Our subset of JavaScript now supports many of the core features of modern programming languages. Moreover, the type system that we have developed provides strong correctness guarantees about the program execution, enabling us to detect many programming errors before a program is actually evaluated. Unfortunately, this type system is also quite limited in its current form. The types that are expressible in our language can only describe program expressions of bounded size. For example, it is impossible to write a well-typed function in our current language that operates on linked lists of unbounded size, where the individual list nodes are implemented using objects. Many common programming idioms cannot be realized due to this limitation of our type system.

In this class, we extend our type system with interface types. An interface type can be thought of as an alias for a type expression. By making interface types recursive, we enable our type language to capture program expressions of arbitrary size. This feature greatly increases the practical expressiveness of our language.

The introduction of recursive types confronts us with the mind-boggling problem of reasoning about infinite type expressions. By exploiting the mathematical concept of coinduction, we will develop a surprisingly simple solution to this problem that leads to a natural generalization of our type inference algorithm from class 23.

**Interfaces**

One of the motivations for introducing interface types is that object types are quite cumbersome to use as is. Interface types allow us to abbreviate complex type expressions with meaningful names, which we can then use as short-hands for these expressions in type annotations.

For example, consider again our counter object class. The following interface type declaration defines a type CounterRep as an alias for the type `{var x: number}`:

```typescript
interface CounterRep {x: number};
```
This type describes the objects that we use for representing the state of counter objects. Similarly, the following interface type declaration defines a type `Counter` that stands for the type of a counter object itself:

```typescript
interface Counter {
    get: () => number;
    inc: () => undefined;
}
```

We can now use these interface types as type abbreviations in our declarations of the counter class function, constructor, and client:

```typescript
const counterClass = function(rep: CounterRep): Counter {
    return {
        get: function() { return rep.x; },
        inc: function() { rep.x = rep.x + 1; }
    }
};

const newCounter = function() {
    const rep = {x: 0};
    return counterClass(rep)
};

const counterClient = function(c: Counter) {
    c.inc();
    c.inc();
    c.inc();
};
```

The code is much easier to read than the earlier version in which the `CounterRep` and `Counter` types where expanded to their defining type expressions.

**Recursive Types**

In general, we extend the concrete syntax of our language with interface type declarations of the form

```
interface T τ; e
```

where `T` is a type name, `τ` a type expression, and `e` an expression. Type annotations in `e` may refer to the type name `T`, which expands to its defining type expression `τ`.

While type aliases are very useful, they do not increase the expressive power of our type system. However, the scope of the type name `T` in an interface declaration is not restricted to the expression `e`. It also includes the defining
expression \( \tau \) itself. That is, interface type declarations can be recursive. For example, the following recursive interface type declarations defines a type \( \text{Hungry} \) that describes functions from \text{number} to \text{Hungry}:

\[
\text{interface Hungry (x: number) => Hungry;}
\]

The following declaration defines a recursive function \( \text{yum} \) that has type \( \text{Hungry} \):

\[
\text{const yum = function yum(x: number): Hungry { return yum;}}
\]

Note that the function \( \text{yum} \) simply takes a parameter of type \text{number} and returns itself, thus satisfying the definition of the recursive type \( \text{Hungry} \). The function \( \text{yum} \) cannot be typed in our earlier type system (i.e., without recursive interface types there exists no type annotation for \( \text{yum} \) that satisfies the typing rules).

The function \( \text{yum} \) can be applied arbitrarily often to a value of type \text{number}. For example, the following sequence of function calls is well-typed:

\[
\text{yum}(1)(3)(2)
\]

Perhaps, the \( \text{yum} \) function does not seem very useful at first. However, its type and behavior effectively describe an output channel that consumes a stream of values and, e.g., writes them to a file.

