# A Design for Type-Directed Programming in Java*
# (Extended Version)

Stephanie Weirich  Liang Huang
Department of Computer and Information Science
University of Pennsylvania
{sweirich,lhuang3}@cis.upenn.edu

University of Pennsylvania Technical Report, MS-CIS-04-11
October 2004

**Abstract**

*Type-directed programming* is an important and widely used paradigm in the design of software. With this form of programming, a program may analyze type information to determine its behavior. By analyzing the structure of data, many operations, such as serialization, cloning, structural equality, and iterators, may be defined once, for all types of data. The benefit of type-directed programming is that as software evolves, operations need not be updated—they will automatically adapt to new data forms. Otherwise, each of these operations must be individually redefined for each type of data, forcing programmers to revisit the same program logic many times during a program's lifetime.

The support for type-directed programming in the Java language includes the `instanceof` operator and the Java Reflection API. These mechanisms allow Java programs to depend on the run-time classes of objects. However, these mechanisms are difficult to use correctly and require needless casting. They also do not integrate well with generics.

In this paper, we describe the design of several expressive new mechanisms for type-directed programming in Java. Our new mechanisms are based on an extension of Java with first-class types (such as NextGen), so they naturally include support for generics. Because these new mechanisms analyze first-class type information directly, instead of examining the run-time class of objects, they can provide strong guarantees about program correctness. Furthermore, our new mechanisms are based on pattern matching, so they naturally and succinctly express many type-directed algorithms.

## 1 Introduction

> *In Common Lisp I have often wanted to iterate through the fields of a struct—to comb out references to a deleted object, for example, or find fields that are uninitialized. I know the structs are just vectors underneath. And yet I can't write a general purpose function that I can call on any struct. I can only access the fields by name, because that's what a struct is supposed to mean.*
>
> *– Paul Graham, "Being Popular"*

The design and structuring of software is a difficult task. Good software engineering requires code that is concise, manageable, reusable and easy to modify. Consequently, modern statically-typed programming languages include abstraction mechanisms such as subtype and parametric polymorphism (the latter is also called generics) to allow programmers to decompose complicated software in useful ways. While these abstraction mechanisms are useful, they do not cover all situations. They do not apply to operations that

*This work is supported by NSF grant CCF-0347289 CAREER:Type-Directed Programming in Object Oriented Languages.

are most naturally defined by the structure of data. These operations require a different set of abstraction mechanisms called *type-directed programming*.

With type-directed programming, the program analyzes type information to determine its behavior. That way, if the arguments to type-directed operations change structure, the operations adapt automatically. Without this mechanism, each of these operations must be individually defined and updated for each type of data, forcing programmers to revisit the same program logic at many times during a program's life cycle. This redundancy increases the chance of error and reduces program maintainability. It makes changing data representations unattractive to programmers because many lines of code must be modified.

A common use of type-directed programming is for serialization. Serialization converts any data object into an appropriate form for display, network transmission, replication (for fault tolerance), or persistent storage. Because serialization is such an important part of modern software systems, yet is difficult to define and maintain without type-directed programming, some languages include primitive mechanisms for it. However, this choice significantly reduces flexibility, preventing programmers from specializing serialization for their particular uses.

So that programmers can define their own version of routines like serialization, the Java programming language [17] includes mechanisms for type-directed programming, such as run-time type identification (with the keyword `instanceof`) and the Reflection API [18]. Figure 1 demonstrates how to implement serialization for cyclic data structures in Java. Reflection provides the method `getClass` to retrieve metadata describing the structure of the run-time class of any object. This metadata supports operations for determining the fields and methods of the run-time class.

The class `Pickle` contains the method `pickle` that converts any object to a string of characters by examining its type structure. So that it may serialize recursive data structures, this operation uses a hash table to record objects previously serialized. For objects that have not been previously serialized, it first determines if the object's class represents one of the primitive types (such as `Integer` or `Boolean`). If so, it uses one of the primitive operations for converting the object to a string. Otherwise, `pickle` recursively serializes each field of the object.

The benefit of implementing serialization in this manner is that it is independent of the class structure of the application. Without this mechanism, each class must implement its own serialization routine. This scattering of program logic throughout the classes means that, as the application is updated, the serialization methods must be continually updated in many disparate places. Even if we do not mind this commingling of concerns, defining and maintaining these distributed methods is tedious and error-prone, especially if the code maintainers are not the original authors. Type-directed programming allows the programmer to define operations in one location (or package) without modification of the rest of the program and without dependence on the specific classes found in other packages. Doing so means that the system may be divided into more coherent semantic units because the new operations do not need to be interspersed into existing modules. It also means that as the existing modules and classes change, the type-directed operations are still valid and do not need to be updated.

While serialization is the most widely cited example, type-directed programming can play a critical role in the development of many other parts of software systems. Many basic operations are most naturally defined over the structure of type information. Besides serialization, cloning (making identical, deep copies of data), structural equality, and iteration (applying an operation to each data element in a collection) may be defined by type structure. Some languages define the most common of these operations natively, speeding software development in some cases, but providing little benefit outside the narrow scope of that predefined functionality.

Type-directed programming is also important at the boundaries of software components. Extensible systems can use type information to ensure the stability of the system. They can check that newly loaded code satisfies the requirements of the running system and provides the necessary interfaces before accepting a dynamic update [21]. For example, the Common Object Model (COM) [12], treats objects abstractly and provides access to clients through one or more interfaces. All objects must implement the interface `IUnknown`, which provides the function `QueryInterface` for clients to call at runtime to determine whether the object implements a particular interface.

```
class Pickle {
  // hash table for cycle detection
  protected HashMap hashMap;
  public String pickle( Object obj ) {
    if (obj == null) return "null";
    // Check to see if we've seen obj before.
    // If not, store a unique id for this object.
    if ( hashMap.containsKey( obj ) )
        return (String)hashMap.get( obj );
    String id =
      "#" + Integer.toString( hashMap.size() + 1 );
    hashMap.put(obj,id);
    // Switch on the class of the object
    Class objClass = obj.getClass();
    if ( obj instanceof Integer ) {
        Integer i = (Integer) obj.intValue();
        return Integer.toString( i );
    } else if ( obj instanceof Boolean ) {
        Boolean b = (Boolean) obj.booleanValue();
        return Boolean.toString( b );
    } else if ... {  // Cases for other base types
    } else if ( objClass.isArray() ) {
        // Case if obj is an array
        ...
    } else try {
        // If obj is not a primitive type or array,
        // then determine all fields of the class
        // and recursively pickle each field of
        // the object, separated by commas.
        String result = "[" +  id + ":"
          + objClass.getName() + " ";
        Field[] f = objClass.getDeclaredFields();
        for ( int i=0; i<f.length; i++ ) {
            f[i].setAccessible(true);
            result += f[i].getName() + "=";
            result += pickle( f[i].get( obj ) );
            if (i < (f.length - 1)) result += ",";
        }
        return result + "]" ;
   } catch ( IllegalAccessException e )
        { return "Impossible"; }
  }
  // Constructor---creates an empty hashtable
  Pickle() {  this.hashMap = new HashMap();  }
}
```

Figure 1: Type-directed Serialization in Java

Furthermore, when interfaces are known during development, type-directed proxies may be used to adapt the interface of a component to a particular situation. For example, if an application always calls each method of a particular class with the same first argument, type-directed programming can define a wrapper for that class that automatically provides that argument [38]. Also, such proxies can be used to log, trace, profile or debug function calls to all of the methods of a specific component [20].

Finally, type-directed programming is also useful at the boundary between software and user. It allows functionality to be automatically reflected to the user as it is added to a system. For example, with JavaBeans [26], a system may examine the interface of a new component to directly provide user-interface control of the component in the form of check boxes, selection lists, buttons, etc.

**Problems with current mechanisms in Java** However, although the Java mechanisms for type directed programming promote program modularity—the serialization routine in Figure 1 may be applied to any object—serialization also demonstrates the flaws of `instanceof` and the Reflection API when compared to type-directed programming in other languages.

For example, using `instanceof` or reflection in Java almost always requires run-time type casting, leading to redundant checks and a potential for dynamic failure. In this example, when `obj` is an `Integer`, it must be cast to the `Integer` class before it may be converted to a string. This cast checks that the run-time class of `obj` is `Integer` even though `instanceof` determines that same fact. Furthermore, in the case that `obj` is not one of the classes representing primitive types, an exception handler for the `IllegalAccessException` must be installed. This exception could be raised by each field access. However, because the only accessed fields are those provided by `getDeclaredFields`, this exception will never be raised. When reflection is used correctly, the run-time casts are redundant. However, because reflection could be used incorrectly, the programmer must consider the situation when the run-time check fails, and must write code to handle exceptions such as `ClassCastException` or `IllegalAccessException`. The fact that these run-time casts must be included in correct code is a symptom of the fact that reflection is a relatively low-level mechanism for defining type-directed operations.

Furthermore, reflection breaks user-defined abstractions in Java. The method `getDeclaredFields` produces a data structure that contains all fields of the object, including those declared to be `protected` or `private`. Therefore, programmers cannot rely on private fields to hide information or enforce program modularity. The call `setAccessible(true)` prevents `IllegalAccessException` from being raised when the `private` and `protected` fields are accessed. To prevent access to private fields, the entire program may be run with a security manager that causes the `setAccessible` command to fail. However, such coarse control falls short of the programmable access control that is provided in other domains.

## 1.1 New mechanisms for Java

In this paper, we propose new mechanisms for type-directed programming in Java that may be used instead of `instanceof` or Java Reflection. These new mechanisms are based on an extension of Java with first-class genericity—one in which the types that instantiate the parameters to generic methods and classes are available at run time. While the current implementation of generics in Java (based on GJ [4]) erases such types before the program is run, extensions such as NextGen [7] provide this type information at run-time. Our new mechanisms analyze this first-class type information directly, instead of examining the run-time class of objects.

There are several advantages to analyzing first-class types instead of the run-time classes of objects.

- Our new mechanisms can provide stronger guarantees of correctness. Just as the introduction of generics allowed some casts to be eliminated from Java programs, this mechanism also can remove potential failure points.

- Our new mechanisms are easier to use. The mechanisms that we propose provide sophisticated type matching capabilities, giving the users a convenient way to program with type information. In particular, these mechanisms can more closely encode the structure of type-directed algorithms.

- Our new mechanisms integrate well with generics. Because of the type erasure implementation of generics, the current implementation of Java Reflection and `instanceof` do not provide accurate information about generic classes and methods. While it is possible to extend `instanceof` and Java Reflection [37] to generics, we think that our mechanisms are a more natural integration.
- Our new mechanisms are expressive in terms of protecting abstraction. Reflection and `instanceof` analyze the most specific type of an object. However, run-time type information that describes an object's type can be any supertype of the actual type. Packages that do not want the complete structure of their objects to be determined through analysis can provide abbreviate versions of the objects' types to type-directed operations.

The structure of this paper is as follows. In the next section, we informally describe mechanisms for analyzing the name and the structure of type parameters. In that section, we also show the expressiveness of our new mechanisms by describing how to implement some of the algorithms that previously required `instanceof` and Java Reflection. In Section 3 we formalize the semantics of these new mechanisms in a Featherweight Java-like calculus [22]. The main result of this paper is that we show that these new mechanisms are type-safe additions to this core calculus. Finally, we discuss related work and possible future extensions of our mechanisms.

# 2 Analyzing type parameters

The basic design of our new mechanisms is similar to the `typecase` operator found in intensional polymorphism [19, 16, 13, 36]. In this section, we describe new operators that analyze the type parameters of generic methods and classes instead of the run-time classes of objects.

