Y, say $[T/X, U/Y]$. Then a derivation of $\vdash C\texttt{<}T\texttt{>}<:D\texttt{<}U\texttt{>}$ exists only if $\vdash D\texttt{<}T\texttt{>}<:D\texttt{<}U\texttt{>}$ and thus $\vdash U<:T$. We are justified in 'inverting' rule BASE to obtain DEBASE that uses the class hierarchy to derive a subtype relationship between two instantiations of the same class.

It is straightforward to prove standard properties of subtype entailment.

Lemma 1 (Substitution). *If $\Delta \vdash T<:U$ then $S\Delta \vdash ST<:SU$ for any substitution $S = [\overline{T}/\overline{X}]$.*

Proof. By induction on the derivation, using a similar property of $\triangleleft$. □

Lemma 2 (Weakening). *If $\Delta \vdash \Delta'$ and $\Delta' \vdash T<:U$ then $\Delta \vdash T<:U$.*

Proof. By induction on the subtyping derivation. □

We will also make use of the following lemma, which states that subtype assertions lift through type formers according to variance.

Lemma 3 (Subtype lifting). *Suppose that $\overline{v}\overline{X} \vdash V$ mono, and for all i, if $v_i \in \{\circ,\texttt{+}\}$ then $\Delta \vdash T_i<:U_i$, and if $v_i \in \{\circ,\texttt{-}\}$ then $\Delta \vdash U_i<:T_i$. Then $\Delta \vdash [\overline{T}/\overline{X}]V<:[\overline{U}/\overline{X}]V$.*

Proof. By induction on the variance validity derivation. □

3.1 Syntax-directed Subtyping

The declarative presentation of subtyping is direct, and it is easy to prove properties such as Substitution and Weakening, but it is not easy to derive an algorithm from the rules: reading the rules backwards, we can always apply rule TRAN to

$$\frac{}{\Psi \succ X <: X}\ \textsc{s-var} \qquad \frac{C\text{<}\overline{vX}\text{>} : \overline{K} \qquad \Psi \succ [\overline{T}/\overline{X}]K_i <: D\text{<}\overline{U}\text{>} \qquad C \neq D}{\Psi \succ C\text{<}\overline{T}\text{>} <: D\text{<}\overline{U}\text{>}}\ \textsc{s-base}$$

$$\frac{\Psi \succ T <: U \quad U <: X \in \Psi \quad T \neq X}{\Psi \succ T <: X}\ \textsc{s-lower} \qquad \frac{X <: T \in \Psi \quad \Psi \succ T <: K}{\Psi \succ X <: K}\ \textsc{s-upper}$$

$$\frac{C\text{<}\overline{vX}\text{>} : \overline{K} \quad \forall i, v_i \in \{\circ, \texttt{+}\} \Rightarrow \Psi \succ T_i <: U_i \text{ and } v_i \in \{\circ, \texttt{-}\} \Rightarrow \Psi \succ U_i <: T_i}{\Psi \succ C\text{<}\overline{T}\text{>} <: C\text{<}\overline{U}\text{>}}\ \textsc{s-con}$$

Fig. 3. Syntax-directed subtyping rules

introduce new subgoals. So we now consider an alternative set of *syntax-directed* subtyping rules, where the structure of the types determines uniquely a rule (scheme) to apply. These are presented in Figure 3.

We write $\Psi \succ T <: U$ to mean that "under context Ψ we can deduce that T is a subtype of U". As is usual, we eliminate the transitivity rule TRAN, rolling it into rules S-BASE, S-UPPER, and S-LOWER. We also work with a different form of context: instead of an arbitrary set of subtype assertions, the context Ψ provides upper or lower bounds for type variables. In place of a hypothesis rule, we have rules S-UPPER and S-LOWER that replace a type variable by one of its bounds.

For transitivity to be admissible, we need to impose some restrictions on the context Ψ. For example, consider the context $\Psi = \{C\text{<}X\text{>} <: Z, Z <: C\text{<}Y\text{>}\}$ for covariant C. Clearly we have $\Psi \succ C\text{<}X\text{>} <: Z$ and $\Psi \succ Z <: C\text{<}Y\text{>}$, but not $\Psi \succ C\text{<}X\text{>} <: C\text{<}Y\text{>}$. We need to add $X <: Y$ to Ψ to achieve this. We define a notion of *consistency* for contexts (see Pottier [14] and Trifonov and Smith [17] for similar ideas).

Definition 1 (Consistency). *A context Ψ is* consistent *if for any pair of assertions $T <: X \in \Psi$ and $X <: U \in \Psi$ it is the case that $\Psi \succ T <: U$.*

We should now have enough to relate the syntax-directed and declarative rules: given a consistent context Ψ that is equivalent to a set of constraints Δ (in the sense that $\Psi \succ \Delta$ and $\Delta \vdash \Psi$), the relation $\Psi \succ - <: -$ should coincide with $\Delta \vdash - <: -$. The proof of this rests on the admissibility of transitivity: if $\Psi \succ T <: U$ and $\Psi \succ U <: V$ then $\Psi \succ T <: V$. Attempts at a direct proof of transitivity fail (for example, by induction on the total height of the derivations). There are two difficult cases. If the first derivation ends with rule S-CON and the second ends with S-BASE then we need to 'push' the premises of S-CON through the second derivation. We use an auxiliary result (Lemma 6) to achieve this. If the first derivation ends with rule S-LOWER (so we have a proof of $\Psi \succ T <: X$) and the second ends with rule S-UPPER (so we have a proof of $\Psi \succ X <: V$) then we need to make use of the consistency of Ψ in the side-conditions of these rules ($T' <: X \in \Psi$ and $X <: U' \in \Psi$) to obtain a derivation of $\Psi \succ T' <: U'$. But to apply the induction hypothesis on this derivation we need to bound its size.

Lemma 4. *For any consistent context Ψ there exists a context Ψ' such that $\Psi \succ \Psi'$ and $\Psi' \succ \Psi$ and satisfying the following property: if $T <: X <: U \in \Psi'$ then*

$$\frac{}{\Psi \succ_{b,m} X{<:}X}\ \textsc{r-var} \qquad \frac{C\texttt{<}\overline{v}\overline{X}\texttt{>}:\overline{K} \qquad \Psi \succ_{b,m} [\overline{T}/\overline{X}]K_i{<:}D\texttt{<}\overline{U}\texttt{>} \qquad C \neq D}{\Psi \succ_{b,m} C\texttt{<}\overline{T}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>}}\ \textsc{r-base}$$

$$\frac{\Psi \succ_{b,m} T{<:}U \quad U{<:}X \in \Psi \quad T \neq X}{\Psi \succ_{b+1,m} T{<:}X}\ \textsc{r-lower} \qquad \frac{X{<:}T \in \Psi \quad \Psi \succ_{b,m} T{<:}K}{\Psi \succ_{b+1,m} X{<:}K}\ \textsc{r-upper}$$

$$\frac{C\texttt{<}\overline{v}\overline{X}\texttt{>}:\overline{K} \quad \forall i, v_i \in \{\circ,\texttt{+}\} \Rightarrow \Psi \succ_{b,n} T_i{<:}U_i \text{ and } v_i \in \{\circ,\texttt{-}\} \Rightarrow \Psi \succ_{b,n} U_i{<:}T_i}{\Psi \succ_{b,n+1} C\texttt{<}\overline{T}\texttt{>}{<:}C\texttt{<}\overline{U}\texttt{>}}\ \textsc{r-con}$$