We can use a similar idea to define the type of a functional counter object. The \text{inc} method of such an object returns a new counter object with the updated counter value, rather than modifying the state of the counter object on which the \text{inc} method is called. The corresponding interface type declaration is

\[
\text{interface Counter { const get: () => number; const inc: () => Counter }}
\]

Here is a possible implementation of the corresponding class and constructor functions for functional counters:

\[
\text{const counterClass = function counterClass(rep: CounterRep): Counter { return { const get: function() { return rep.x }, const inc: function() { const repl = { x: rep.x + 1 }; return counterClass(repl); }}; const newCounter = function(): Counter { }
\]
Given these declarations, the following expression evaluates to 2:

```javascript
newCounter().inc().inc().get()
```

Recursive Data Structures

Other common programming idioms that we can support using recursive interface types include implementations of recursive linked data structures such as lists and trees. For example, the following type declaration describes the type of a node in a linked list that stores numeric data values:

```typescript
interface Node {
    data: number;
    next: Node;
}
```

We can use this type to implement mutable linked list objects. The following type declaration describes the signature of the methods provided by such linked list objects:

```typescript
interface List {
    add: (x: number) => ()
    remove: () => number
    isEmpty: () => boolean
}
```

A list object provides an `add` method for inserting a new value at the front of the list, a `remove` method for removing the first element from the list, and an `isEmpty` method for checking whether the list is empty. The internal representation of a list object provides a reference to the first node of the list:

```typescript
interface ListRep {
    first: Node;
}
```

Figure 1 shows a possible implementation of the corresponding class and constructor functions. Note that we here assume that our language provides a value `null` whose type is a subtype of all object types. The `null` value allows us to implement the base cases of recursive linked data structures.

Extending the Type Language

We incorporate interface types into our language by extending type expressions with type variables `TVar` and recursive interface type expressions:

```plaintext
T ∈ TVar
τ ∈ Typ ::= bool | number | (x:τ₁) → τ₂ | {f:τ} | T | Interface T τ
```
const listClass = function(l: ListRep): List {
    return {
        add: function(x: number) {
            const n = { data: x, next: l.first };
            l.first = n;
        },
        remove: function() {
            const x = l.first.data;
            l.first = l.first.next;
            return x;
        },
        isFirst: function() {
            return l.first == null;
        }
    };
}

const newList = function(): List {
    return listClass({first: null});
};

Figure 1: Implementation of a linked list data structure

In our concrete syntax, we further consider a new type of program expressions that are interface type declarations of the form interface $T\;\tau;\;e$, as in our earlier discussion. In our abstract syntax, we represent such expressions by substituting all occurrences of $T$ in the body $e$ of the type declaration expression by the declared interface type: $e[(\text{Interface } T\;\tau)/T]$. That is, the abstract syntax of program expressions remains unchanged compared to our previous classes.

**Meaning of Recursive Types**

So far, we have used finite trees as the mathematical representation of expressions. With the introduction of recursive interface types, we interpret expressions as potentially infinite trees.

In order to develop our intuition for this representation, consider again the example of the “hungry” interface type expression:

$$\tau_H = \text{Interface } H \ (\{x : \text{number} \} \Rightarrow H)$$

We can unfold the type $\tau_H$ by taking its defining type expression

$$(x : \text{number}) \Rightarrow H$$

and then substituting all occurrences of the type variable $H$ with $\tau_H$ itself. This
yields the type expression:

\[ (x : \text{number}) \Rightarrow \tau_H = (x : \text{number}) \Rightarrow (\text{Interface } H (x : \text{number}) \Rightarrow H) \]

We can repeat this unfolding process ad infinitum, yielding an infinite type expression

\[ (x : \text{number}) \Rightarrow (x : \text{number}) \Rightarrow (x : \text{number}) \Rightarrow \ldots \]

We can draw this expression as an infinite tree:

```
  \Rightarrow
  \downarrow
  x : \text{number}
```

We provide a formal mathematical representation of recursive types as infinite trees.

**Types as Trees.** Before we move to recursive types, let us go back to the simpler case of non-recursive types. In the following, we will consider the simpler grammar of type expressions without interface types, type variables, and object types:

\[ \tau \in \text{Typ} := \text{bool} | \text{number} | (x : \tau_1) \Rightarrow \tau_2 \]

The following discussion easily extends to a type language with object types. We omit object types only for the sake of simplicity.