We can roughly divide the mechanisms into two categories: those that determine the name of the run-time type (analogous to `instanceof`) and those that determine its structure (analogous to reflective mechanisms). Both sorts of mechanisms are necessary: recall that the implementation of serialization required both `instanceof` and reflection.

## 2.1 Nominal Analysis

We begin with name-based (also called nominal) analysis of run-time type information. Consider a new expression form for Java called `ifsubtypeof`. This expression form is a conditional—it chooses one of two branches based on whether a type variable is a subtype of a specific type at run-time. If the condition holds, the type checker can perform *type refinement*—the types of variables mentioning the analyzed type parameter change in the branch, eliminating redundant type casts [13]. For example, if `x` has type `T`, then in the case below, we know that `T` is a subtype of `Integer` and that `x` does not need to be cast to `Integer` before being used.

```
T x;
ifsubtypeof(T, Integer) {
   // T=Integer in this branch so x has type Integer
}
```

Furthermore, type variables create equations between types. If we determine the run-time identity of a single type variable, we may discover the class of many objects.

```
List<T> x;
ifsubtypeof(T, Integer) {
  // here we know T is a subtype of Integer,
  // so all of the elts of the list x are Integers.
}
```

By analyzing type parameters we remove potential failure points from the program. Otherwise, when examining the run-time classes of references to objects directly, their types can change unexpectedly, due to aliasing. Therefore, we cannot statically eliminate casts such as:

```
if (x.field instanceof Integer)
  writeInt ((Integer)x.field);
```

because we have no guarantee that the run-time type of `x.field` remains constant. The example code below demonstrates such a failure. Even though we determine that the class of `a.field` is `Boolean` before we use it, a method call can change that field to be an object of some incompatible class, such as `Integer`.

```
class A { Object field; }
class Example {
  A a;
  Example(A a) { this.a = a; }
  public void example() {
      if (a.field instanceof Boolean) {
          // Method call changes a.field
          f();
          // This cast fails unexpectedly
          Boolean b = (Boolean)a.field;
      }
  }
  public void f() {
     a.field = new Integer(3);
  }
}
```

Furthermore, because of concurrency, we do not need an explicit call to the method `f` to cause the type cast to fail. A concurrent thread could also change `a.field` at exactly the wrong time. In contrast, by analyzing type parameters, we remove this possible failure. Consider the analogous code:

```
class A<T> { T field; }
class Example<T> {
  T a;
  Example(A<T> a) { this.a = a }
  public void example() {
     ifsubtypeof (T, Boolean) {
        f();
        // No cast and no failure point
        Boolean b = a.field;
     }
  }
  public void f() {
     ifsubtypeof  (T, Integer) {
        // Only change field if it is Integer
        a.field = new Integer(3);
     }
  }
}
```

In this code, when the `Example` class is created, the parameter `T` must be instantiated. If it is instantiated with `Boolean`, then all other aliases must also think that it is a `Boolean` and so can only update `field` with a compatible class. Otherwise, if it is instantiated with `Object`, other classes could change `field` to any class, but the `ifsubtypeof` expression would not return true.

One limitation with `ifsubtypeof` is with parameterized classes. What if we wished to determine whether the type `T` was a list, without knowing what elements are contained in the list? We cannot use `ifsubtypeof` because we must compare `T` against `List<U>` where `U` is some type. Note that using `List<Object>` is not sufficient because of invariance of type parameters: `List<Integer>` is not a subtype of `List<Object>`.

We can make nominal analysis more powerful with pattern matching. The expression form, called `typematch`, generalizes `ifsubtypeof`. This expression matches an argument type `T` against a number of type patterns—in other words, types that contain unbound variables. Like `ifsubtypeof` the type checker can refine the static type information to correspond to the branch of the expression.

```
X x;
typematch X with
   Integer:  // Here x is an integer
   List<Y>:  // Here x is a list of Ys
             //   and we can analyze Y further.
   default:  // Here we know nothing about x
```

If a pattern does not contain any free type variables, then `typematch` behaves the same as `ifsubtypeof`. We can implement `ifsubtypeof (T,U)` $e_1$ $e_2$ with:

```
typematch T with
   U :  e1  // true branch
   default : e2  // false branch
```

## 2.2 Structural Analysis

The expressions `ifsubtypeof` and `typematch` corresponded to (and generalized) operations such as `instanceof` that are useful when we know that the class could be one of a finite number. But what if we know nothing about the class?

Being able to determine the structure of classes is also important to type-directed operations. The Java Reflection API provides this sort of capability for finding out information about the run-time class of objects. This information is used for a number of purposes, including:

- for generic algorithms such as serialization and visitors that iterate over all fields of an object.
- for interfacing with newly loaded objects, by determining the general interface that an object satisfies.
- for reflecting the functionality of an object to a user-interface—such as creating "properties" for the fields that exist in an object.
- for testing, by finding all zero-argument methods whose names start with "test" and invoking them.

Our goal is to allow programs to discover the structure of first-class types. This is an ambitious goal. The Java Tutorial [18] says that Java Reflection may be used to:

- *Determine the class of an object.*
- *Get information about a class's modifiers, fields, methods, constructors, and superclasses.*
- *Find out what constants and method declarations belong to an interface.*
- *Create an instance of a class whose name is not known until runtime.*
- *Get and set the value of an object's field, even if the field name is unknown to your program until runtime.*
- *Invoke a method on an object, even if the method is not known until runtime.*

- *Create a new array, whose size and component type are not known until runtime, and then modify the array's components.*

However, for one of the operations, we already have the capability in NextGen. We can create an instances of a statically unknown class by using a type parameter. Other operations, such as determining the name of a class or the number of methods or fields are rather trivial to add to the language. The semantics of the operations below are fairly straightforward.

- `getClassName<T>` Returns the name of the class as a string.
- `getFieldName<T,f>` Returns the name of the field `f` in class `T` as a string. (In this operation, `f` is an accessor variable, described below.)
- `getMethodName<T,m>` Returns the name of the method `m` in class `T` as a string.
- `numFields<T>` Returns the number of fields as an integer.
- `numMethods<T>` Returns the number of methods as an integer.

What is more difficult is providing a mechanism for iterating over the structure of the class, including its fields and methods. An open question that arises in this context is how to simply and soundly examine the names and types of the fields and methods declared for objects. (A similar question has already arisen for record and variant types in implementations of type-directed programming in functional languages.) The problem is that the structure of these types takes many steps to fully determine. How many fields and methods are there? What are their names and types?

Some systems have incorporated ad hoc solutions to this problem with respect to records and variants. For example, Haskell type classes [40] require help from the user—they cannot automatically generate operations for variant and record types. Generic Haskell [10] converts variant and record types into an internal representation to define basic operations, leading to a mismatch between the definition of the type-directed operation and the types at which it is used. In Figure 1 we saw that Java Reflection uses accessors such as `getFields` and refers to method and field names as strings, but cannot statically guarantee the correctness of accessing fields or invoking methods.

**Iterating over the fields of a class** Our approach to the problem of safely discovering and accessing the fields and methods of object types is to add new expression forms for that purpose. For example, the following form iterates over the fields in a class T, binding the type parameter `X` to the type of each field and an *accessor variable* (a new form of variable) to the name of each field. We might use `forfield` as follows:

```
T obj;
// iterate over all of the fields of T
forfield (X f in T) {
    // bind type parameter X and accessor
    // variable f. Then refine the type
    // T so that it contains one field "f"
    // of type X
   X fieldVal = obj.f;
    // Can analyze X like any other type
   print<X>(fieldVal);
}
```

This new expression form is not as simple as it appears. In the body of `forfield`, the identity of the type `T` should be refined to be a type that includes a field called `f` of type `X`. However, the new type of `obj` (containing a single field `f`) probably does not exist. Because Java's type system requires that objects may only be assigned types that are the names of pre-defined classes, there most likely will not be a class

```
class Pickle {
  // hash table for cycle detection
  protected HashMap hashMap;
  public String <T> pickle( T obj ) {
    if (obj == null) return "null";
    // Check to see if we've seen obj before.
    // If not, store a unique id for this object.
    if ( hashMap.containsKey( obj ) )
        return (String)hashMap.get( obj );
    String id =
      "#" + Integer.toString( hashMap.size() + 1 );
    hashMap.put(obj,id);
    // Switch on the class of the object
    typematch T with
      Integer: return Integer.toString(obj);
      Boolean: return Boolean.toString(obj);
      ... :  // Cases for other base types
      X[] :  // Case for arrays
             // X is the type of elts in the array
            { ... }
      default: {
       String result = "[" + id + ":"
                     + getName<T> + " ";
       Int i=0;
       forfield ( X f in T ) {
          result += getFieldName<T,f>;
          result += pickle<X>(obj.f);
          if (i < numFields<T> ) result += ",";
          i++;
       }
       return result + "]" ;
    }
  }
  Pickle() {  this.hashMap = new HashMap();  }
}
```

Figure 2: Pickling with structural type analysis

with the right structure that we can refine `T` to. Therefore, as described in the next section, we add a very limited form of *structural* object types to Java. This addition is not surprising given that we are verifying the *structural* analysis of object types.

Using these mechanisms, we can rewrite the Pickling example as shown in Figure 2. In this example, we use `typematch` to determine whether `T` is a base type or an array type. If it is neither, then `forfield` iterates over the fields in the object, calling the serialization routine recursively.

**Pattern matching fields and methods** A generalization of `forfield` is to allow the type variable that represents the type of the field to be a type pattern. This pattern may be a literal type, in which case, the body executes for each field with that type:

```
T obj = ...;
forfield (Integer f in T) {
   // Increment all integer-valued fields in obj.
  obj.f = obj.f + 1;
}
```

or the pattern may be arbitrarily more complicated. For example, it may select all fields of the class that are arrays (no matter what type of elements that they contain) or all static fields in a class.

The `formethod` expression iterates over the methods found in a class. Type patterns are very important for this iteration. For example, suppose we would like to pick out all methods in a class that take no arguments, return `void`, and whose name starts with "test". We may do so with the following code:

```
static <T> void runtests (T x) {
  formethod( void m() in T ) {
     if ( getMethodName<m>.startsWith("test") ) {
        x.m();
     }
  }
}
```

Another way to test methods is to apply them to random inputs. Suppose we wish to test all single argument methods of a class. The problem is that if we do not know what the class of that argument is, how do we generate a random instance of it to test the method with? However, if those classes have static methods for generating random instances, we can use `formethod` to find such methods. (This example is inspired by QuickCheck, a package for testing Haskell programs [9].)

```
static <T> void runtests (T obj) {
  formethod( void m(X) in T ) {
     formethod (static X n() in X ) {
       if ( getMethodName<n>.startsWith("random") ) {
         obj.m( X.n() );
       }
     }
  }
}
```

A difference between iterating over fields and iterating over methods is that because of method type parameters and multi-argument methods, it is impossible to write a pattern general enough to match every method in a class. The pattern must specify the number of type and term parameters of the method. However, despite this limitation, the above examples show that method iteration is a useful tool for type-directed programming.

# 3 Semantics

To more fully describe the semantics of our new type-analysis operators and to provide some assurance that they are sound within the context of the Java programming language, we next formalize a small Java-like language extended with these constructs. Below, we describe the core language, and then in the next two sections extend it with nominal and structural analysis.