Fig. 4. Ranked syntax-directed subtyping rules

there is a derivation $\Psi' \succ T{<:}U$ in which all uses of rules S-LOWER *and* S-UPPER *are trivial, namely, that the premise is of the form $\Psi' \succ V{<:}V$.*

Proof. By consistency of Ψ, for any $T{<:}X{<:}U \in \Psi$ we have a derivation of $\Psi \succ T{<:}U$. For every sub-derivation that ends with the conclusion $\Psi \succ V{<:}X$, add $V{<:}X$ to the context, likewise for every sub-derivation that ends with $\Psi \succ X{<:}V$, add $X{<:}V$ to the context. If we repeat this process the resultant context $\Psi' \supseteq \Psi$ has the desired property. □

Figure 4 presents a 'ranked' variant of the syntax-directed rules, where the judgment $\Psi \succ_{b,m} T{<:}U$ is indexed by natural numbers b and m, where b is a bound on the height of the derivation with respect to rules R-LOWER and R-UPPER, and n is a bound on the height with respect to rule R-CON. Note that rule R-BASE does not count towards either measure: in our proofs we make use of the following lemma that lets us elide inheritance.

Lemma 5 (Variant inheritance). *If $\Psi \succ_{b,m} C\texttt{<}\overline{T}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>}$ then $C\texttt{<}\overline{T}\texttt{>} \lhd D\texttt{<}\overline{V}\texttt{>}$ and $\Psi \succ_{b,m} D\texttt{<}\overline{V}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>}$ for some $\overline{V}$.*

Proof. By induction on the subtyping derivation. □

Lemma 6. *Fix some n, c, l, $\overline{T}$, $\overline{U}$. Suppose that for any $m \leqslant n$, any $b \leqslant c$ and any W, if $\Psi \succ_{b,m} T_i{<:}W$ and $v_i \in \{\circ,\texttt{+}\}$, then there exists r such that $\Psi \succ_{b+l,r} U_i{<:}W$. Likewise suppose that for any $m \leqslant n$, any $b \leqslant c$ and any W, if $\Psi \succ_{b,m} W{<:}T_i$ and $v_i \in \{\circ,\texttt{-}\}$, then there exists r such that $\Psi \succ_{b+l,r} W{<:}U_i$.*

1. *For any T and V such that $\overline{v}\overline{X} \vdash T$ mono, if $\Psi \succ_{c,n} [\overline{T}/\overline{X}]T{<:}V$ then $\Psi \succ_{c+l,r} [\overline{U}/\overline{X}]T{<:}V$ for some r.*
2. *For any U and V such that $\neg\overline{v}\overline{X} \vdash U$ mono, if $\Psi \succ_{c,n} V{<:}[\overline{T}/\overline{X}]U$ then $\Psi \succ_{c+l,r} V{<:}[\overline{U}/\overline{X}]U$ for some r.*

Proof. By simultaneous induction on the subtyping derivations in (1) and (2).

Lemma 7 (Transitivity). *Let Ψ be a consistent context. If $\Psi \succ T{<:}U$ and $\Psi \succ U{<:}V$ then $\Psi \succ T{<:}V$.*

Proof. Using Lemma 4, assume that Ψ satisfies the stronger conditions described there. We now prove the following equivalent 'ranked' statement. If $\mathcal{D}_1$ and $\mathcal{D}_2$ are derivations of $\Psi \succ_{b,m} T{<:}U$ and $\Psi \succ_{c,n} U{<:}V$, then $\Psi \succ_{b+c,r} T{<:}U$ for some r. We proceed by induction on $(b+c, m+n)$, ordered lexicographically. We make use of Lemma 6 for R-CON against R-BASE. □

Theorem 1 (Equivalence of syntax-directed and declarative rules). *Provided Ψ is consistent, $\Psi \succ \Delta$ and $\Delta \vdash \Psi$, then $\Psi \succ T{<:}U$ iff $\Delta \vdash T{<:}U$.*

Proof. By induction on the derivations, using Lemma 7 for rule TRAN. □

3.2 Subtyping Algorithm

Our syntax-directed rules can be interpreted as a procedure for checking subtypes: if a subtype assertion holds, then the procedure terminates with result *true*. To show that the procedure will terminate with *false* if the relation does *not* hold, it suffices to find some measure on subtype judgments that strictly decreases from conclusion to premises of the syntax-directed rules. Unfortunately, there is no such measure for the rules of Figure 3. Consider the following classes:

$$N\texttt{<-}X\texttt{>:object} \qquad \text{and} \qquad C : N\texttt{<}N\texttt{<}C\texttt{>>}$$

Now consider checking the subtype assertion $C{<:}N\texttt{<}C\texttt{>}$. If we attempt to construct a derivation, we end up back where we started:

$$\dfrac{\dfrac{\vdots}{\succ C{<:}N\texttt{<}C\texttt{>}}}{\dfrac{\succ N\texttt{<}N\texttt{<}C\texttt{>>}{<:}N\texttt{<}C\texttt{>}}{\succ C{<:}N\texttt{<}C\texttt{>}}\ \text{S-BASE}}\ \text{S-CON}$$

A similar issue arises with constraints. Suppose that $\Psi = \{X{<:}N\texttt{<}N\texttt{<}X\texttt{>>}\}$, and consider checking the subtype assertion $X{<:}N\texttt{<}X\texttt{>}$. Even simple equations on type variables, expressed as bounds, such as $\{X{<:}Y, Y{<:}X\}$, can induce looping behaviour, for example testing $X{<:}\texttt{object}$.