First, we assign a natural number to each of the syntactic constructs in this grammar in order to identify them uniquely: \text{bool} is assigned 0, \text{number} is assigned 1, the function type constructor \Rightarrow is assigned 2, the separator symbol : is assigned 3. We assume that variable names are also represented by natural numbers. Now consider the type expression

\[ \tau = (x : \text{number}) \Rightarrow (y : \text{bool}) \Rightarrow \text{bool} \]

If we assume that variable \(x\) is represented by 0 and \(y\) is represented by 1, then the type expression \(\tau\) can be drawn as the tree

```
  \Rightarrow
  \downarrow
  x : \text{number}
  \downarrow
  \Rightarrow
  x : \text{number}
  \downarrow
  y : \text{bool}
```
In order to obtain a mathematical representation of such trees, we number the children of each node consecutively. We can draw these numbers as labels on the edges of the tree. For example, the tree of the type $\tau$ now looks as follows:

Given this representation, we identify each node in the tree by the sequence of edge labels on the path from the root of the tree to the node. For example, the root node is identified by the empty sequence $\epsilon$ and the left-most node in the above tree is identified by the sequence 00. The tree representation of a type is then encoded by a function $t$ that maps the nodes in the tree, identified by their path label sequences, to their node labels.

In general, we say that a function $t : \mathbb{N}^* \rightarrow \mathbb{N}$ is a tree if its domain is prefix closed. We call a function $t : \mathbb{N}^* \rightarrow \mathbb{N}$ prefix closed if for all $p \in \mathbb{N}^*$ and $n \in \mathbb{N}$, if $np \in \text{dom}(t)$, then $p \in \text{dom}(t)$. We denote by $\text{Tree}$ the set of all trees and we call a tree $t$ finite if $\text{dom}(t)$ is finite. Observe that the set $\text{Tree}$ contains both finite and infinite trees since $\text{dom}(t)$ may also be infinite.

We formalize the encoding of a type $\tau \in \text{Typ}$ to its tree representation by the following function:

$$
treeof(\tau) : \mathbb{N}^* \rightarrow \mathbb{N}
$$

$$
treeof(\text{bool})(\epsilon) = 0
$$

$$
treeof(\text{number})(\epsilon) = 1
$$

$$
treeof((x : \tau_1) \Rightarrow \tau_2)(\epsilon) = 2
$$

$$
treeof((x : \tau_1) \Rightarrow \tau_2)(0) = 3
$$

$$
treeof((x : \tau_1) \Rightarrow \tau_2)(00) = x
$$

$$
treeof((x : \tau_1) \Rightarrow \tau_2)(01p) = treeof(\tau_1)(p)
$$

$$
treeof((x : \tau_1) \Rightarrow \tau_2)(01p) = treeof(\tau_2)(p)
$$
In particular, the tree of our running example \( \tau = (x: \text{number}) \Rightarrow \{f: \text{bool}\} \) is given by

\[
\text{treeof}(\tau) = \{\epsilon \mapsto 2, 0 \mapsto 3, 00 \mapsto 0, 01 \mapsto 1, 1 \mapsto 2, 10 \mapsto 3, 101 \mapsto 1, 100 \mapsto 0, 11 \mapsto 0\}
\]

**Recursion and Finite Types.** In the following, we identify the types \( \tau \in \text{Typ} \) with their tree encodings \( \text{treeof}(\tau) \). In our earlier discussion of grammars for the abstract syntax of expressions, we have defined the meaning of a grammar as trees that can be constructed from the grammar rules in finitely many steps. In this interpretation, types \( \tau \) are finite trees. Let us denote the set of trees that can be constructed this way as the set \( \text{Typ}_{\text{fin}} \). We can define the set \( \text{Typ}_{\text{fin}} \) formally by interpreting the grammar rules as the definition of a generator function:

\[
G_{\text{Typ}} : 2^{\text{Tree}} \to 2^{\text{Tree}}
\]

\[
G\{\text{typ}\}(T) = \{\text{bool}\} \cup \{\text{number}\} \cup \{ (x: \tau_1) \Rightarrow \tau_2 \mid \tau_1, \tau_2 \in T \}
\]

The generator function \( G_{\text{Typ}} \) takes a set of trees \( T \subseteq \text{Tree} \) and generates a new set of trees \( G_{\text{Typ}}(T) \) according to the construction rules of the grammar. You should convince yourself that we can indeed use \( G_{\text{Typ}} \) to construct all types by consecutively applying \( G_{\text{Typ}} \) to the empty set. That is, \( G_{\text{Typ}}(\emptyset) \) contains the types \( \text{bool} \) and \( \text{number} \), the set \( G_{\text{Typ}}(G_{\text{Typ}}(\emptyset)) \) contains all function types whose parameter and result types are drawn from \( G_{\text{Typ}}(\emptyset) \), and so forth. By taking the union of all the sets that we can construct using this iterative process, we obtain the least fixed point of \( G_{\text{Typ}} \), which is the set \( \text{Typ}_{\text{fin}} \):

\[
\text{Typ}_{\text{fin}} = \emptyset \cup G_{\text{Typ}}(\emptyset) \cup G_{\text{Typ}}(G_{\text{Typ}}(\emptyset)) \cup \ldots
\]