Like Featherweight Generic Java (FGJ) [22] this language is a functional core of an object-oriented language with nominal subtyping. This language includes only top-level class definitions, object instantiation, field access, method invocation, and type casts. We omit many of the features of Java that are orthogonal to our study, such as mutation, concurrency, exceptions, and interfaces.

| | | | |
|---|---|---|---|
| Class names | $\texttt{C},\texttt{D}$ | | |
| Expression variables | $\texttt{x},\texttt{y}$ | | |
| Type variables | $\texttt{X},\texttt{Y},\texttt{Z}$ | | |
| Field names | $\texttt{f},\texttt{g}$ | | |
| x Method names | $\texttt{m},\texttt{n}$ | | |
| Types | $\texttt{S},\texttt{T},\texttt{U}$ | $::=$ | $\texttt{X} \mid \texttt{N}$ |
| Non-variable types | $\texttt{N},\texttt{P},\texttt{Q}$ | $::=$ | $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ |
| Class declarations | $\texttt{CL}$ | $::=$ | $\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N\{ } \overline{\texttt{T}}\ \overline{\texttt{f}} = \overline{\texttt{v}};\ \overline{\texttt{M}}\ \texttt{\}}$ |
| Method declarations | $\texttt{M}$ | $::=$ | $\texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> T m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}}$ |
| Method types | $\texttt{MT}$ | $::=$ | $\texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>(}\overline{\texttt{T}}\texttt{)}\rightarrow\texttt{T}$ |
| Expressions | $\texttt{e}$ | $::=$ | $\texttt{x} \mid \texttt{e.f} \mid \texttt{e.m <T>(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new T(}\overline{\texttt{e}}\texttt{)} \mid \texttt{(T)e}$ |
| Values | $\texttt{v}$ | $::=$ | $\texttt{new C(}\overline{\texttt{v}}\texttt{)}$ |
| Type contexts | $\Delta$ | $::=$ | $\varnothing \mid \Delta,\ \texttt{X} <: \texttt{T}$ |
| Term contexts | $\Gamma$ | $::=$ | $\varnothing \mid \Gamma,\ \texttt{x:T}$ |

Figure 3: Core Syntax

Like NextGen [7], this core language has a type-passing semantics. We allow type parameters in places that Generic Java does not. For example, type parameters may be used in the arguments of run-time type casts.

Furthermore, we allow type parameters to be used for object instantiation. To support abstract object creation in this model, all classes must declare instance initializers for all fields. In a `new X(`$\overline{\texttt{e}}$`)` expression, we do not know statically how many arguments to supply to the constructor. However, if each class has default values for each field, if not enough values are supplied with the new expression, then the default values may be used for the field. To simplify the semantics of the language we require that the initializers be syntactic values in a class declaration. We could relax this restriction by defining evaluation of class declarations in the class table.

The abstract syntax of the language is shown in Figure 3. The conventions for meta-variables are also listed in this table. Like FGJ, we greatly abuse the sequence notation, for example using $\overline{\texttt{T}}\ \overline{\texttt{f}}$ to refer to $\texttt{T}_0\ \texttt{f}_1,\ \ldots,\ \texttt{T}_0\ \texttt{f}_n$. The notation $|\overline{\texttt{T}}|$ is the length of the sequence. Sequences of names (such as for types, variables, fields and methods) are required to contain no duplicates. Additionally, `this` should not be the name of a field or a variable.

The semantics of this core calculus is very similar to that of FGJ. We include the rules and auxiliary functions for typing in Figures 4, 5, 6, and 7. To make our model closer to Java, we choose to give a small-step call-by-value semantics instead general reduction rules.

There are a few notable differences between this calculus and FGJ. In preparation for our extensions to the calculus, we define the upper bound of `T` in $\Delta$, written $bound_\Delta(\texttt{T})$, recursively. The typing rules that differ from FGJ include T-Class, where we check that the initializers for the fields are well formed and T-New where, because of the presence of field initializers, fewer arguments than fields may be supplied to a `new` expression.

## 3.1 Nominal analysis

The expression form `typematch T with` $\overline{\texttt{T}} : \overline{\texttt{e}}$ `default : e` allows programmers to pattern match the names of run-time type information. The semantics related to this expression are in Figure 8. The dynamic semantics of this expression form relies on the auxiliary function $matches(\texttt{T},\texttt{U})$. When this function is defined, `T` can

$$\frac{\mathit{bound}_\Delta(\Delta(\mathtt{X})) = \mathtt{N}}{\mathit{bound}_\Delta(\mathtt{X}) = \mathtt{N}}\ \text{[B-Var]}$$

$$\mathit{bound}_\Delta(\mathtt{N}) = \mathtt{N}\ \text{[B-NonVar]}$$

$$\frac{}{\mathit{fields}(\mathtt{Object}) = \bullet}\ \text{[F-Object]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{}\overline{\mathtt{T}'}\ \overline{\mathtt{f}'}\ \mathtt{=}\overline{\mathtt{v}'}\mathtt{;}\overline{\mathtt{M}}\mathtt{\}} \qquad \mathit{fields}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}) = \ \overline{\mathtt{T}''}\ \overline{\mathtt{f}''}}{\mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{T}''}\ \overline{\mathtt{f}''}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}](\overline{\mathtt{T}'}\ \overline{\mathtt{f}'})}\ \text{[F-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{}\overline{\mathtt{T}'}\ \overline{\mathtt{f}}\ \mathtt{=}\overline{\mathtt{v}}\mathtt{;}\overline{\mathtt{M}}\mathtt{\}}}{\mathit{fieldval}(\mathtt{f}_i, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{v}_i}\ \text{[FV-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{}\overline{\mathtt{T}'}\ \overline{\mathtt{f}}\ \mathtt{=}\overline{\mathtt{v}}\mathtt{;}\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{f} \notin \overline{\mathtt{f}}}{\mathit{fieldval}(\mathtt{f}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{fieldval}(\mathtt{f}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})}\ \text{[FV-Super]}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{}\overline{\mathtt{S}}\ \ \overline{\mathtt{f}}\ \mathtt{=}\overline{\mathtt{v}}\mathtt{;}\ \overline{\mathtt{M}}\ \ \mathtt{\}} \\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}} \end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}](\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}}\ \to \mathtt{U})}\ \text{[MT-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{}\overline{\mathtt{S}}\ \ \overline{\mathtt{f}}\ \mathtt{=}\overline{\mathtt{v}}\mathtt{;}\ \overline{\mathtt{M}}\ \ \mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})}\ \text{[MT-Super]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{...;}\ \overline{\mathtt{M}}\ \ \mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{S}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}, \overline{\mathtt{Y}} \mapsto \overline{\mathtt{S}}]\mathtt{e})}\ \text{[MB-Class]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\{...;}\ \overline{\mathtt{M}}\ \ \mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{S}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{mbody}(\mathtt{m<}\overline{\mathtt{S}}\mathtt{>}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})}\ \text{[MB-Super]}$$

Figure 4: Auxillary operations

match the pattern `U`. In the first computation rule for `typematch`, the argument of the pattern match must be a closed type `N`. If this type matches the first pattern, then the produced substitution $\Sigma$ replaces the pattern variables in the branch. The notation $\Sigma(\mathtt{e})$ stands for this simultaneous substitution.

If the first pattern does not match—if we cannot derive the match judgment—then the semantics discards the first pattern and tries to find a match from the remaining patterns. Because every match expression must end with a default branch, some branch will be taken. The operational semantics of this language is deterministic. For simplicity, we designed the semantics such that if several patterns match the analyzed type, then the first match is taken. However it would be possible for the operation of `typematch` to select the most precise pattern.

The definition of *matches* is at the top of Figure 8. Because of the invariance of the arguments to parameterized classes, we must determine not just when a type could be a subtype of a pattern (after some substitutions) but also when it could equal the pattern (after some substitutions). For this reason we define both $\mathit{matches}(\mathtt{S},\mathtt{T})$ and $\mathit{equals}(\mathtt{S},\mathtt{T})$. The first rule states that we can always match a type to a pattern variable, and so produces the substitution that replaces the pattern variable with the type. There is a

$$\Delta \vdash \mathtt{T} <: \mathtt{Object} \text{ [S-OBJECT]}$$

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \text{ [S-REFL]}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \text{ [S-TRANS]}$$

$$\frac{}{\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})} \text{ [S-VAR]}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\{\ \ldots\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}} \text{ [S-CLASS]}$$

$$\frac{}{\Delta; \Gamma \vdash x \in \Gamma(x)} \text{ [T-VAR]}$$

$$\frac{\begin{array}{c} bound_{\Delta}(\mathtt{T}) = \mathtt{N} \\ fields(\mathtt{N}) = \overline{\mathtt{T}}\ \ \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}'} \qquad |\overline{\mathtt{T}'}| \leq |\overline{\mathtt{T}}| \qquad \Delta \vdash \mathtt{T}'_i <: \mathtt{T}_i \ \ (for\ 1 \leq i \leq |\overline{\mathtt{T}'}|) \end{array}}{\Delta; \Gamma \vdash \mathtt{new\ \ T(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}} \text{ [T-NEW]}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \\ bound_{\Delta}(\mathtt{T}_0) = \mathtt{N} \qquad mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}} \text{ [T-INVK]}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad bound_{\Delta}(\mathtt{T}_0) = \mathtt{N} \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \ \overline{\mathtt{f}}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}_i} \text{ [T-FIELD]}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}}{\Delta; \Gamma \vdash \mathtt{(T)e} \in \mathtt{T}} \text{ [T-CAST]}$$

Figure 5: Subtyping and expression typing

similar rule for $equals(\mathtt{S}, \mathtt{T})$. To match a non-variable type to a pattern, we must either match it to the same class, where all of the type arguments are equal, or we must see if its superclass matches the pattern. Likewise, to determine if a non-variable type is equal to a pattern, the pattern must be for the same class, and all of the type arguments must be equal.

To type check a pattern match expression, we check each branch in a refined context that assumes that the analyzed type is a subtype of the pattern. This context refinement means that, unlike `instanceof`, we do not need to cast an expression to match the new type. For example, the following code type checks in this calculus because we add the constraint that `X <: Boolean` in the case for `Boolean` :

```
<X> Integer m (X t) {
  typematch X with
     Boolean: if t { return 1; } else { return 0; }
     default: return -1;
}
```

Technically, we refine the context by adding a special assumption $\mathtt{S} \ll: \mathtt{T}$. If such a refinement is in a context

**Well-formed types:**

$$\Delta \vdash \mathtt{Object}\ \text{ok}\ \text{[WF-Object]}$$

$$\frac{\mathtt{X} \in \text{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}}\ \text{[WF-Var]}$$

$$\frac{\Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}} \qquad CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\ \ldots\ \}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \text{[WF-Class]}$$

**Valid method overriding:**

$$\frac{\mathit{mtype}(\mathtt{m},\mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}_0 \qquad \mathtt{implies}\ \overline{\mathtt{P}}, \overline{\mathtt{T}} = (\overline{\mathtt{Q}}, \overline{\mathtt{U}})\ \text{ and }\ \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{T}_0 <: \mathtt{U}_0}{\mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}_0)}$$

**Method typing:**

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \text{ok} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} <: \mathtt{T} \\ CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\{\ \ldots\ \} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)}\{\mathtt{return}\ \mathtt{e}_0;\}\ \text{ok}\ \mathtt{in}\ \mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}}\ \text{[T-Method]}$$

**Class typing:**

$$\frac{\begin{array}{c}\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}\ \text{ok} \qquad \overline{\mathtt{M}}\ \text{ok}\ \mathtt{in}\ \mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \\ \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}'}\ \overline{\mathtt{f}'} \qquad \overline{\mathtt{f}'} \cap \overline{\mathtt{f}} = \varnothing \qquad \overline{\mathtt{X}} <: \overline{\mathtt{N}};\ \mathtt{this}:\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \overline{\mathtt{v}} \in \overline{\mathtt{S}} <: \overline{\mathtt{T}}\end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \mathtt{=}\overline{\mathtt{v}};\ \overline{\mathtt{M}}\}\ \text{ok}}\ \text{[T-Class]}$$

Figure 6: Core well-formedness rules

$\Delta$, we can conclude that $\Delta \vdash \mathtt{S} <: \mathtt{T}$. This new assumption also appears in the definition of *bounds*. There are no restrictions about what refinement assumptions we may add to the context. If they are not satisfiable (for example, assuming $\mathtt{C} \ll: \mathtt{D}$ when there is no relationship between the classes `C` and `D`) then the branch of `typematch` that introduced that assumption could never be taken. A smart type checker could soundly omit the checking of such branches.