These examples can be dealt with straightforwardly if the algorithm keeps a set of goals 'already seen', returning *false* when asked to prove an assertion from the set. Unfortunately this solution is not universal. Consider these definitions:

$$N\texttt{<-}X\texttt{>:object} \qquad \text{and} \qquad D\texttt{<}Y\texttt{>}:N\texttt{<}N\texttt{<}D\texttt{<}D\texttt{<}Y\texttt{>>>>}$$

Now consider checking $D^m\texttt{<}T\texttt{>}{<:}N\texttt{<}D^m\texttt{<}T\texttt{>>}$ where D^m has the obvious interpretation as m iterations of the type constructor D:

$$\begin{array}{c}
\vdots \\
\succ D^{m+1}\texttt{<}T\texttt{>}{<:}N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>} \\
\hline
\succ N\texttt{<}N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>>}{<:}N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>} \quad \text{S-CON} \\
\hline
\succ D^{m}\texttt{<}T\texttt{>}{<:}N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>} \quad \text{S-BASE} \\
\hline
\succ N\texttt{<}N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>>}{<:}N\texttt{<}D^{m}\texttt{<}T\texttt{>>} \quad \text{S-CON} \\
\hline
\succ D^{m}\texttt{<}T\texttt{>}{<:}N\texttt{<}D^{m}\texttt{<}T\texttt{>>} \quad \text{S-BASE}
\end{array}$$

```
$\mathrm{Sub}(\Xi, \Psi, T, U) =$
  if $(T, U) \in \Xi$ then *false*
  else let $\mathrm{Sub}'(T', U') = \mathrm{Sub}(\{(T, U)\} \cup \Xi, \Psi, T', U')$ in
  case $T, U$ of
    $X, X \Rightarrow$ *true*
    $T, X \Rightarrow \bigvee_{T_i <: X \in \Psi} \mathrm{Sub}'(T, T_i)$
    $X, K \Rightarrow \bigvee_{X <: T_i \in \Psi} \mathrm{Sub}'(T_i, K)$
    $C\texttt{<}\overline{T}\texttt{>}, D\texttt{<}\overline{U}\texttt{>} \Rightarrow \bigvee_{K \in \overline{K}} \mathrm{Sub}'([\overline{T}/\overline{X}]K, D\texttt{<}\overline{U}\texttt{>})$, if $C \neq D$ and $C\texttt{<}\overline{v}\overline{X}\texttt{>} : \overline{K}$
    $C\texttt{<}\overline{T}\texttt{>}, C\texttt{<}\overline{U}\texttt{>} \Rightarrow \bigwedge_{i \mid v_i \in \{\circ, +\}} \mathrm{Sub}'(T_i, U_i) \wedge \bigwedge_{i \mid v_i \in \{\circ, -\}} \mathrm{Sub}'(U_i, T_i)$, if $C\texttt{<}\overline{v}\overline{X}\texttt{>} : \overline{K}$
```

Fig. 5. Subtyping algorithm

After four rules we end up at the subgoal $D^{m+1}\texttt{<}T\texttt{>} <: N\texttt{<}D^{m+1}\texttt{<}T\texttt{>>}$, demonstrating that there is no derivation.

We have not yet devised an algorithm that can check this assertion; nor have we proved that the problem is undecidable. Instead, we impose a natural restriction on class hierarchies that guarantees termination. We introduce the notion of *closure* of a set of types under decomposition and inheritance.

Definition 2 (Closure of types). *A set of types $\mathcal{S}$ is* closed *if whenever $C\texttt{<}\overline{T}\texttt{>} \in \mathcal{S}$ then $\overline{T} \subseteq \mathcal{S}$ (decomposition) and whenever $K \in \mathcal{S}$ and $K \lhd K'$ then $K' \in \mathcal{S}$ (inheritance). The* closure *of $\mathcal{S}$ is the least closed superset of $\mathcal{S}$.*

Now consider the closure of the set $\{D\texttt{<object>}\}$ with respect to the above class definitions. It is easy to see that it is infinite. We rule out such classes; in fact, the .NET Common Language Runtime already imposes the same restriction [4, Partition II, §9.2], which enables eager loading of superclasses.

Definition 3 (Finitary definitions). *A set of class definitions is* finitary *if for any set of types $\mathcal{S}$ making use of those classes, its closure is finite.*

Fortunately, there is an algorithm that can check whether or not a set of class definitions is finitary [18, §6].

Figure 5 presents our subtyping algorithm in functional style. The additional parameter Ξ is a set of pairs of types representing subtype assertions already visited. The algorithm assumes that class definitions are finitary.

Definition 4 (Small derivations). *A derivation of $\Psi \succ T <: U$ is* small *if each proper sub-derivation has a conclusion other than $\Psi \succ T <: U$, and is itself small.*

It is easy to see that an arbitrary derivation can be transformed into a small derivation. We make use of this fact in the proof of completeness.

Theorem 2 (Soundness and completeness of subtyping algorithm). $\mathrm{Sub}(\{\}, \Psi, T, U) = $ *true iff* $\Psi \succ T <: U$.

Proof. Soundness ($\Rightarrow$). By induction on the call tree. *Completeness* ($\Leftarrow$). Let $\mathcal{P} = \{(T, U) \mid \Psi \succ T <: U$ is a sub-derivation of $\mathcal{D}\}$. We show, by induction on $\mathcal{D}$, that if $\mathcal{D}$ is a small derivation of $\Psi \succ T <: U$ and $\Xi \cap \mathcal{P} = \emptyset$ then $\mathrm{Sub}(\Xi, \Psi, T, U) =$ *true*. □

Theorem 3 (Termination). *For any Ξ, any consistent, finite Ψ and any T and U, the procedure* $\mathrm{Sub}(\Xi, \Psi, T, U)$ *terminates with result* true *or* false.

Proof. Let $\mathcal{S} = \{T, U\} \cup \{T \mid T{<:}U \in \Psi\} \cup \{U \mid T{<:}U \in \Psi\}$. Call its closure $\mathcal{T}$, which is finite if we assume finitary class definitions. Then it is easy to see that at each recursive call to Sub, the cardinality of $(\mathcal{T} \times \mathcal{T}) \setminus \Xi$ decreases by one. Hence the algorithm terminates. $\square$

3.3 Constraint Closure

There remains one piece of the subtyping jigsaw to put in place: given a set of constraints Δ, as declared or inherited by a method, determining an equivalent, consistent context Ψ, used as input to the subtyping algorithm when type-checking the body of the method.

Not all constraint sets are useful: in particular, constraints between types that are unrelated in the hierarchy can never be satisfied. That's not enough, though: a constraint set may *entail* unsatisfiable constraints. (For example, the set $\{C{<:}X, X{<:}D\}$ is unsatisfiable if C is unrelated to D in the class hierarchy.) So we define a notion of *closure* for constraint sets.