Equivalently, \( \text{Typ}_{\text{fin}} \) is the smallest set \( T \) that satisfies the closure condition \( G_{\text{Typ}}(T) \subseteq T \).

The definition of \( \text{Typ}_{\text{fin}} \) as the least fixed point of the generator function \( G_{\text{Typ}} \) is just a reformulation of the interpretation of a grammar as a structurally recursive definition of a set of trees. In general, all structurally recursive definitions have in common that they can only be used to construct finite objects. Can we construct infinite objects in a similar fashion?

**Corecursion and Infinite Types.** Instead of constructing the least fixed point of the generator function \( G_{\text{Typ}} \), we can also construct its greatest fixed point. The greatest fixed point of \( G_{\text{Typ}} \) is the largest set of trees \( T \) that satisfies the consistency condition \( T \subseteq G_{\text{Typ}}(T) \). The greatest fixed point of \( G_{\text{Typ}} \) yields the set of all finite and infinite types. Let us denote this set by \( \text{Typ}_{\text{inf}} \). Note that \( \text{Typ}_{\text{inf}} \) is a strict subset of \( \text{Tree} \). The trees in \( \text{Typ}_{\text{inf}} \) still adhere to the grammar rules of types, except that these types are now allowed to be infinite trees. In principle, we can construct the set \( \text{Typ}_{\text{inf}} \) by applying \( G_{\text{Typ}} \) iteratively to the set of all trees \( \text{Tree} \) and intersecting the result sets:

\[
\text{Typ}_{\text{inf}} = \text{Tree} \cap G_{\text{Typ}}(\text{Tree}) \cap G_{\text{Typ}}(G_{\text{Typ}}(\text{Tree})) \cap \ldots
\]

We refer to this dual interpretation of a grammar as corecursion.
Recursive Types. Let us now consider the extension of our simplified type language with recursive interface types:

\[
T \in \text{TVar} \\
\tau \in \text{Typ}_{\text{rec}} ::= \text{bool} \mid \text{number} \mid (x : \tau_1) \Rightarrow \tau_2 \mid \text{Interface } T \ \tau
\]

The types in \( \text{Typ}_{\text{rec}} \) define a subset of the types in \( \text{Typ}_{\text{inf}} \). We can make this connection explicit by extending the function \( \text{treeof} \) to \( \text{Typ}_{\text{rec}} \). However, there are certain recursive types that have no meaningful representation as trees, e.g., the type \( \text{Interface } T \ T \). We require that type variables of interface types must occur under some type constructor. More generally, we say that a type \( \tau \in \text{Typ}_{\text{rec}} \) is well-formed if \( \tau \) is closed and contains no subexpressions of the form

\[
\text{Interface } T_1 \ (\text{Interface } T_2 \ (\ldots (\text{Interface } T_n \ T_1) \ldots))
\]

For a well-formed type \( \tau \in \text{Typ}_{\text{rec}} \) we can then define:

\[
\text{treeof}(\tau) : \mathbb{N}^* \rightarrow \mathbb{N} \\
\text{treeof}(\text{bool})(\epsilon) = 0 \\
\text{treeof}(\text{number})(\epsilon) = 1 \\
\text{treeof}(\ (x : \tau_1) \Rightarrow \tau_2)(\epsilon) = 2 \\
\text{treeof}(\ (x : \tau_1) \Rightarrow \tau_2)(0) = 3 \\
\text{treeof}(\ (x : \tau_1) \Rightarrow \tau_2)(00) = x \\
\text{treeof}(\ (x : \tau_1) \Rightarrow \tau_2)(01p) = \text{treeof}(\tau_1)(p) \\
\text{treeof}(\ (x : \tau_1) \Rightarrow \tau_2)(01p) = \text{treeof}(\tau_2)(p) \\
\text{treeof}(\text{Interface } T \ \tau)(p) = \text{treeof}(\tau[(\text{Interface } T \ \tau)/T])(p)
\]

Note that the last case of the definition is well-defined due to the restrictions imposed on well-formed types.