Furthermore, for simplicity we do not make any "deep" conclusions from type matching assumptions. For example, from $\mathtt{C<X>} \ll: \mathtt{C<Y>}$ it would be sound to also conclude that the type `X` equals `Y`. A more sophisticated calculus could incorporate such deductions.

We have shown that this language (as well as the extension for structural analysis described in the next section) is type sound, using a similar proof to that for FGJ [22]. See Appendix A for the proof.

## 3.2 Structural analysis

Using structural type analysis, we would like to be able to determine what fields and methods are present in a class, and then access those fields and invoke those methods. We will do so with two new expression forms. The syntax of the language with the new form appears in Figure 9. Because of the functional nature of this language, these forms are designed as "folds". A language with mutation could simplify these forms into iteration, such as the `forfield` and `formethod` expression forms described in Section 2.

An expression $\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{S}\ \mathtt{f}_\mathtt{x} \in \mathtt{T})\ \mathtt{e}'$ iterates over the fields of the type `T`. The semantics of this expression appears in Figure 10. The variable `x` is an accumulator, initialized with the value of `e`. The expression $\mathtt{e}'$ executes once for each field whose type matches the pattern `S`. The variable $\mathtt{f}_\mathtt{x}$ is an accessor

$$\frac{\varnothing \vdash \mathtt{N} <: \mathtt{P}}{\mathtt{(P)(new\ N(}\overline{\mathtt{v}}\mathtt{))} \mapsto \mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{)}}\ \text{[E-CAST]}$$

$$\frac{\mathit{mbody}(\mathtt{m}<\overline{\mathtt{P}}>, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e})}{\mathtt{(new\ N(}\overline{\mathtt{v}}\mathtt{)).m}<\overline{\mathtt{P}}>(\overline{\mathtt{v}'}) \mapsto [\overline{\mathtt{x}} \mapsto \overline{\mathtt{v}'}, \mathtt{this} \mapsto \mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{)}]\mathtt{e}}\ \text{[E-INVK]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad i \leq |\overline{\mathtt{v}}|}{(\mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{)}).\mathtt{f}_i \mapsto \mathtt{v}_i}\ \text{[E-FIELD]}$$

$$\frac{\mathit{fieldval}(\mathtt{f}_i, \mathtt{N}) = \mathtt{v} \qquad |\overline{\mathtt{v}}| < i}{(\mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{)}).\mathtt{f}_i \mapsto \mathtt{v}}\ \text{[E-DEFAULT]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{(\mathtt{N})\mathtt{e} \mapsto \mathtt{e}'}\ \text{[EC-CAST]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{e.f} \mapsto \mathtt{e}'\mathtt{.f}}\ \text{[EC-FIELD]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{e.m}<\overline{\mathtt{N}}>(\overline{\mathtt{e}}) \mapsto \mathtt{e}'\mathtt{.m}<\overline{\mathtt{T}}>(\overline{\mathtt{e}})}\ \text{[EC-INVK-RECV]}$$

$$\frac{\mathtt{e}_i \mapsto \mathtt{e}_i'}{\mathtt{v.m}<\overline{\mathtt{N}}>(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) \mapsto \mathtt{v.m}<\overline{\mathtt{N}}>(\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}})}\ \text{[EC-INVK-ARG]}$$

$$\frac{\mathtt{e}_i \mapsto \mathtt{e}_i'}{\mathtt{new\ N(}\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}\mathtt{)} \mapsto \mathtt{new\ N(}\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}}\mathtt{)}}\ \text{[EC-NEW-ARG]}$$

Figure 7: Core evaluation rules

variable—a variable referring to the current name of the field. The index $i$ is the index of the current field. In source programs the index should always be 1.

The operational semantics of `fieldfold` is defined by four rules. In the first rule, the index is out of range for the fields of the analyzed class so the accumulator is returned. In the second rule, the index refers to a field in the class, and the type of that field matches the type in the pattern. In that case, the body of `fieldfold` becomes the new accumulator, after substituting the current accumulator for `x`, the current field name for the accessor variable, and the types generated by the pattern match. The next rule is used when the type of the current field does not match the pattern, skipping any analysis of that field. Finally, a congruence rule allows the accumulator to be evaluated to a value.

To type check `fieldfold` we create a refined context to check the body of `fieldfold`. A refinement assumption $\mathtt{S} \lll: \{\mathtt{T}\ \mathtt{f_x}\}$ is present in a context, it means that any expression of type `S`(or any subtype of `S`) may project a field $\mathtt{f_x}$ with type `T`. This assumption is used in the rule T-FIELDVAR to check a field access when the accessor is a variable. The rule for checking a field access for constant accessors is unchanged.

Method folding (see Figure 11) behaves analogously to field folding. The operational semantics iterates through the methods of an object, executing the body of the fold for each matching method type. Determining if a method type matches is a little more complicated than determining a matching field type. Like method overriding, we require equality patterns for the bounds and the arguments to the method, but we allow the return type to be a subtype.

The following two substitution lemmas are important for showing the soundness of field and method folding.

# 4 Related Work

There are several different linguistic mechanisms that support type-directed programming in other languages.

**Determining type names** Initially, type-directed programming was implemented by mechanisms to hide the *names* of types at compile time (via a dynamic type, variously called `any`, `REFANY`, or `Object`) and to recover the type name at run time (such as `INSPECT`, `instanceof` or `TYPECASE`). The languages Simula-67 [3], CLU [32], Cedar/Mesa [29], Modula-2+ and Modula-3 [6] have such mechanisms. Haskell type classes [40] also base execution on the names of types, but decompose type-directed operations in a different manner.

**Reflecting types as data** However, as well as determining names of types, in languages with composite types, such as arrays, tuples, records, and variants, it is important to be able to examine that structure. Java Reflection (like mechanisms in many other languages, such as Amber [5], Cedar/Mesa [29], and C# [23]) provides a mechanism to *reflect* type information into a data structure. However, even though programmers can define operations based on the type of a value, this mechanism cannot tie the type of operations to the reflected type information stored in the data structure. As a result, static type checking is compromised, as we saw in the serialization example in Figure 1. Type-directed operations rely on run-time casts to guarantee their type correctness.

To give users control over run-time type information, but still provide strong static type checking, Crary et al. [13] designed a language that reflects type information into run-time data structures. This language uses a form of dependent type to permit static type checking of type-directed operations. Weirich later showed that the dependency in this language was simple enough to be encoded with higher-order parametric polymorphism [41]. Cheney and Hinze [8] used a similar idea to implement this language as a Haskell library.

**Pattern matching types** Functional languages with strong static type checking deliberately omit (or strongly discourage) mechanisms for run-time type casts. To support the analysis of composite types in the ML language [33], The `Dynamic` type of Abadi et al. [1, 2] and of Leroy [31] uses a special elimination form (called `typecase`) similar to pattern matching. The branches of this form bind type variables to the subcomponents of composite types and create an alias to the dynamic value with the discovered type.

```
fun tostring (dv:Dynamic) =
  typecase dv of
    (v:String) =>
      (* Here v (= to dv) has type String *)
    (v:X*Y)    =>
      (* Here v is a product of type X*Y *)
```

However, with the above mechanism, only type information that is stored in dynamic values can be analyzed. In contrast, intensional polymorphism [19], extensional polymorphism [14] and structural polymorphism [34, 35] are mechanisms that analyze explicit type parameters. These frameworks requires that the language semantics propagate type information at run-time, independently of values. For example, `tostring` implemented with intensional polymorphism analyzes the type parameter `'a`, which is the type of the argument `v`.

```
fun tostring 'a (v : 'a) =
  typecase 'a of
    String => (* Here v has type String *)
    X*Y    => (* Here v has type X*Y *)
```

The G'Caml [15] language is a current extension of O'Caml [30] with extensional polymorphism. Intensional polymorphism also introduced type analysis to the type language allowing the type of type-directed operations to non-parametrically depend on the analyzed type. For example, an operation that swaps the components of embedded tuples must be assigned a type that reflects this transposition. Trifonov et al. [39] extended intensional polymorphism to first-class polymorphic and existential types.

**Polytypic/generic programming** All of the above mechanisms rely on run-time type information, either associated with a dynamic value or independently propagated. A separate line of research, sometimes called polytypic or generic programming, aims to automatically generate type-directed operations at compile time. Because they are based in category theory, the mechanisms in this line of research can define operations, such as maps and folds, that are defined by *parameterized types* (also called *type constructors*). The Charity [11] language automatically generates maps and folds for datatypes at compile time, but cannot be extended with new type-directed operations. Functorial ML [28] uses combinators to define parameterized types and then defines type-directed operations based on these combinators. The ideas behind this theory were incorporated into the FiSH language [27]. Polytypic programming [24, 25] extends Haskell with a way to define type-directed operations. However, it is limited in its domain—it cannot handle mutually recursive, nested or multiparameter datatypes, or datatypes that containing functions. Generic Haskell similarly extends the Haskell language, based on the work of Ralf Hinze [10]. In this framework, parameterized types are built from the simply-typed lambda-calculus, and a type-directed operation is an interpretation of that lambda-calculus term. Weirich's work on higher-order intensional type analysis [42] unifies this rich line of research with run-time type analysis. It extends Hinze's work with run-time type information and allows the definition of these operations in languages with first-class polymorphism.

# 5 Conclusions and Future work

This paper describes a new approach to the design of mechanisms for run-time type analysis in Java. Instead of basing execution on the run-time classes of objects, `typematch`, `fieldfold` and `methfold` examine the structure of first-class type information. Because of this approach, these mechanisms may statically refine the context to reflect the dynamic type discovery. As a result, type-directed operations may be expressed with our mechanisms without the need for type casting.

There are several extensions to this design that we plan to explore in the future. The first is to provide a way for programmers to assign type parameters to the run-time type of objects. That way, type information does not need to be explicitly passed throughout the program. For example, one might use this new capability as follows:

```
public void f(Object x) {
  // T is run-time type of x
  <T> local = x;
  // local is an alias for x with that type
  typeDirectedMethod<T>(local);
}
```

A drawback of this extension is a loss of abstraction—this extension provides a way for anyone to discover the most precise type of an object.

Another extension would allow us to discover more information about run-time types. For example, we could add a type operator `Super<T>` that would return the supertype of a class as a new type parameter. (For `Object` it would just return `Object`.) With this operator, we could print out the class hierarchy above a particular class with the following code:

```
static <X>void printSuperclasses() {
  typematch Super<X>  with
     Object: return;
     default: {
       System.out.println(getClassName<X>);
       printSuperClasses<Super<X>>();
    }
  }
}
```

The reason that we have not done so in the current version of the language is for simplicity. Introducing such a type operator means that we have a non-trivial definition of type equivalence.