Definition 5 (Closure). *A constraint set Δ is* closed *if it is closed under transitivity, inheritance and decomposition:*

- *If $T{<:}U \in \Delta$ and $U{<:}V \in \Delta$ then $T{<:}V \in \Delta$.*
- *If $C\texttt{<}\overline{T}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>} \in \Delta$ and $C\texttt{<}\overline{T}\texttt{>} \lhd D\texttt{<}\overline{V}\texttt{>}$ with $D\texttt{<}\overline{v}\overline{X}\texttt{>}:\overline{K}$ then for each i, if $v_i \in \{\circ, +\}$ then $V_i{<:}U_i \in \Delta$ and if $v_i \in \{\circ, -\}$ then $U_i{<:}V_i \in \Delta$.*

The closure *of Δ, written $\mathrm{Cl}(\Delta)$, is the least closed superset of Δ.*

Definition 6 (Consistency of constraint sets). *A constraint set Δ is* consistent *if for any constraint $C\texttt{<}\overline{T}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>} \in \mathrm{Cl}(\Delta)$ there exists some $\overline{V}$ such that $C\texttt{<}\overline{T}\texttt{>} \lhd D\texttt{<}\overline{V}\texttt{>}$.*

To construct a context Ψ from a constraint set Δ we make use of a partial function Dec which takes an arbitrary constraint $T{<:}U$ and produces a set of constraints on type variables through a combination of inheritance and decomposition (Pottier defines a similar notion [14]).

$$\begin{aligned}
\mathrm{Dec}(X{<:}T) &= \{X{<:}T\} \\
\mathrm{Dec}(T{<:}X) &= \{T{<:}X\} \\
\mathrm{Dec}(C\texttt{<}\overline{V}\texttt{>}{<:}D\texttt{<}\overline{U}\texttt{>}) &= \begin{cases} \bigcup_{i \mid v_i \in \{\circ,+\}} \mathrm{Dec}(T_i{<:}U_i) \cup \bigcup_{i \mid v_i \in \{\circ,-\}} \mathrm{Dec}(U_i{<:}T_i) \\ \quad \text{if } C\texttt{<}\overline{V}\texttt{>} \lhd D\texttt{<}\overline{T}\texttt{>} \text{ for some } \overline{T} \text{ where } D\texttt{<}\overline{v}\overline{X}\texttt{>}:\overline{K} \\ \text{undefined otherwise.} \end{cases}
\end{aligned}$$

We combine this with transitive closure in the following Lemma.

$$
\begin{array}{rll}
\text{(class def)}\ cd & ::= & \texttt{class}\ C\texttt{<}\overline{v}\,\overline{X}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta\ \{\overline{P}\ \overline{T}\,\overline{f}\texttt{;}\ kd\ \overline{md}\} \\
\text{(constr def)}\ kd & ::= & \texttt{public}\ C\texttt{(}\overline{T}\,\overline{f}\texttt{)}\ \texttt{: base(}\overline{f}\texttt{)}\ \{\texttt{this.}\overline{f}\ \texttt{=}\ \overline{f}\texttt{;}\} \\
\text{(field qualifier)}\ P & ::= & \texttt{public readonly} \\
\text{(method qualifier)}\ Q & ::= & \texttt{public virtual}\ \mid\ \texttt{public override} \\
\text{(method def)}\ md & ::= & Q\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\,\overline{x}\texttt{)}\ \texttt{where}\ \Delta\ \{\texttt{return}\ e\texttt{;}\} \\
\text{(expression)}\ e & ::= & x\ \mid\ e\texttt{.}f\ \mid\ e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}\ \mid\ \texttt{new}\ K\texttt{(}\overline{e}\texttt{)}\ \mid\ \texttt{(}T\texttt{)}e \\
\text{(value)}\ v, w & ::= & \texttt{new}\ K\texttt{(}\overline{v}\texttt{)} \\
\text{(typing environment)}\ \Gamma & ::= & \overline{X}, \overline{x} : \overline{T}, \Delta \\
\text{(method signature)} & ::= & \texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{T} \to T \quad (\overline{X}\text{ is bound in }\Delta, \overline{T}, T) \\
\text{(substitutions)} & ::= & [\overline{T}/\overline{X}], [\overline{e}/\overline{x}]
\end{array}
$$

Fig. 6. Syntax of C$^\sharp$ minor with variance and constraints

Lemma 8 (Context construction). *Let Δ be a set of constraints. Define*

$$
\begin{aligned}
\Psi_0 &= \textstyle\bigcup_{T<:U\in\Delta} \mathrm{Dec}(T<:U) \\
\Psi_{n+1} &= \Psi_n \cup \textstyle\sum_{T<:X<:U\in\Psi_n} \mathrm{Dec}(T<:U).
\end{aligned}
$$

If the class definitions are finitary, and Δ is consistent, then Ψ_n is defined for each n and has a fix-point $\Psi = \Psi_\infty$. Then Ψ is consistent, $\Delta \vdash \Psi$ and $\Psi \succ \Delta$.

This provides a means of computing a consistent Ψ that models a set of constraints Δ, or rejecting the constraints as unsatisfiable if they are found to be inconsistent. In practice one might want to *simplify* constraints further, using techniques such as those described by Pottier [14], though constraint sets in C$^\sharp$ are unlikely to be large.

4 C$^\sharp$ minor with Variance and Generalized Constraints

In this section we formalize variance and generalized constraints as extensions of a small, but representative fragment of C$^\sharp$. After presenting the type system and operational semantics, we prove the usual *Preservation* and *Progress* theorems (Theorems 4 and 5) that establish *Type Soundness* (Theorem 6). Preservation tells us that program evaluation preserves types. Progress tells us that well-typed programs are either already fully evaluated, may be evaluated further, or are stuck, but only at the evaluation of an illegal cast (but not, say, at an undefined runtime member lookup). The fact that we have to accommodate stuck programs has nothing to do with our extensions; it is just the usual symptom of supporting runtime-checked downcasts.

We formulate our extensions for 'C$^\sharp$ minor' [9], a small, purely-functional subset of C$^\sharp$ version 2.0 [15, 6]. Its (extended) syntax, typing rules and small-step reduction semantics are presented in Figures 6–8. To aid the reader, we emphasize the essential differences to basic C$^\sharp$ minor using shading. C$^\sharp$ minor itself is based on Featherweight GJ [7] and has similar aims: it is just enough for our purposes but does not "cheat" – valid (constraint-free) programs in C$^\sharp$ minor really are valid C$^\sharp$ programs. The differences from FGJ are as follows:

Subtyping:

$$\text{(sub-incl)}\ \frac{\Delta \vdash T\texttt{<:}U}{\overline{X}, \overline{x}{:}\overline{T}, \Delta \vdash T \texttt{<:}\ U}$$

Well-formed types and constraints:

$$\frac{}{\Gamma \vdash \texttt{object}\ \mathit{ok}} \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ \mathit{ok}} \qquad \frac{\begin{array}{c}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\,\overline{X}\texttt{>} : K\ \texttt{where}\ \Delta\ \{\ldots\} \\ \Gamma \vdash \overline{T}\ \mathit{ok} \quad \Gamma \vdash [\overline{T}/\overline{X}]\Delta\end{array}}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathit{ok}}$$

$$\frac{\Delta \equiv \overline{T}\texttt{<:}\overline{U} \quad \Gamma \vdash \overline{T}, \overline{U}\ \mathit{ok}}{\Gamma \vdash \Delta\ \mathit{ok}}$$