We have seen that recursive interface types correspond to infinite types that can be constructed by corecursion using our simpler type language without recursive types. This treatment of recursive types easily extends to languages with additional type constructors such as object types.

Recursive Types and Subtyping

Now that we understand the meaning of recursive types, we are ready to study what are the implications of our observations for the typing and subtyping relations. From a mathematical point, there is very little that we need to do to generalize our definitions of the typing and subtype relations to infinite types. First, observe that the typing relation directly generalizes to recursive types. All that we need to do is replace the usages of the subtype relation for non-recursive types in the inference rules of the typing relation by a new subtype relation for recursive types. The same applies for the usages of the join operator. Thus, it remains to show how we can define the subtype relation on recursive types.
Again, the answer is surprisingly simple. All that we need to do is reinterpret our old definition of the subtype relation appropriately. Recall that the subtype relation is defined by inference rules using recursion on the structure of type expressions. Similar to our grammar rules for type expressions, we can interpret these inferences rules as generator functions on trees. The only difference is that the generator function now work on sets of pairs of trees instead of sets of trees.

For example, consider the subtyping rule for function types:

\[
\begin{align*}
C \vdash \tau_2 &<: \tau'_2, \\
C \vdash \tau'_1 &<: \tau_1, \\
C \vdash ((x : \tau_1) \Rightarrow \tau_2) &<: ((y : \tau'_1) \Rightarrow \tau'_2)
\end{align*}
\]

This rule can also be viewed as the definition of a generator function \( G_{\text{SubFun}} \) on sets of pairs of trees:

\[
G_{\text{SubFun}} : 2^{(\text{Tree} \times \text{Tree})} \rightarrow 2^{(\text{Tree} \times \text{Tree})}
\]

\[
G_{\text{SubFun}}(R) = \{ ((x : \tau_1) \Rightarrow \tau_2, (y : \tau'_1) \Rightarrow \tau'_2) | (\tau'_1, \tau_1) \in R, (\tau_2, \tau'_2) \in R \}
\]

Similarly, the rules \( \text{SubRefl} \) and \( \text{SubObj} \) define corresponding generator functions. If we view the subtype relation as a binary relation on trees \(<: \subseteq \text{Tree} \times \text{Tree}, \) then it simply corresponds to the least fixed point of the above generator functions, restricted to \( \text{Typ}^\text{fin} \). Alternatively, we can take the corecursive interpretation of the inference rules by considering the greatest fixed point of the induced generator functions. This way we obtain a more general subtyping relation that also works for infinite types.

Although, the corecursive interpretation of the old subtyping rules gives us the correct notion of subtyping, it does not immediately tell us how we can actually check the subtyping relation on recursive types. For example, consider the following two recursive types:

\[
\tau_1 = \text{Interface ResetCounter} \{
\begin{align*}
\text{const} &\text{ get : } () \Rightarrow \text{number}; \\
\text{const} &\text{ inc : } () \Rightarrow \text{ResetCounter}; \\
\text{const} &\text{ reset : } () \Rightarrow \text{ResetCounter}
\end{align*}
\}
\]

\[
\tau_2 = \text{Interface Counter} \{
\begin{align*}
\text{const} &\text{ get : } () \Rightarrow \text{number}; \\
\text{const} &\text{ inc : } () \Rightarrow \text{Counter}
\end{align*}
\}
\]

Intuitively, we expect that \( \tau_1 <: \tau_2 \) holds. Though, how do we actually infer this fact given that the subtype relation is only defined indirectly on the infinite unfoldings of \( \tau_1 \) and \( \tau_2 \)? It appears that we have to construct an infinite subtype derivation to confirm that the two types are indeed in the subtype relation. When we define the subtyping relation as a least fixed point (i.e., we restrict ourselves to finite types), this problem does not occur. The subtyping relation
is then defined by structural recursion. In turn, this allowed us to use the well-foundedness of the recursion to check the subtyping relation by reducing it to checks on smaller and smaller type expressions. That is, we checked the subtype relation by exploiting the induction principle. This technique no longer works for infinite types.