Finally, we plan to explore ways to make the structural operators described in this calculus more flexible. There are several ways to provide more expressiveness in the analysis of object types. Through the use of dependent type systems, we might be able to design a calculus that could type check following code:

```
T obj;
 // New reflective form to retrieve the
 // fields of a type parameter.
Field[] fs = T.getFields();
if (fs.length > 1 &&
    fs[0].getType() subtypeof B &&
    fs[0].getName() == "x") {
     // Refine the type T to include at
     // least one field "x" of type B
    B fieldVal = obj.x;
}
```

With dependent types, the type of a term can be determined by arbitrary values of other terms. To make type checking in such a system tractable, we must limit what terms can determine type structure. Finding an expressive but tractable set of restrictions that the programmer can understand will require careful engineering.

# References


[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, Lund, Sweden, 1973.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[5] Luca Cardelli. Amber. In Guy Coisineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70. Springer-Verlag, 1986.

[6] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation, Systems Research Center, November 1989.

[7] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[8] James Cheney and Ralf Hinze. Poor man's dynamics and generics. In Manuel M. Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*. ACM Press, 2002.

[9] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, Montreal, CA, September 2000.

[10] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[11] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[12] Microsoft COM technologies, January 2002. `http://www.microsoft.com/com/default.asp`.

[13] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[14] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, January 1995.

[15] Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, 2001.

[16] Neal Glew. Type dispatch for named hierarchical types. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 172–182, Paris, France, September 1999.

[17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[18] Dale Green. Trail: The reflection API. In Mary Campione, Kathy Walrath, Alison Huml, and Tutorial Team, editors, *The Java Tutorial Continued: The Rest of the JDK(TM)*. Addison-Wesley Pub Co, 1998. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

[19] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[20] Tom Harpin. Using `java.lang.reflect.proxy` to interpose on Java class methods. `http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/`, July 2001.

[21] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.

[22] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[23] International Organisation for Standardization and International Electrotechnical Commission. *ISO/IEC 23270:2003 Information technology–C# Language Specification*, April 2003.

[24] Patrick Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[25] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000.

[26] JavaBeans: The only component for Java technology, May 2002. `http://java.sun.com/products/javabeans/`.

[27] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[28] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[29] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[30] Xavier Leroy. *The Objective Caml System, Release 3.06.* Institut National de Recherche en Informatique et Automatique (INRIA), 2002.

[31] Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 406–426. Springer-Verlag, August 1991.

[32] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU reference manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[33] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[34] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[35] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.*, 1998.

[36] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(2):159–209, 2003.

[37] Jose H. Solorzano and Suad Alagić. Paramteric polymorphism for Java: A reflective solution. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 216–225, Vancouver, Canada, 1998.

[38] Paul Tremblett. Java reflection. *Dr. Dobbs Journal*, January 1998.

[39] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, September 2000.

[40] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[41] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106, Genova, Italy, 2001. Springer.

[42] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, April 2002.

[43] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

**Syntax**

| | | | |
|---|---|---|---|
| Expressions | $\mathtt{e}$ | $::=$ | $\ldots \mid \mathtt{typematch}\ \mathtt{T}\ \mathtt{with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}$ |
| Type contexts | $\Delta$ | $::=$ | $\ldots \mid \mathtt{S} \ll: \mathtt{T}$ |
| Simultaneous type substitutions | $\Sigma$ | $::=$ | $\cdot \mid \Sigma, \mathtt{X} \mapsto \mathtt{T}$ |

**Minimal Context:**

$$\frac{\Delta \vdash \mathtt{T}\ \text{ok} \qquad dom(\Delta) = FV(\mathtt{T}) \qquad \forall \mathtt{X} \in dom(\Delta),\ \Delta(\mathtt{X}) = \mathtt{Object}}{\Delta \vdash \mathtt{T}\ \text{minok}}$$

**Consistency of Substitution**

$$\frac{\forall\ \mathtt{X} \mapsto \mathtt{T} \in \Sigma \ \text{ and }\ \mathtt{Y} \mapsto \mathtt{S} \in \Sigma,\ \ \mathtt{X} = \mathtt{Y} \text{ implies } \mathtt{T} = \mathtt{S}}{\Sigma \text{ consistent}}$$

**Matching**

$$matches(\mathtt{N}, \mathtt{X}) = \mathtt{X} \mapsto \mathtt{N}$$

$$matches(\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = equals(\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$$

$$\frac{CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \triangleleft\ \mathtt{N}\{\ \ldots\ \}}{matches(\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}, \mathtt{T}) = matches([\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{N}, \mathtt{T})}$$

$$equals(\mathtt{N}, \mathtt{X}) = \mathtt{X} \mapsto \mathtt{N}$$

$$\frac{equals(\overline{\mathtt{S}}, \overline{\mathtt{T}}) = \overline{\Sigma} = \Sigma \qquad \Sigma \text{ consistent}}{equals(\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \Sigma}$$

**Computation**

$$\frac{matches(\mathtt{N}, \mathtt{T}) = \Sigma}{\mathtt{typematch}\ \mathtt{N}\ \mathtt{with}\ \mathtt{T} : \mathtt{e}\ \ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \mapsto \Sigma(\mathtt{e})}\ \text{[R-Match]}$$

$$\frac{matches(\mathtt{N}, \mathtt{T}) \text{ is not defined}}{\mathtt{typematch}\ \mathtt{N}\ \mathtt{with}\ \mathtt{T} : \mathtt{e}\ \ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \mapsto \mathtt{typematch}\ \mathtt{N}\ \mathtt{with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}'}\ \text{[R-NoMatch]}$$

$$\frac{}{\mathtt{typematch}\ \mathtt{N}\ \mathtt{with}\ \ \ \mathtt{default} : \mathtt{e}' \mapsto \mathtt{e}'}\ \text{[R-Default]}$$

**Static Semantics:**

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \Delta \qquad bound_\Delta(\mathtt{T}) = \mathtt{N}}{bound_\Delta(\mathtt{S}) = \mathtt{N}}\ \text{[B-Refine]}$$

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{S} <: \mathtt{T}}\ \text{[S-Refine]}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{T}\ \text{ok} \qquad \Delta \vdash \mathtt{U}\ \text{ok} \\ \Delta; \Gamma \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \qquad (1 \le i \le |\overline{\mathtt{T}}|) \quad \Delta_i \vdash \mathtt{T}_i\ \text{minok} \quad \Delta, \Delta_i, \mathtt{T} \ll: \mathtt{T}_i; \Gamma \vdash \mathtt{e}_i \in \mathtt{U}_i <: \mathtt{U}\end{array}}{\Delta; \Gamma \vdash \mathtt{typematch}\ \mathtt{T}\ \mathtt{with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \in \mathtt{U}}\ \text{[T-Refine]}$$

Figure 8: Nominal type pattern matching

| | | | |
|---|---|---|---|
| Class names | $\mathtt{C,D}$ | | |
| Expression variables | $\mathtt{x,y}$ | | |
| Type variables | $\mathtt{X,Y,Z}$ | | |
| Types | $\mathtt{S,T,U}$ | $::=$ | $\mathtt{X} \mid \mathtt{N}$ |
| Non-variable types | $\mathtt{N,P,Q}$ | $::=$ | $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ |
| Class decls | $\mathtt{CL}$ | $::=$ | $\mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft\ \mathtt{N}\ \{\ \overline{\mathtt{T}}\ \ \overline{\mathtt{f_i}} = \overline{\mathtt{v}};\ \overline{\mathtt{M}}\ \}$ |
| Method decl | $\mathtt{M}$ | $::=$ | $\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \mathtt{T}\ \mathtt{m_i}(\overline{\mathtt{T}}\ \ \overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \}$ |
| Expression | $\mathtt{e}$ | $::=$ | $\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}}) \mid \mathtt{new\ T}(\overline{\mathtt{e}}) \mid (\mathtt{T})\mathtt{e}$ |
| | | $\mid$ | $\mathtt{typematch\ T\ with\ \overline{T} : \overline{e}\ \ default : e}$ |
| | | $\mid$ | $\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{T})\ \mathtt{e}'$ |
| | | $\mid$ | $\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e}'$ |
| Method types | $\mathtt{MT}$ | $::=$ | $\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ (\overline{\mathtt{T}}) \rightarrow \mathtt{T}$ |
| Field accessor | $\mathtt{f}$ | $::=$ | $\mathtt{f_i} \mid \mathtt{f_x}$ |
| Method accessor | $\mathtt{m}$ | $::=$ | $\mathtt{m_i} \mid \mathtt{m_x}$ |
| Values | $\mathtt{v}$ | $::=$ | $\mathtt{new\ C}(\overline{\mathtt{v}})$ |
| Type context | $\Delta$ | $::=$ | $\varnothing \mid \Delta, \mathtt{X} <: \mathtt{S} \mid \Delta, \mathtt{S} \lll: \mathtt{T} \mid \Delta, \mathtt{T} \lll: \{\mathtt{T}\ \mathtt{f_x}\} \mid \Delta, \mathtt{T} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}$ |
| Term context | $\Gamma$ | $::=$ | $\varnothing \mid \Gamma, \mathtt{x} : \mathtt{S}$ |

Figure 9: Syntax Additions for Structural Analysis

**Computation**

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad i > |\overline{\mathtt{f}}|}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}}\ \text{[E-FFBase]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad 1 \le i \le |\overline{\mathtt{f}}| \qquad \mathit{matches}(\mathtt{T}_i, \mathtt{T}) = \Sigma}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{f_x} \mapsto \mathtt{f}_i]\Sigma(\mathtt{e}); \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}}\ \text{[E-FFMatch]}$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad 1 \le i \le |\overline{\mathtt{f}}| \qquad \mathit{matches}(\mathtt{T}_i, \mathtt{T}) \text{ is not defined}}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}}\ \text{[E-FFSkip]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}_0 \mapsto \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}'; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}_0}\ \text{[E-FFCong]}$$

**Static semantics**

$$\frac{\begin{array}{c} i > 0 \qquad \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \\ \Delta \vdash \mathtt{T}' \text{ ok} \qquad \Delta' \vdash \mathtt{T} \text{ minok} \qquad \Delta, \Delta', \mathtt{T}' \lll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\Delta; \Gamma \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{T}')\ \mathtt{e}' \in \mathtt{U}}\ \text{[T-FieldFold]}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \lll: \{\mathtt{T}\ \mathtt{f_x}\}}{\Delta; \Gamma \vdash \mathtt{e.f_x} \in \mathtt{T}}\ \text{[T-FieldVar]}$$

Figure 10: Field folding

**Method type matching**

$$\frac{\begin{array}{c} equals(\overline{\mathtt{N}}, \overline{\mathtt{P}}) = \overline{\Sigma} = \Sigma_1 \\ equals(\overline{\mathtt{T}}, \overline{\mathtt{U}}) = \overline{\Sigma'} = \Sigma_2 \qquad matches(\mathtt{T}, \mathtt{U}) = \Sigma'' \qquad \Sigma = \Sigma_1, \Sigma_2, \Sigma'' \qquad \Sigma \text{ consistent} \end{array}}{matches(\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \rightarrow \mathtt{T}, \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \rightarrow \mathtt{U}) = \Sigma}$$

**Method Type Wellformedness**

$$\frac{\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \overline{\mathtt{T}}, \mathtt{T} \text{ ok}}{\Delta \vdash \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \rightarrow \mathtt{T} \text{ ok}} \text{ [MTOK]}$$

**Method Type Minimal Context**

$$\frac{\Delta \vdash \mathtt{MT} \text{ ok} \qquad dom(\Delta) = FV(\mathtt{MT}) \qquad \forall \mathtt{X} \in dom(\Delta),\ \Delta(\mathtt{X}) = \mathtt{Object}}{\Delta \vdash \mathtt{MT} \text{ minok}}$$