Typing:

$$\text{(ty-var)}\ \frac{}{\Gamma, x{:}T \vdash x : T} \qquad \text{(ty-fld)}\ \frac{\Gamma \vdash e : K\ \mathit{fields}(K) = \overline{P}\ \overline{T}\ \overline{f}}{\Gamma \vdash e.f_i : T_i}$$

$$\text{(ty-cast)}\ \frac{\Gamma \vdash U\ \mathit{ok} \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{(}U\texttt{)}e : U} \quad \text{(ty-sub)}\ \frac{\Gamma \vdash e : T \quad \Gamma \vdash U\ \mathit{ok} \quad \Gamma \vdash T \texttt{<:}\ U}{\Gamma \vdash e : U}$$

$$\text{(ty-new)}\ \frac{\Gamma \vdash K\ \mathit{ok} \quad \mathit{fields}(K) = \overline{P}\ \overline{T}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash \texttt{new}\ K\texttt{(}\overline{e}\texttt{)} : K}$$

$$\text{(ty-meth)}\ \frac{\begin{array}{lll}\Gamma \vdash e : K & \mathit{mtype}(K.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{U} \to U & \\ \Gamma \vdash \overline{T}\ \mathit{ok} & \Gamma \vdash [\overline{T}/\overline{X}]\Delta & \Gamma \vdash \overline{e} : [\overline{T}/\overline{X}]\overline{U}\end{array}}{\Gamma \vdash e.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} : [\overline{T}/\overline{X}]U}$$

Method and Class Typing:

$$\text{(ok-virtual)}\ \frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\,\overline{X}\texttt{>} : K\ \texttt{where}\ \Delta_1\ \{\ldots\}\ \ \mathit{mtype}(K.m)\ \text{not defined} \\ \neg\overline{v}\overline{X} \vdash \Delta_2, \overline{T}\ \mathit{mono} \quad \overline{v}\overline{X} \vdash T\ \mathit{mono} \quad \Delta_1, \Delta_2\ \mathit{consistent} \\ \overline{X}, \overline{Y}, \Delta_1, \Delta_2 \vdash T, \overline{T}, \Delta_2\ \mathit{ok} \quad \overline{X}, \overline{Y}, \Delta_1, \Delta_2, \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T\end{array}}{\vdash \texttt{public virtual}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \texttt{where}\ \Delta_2\ \{\texttt{return}\ e\texttt{;}\}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\text{(ok-override)}\ \frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\,\overline{X}\texttt{>} : K\ \texttt{where}\ \Delta_1\ \{\ldots\} \\ \mathit{mtype}(K.m) = \texttt{<}\overline{Y}\ \texttt{where}\ \Delta_2\texttt{>}\overline{T} \to T \\ \Delta_1, \Delta_2\ \mathit{consistent} \quad \overline{X}, \overline{Y}, \Delta_1, \Delta_2, \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T\end{array}}{\vdash \texttt{public override}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\text{(ok-class)}\ \frac{\begin{array}{lll}\overline{v}\overline{X} \vdash K, \overline{T}\ \mathit{mono} & \Delta\ \mathit{consistent} & \overline{X}, \Delta \vdash K, \Delta, \overline{T}\ \mathit{ok} \\ \mathit{fields}(K) = \overline{P}\ \overline{U}\ \overline{g} & \overline{f}\ \text{and}\ \overline{g}\ \text{disjoint} & \\ \vdash \overline{md}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>} & \multicolumn{2}{l}{\mathit{kd} = \texttt{public}\ C\texttt{(}\overline{U}\ \overline{g}, \overline{T}\ \overline{f}\texttt{)}\ \texttt{base(}\overline{g}\texttt{)}\ \{\texttt{this.}\overline{f}\texttt{=}\overline{f}\texttt{;}\ \}}\end{array}}{\vdash \texttt{class}\ C\texttt{<}\overline{v}\,\overline{X}\texttt{>} : K\ \texttt{where}\ \Delta\ \{\overline{P}\ \overline{T}\ \overline{f}\texttt{;}\ \mathit{kd}\ \overline{md}\}\ \mathit{ok}}$$

Fig. 7. Typing rules for C$^\sharp$ minor with variance and constraints

Operational Semantics:
(reduction rules)

$$\text{(r-fld)}\ \frac{\mathit{fields}(K) = \overline{P}\ \overline{T}\ \overline{f}}{\texttt{new}\ K\texttt{(}\overline{v}\texttt{)}\texttt{.}f_i \rightarrow v_i} \quad \text{(r-meth)}\ \frac{\mathit{mbody}(K.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle}{\texttt{new}\ K\texttt{(}\overline{v}\texttt{)}\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{w}\texttt{)} \rightarrow [\overline{w}/\overline{x}, \texttt{new}\ K\texttt{(}\overline{v}\texttt{)}/\texttt{this}]e'}$$

$$\text{(r-cast)}\ \frac{\vdash K <: T}{\texttt{(}T\texttt{)}\texttt{new}\ K\texttt{(}\overline{v}\texttt{)} \rightarrow \texttt{new}\ K\texttt{(}\overline{v}\texttt{)}}$$

(evaluation rules)

$$\text{(c-new)}\ \frac{e \rightarrow e'}{\texttt{new}\ K\texttt{(}\overline{v}, e, \overline{e}\texttt{)} \rightarrow \texttt{new}\ K\texttt{(}\overline{v}, e', \overline{e}\texttt{)}} \quad \text{(c-fld)}\ \frac{e \rightarrow e'}{e\texttt{.}f \rightarrow e'\texttt{.}f}$$

$$\text{(c-cast)}\ \frac{e \rightarrow e'}{\texttt{(}T\texttt{)}e \rightarrow \texttt{(}T\texttt{)}e'} \quad \text{(c-meth-rcv)}\ \frac{e \rightarrow e'}{e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \rightarrow e'\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}}$$

$$\text{(c-meth-arg)}\ \frac{e \rightarrow e'}{v\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{v}, e, \overline{e}\texttt{)} \rightarrow v\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{v}, e', \overline{e}\texttt{)}}$$

Field lookup:

$$\frac{}{\mathit{fields}(\texttt{object}) = \{\}} \qquad \frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\ \overline{X}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta\ \{\ \overline{P_1}\ \overline{U_1}\ \overline{f_1}\texttt{;}\ kd\ \overline{md}\} \\ \mathit{fields}([\overline{T}/\overline{X}]K) = \overline{P_2}\ \overline{U_2}\ \overline{f_2}\end{array}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{P_2}\ \overline{U_2}\ \overline{f_2}, \overline{P_1}\ [\overline{T}/\overline{X}]\overline{U_1}\ \overline{f_1}}$$