Induction and Coinduction

We can reformulate the induction principle in terms of least fixed points as follows. Suppose that we are given a base set \( X \) (such as the set all trees \( X = \text{Tree} \)), or the set of all pairs of trees \( X = \text{Tree} \times \text{Tree} \) and a generator function \( G : 2^X \rightarrow 2^X \) whose least fixed point \( \text{lfp}(G) \subseteq X \) inductively defines a subset of \( X \) (such as the set of all tree encodings of finite types \( \text{Typ}_{\text{fin}} \subseteq \text{Tree} \), or the subtype relation on such types \( < : \subseteq \text{Tree} \times \text{Tree} \)). Then the induction principle says that for all sets \( C \subseteq X \), if \( G(C) \subseteq C \), then \( \text{lfp}(G) \subseteq C \). In other words, if \( C \) describes a property that we want to prove about the elements of our inductively defined set \( \text{lfp}(G) \), then it suffices to prove that \( C \) is closed under \( G \). The proof technique is then as follows: assuming that \( x \in C \) holds for some arbitrary \( x \) (the induction hypothesis) we prove that \( G(\{x\}) \subseteq C \).

We can obtain a dual proof technique by replacing the least fixed point \( \text{lfp}(G) \) of the generator function with its greatest fixed point \( \text{gfp}(G) \). This principle is referred to as the coinduction principle. The coinduction principle says that given a set \( C \subseteq X \) if \( C \subseteq G(C) \), then \( C \subseteq \text{gfp}(G) \). This principle enables us to prove membership in the greatest fixed point, i.e., if we want to prove \( x \in \text{gfp}(G) \), we have to find a set \( C \) such that \( x \in C \) and \( C \subseteq G(C) \).

The coinduction principle allows us to prove properties of infinite objects that are defined by corecursion, such as the infinite trees represented by recursive types. Although coinduction is less well-known than induction, it has many important applications in computer science.

Checking the Subtype Relation using Coinduction

We apply the coinduction principle to our specific problem of checking whether two (possibly) infinite types \( \tau_1, \tau_2 \in \text{Typ}_{\text{inf}} \) are in the subtype relation \( \tau_1 < : \tau_2 \). Observe that this problem is equivalent to checking whether \( (\tau_1, \tau_2) \in \text{gfp}(G_<) \) where \( G_< \) is the generator function that we obtain from the inference rules of the subtype relation. The question is then: given \( \tau_1, \tau_2 \in \text{Typ}_{\text{inf}} \), how do we construct a set \( C \) of pairs of types such that \( (\tau_1, \tau_2) \in C \) and \( C \subseteq G_<.(C) \)? The idea is to construct the set \( C \) iteratively while we move the derivation of the subtype relation downwards into the subtrees of the two infinite trees represented by \( \tau_1 \) and \( \tau_2 \).

To better understand this argument, let us reconsider our concrete example of the types \( \tau_1 \) and \( \tau_2 \) that describe \( \text{ResetCounter} \), respectively, \( \text{Counter} \) objects. To prove that \( \tau_1 < : \tau_2 \) holds, we start with an empty hypothesis set \( C = \emptyset \). Since \( \tau_1 \) and \( \tau_2 \) are interface types, we first need to unfold their definitions before
we can make progress. Unfolding each of the definitions once yields the types:

\[
\tau'_1 = \{ \\
\quad \text{const } \text{get}: () \Rightarrow \text{number}; \\
\quad \text{const } \text{inc}: () \Rightarrow \tau_1; \\
\quad \text{const } \text{reset}: () \Rightarrow \tau_1 \\
\}
\]

\[
\tau'_2 = \{ \\
\quad \text{const } \text{get}: () \Rightarrow \text{number}; \\
\quad \text{const } \text{inc}: () \Rightarrow \tau_2 \\
\}
\]