**Method Type Subtyping**

$$\frac{\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash (\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \rightarrow \mathtt{T}) <: (\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \overline{\mathtt{T}} \rightarrow \mathtt{U})} \text{ [MTS}\textsc{ub}\text{]}$$

**Computation**

$$\frac{mtype(\mathtt{m}_i, \mathtt{N}) \textit{ is undefined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}} \text{ [E-MFB}\textsc{ase}\text{]}$$

$$\frac{mtype(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad matches(\mathtt{MT}_i, \mathtt{MT}) = \Sigma}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{m_x} \mapsto \mathtt{m_i}]\Sigma(\mathtt{e}); \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-MFM}\textsc{atch}\text{]}$$

$$\frac{mtype(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad matches(\mathtt{MT}_i, \mathtt{MT}) \text{ is not defined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}} \text{ [E-MFS}\textsc{kip}\text{]}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e}'}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}_0 \mapsto \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}'; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}_0} \text{ [E-MFC}\textsc{ong}\text{]}$$

**Static semantics**

$$\frac{\begin{array}{c} i > 0 \qquad \Delta' \vdash \mathtt{MT} \text{ minok} \\ \Delta \vdash \mathtt{T} \text{ ok} \qquad \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \qquad \Delta, \Delta', \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\Delta; \Gamma \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e}' \in \mathtt{U}} \text{ [T-M}\textsc{eth}\text{F}\textsc{old}\text{]}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \overline{\mathtt{T}} \text{ ok} \\ \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta \vdash \mathtt{T}_0 \ll: \{(\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U})\ \mathtt{m_x}\} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta; \Gamma \vdash \mathtt{e.m_x<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}} \text{ [T-I}\textsc{nvk}\text{V}\textsc{ar}\text{]}$$

Figure 11: Method folding

$$\frac{\mathtt{T} \ll: \{\mathtt{U}\ \mathtt{f}_\mathtt{x}\} \in \Delta}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{U}\ \mathtt{f}_\mathtt{x}\}}\ [\text{RF-}\textsc{Hyp}]$$

$$\frac{\Delta \vdash \mathtt{U} \ll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\} \qquad \Delta \vdash \mathtt{S} <: \mathtt{U}}{\Delta \vdash \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\}}\ [\text{RF-}\textsc{Trans}]$$

$$\frac{\mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m}_\mathtt{x}\} \in \Delta}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m}_\mathtt{x}\}}\ [\text{RM-}\textsc{Hyp}]$$

$$\frac{\Delta \vdash \mathtt{U} \ll: \{\mathtt{MT}\ \mathtt{m}_\mathtt{x}\} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m}_\mathtt{x}\}}\ [\text{RM-}\textsc{Trans}]$$

Figure 12: Structural refinement

# A Proofs

In the following, we show that the new extensions that we propose in this paper are type sound. The proof of type soundness is fairly straightforward and resembles the analogous proof of Igarashi, Pierce and Wadler for FGJ [22]. Following the syntactic technique popularized by Wright and Felleisen [43], we show soundness by showing that our calculus satisfies the preservation (Lemma A.19) and progress (Lemma A.20) lemmas. However, before we may prove those lemmas, we must show a number of results about the metatheory of our language.

**Notation A.1** *We write* $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T} <: \texttt{U}$ *as a shorthand for* $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$ *with* $\Delta \vdash \texttt{T} <: \texttt{U}$*.*

**LEMMA A.2 (Consistent Substitution)** If $\Sigma = \Sigma_1, \ldots, \Sigma_n$ and $\Sigma$ consistent and $dom(\Sigma_i) = FV(\texttt{T})$ then $\Sigma_i(\texttt{T}) = \Sigma(\texttt{T})$.

**Proof:** immediate

The following lemma is important to show that the substitution produced by matching is a unifier for the two types. This lemma is important to show that typing is preserved in the evaluation of `typematch`, `fieldfold` and `methfold`.

**LEMMA A.3 (Matches)**

1. If $\varnothing \vdash \texttt{N}$ ok and $equals(\texttt{N}, \texttt{T}) = \Sigma$ then $\texttt{N} = \Sigma(\texttt{T})$ and $dom(\Sigma) = FV(\texttt{T})$.
2. If $\varnothing \vdash \texttt{N}$ ok and $matches(\texttt{N}, \texttt{T}) = \Sigma$ then $\varnothing \vdash \texttt{N} <: \Sigma(\texttt{T})$ and $dom(\Sigma) = FV(\texttt{T})$ and $\varnothing \vdash \Sigma(\texttt{T})$ ok.
3. If $\varnothing \vdash \texttt{MT}$ ok and $matches(\texttt{MT}, \texttt{MT}') = \Sigma$ then $\varnothing \vdash \texttt{MT} <: \Sigma(\texttt{MT}')$ and $dom(\Sigma) = FV(\texttt{MT}')$ and $\varnothing \vdash \Sigma(\texttt{MT}')$ ok.

**Proof:**

1. By induction on the derivation $equals(\texttt{N}, \texttt{T}) = \Sigma$.

   **Case** EQ-VAR: $equals(\texttt{N}, \texttt{T}) = equals(\texttt{N}, \texttt{X}) = \texttt{X} \mapsto \texttt{N} = \Sigma$. immediate.

   **Case** EQ-PAR: $equals(\texttt{N}, \texttt{T}) = equals(\texttt{C<}\overline{\texttt{S}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = equals(\overline{\texttt{S}}, \overline{\texttt{T}}) = \overline{\Sigma} = \Sigma$.
   Since $equals(\texttt{S}_i, \texttt{T}_i) = \Sigma_i$, by induction hypothesis, $\texttt{S}_i = \Sigma_i(\texttt{T}_i)$ and $dom(\Sigma_i) = FV(\texttt{T}_i)$.
   So by Lemma A.2 (Consistent Substitution), $\Sigma(\texttt{T}_i) = \Sigma_i(\texttt{T}_i) = \texttt{S}_i$ thus $\Sigma(\overline{\texttt{T}}) = \overline{\texttt{S}}$.
   Therefore
   $$dom(\Sigma) = \bigcup_i dom(\Sigma_i) = \bigcup_i FV(\texttt{T}_i) = FV(\overline{\texttt{T}}) = FV(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = FV(\texttt{T})$$
   and
   $$\Sigma(\texttt{T}) = \Sigma(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{C<}\Sigma(\overline{\texttt{T}})\texttt{>} = \texttt{C<}\overline{\texttt{S}}\texttt{>} = \texttt{N}$$
   with $\varnothing \vdash \Sigma(\texttt{T})$ ok.

2. By induction on the derivation $matches(\texttt{N}, \texttt{T}) = \Sigma$.

   **Case** M-VAR: $matches(\texttt{N}, \texttt{T}) = matches(\texttt{N}, \texttt{X}) = \texttt{X} \mapsto \texttt{N} = \Sigma$. immediate.

   **Case** M-PAR: $matches(\texttt{N}, \texttt{T}) = matches(\texttt{C<}\overline{\texttt{S}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = equals(\texttt{C<}\overline{\texttt{S}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \Sigma$.

By part 1 of this lemma, $\mathtt{N} = \Sigma(\mathtt{T})$ and $dom(\Sigma) = FV(\mathtt{T})$, thus $\varnothing \vdash \mathtt{N} <: \Sigma(\mathtt{T})$ and $\varnothing \vdash \Sigma(\mathtt{T})$ ok (since $\varnothing \vdash \mathtt{N}$ ok).

**<u>Case</u>** <u>M-SUPER:</u>

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\lhd P\ \{\ ...\ \}}}{matches(\mathtt{N},\mathtt{T}) = matches(\mathtt{C<\overline{S}>},\mathtt{T}) = matches([\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{P},\mathtt{T}) = \Sigma}$$

Let $\mathtt{U} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{P}$. By induction hypothesis, $\varnothing \vdash \mathtt{U} <: \Sigma(\mathtt{T})$ and $dom(\Sigma) = FV(\mathtt{T})$ and $\varnothing \vdash \Sigma(\mathtt{T})$ ok.

Then By the rule S-CLASS, $\varnothing \vdash \mathtt{N} <: \mathtt{U}$ and by the rule S-TRANS, $\varnothing \vdash \mathtt{N} <: \Sigma(\mathtt{T})$.

3. By induction on the derivation of $matches(\mathtt{MT},\mathtt{MT'})$

$$\frac{\begin{array}{c}\mathtt{MT} = \mathtt{<\overline{X} \lhd \overline{N}>\ \overline{T} \to T} \qquad \mathtt{MT'} = \mathtt{<\overline{X} \lhd \overline{P}>\ \overline{U} \to U} \qquad equals(\overline{\mathtt{N}},\overline{\mathtt{P}}) = \overline{\Sigma} = \Sigma_1 \\ equals(\overline{\mathtt{T}},\overline{\mathtt{U}}) = \overline{\Sigma'} = \Sigma_2 \qquad matches(\mathtt{T},\mathtt{U}) = \Sigma'' \qquad \Sigma = \Sigma_1,\Sigma_2,\Sigma'' \qquad \Sigma \text{ consistent}\end{array}}{matches(\mathtt{MT},\mathtt{MT'}) = \Sigma}$$

From $\varnothing \vdash \mathtt{MT}$ ok we know that $\varnothing \vdash \overline{\mathtt{N}},\overline{\mathtt{T}},\mathtt{T}$ ok.

From part 1 of this lemma, we have $\Sigma_1(\overline{\mathtt{P}}) = \overline{\mathtt{N}}$ with $dom(\Sigma_1) = FV(\overline{\mathtt{P}})$ and $\Sigma_2(\overline{\mathtt{U}}) = \overline{\mathtt{T}}$ with $dom(\Sigma_2) = FV(\overline{\mathtt{U}})$.

Then by $\Sigma$ consistent and the Consistency Lemma (A.2)

$$\Sigma(\mathtt{MT'}) = (\Sigma'',\Sigma_1,\Sigma_2)(\mathtt{MT'}) = \mathtt{<\overline{X} \lhd \Sigma_1(\overline{P})>\ \Sigma_2(\overline{T}) \to \Sigma''(U)} = \mathtt{<\overline{X} \lhd \overline{N}>\ \overline{T} \to (\Sigma''(U))}$$

From part 2 of this lemma, we have $\varnothing \vdash \mathtt{T} <: \Sigma''(\mathtt{U})$, $\varnothing \vdash \Sigma''(\mathtt{U})$ ok and $dom(\Sigma'') = FV(\mathtt{U})$.

Then by the rule MTSUB (Method Type Subtyping, Fig. 11), $\varnothing \vdash \mathtt{MT} <: \Sigma(\mathtt{MT'})$.

By the rule MTOK (Fig. 11), we have $\varnothing \vdash \Sigma(\mathtt{MT'})$ ok.

Finally,

$$dom(\Sigma) = dom(\Sigma_1) \cup dom(\Sigma_2) \cup dom(\Sigma'') = FV(\overline{\mathtt{P}}) \cup FV(\overline{\mathtt{U}}) \cup FV(\mathtt{U}) = FV(\mathtt{MT'})$$

The following collection of lemmas shows that although structural refinedment is part of the type context, it does not make a difference for many judgements. For these judgments, structural refinement assumptions may be dropped.