Method lookup:

$$\frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\ \overline{X_1}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta\ \{\ldots\ \overline{md}\} \\ m \text{ not defined } \texttt{public virtual} \text{ in } \overline{md}\end{array}}{\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = \mathit{mtype}([\overline{T_1}/\overline{X_1}]K.m)}$$

$$\frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X_1}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta_1\ \{\ldots\ \overline{md}\} \\ \texttt{public virtual}\ U\ m\texttt{<}\overline{X_2}\texttt{>(}\overline{U}\ \overline{x}\texttt{)}\ \texttt{where}\ \Delta_2\ \{\texttt{return}\ e\texttt{;}\} \in \overline{md}\end{array}}{\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = [\overline{T_1}/\overline{X_1}](\texttt{<}\overline{X_2}\ \texttt{where}\ \Delta_2\texttt{>}\overline{U} \rightarrow U)}$$

Method dispatch:

$$\frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\ \overline{X_1}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta\ \{\ldots\ \overline{md}\} \\ m \text{ not defined in } \overline{md}\end{array}}{\mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \mathit{mbody}([\overline{T_1}/\overline{X_1}]K.m\texttt{<}\overline{T_2}\texttt{>})}$$

$$\frac{\begin{array}{l}\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{v}\ \overline{X_1}\texttt{>}\ \texttt{:}\ K\ \texttt{where}\ \Delta_1\ \{\ldots\ \overline{md}\} \\ Q\ U\ m\texttt{<}\overline{X_2}\texttt{>(}\overline{U}\ \overline{x}\texttt{)}\ \texttt{where}\ \Delta_2\ \{\texttt{return}\ e\texttt{;}\} \in \overline{md}\end{array}}{\mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \langle \overline{x}, [\overline{T_1}/\overline{X_1}, \overline{T_2}/\overline{X_2}]e \rangle}$$

Fig. 8. Evaluation rules and helper definitions for C$^\sharp$ minor with variance and constraints

– Instead of bounds on type parameters, we allow *subtype constraints* on types, specified at class and virtual method definitions but implicitly inherited at method overrides. In this way, a virtual method may further constrain its outer class type parameters as well as its own method type parameters.
– We include a separate rule for subsumption instead of including subtyping judgments in multiple rules.
– We fix the reduction order to be call-by-value.

Like Featherweight GJ, this language does not include object identity and encapsulated state, which arguably are defining features of the object-oriented programming paradigm, nor does it model interfaces. It does include dynamic dispatch, generic methods and classes, and runtime casts. Despite the lack of mutation, unrestricted use of variant type parameters leads to unsoundness:

```
class C<-X> { public readonly X x; public C(X x) { this.x = x; } }
// Interpret a Button as a string!
((C<string>) (new C<object>(new Button()))).x
```

For readers unfamiliar Featherweight GJ we summarize the language here.

Type variables X, types T, classes C, constructed types K, constraints $T{<:}U$, constraint lists Δ and indeed the declarative subtyping relation $\Delta \vdash T{<:}U$ are as in Section 3 and not re-defined here; `object` abbreviates `object<>`.

A **class definition** *cd* consists of a class name C with formal, variance-annotated type parameters $\overline{v}\,\overline{X}$, single base class (superclass) K, constraints Δ, constructor definition kd, typed instance fields $\overline{P}\ \overline{T}\ \overline{f}$ and methods $\overline{md}$. Method names in $\overline{md}$ must be distinct *i.e.* there is no support for overloading.

A **field qualifier** P is always `public readonly`, denoting a publicly accessible field that can be read, but not written, outside the constructor. Readonly fields behave covariantly.

A **method qualifier** Q is either `public virtual`, denoting a publicly-accessible method that can be inherited or overridden in subclasses, or `public override`, denoting a method that overrides a method of the same name and type signature in some superclass.

A **method definition** *md* consists of a method qualifier Q, a return type T, name m, formal type parameters $\overline{X}$, formal argument names $\overline{x}$ and types $\overline{T}$, constraints Δ and a body consisting of a single statement `return` e`;`. The constraint-free sugar $Q\ T\ m$`<`$\overline{X}$`>(`$\overline{T}\ \overline{x}$`)` {`return` e`;`} abbreviates a declaration with an empty `where` clause ($|\Delta| = 0$). By design, the typing rules only allow constraints to be placed on a *virtual* method definition: constraints are inherited, modulo base-class instantiation, by any overrides of this virtual method. Implicitly inheriting constraints matches C$^\sharp$'s implicit inheritance of bounds on type parameters. Note that if Δ contains a bound on a class type parameter, then it may become a general constraint between types in any overrides of this method (by virtue of base class specialization). This is why we accommodate arbitrary constraints, not just bounds, in constraint sets Δ.

A **constructor** *kd* initializes the fields of the class and its superclass.

An **expression** e can be a method parameter x, a field access $e.f$, the invocation of a virtual method at some type instantiation $e.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}$ or the creation of an object with initial field values $\texttt{new}\ K\texttt{(}\overline{e}\texttt{)}$. A **value** v is a fully-evaluated expression, and (always) has the form $\texttt{new}\ K\texttt{(}\overline{v}\texttt{)}$.

A **class table** $\mathcal{D}$ maps class names to class definitions. The distinguished class `object` is not listed in the table and is dealt with specially.

A typing environment Γ has the form $\Gamma = \overline{X}, \overline{x{:}T}, \Delta$ where free type variables in $\overline{T}$ and Δ are drawn from $\overline{X}$. We write $\cdot$ to denote the empty environment. Judgment forms are as follows. The subtype judgment $\Gamma \vdash T <: U$ extracts Δ from Γ and defers to subtype judgment of Figure 2. To do this we define the predicates $C\texttt{<}\overline{vX}\texttt{>:}\overline{K}$ of Section 3 to mean $|\overline{K}| = 0$ and $C\texttt{<}\overline{vX}\texttt{>} \equiv \texttt{object<>}$ or $|\overline{K}| = 1$ and $\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{vX}\texttt{>}\ \texttt{:}\ K_1\ \ldots$. The formation judgment $\Gamma \vdash T\ ok$ states "in typing environment Γ, the type T is well-formed with respect to the class table, type variables and constraints declared in Γ". The typing judgment $\Gamma \vdash e : T$ states that "in the context of a typing environment Γ, the expression e has type T" with type variables in e and T drawn from Γ. The method well-formedness judgment $\vdash md\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}$ states that "method definition md is valid in class $C\texttt{<}\overline{X}\texttt{>}$." The class well-formedness judgment $\vdash cd\ ok$ states that "class definition cd is valid". The judgment $e \rightarrow e$ states that "(closed) expression e reduces, in one step, to (closed) expression e'." As usual, the reduction relation is defined by both primitive *reduction* rules and contextual *evaluation* rules.

All of the judgment forms and helper definitions of Figures 7 and 8 assume a class table $\mathcal{D}$. When we wish to be more explicit, we annotate judgments and helpers with $\mathcal{D}$. We say that $\mathcal{D}$ is a *valid* class table if $\vdash^{\mathcal{D}} cd\ ok$ for each class definition cd in $\mathcal{D}$ and the class hierarchy is a tree rooted at `object` (which we could easily formalise but do not).