The unfolding step reduces the problem of checking whether \( \tau_1 \prec \tau_2 \) holds under assumption \( C = \emptyset \) to the problem of checking whether \( \tau'_1 \prec \tau'_2 \) holds under assumption \( C = \{ \tau_1 \prec \tau_2 \} \). Since \( \tau'_1 \) and \( \tau'_2 \) are object types, we can use the subtyping rules for object types to reduce the problem further. We observe that \( \tau'_1 \) has all the fields that \( \tau'_2 \) has and the common fields have the same mutability in both types. Furthermore, since the fields are all \text{const}, it remains to show that the types of the common fields are again in the subtype relation. For field \text{get}, this means that we only need to show that \( \text{number} \prec \text{number} \) holds, and for field \text{inc} we need to show that \( \tau_1 \prec \tau_2 \) holds. The fact \( \text{number} \prec \text{number} \) immediately follows from the rule \text{SubRefl}. On the other hand, \( \tau_1 \prec \tau_2 \) is exactly the condition that we originally set out to prove. However, in the meantime we have unfolded the definitions of \( \tau_1 \) and \( \tau_2 \) and we have so far succeeded with our subtype derivation. We can therefore use the fact that \( \tau_1 \prec \tau_2 \) is now in our hypothesis set \( C \) to immediately discharge this reoccurring proof obligation. At this point, we have completed all subderivations and we conclude that \( \tau_1 \prec \tau_2 \) indeed holds.

We formalize this coinductive subtyping algorithm using the judgment form \( C \vdash \tau_1 \prec \tau_2 \). This judgment says that under coinduction hypothesis set \( C \), the type \( \tau_1 \) is a subtype of type \( \tau_2 \). Figure 3 shows the algorithmic subtyping rules from class 23, adapted to the new judgment form. Note that the only difference compared to the rules for non-recursive types is the threading of the coinduction hypothesis set \( C \). The rules in Figure 3 are supplemented by the rules in Figure 2. The rule \text{SubCoind} ties the coinductive knot by concluding \( \tau \prec \tau' \) if this fact has already been proved on the current derivation trail (signified by the membership of \( \tau \prec \tau' \) in the hypothesis set \( C \)). The rules \text{SubInterface} \_1 and \text{SubInterface} \_2 take care of unfolding the definitions of interface types on either side of the subtype relation. The given pair is then added to the hypothesis set \( C \) so that all occurrences of the same pair in the subderivations can be immediately discharged using the rule \text{SubCoind}.

**Correctness.** The coinduction principle ensures the correctness of the subtyping algorithm, i.e., if we can derive \( \emptyset \vdash \tau_1 \prec \tau_2 \), then \((\tau_1, \tau_2) \in \text{gfp}(G_{\prec})\)
\[(\tau <: \tau') \in C\]
\[C \vdash \tau <: \tau \quad \text{SubCoind}\]

\[
\begin{align*}
C \cup \{(\text{Interface } T \tau) <: \tau'[\text{Interface } T \tau]/T]\} &\vdash \tau'(\text{Interface } T \tau)/T \:< \tau' \quad \text{SubInterface}_1 \\
C \vdash (\text{Interface } T \tau) <: \tau' \\
C \cup \{\tau <: (\text{Interface } T' \tau')\} &\vdash \tau <: \tau'(\text{Interface } T' \tau'/T') \quad \text{SubInterface}_2
\end{align*}
\]

Figure 2: Subtyping rules for recursive types

\[
C \vdash \tau <: \tau \quad \text{SubRefl}
\]

\[
\begin{align*}
C \vdash \tau_2 <: \tau'_2 &\quad C \vdash \tau'_1 <: \tau_1 \\
C \vdash ((x: \tau_1) \Rightarrow \tau_2) <: ((y: \tau'_1) \Rightarrow \tau'_2) \\
\{g_1, \ldots, g_m\} &\subseteq \{f_1, \ldots, f_n\}
\end{align*}
\]

for all \(i, j\), if \(f_i = g_j\), then \(\text{mut}_i = \text{mut}'_j = \text{const}\) and \(C \vdash \tau_i <: \tau'_j\)

\[\text{or } \text{mut}_i = \text{mut}'_j = \text{var} \text{ and } \tau_i = \tau'_j\]

\[C \vdash \{\text{mut}_1 f_1: \tau_1; \ldots; \text{mut}_n f_n: \tau_n\} <: \{\text{mut}'_1 g_1: \tau'_1; \ldots; \text{mut}'_m g_m: \tau'_m\} \quad \text{SubObj}\]

Figure 3: Subtyping rules for the old type constructors. The only change compared to class 23 is the threading of the coninduction hypothesis set \(C\).
meaning that $\tau_1$ is a subtype of $\tau_2$. Proving that the algorithm always terminates is more involved. Termination follows from the fact that the trees obtained from the unfolding of recursive types are not arbitrarily complex. In fact, the trees of recursive types are regular, which means that they only comprise finitely many subtrees. Consequently, in order to check whether $\tau_1 <: \tau_2$ holds, we only need to construct a finite coinduction hypothesis set $C$. Moreover, to construct this set, it suffices to unfold $\tau_1$ and $\tau_2$ finitely often.