**<u>LEMMA</u> A.4 (Structural Refinement Strengthening)**

1. If $\mathtt{S} \ll: \{\mathtt{T\ f_x}\},\Delta \vdash \mathtt{U} <: \mathtt{V}$ then $\Delta \vdash \mathtt{U} <: \mathtt{V}$.
2. If $\mathtt{S} \ll: \{\mathtt{MT\ m_x}\},\Delta \vdash \mathtt{U} <: \mathtt{V}$ then $\Delta \vdash \mathtt{U} <: \mathtt{V}$.
3. If $\mathtt{S} \ll: \{\mathtt{T\ f_x}\} \vdash \mathtt{U}$ ok then $\varnothing \vdash \mathtt{U}$ ok.
4. If $\mathtt{S} \ll: \{\mathtt{MT\ m_x}\} \vdash \mathtt{U}$ ok then $\varnothing \vdash \mathtt{U}$ ok.
5. If $bound_{\mathtt{S} \ll: \{\mathtt{T\ f_x}\}}(\mathtt{U}) = \mathtt{N}$ then $bound_{\varnothing}(\mathtt{U}) = \mathtt{N}$.
6. If $bound_{\mathtt{S} \ll: \{\mathtt{MT\ m_x}\}}(\mathtt{U}) = \mathtt{N}$ then $bound_{\varnothing}(\mathtt{U}) = \mathtt{N}$.
7. If $\mathtt{S} \ll: \{\mathtt{T\ f_x}\} \vdash \mathtt{U}$ minok then $\varnothing \vdash \mathtt{U}$ minok.
8. If $\mathtt{S} \ll: \{\mathtt{MT\ m_x}\} \vdash \mathtt{U}$ minok then $\varnothing \vdash \mathtt{U}$ minok.

**Proof:**

1. By induction on the derivation of $\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}, \Delta \vdash \mathtt{U} <: \mathtt{V}$.

   **Case** S-Refl: Immediate by the rule S-Refl.

   **Case** S-Trans: Immediate by induction hypothesis(twice) and the rule S-Trans itself.

   **Case** S-Var: Impossible.

   **Case** S-Class: Immediate by the rule S-Class.

   **Case** S-Refine:

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}, \Delta}{\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}, \Delta \vdash \mathtt{S} <: \mathtt{T}}\ \text{[S-Refine]}$$

   From $\mathtt{S} \ll: \mathtt{T} \in \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}, \Delta$ we know $\mathtt{S} \ll: \mathtt{T} \in \Delta$.
   So by the rule S-Refine again, we have $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

2. Similar to the previous part.

3. By induction on the derivation $\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \mathtt{U}$ ok.

   **Case** WF-Object: Immediate.

   **Case** WF-Var: Impossible.

   **Case** WF-Class:

$$\frac{\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}} \qquad CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\ \ldots\ \}}{\mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \text{[WF-Class]}$$

   By induction hypothesis, $\varnothing \vdash \overline{\mathtt{T}}$ ok.
   By part 1 of this lemma, $\varnothing \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}}$.
   Then by the rule WF-Class again, we have $\varnothing \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ ok.

4. Similar to the previous part.
5. Immediate from the definition of *bound*.
6. Similar to the previous part.
7. Trivial.
8. Trivial.

However, when a structural refinement assumption is used in a structural refinement judgment, we can say a few things about it.

**LEMMA A.5 (Subtyping from Structural Refinement)**

1. If $\mathtt{V} \ll: \{\mathtt{T'}\ \mathtt{f_x}\} \vdash \mathtt{S} \ll: \{\mathtt{T}\ \mathtt{f_x}\}$ then $\mathtt{T} = \mathtt{T'}$ and $\varnothing \vdash \mathtt{S} <: \mathtt{V}$.
2. If $\mathtt{S} \ll: \{\mathtt{MT'}\ \mathtt{m_x}\} \vdash \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}$ then $\mathtt{MT} = \mathtt{MT'}$ and $\varnothing \vdash \mathtt{T} <: \mathtt{S}$.

**Proof:**

1. By induction on the derivation $\mathtt{V} \ll: \{\mathtt{T}' \ \mathtt{f_x}\} \vdash \mathtt{S} \ll: \{\mathtt{T} \ \mathtt{f_x}\}$ (See Fig. 9).

   **Case** RF-HYP: Immediate.

   **Case** RF-TRANS:

$$\frac{\mathtt{V} \ll: \{\mathtt{T}' \ \mathtt{f_x}\} \vdash \mathtt{U} \ll: \{\mathtt{T} \ \mathtt{f_x}\} \qquad \mathtt{V} \ll: \{\mathtt{T}' \ \mathtt{f_x}\} \vdash \mathtt{S} <: \mathtt{U}}{\mathtt{V} \ll: \{\mathtt{T}' \ \mathtt{f_x}\} \vdash \mathtt{S} \ll: \{\mathtt{T} \ \mathtt{f_x}\}} \text{[RF-TRANS]}$$

   By induction hypothesis, $\mathtt{T} = \mathtt{T}'$ and $\varnothing \vdash \mathtt{U} <: \mathtt{V}$. By part 2 of this lemma and $\mathtt{V} \ll: \{\mathtt{T}' \ \mathtt{f_x}\} \vdash \mathtt{S} <: \mathtt{U}$, we have $\varnothing \vdash \mathtt{S} <: \mathtt{U}$. Then by the rule S-TRANS, $\varnothing \vdash \mathtt{S} <: \mathtt{V}$.

2. By induction on the derivation $\mathtt{S} \ll: \{\mathtt{MT}' \ \mathtt{m_x}\} \vdash \mathtt{T} \ll: \{\mathtt{MT} \ \mathtt{m_x}\}$.

   **Case** RM-HYP: Immediate.

   **Case** RM-TRANS:

$$\frac{\mathtt{S} \ll: \{\mathtt{MT}' \ \mathtt{m_x}\} \vdash \mathtt{U} \ll: \{\mathtt{MT} \ \mathtt{m_x}\} \qquad \mathtt{S} \ll: \{\mathtt{MT}' \ \mathtt{m_x}\} \vdash \mathtt{T} <: \mathtt{U}}{\mathtt{S} \ll: \{\mathtt{MT}' \ \mathtt{m_x}\} \vdash \mathtt{T} \ll: \{\mathtt{MT} \ \mathtt{m_x}\}} \text{[RM-TRANS]}$$

   By induction hypothesis, $\mathtt{MT} = \mathtt{MT}'$ and $\varnothing \vdash \mathtt{U} <: \mathtt{S}$. By part 1 of this lemma and $\mathtt{S} \ll: \{\mathtt{MT}' \ \mathtt{m_x}\} \vdash \mathtt{T} <: \mathtt{U}$, we have $\varnothing \vdash \mathtt{T} <: \mathtt{U}$. Then by the rule S-TRANS, $\varnothing \vdash \mathtt{T} <: \mathtt{S}$.

Even though we did not explicitly add rules for reflexivity and transitivity for method type subtyping, those properties hold for our system.

**LEMMA A.6 (Method Type Subtyping)**

1. $\Delta \vdash \mathtt{MT} <: \mathtt{MT}$.
2. If $\Delta \vdash \mathtt{MT} <: \mathtt{MT}'$ and $\Delta \vdash \mathtt{MT}' <: \mathtt{MT}''$ then $\Delta \vdash \mathtt{MT} <: \mathtt{MT}''$.

   **Proof:**

1. Immediate from the rule S-REFL and the rule MTSUB (definition of method type subtyping).
2. Immediate from the rule S-TRANS and the rule MTSUB (definition of method type subtyping).

The next lemma shows the relationship between the fields and methods of a type and its subtypes. Basically, subtypes must have the same fields, with possibly some additional ones. Also, the type of a method in one class must be a subtype of its type in classes that are supertypes.

**LEMMA A.7 (Fields and Methods for Subtypes)**

1. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $bound_\Delta(\mathtt{T}) = \mathtt{P}$, then $bound_\Delta(\mathtt{S}) = \mathtt{N}$ for some $\mathtt{N}$ and $fields(\mathtt{N}) = fields(\mathtt{P}); \overline{\mathtt{T}} \ \overline{\mathtt{f}}$.
2. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $bound_\Delta(\mathtt{T}) = \mathtt{P}$ and $mtype(\mathtt{m_i}, \mathtt{P}) = \mathtt{MT}_i$, then $bound_\Delta(\mathtt{S}) = \mathtt{N}$ for some $\mathtt{N}$ and $\Delta \vdash mtype(\mathtt{m_i}, \mathtt{N}) <: \mathtt{MT}_i$.

   **Proof:**

1. By induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

   Note that by adding the rule B-Refine, the calculus becomes *nondeterministic*. But since the ruleset for $bound_\Delta(\mathtt{T})$ is an (proper) superset of that of [22], everything derivable there is still derivable here. So we just show the new case:

   **Case** S-Refine:

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{S} <: \mathtt{T}} \text{ [S-Refine]}$$

   By the rule B-Refine

$$\frac{\mathtt{S} \ll: \mathtt{T} \in \Delta \qquad bound_\Delta(\mathtt{T}) = \mathtt{P}}{bound_\Delta(\mathtt{S}) = \mathtt{P}} \text{ [B-Refine]}$$

   Let $\mathtt{N} = \mathtt{P}$ trivially finishes the case.

2. same as above.

Before we can show the preservation lemma, we must prove a number of substitution lemmas. The first is that if we have made an assumption about the structure about the type when type checking an expression, that turns out to be satisfiable, we can substitute the appropriate field or method name into that expression.

**LEMMA A.8 (Field and Method Substitutions)**

1. If $\mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma \vdash \mathtt{e} \in \mathtt{U}$ and *fields*$(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ and $\varnothing \vdash \mathtt{T}_i <: \mathtt{T}$ then $\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}]\mathtt{e} \in \mathtt{S} <: \mathtt{U}$.
2. If $\mathtt{N} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}; \Gamma \vdash \mathtt{e} \in \mathtt{U}$ and *mtype*$(\mathtt{m_i}, \mathtt{N}) = \mathtt{MT}_i$ and $\varnothing \vdash \mathtt{MT}_i <: \mathtt{MT}$ then $\varnothing; \Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\mathtt{e} \in \mathtt{S} <: \mathtt{U}$.

**Proof:**

1. Let $\Delta = \mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\}$. By induction on the derivation of $\mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma \vdash \mathtt{e} \in \mathtt{U}$. Only the first case is interesting.