The operation $mtype(T.m)$, given a statically known class $T \equiv C\texttt{<}\overline{T}\texttt{>}$ and method name m, looks up the generic signature of method m, by traversing the class hierarchy from C to find its virtual definition. The operation also computes the inherited constraints of m so it cannot simply return the syntactic signature of an intervening override but must examine its virtual definition.

The operation $mbody(T.m\texttt{<}\overline{T}\texttt{>})$, given a runtime class $T \equiv C\texttt{<}\overline{U}\texttt{>}$, method name m and method instantiation $\overline{T}$, walks the class hierarchy from C to find the most specific override of the virtual method, returning its instantiated body.

Now some comments on the differences in our rules. Rule (ty-meth) imposes an additional premise: the actual, instantiated constraints of the method signature (if any) must be derivable from the constraints in the context. In turn, rules (ok-virtual) and (ok-override) add the class constraints and any declared or inherited formal method constraints to the environment, before checking the method body: the body may assume the constraints hold, thus allowing more code to type-check. Note that we may apply subsumption, including subtyping through variance, to the receiver of a method call or field lookup: for safety, the run-time type or signature of the field or method must always be a subtype of this static type. To this end, rule (ok-class) restricts field types to be monotonic in the variance of class type parameters. Because the base class must be monotonic too, this property is preserved by the types of any inherited fields (it is easy to show that monotonicity is preserved by monotonic substitution). Rule (ok-virtual) on the other hand, requires the method constraints and argument types to be anti-monotonic in the variance of the class type parameters, but the return type to be monotonic.

Our type checking rules are not algorithmic in their current form. In particular, the rules do not give a strategy for proving subtyping judgments and the type checking rules for expressions are not syntax-directed because of rule (ty-sub). As a concession to producing an algorithm, rules (ok-virtual) and (ok-override) require that the declared constraints in Δ are *consistent*. This ensures that an algorithm will only have to cope with the bodies of methods that have consistent constraints. This does not rule out any useful programs: methods with inconsistent constraints are effectively dead, since the pre-conditions for calling them can never be established. However, imposing consistency means that subtype relations can be decided by appealing to our subtyping algorithm.

Nevertheless, our proof of Type Soundness does *not* rely on the notion of *consistency*. Type soundness holds even if we omit the consistency premises.

We now outline the proof (eliding standard lemmas like *Well-formedness*, *Weakening* and *Inversion*). The class table implicit in all results is assumed to be valid.

We prove the usual type and term substitution properties that follow, but a key lemma for our system is Lemma 11, that lets us discharge proven hypothetical constraints from various judgment forms (a similar lemma appears in [10], but for equations).

Lemma 9 (Substitution Property for Lookup).

– *If* $fields(K) = \overline{P}\ \overline{T}\ \overline{f}$ *then* $fields([\overline{U}/\overline{Y}]K) = \overline{P}\ [\overline{U}/\overline{Y}]\overline{T}\ \overline{f}$.
– $mtype(K.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{T} \rightarrow T$ *implies* $mtype(([\overline{U}/\overline{Y}]K).m) = [\overline{U}/\overline{Y}](\texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{T} \rightarrow T)$.
– $mtype(K.m)$ *is undefined then* $mtype(([\overline{U}/\overline{Y}]K).m)$ *is undefined.*

Lemma 10 (Substitution for types). *Let* $\mathcal{J}$ *range over the judgment forms of subtyping* ($T{<:}U$)*, type well-formedness* (T *ok*) *and typing* ($e : T$)*:*
If $\overline{X}, \overline{Y}, \overline{x}{:}\overline{T}, \Delta \vdash \mathcal{J}$ *and* $\overline{Y} \vdash \overline{U}$ *ok then* $\overline{Y}, \overline{x}{:}[\overline{U}/\overline{X}]\overline{T}, [\overline{U}/\overline{X}]\Delta \vdash [\overline{U}/\overline{X}]\mathcal{J}$.

Proof. Straightforward induction on the derivation of $\mathcal{J}$, using Lemma 9. □

Lemma 11 (Constraint Elimination). *Let* $\mathcal{J}$ *range over the judgment forms of subtyping* ($T{<:}U$)*, type well-formedness* (T *ok*) *and typing* ($e : T$)*:*
If $\Gamma, \Delta \vdash \mathcal{J}$ *and* $\Gamma \vdash \Delta$ *then* $\Gamma \vdash \mathcal{J}$.

Proof. Induction on the derivation of $\mathcal{J}$. □

Lemma 12 (Substitution for terms). *If* $\Gamma, \overline{x}{:}\overline{T} \vdash e : T$ *and* $\Gamma \vdash \overline{v} : \overline{T}$ *then* $\Gamma \vdash [\overline{v}/\overline{x}]e : T$.

Proof. By induction on the typing derivation. □

To prove Preservation we also need the following properties of (ground) subtyping. The first two lemmas tell us that the types of members are preserved by subtyping, but only up to subtyping, since fields and method signatures behave covariantly (subtyping on method signatures may be defined in the usual contra-co fashion, treating constraints contra-variantly). The proofs of these lemmas rely on the monotonicity restrictions on base classes, fields and method signatures enforced by rules (ok-class) and (ok-virtual): these, in turn, justify appeals to Lemma 3 in the proofs.

Lemma 13 (Field Preservation). *If* $\cdot \vdash T, U$ *ok and* $\cdot \vdash T <: U$*, then* $\mathit{fields}(U) = \overline{P}\ \overline{U}\ \overline{g}$ *and* $\mathit{fields}(T) = \overline{P}\ \overline{T}\ \overline{f}$ *implies* $\cdot \vdash T_i <: U_i$ *and* $f_i = g_i$ *for all* $i \leq |\overline{g}|$.

Lemma 14 (Signature Preservation).
If $\cdot \vdash T, U$ *ok and* $\cdot \vdash T <: U$ *then* $\mathit{mtype}(U.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta_1\texttt{>}\overline{V}_1 \rightarrow V_1$ *implies* $\mathit{mtype}(T.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta_2\texttt{>}\overline{V}_2 \rightarrow V_2$ *where* $\Delta_1 \vdash \Delta_2$ *and* $\cdot \vdash \overline{V}_1 <: \overline{V}_2$ *and* $\cdot \vdash V_2 <: V_1$.

Lemma 15 (Soundness for Dispatch). *If* $\mathit{mbody}(T.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e\rangle$ *then, provided* $\cdot \vdash T, \overline{T}$ *ok and* $\mathit{mtype}(T.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{U} \rightarrow U$ *and* $\cdot \vdash [\overline{T}/\overline{X}]\Delta$*, there must be some type* V *such that* $\cdot \vdash V$ *ok,* $\cdot \vdash T <: V$ *and* $\overline{x}{:}[\overline{T}/\overline{X}]\overline{U}, \texttt{this}{:}V \vdash e : [\overline{T}/\overline{X}]U$.