Computing Joins and Meets Coinductively

Similar to checking membership in the subtype relation, we can use coinduction to compute joins and meets of recursive types. To this end, we introduce the judgment forms $C \vdash \tau_1 \sqcup \tau_2 = \tau$ and $C \vdash \tau_1 \sqcap \tau_2 = \tau$ which are the coinductive versions of our join and meet functions from class 23. The new inference rules for these functions are shown in Figures 4, 5, and 6. Note that the rules in Figure 5 and 6 are again almost identical to the corresponding rules for non-recursive types given in class 23. The only difference is the threading of the coinduction hypothesis set $C$.

Figure 4: Rules for computing joins and meets of recursive interface types
Figure 5: Rules for computing joins for the old type constructors. The only change compared to class 23 is the threading of the coinduction hypothesis set $C$. The rules for propagating the non-existence of a join have been elided.
\[
\begin{align*}
\tau &\in \{\text{bool, number}\} & \text{MeetBasic} \\
C \vdash \tau \cap \tau &= \tau \\
\tau_1 \neq \tau_2 \quad \tau_1 &\in \{\text{bool, number}\} & \text{MeetBasicFail}_1 \\
C \vdash \tau_1 \cap \tau_2 = \bot \\
\tau_1 \neq \tau_2 \quad \tau_2 &\in \{\text{bool, number}\} & \text{MeetBasicFail}_2 \\
C \vdash \tau_1 \cap \tau_2 = \bot \\
C \vdash \{\} \cap \{\	ext{mut}_g \ g : \tau_g\} \equiv \{\	ext{mut}_g \ g : \tau_g\} & \text{MeetObjEmp} \\
C \vdash \{\text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g; \text{mut}_g \ g' : \tau_g'\} = \{\text{mut}_k \ k : \tau_k\} \quad C \vdash \tau_1 \cap \tau_2 = \tau_h & \quad \tau = \{\text{const} \ h : \tau_h; \text{mut}_k \ k : \tau_k\} \\
C \vdash \{\text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g; \text{mut}_g \ g' : \tau_g'\} = \{\text{mut}_k \ k : \tau_k\} & \text{MeetObjC} \\
C \vdash \{\text{var} \ h : \tau_h; \text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g; \text{var} \ h : \tau_h; \text{mut}_g \ g' : \tau_g'\} = \tau & \text{MeetObjV}_1 \\
\tau_1 &\neq \tau_2 & \text{MeetObjV}_2 \\
C \vdash \{\text{mut}_1 \ h : \tau_1; \text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g; \text{mut}_2 \ h : \tau_2; \text{mut}_g \ g' : \tau_g'\} = \bot & \text{MeetObjNe} \\
C \vdash \{\text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g\} = \{\text{mut}_k \ k : \tau_k\} & \text{MeetObjNo} \\
\quad h \notin \{\emptyset\} & \quad \tau = \{\text{mut}_1 \ h : \tau_1; \text{mut}_k \ k : \tau_k\} \\
C \vdash \{\text{mut}_1 \ h : \tau_1; \text{mut}_f \ f : \tau_f\} \cap \{\text{mut}_g \ g : \tau_g\} = \tau & \text{MeetObjNo} \\
\quad C \vdash \tau_1 \cup \tau_1' = \tau_1'' & \quad C \vdash \tau_2 \cap \tau_2' = \tau_2'' & \quad \tau'' = (x : \tau_1'') \Rightarrow \tau_2'' & \text{MeetFun} \\
\quad C \vdash (x : \tau_1) \Rightarrow \tau_2 & \quad C \vdash (y : \tau_1') \Rightarrow \tau_2' & \text{MeetFun} \\
\end{align*}
\]

Figure 6: Rules for computing meets for the old type constructors. The only change compared to class 23 is the threading of the conduction hypothesis set \(C\). The rules for propagating the non-existence of a meet have been elided.