Proof. By induction on the relation $\mathit{mbody}(T.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e\rangle$ using Substitution Lemmas 10 and 9 and Lemma 11. □

Theorem 4 (Preservation). *If* $\cdot \vdash e : T$ *then* $e \rightarrow e'$ *implies* $\cdot \vdash e' : T$.

Proof. By induction on the reduction relation using Lemmas 12–15. □

The proof of Progress relies on Lemma 16. The lemma guarantees the presence of a dynamically resolved field or method body, given the existence of a member of the same name in a statically known superclass.

Lemma 16 (Runtime Lookup). *If* $\cdot \vdash T, U$ *ok and* $\cdot \vdash T <: U$ *then*

- $\mathit{fields}(U) = \overline{P}\ \overline{U}\ \overline{g}$ *implies* $\mathit{fields}(T) = \overline{P}\ \overline{T}\ \overline{f}$*, for some* $\overline{T}, \overline{f}$*, with* $\cdot \vdash T_i <: U_i$ *and* $f_i = g_i$ *for all* $i \leq |\overline{g}|$.
- $\mathit{mtype}(U.m) = \texttt{<}\overline{X}\ \texttt{where}\ \Delta\texttt{>}\overline{V} \rightarrow V$ *implies* $\mathit{mbody}(T.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e\rangle$ *for some* $\overline{x}, e$ *with* $|\overline{x}| = |\overline{V}|$.

To state the Progress Theorem in the presence of casts, as for FGJ, we first characterize the implicit *evaluation contexts*, $\mathcal{E}$, defined by the evaluation rules:

$$\mathcal{E} ::= [] \mid \texttt{new}\ K\texttt{(}\overline{v}, \mathcal{E}, \overline{e}\texttt{)} \mid \mathcal{E}\texttt{.}f \mid \mathcal{E}\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \mid v\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{v}, \mathcal{E}, \overline{e}\texttt{)} \mid \texttt{(}T\texttt{)}\mathcal{E}$$

We define $\mathcal{E}[e]$ to be the obvious expression obtained by replacing the unique hole [] in $\mathcal{E}$ with e.

Theorem 5 (Progress). *If* $\cdot \vdash e : T$ *then:*

- $e = v$ *for some value* v *(e is fully evaluated), or*
- $e \rightarrow e'$ *for some* e' *(e can make progress), or*
- $e = \mathcal{E}[\texttt{(}U\texttt{)new}\,K\texttt{(}\overline{v}\texttt{)}]$*, for some evaluation context* $\mathcal{E}$*, types* U *and* K *and values* $\overline{v}$ *where* $\not\vdash K <: U$ *(e is stuck, but only at the evaluation of a failed cast).*

Proof. By (strong) induction on the typing relation, applying Lemma 16. □

Theorem 6 (Type Soundness). *Define* $e \rightarrow^\star e'$ *to be the reflexive, transitive closure of* $e \rightarrow e'$*. If* $\cdot \vdash e : T$*,* $e \rightarrow^\star e'$ *with* e' *a normal form, then either* e' *is a value with* $\cdot \vdash e' : T$*, or a stuck expression of the form* $\mathcal{E}[\texttt{(}U\texttt{)new}\,K\texttt{(}\overline{v}\texttt{)}]$ *where* $\not\vdash K <: U$*.*

Proof. An easy induction over $e \rightarrow^\star e'$ using Theorems 5 and 4. □

5 Conclusion

We have described and formalized a significant generalization of the C$^\sharp$ generics design. Generalized constraints, in particular, are useful in their own right, and easy to understand. In a sense, they are simply a lifting of a restriction imposed in both Java and C$^\sharp$: that the type on the left of a class constraint must be a class type parameter, and that the type on the left of a method constraint must be a method type parameter.

The practicality of definition-site variance is less clear, bearing in mind the refactoring of libraries that is necessary to make good use of the feature. The experience of Scala users will be valuable, as Scala adopts a very similar design for variant types.

For future work, we would like to develop an algorithm for – or prove undecidable – the extension of subtyping to infinitary inheritance. These results would transfer almost directly to variant subtyping in Viroli and Igarashi's system [8] and to wildcards in Java, which have similar inheritance and variance behaviour.

Our formalization could be extended to support interfaces, and perhaps also mutable fields in objects. Finally, we are studying the generalization of our previous work on type-preserving translations from variants of System F into C$^\sharp$, providing some handle on the expressivity of the extensions. It does not seem possible to translate Full $F_{<:}$, for which subtyping is undecidable [13]. Neither is it possible to translate Kernel $F_{<:}$. But a third variant, called $F_{<:}^{\top}$ [3], can be translated into C$^\sharp$ with variance and upper bounds.

The first author has completed a prototype implementation of variant interfaces, variant delegates, and generalized constraints. We hope to release this as a 'diff' to the shared source release of C$^\sharp$ 2.0.

References

1. P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications/European Conference on Object-Oriented Programming (OOPSLA/ECOOP'90)*, pages 161–168. ACM Press, 1990.
2. R. Cartwright and G. L. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.
3. G. Castagna and B. Pierce. Decidable bounded quantification. In *Proceedings of the Twenty-First ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon.* ACM, Jan. 1994.
4. ECMA International. ECMA Standard 335: Common Language Infrastructure, 3rd edition, June 2005. Available at `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.
5. N. G. Fruja. Type Safety of Generics for the .NET Common Language Runtime. In P. Sestoft, editor, *European Symposium on Programming*, pages 325–341. Springer-Verlag, Lecture Notes in Computer Science 3924, 2006.
6. A. Hejlsberg, S. Wiltamuth, and P. Golde. C# version 2.0 specification, 2005. See `http://msdn.microsoft.com/vcsharp/team/language/default.aspx`.
7. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
8. A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006. To appear.
9. A. Kennedy and D. Syme. Transposing F to C$^\sharp$: Expressivity of parametric polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16:707–733, 2004.
10. A. J. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, San Diego, October 2005. ACM.
11. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification, 2005. Available from `http://scala.epfl.ch/`.
12. M. Odersky and M. Zenger. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA).* ACM, 2005.
13. B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.
14. F. Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, Nov. 2001.
15. P. Sestoft and H. I. Hansen. *C# Precisely.* MIT Press, October 2004.
16. M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2005.
17. V. Trifonov and S. Smith. Subtyping constrained types. In *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, Sept. 1996.
18. M. Viroli and A. Natali. Parametric polymorphism in Java through the homogeneous translation LM: Gathering type descriptors at load-time. Technical Report DEIS-LIA-00-001, Università degli Studi di Bologna, April 2000.