   **Case** T-FieldVar:

$$\frac{\mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \mathtt{T}_0 \ll: \{\mathtt{U}\ \mathtt{f_x}\}}{\mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma \vdash \mathtt{e}_0.\mathtt{f_x} \in \mathtt{U}} \text{ [T-FieldVar]}$$

   By $\mathtt{N} \ll: \{\mathtt{T}\ \mathtt{f_x}\} \vdash \mathtt{T}_0 \ll: \{\mathtt{U}\ \mathtt{f_x}\}$ (from inversion) and Lemma A.5 (Subtyping from Structural Refinement), it must be the case that

$$\mathtt{U} = \mathtt{T} \quad \text{and} \quad \varnothing \vdash \mathtt{T}_0 <: \mathtt{N}$$

   And by induction hypothesis we have

$$\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}]\mathtt{e}_0 \in \mathtt{T}'_0 <: \mathtt{T}_0$$

   So by the rule S-Trans we have

$$\varnothing \vdash \mathtt{T}'_0 <: \mathtt{N}$$

   Then by Lemma A.7 (Field and Methods for Subtypes) with $bound_\varnothing(\mathtt{N}) = \mathtt{N}$ we know $bound_\varnothing(\mathtt{T}'_0) = \mathtt{N}'$ for some $\mathtt{N}'$ and

$$fields(\mathtt{N}') = fields(\mathtt{N}); \overline{\mathtt{T}'}\ \overline{\mathtt{f}'} \ = \ \overline{\mathtt{T}}\ \overline{\mathtt{f}}; \overline{\mathtt{T}'}\ \overline{\mathtt{f}'}$$

Then by the rule T-Field and $\varnothing \vdash \mathtt{T}_i <: \mathtt{T}$, we have

$$\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}](\mathtt{e}_0.\mathtt{f_i}) = ([\mathtt{f_x} \mapsto \mathtt{f_i}]\mathtt{e}_0).\mathtt{f_i} \in \mathtt{T}_i <: \mathtt{T} = \mathtt{U}$$

**Case** T-Var: Impossible.

**Case** T-Cast: Immediate from induction hypothesis and applying the rule T-Cast again.

**Case** T-Field:

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{N} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}} \quad \overline{\mathtt{f}}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_j \in \mathtt{T}_i} \text{ [T-Field]}$$

By induction hypothesis,

$$\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}]\mathtt{e}_0 \in \mathtt{T}'_0 <: \mathtt{T}_0$$

By Lemma A.7 (Fields and Methods for Subtypes) with $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{N}$, we have $\mathit{bound}_\varnothing(\mathtt{T}'_0) = \mathtt{N}'$ for some $\mathtt{N}'$ and

$$\mathit{fields}(\mathtt{N}') = \mathit{fields}(\mathtt{N}); \overline{\mathtt{T}'}\ \overline{\mathtt{f}'} \ = \ \overline{\mathtt{T}}\ \overline{\mathtt{f}}; \overline{\mathtt{T}'}\ \overline{\mathtt{f}'}$$

Then by the rule T-Field, we conclude with

$$\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}]\mathtt{e}_0.\mathtt{f_i} \in \mathtt{T}_j <: \mathtt{T}_i$$

**Case** T-New:

$$\frac{\begin{array}{c}\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{N} \\ \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}} \quad \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}'} \qquad |\overline{\mathtt{T}'}| \leq |\overline{\mathtt{T}}| \qquad \Delta \vdash \mathtt{T}'_i <: \mathtt{T}_i \quad (\mathit{for}\ 1 \leq i \leq |\overline{\mathtt{T}'}|)\end{array}}{\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{T}(\overline{\mathtt{e}}) \in \mathtt{T}} \text{ [T-New]}$$

By induction hypothesis,

$$\varnothing; \Gamma \vdash [\mathtt{f_x} \mapsto \mathtt{f_i}]\overline{\mathtt{e}} \in \overline{\mathtt{S}} <: \overline{\mathtt{T}'}$$

From inversion we know

$$\Delta \vdash \mathtt{T}'_i <: \mathtt{T}_i \quad (\mathit{for}\ 1 \leq i \leq |\overline{\mathtt{T}'}|)$$

By Lemma A.4 (Structural Refinement Strengthening), we have

$$\varnothing \vdash \mathtt{T}'_i <: \mathtt{T}_i \quad (\mathit{for}\ 1 \leq i \leq |\overline{\mathtt{T}'}|)$$

By the rule S-Trans,

$$\varnothing \vdash \mathtt{S}_i <: \mathtt{T}_i \quad (\mathit{for}\ 1 \leq i \leq |\overline{\mathtt{T}'}|)$$

Now applying the rule T-New again finishes the case.

**Case** T-Invk:

Easy and similar to the above case: just apply induction hypothesis, Lemma A.4 (Structural Refinement Strengthening), the rule S-Trans, and the rule T-Invk again.

**Case** T-Refine: Easy, by induction hypothesis, Lemma A.4 and the rule T-Refine again.

**Case** T-FieldFold, T-MethFold: Similar to the above case.

**Case** T-InvkVar: Impossible.

2. Let $\Delta = \mathtt{N} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}$. By induction on the derivation of $\mathtt{N} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}; \Gamma \vdash \mathtt{e} \in \mathtt{U}$. Only the first case is interesting.

<u>**Case** T-InvkVar:</u>

$$\frac{\Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \overline{\mathtt{T}}\ \text{ok}\\ \Delta \vdash \mathtt{T}_0 \ll: \{(\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U})\ \mathtt{m_x}\}\end{array} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m_x}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}} \text{[T-InvkVar]}$$

By induction hypothesis,

$$\varnothing;\Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\mathtt{e}_0 \in \mathtt{T}_0' <: \mathtt{T}_0$$

From $\mathtt{N} \ll: \{\mathtt{MT}\ \mathtt{m_x}\} \vdash \mathtt{T}_0 \ll: \{(\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U})\ \mathtt{m_x}\}$ and Lemma A.5 (Subtyping from Structural Refinement), it must be

$$\mathtt{MT} = (\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}) \ \text{ and } \ \varnothing \vdash \mathtt{T}_0 <: \mathtt{N}$$

By the rule S-Trans we have

$$\varnothing \vdash \mathtt{T}_0' <: \mathtt{N}$$

Then by $\mathit{mtype}(\mathtt{m_i}, \mathtt{N}) = \mathtt{MT}_i$ and Lemma A.7 (Fields and Methods for Subtypes) with $\mathit{bound}_\varnothing(\mathtt{N}) = \mathtt{N}$ we know $\mathit{bound}_\varnothing(\mathtt{T}_0') = \mathtt{N}'$ for some $\mathtt{N}'$ and

$$\varnothing \vdash \mathit{mtype}(\mathtt{m_i}, \mathtt{N}') <: \mathtt{MT}_i$$

With $\varnothing \vdash \mathtt{MT}_i <: \mathtt{MT}$ and the rule MTS-Trans (transitivity of method type subtyping), we have

$$\varnothing \vdash \mathit{mtype}(\mathtt{m_i}, \mathtt{N}') <: \mathtt{MT}$$

By $\mathtt{MT} = (\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U})$ and inversion of the rule MTSub (definition of method type subtyping), it must be the case

$$\mathit{mtype}(\mathtt{m_i}, \mathtt{N}') = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U}' \ \text{ and } \ \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{U}' <: \mathtt{U}$$

From inversion of the rule T-InvkVar we know

$$\Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \ \text{ and } \ \Delta \vdash \overline{\mathtt{T}}\ \text{ok}$$

By Lemma A.4 (Structural Refinement Strengthening), we have

$$\varnothing \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \ \text{ and } \ \varnothing \vdash \overline{\mathtt{T}}\ \text{ok}$$

By induction hypothesis and the rule S-Trans we have

$$\varnothing;\Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\overline{\mathtt{e}} \in \overline{\mathtt{S}'} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}$$

Now we apply the rule T-Invk:

$$\frac{\mathit{bound}_\varnothing(\mathtt{T}_0') = \mathtt{N}' \qquad \begin{array}{c}\varnothing;\Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\mathtt{e}_0 \in \mathtt{T}_0' \qquad \varnothing;\Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\overline{\mathtt{e}} \in \overline{\mathtt{S}'} \qquad \varnothing \vdash \overline{\mathtt{T}}\ \text{ok}\\ \mathit{mtype}(\mathtt{m}_i, \mathtt{N}') = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U}' \qquad \varnothing \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}}\end{array} \qquad \varnothing \vdash \overline{\mathtt{S}'} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}}{\varnothing;\Gamma \vdash ([\mathtt{m_x} \mapsto \mathtt{m_i}]\mathtt{e}_0).\mathtt{m_i}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}([\mathtt{m_x} \mapsto \mathtt{m_i}]\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}'} \text{[T-Invk]}$$

We can also apply Lemma A.13 (Subtyping) to $\overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{U}' <: \mathtt{U}$ and get

$$\varnothing \vdash [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}' <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}$$

So finally

$$\varnothing;\Gamma \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\mathtt{e}_0.\mathtt{m_i}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}([\mathtt{m_x} \mapsto \mathtt{m_i}]\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}' <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}$$

<u>**Case** T-FieldVar:</u> Impossible.

*Other cases analagous to the corresponding cases in part 1 of the lemma.*

The next few lemmas deal with context *strengthening*. These lemmas show that assumptions of the form $\mathtt{S} \ll: \mathtt{T}$ may be dropped from a context $\Delta$ when $\Delta \vdash \mathtt{S} <: \mathtt{T}$. Before we can prove the strengthening lemmas (A.11) we must first show how these trivial refinements interact with determining the bound of type variables.

**LEMMA A.9 (Subtyping of Bound)**

1. If $\Delta' = \mathtt{S} \ll: \mathtt{T}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\mathit{bound}_{\Delta,\Delta'}(\mathtt{U}) = \mathtt{N}$ then $\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{P}$ for some $\mathtt{P}$ where $\Delta \vdash \mathtt{P} <: \mathtt{N}$.
2. If $\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{P}$ then $\Delta \vdash \mathtt{U} <: \mathtt{P}$.
3. (Corollary) If $\Delta' = \mathtt{S} \ll: \mathtt{T}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\mathit{bound}_{\Delta,\Delta'}(\mathtt{U}) = \mathtt{N}$ then $\Delta \vdash \mathtt{U} <: \mathtt{N}$.

**Proof:**

1. By induction on the derivation of $\mathit{bound}_{\Delta,\Delta'}(\mathtt{U}) = \mathtt{N}$, with a case analysis of the last rule used.

**Case** B-NonVar:

It must be the case that $\mathtt{U} = \mathtt{N}$, so $\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{N}$.

**Case** B-Var:

So $\mathtt{U} = \mathtt{X}$ and $\mathit{bound}_{\Delta,\Delta'}(\mathtt{X}) = \mathtt{N}$ and $\mathit{bound}_{\Delta,\Delta'}((\Delta,\Delta')(\mathtt{X})) = \mathtt{N}$.

By induction hypothesis,

$$\mathit{bound}_{\Delta}((\Delta,\Delta')(\mathtt{X})) = \mathtt{P} \quad \text{and} \quad \Delta \vdash \mathtt{P} <: \mathtt{N}$$

Since $\Delta' = \mathtt{S} \ll: \mathtt{T}$, we have $(\Delta,\Delta')(\mathtt{X}) = \Delta(\mathtt{X})$ and thus $\mathit{bound}_{\Delta}(\Delta(\mathtt{X}))=\mathtt{P}$.

By the rule B-Var again, we have $\mathit{bound}_{\Delta}(\mathtt{X})=\mathtt{P}$ so

$$\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{P} \quad \text{and} \quad \Delta \vdash \mathtt{P} <: \mathtt{N}$$

**Case** B-Refine:

$$\frac{\mathtt{U} \ll: \mathtt{T}' \in \Delta,\Delta' \qquad \mathit{bound}_{\Delta,\Delta'}(\mathtt{T}') = \mathtt{N}}{\mathit{bound}_{\Delta,\Delta'}(\mathtt{U}) = \mathtt{N}} \text{[B-Refine]}$$

By induction hypothesis, $\mathit{bound}_{\Delta}(\mathtt{T}') = \mathtt{N}'$ and $\Delta \vdash \mathtt{N}' <: \mathtt{N}$ for some $\mathtt{N}'$.

**Subcase** $\mathtt{U} \ll: \mathtt{T}' \in \Delta'$:

So $\mathtt{U} = \mathtt{S}$ and $\mathtt{T}' = \mathtt{T}$ and thus $\Delta \vdash \mathtt{U} <: \mathtt{T}'$

By Lemma A.10 (Monotonicity of Bound),

$$\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{P} \quad \text{and} \quad \Delta \vdash \mathtt{P} <: \mathtt{N}'$$

for some $\mathtt{P}$. Finally by the rule S-Trans, $\Delta \vdash \mathtt{P} <: \mathtt{N}$.

**Subcase** $\mathtt{U} \ll: \mathtt{T}' \in \Delta$: By the rule B-Refine again,

$$\frac{\mathtt{U} \ll: \mathtt{T}' \in \Delta \qquad \mathit{bound}_{\Delta}(\mathtt{T}') = \mathtt{N}'}{\mathit{bound}_{\Delta}(\mathtt{U}) = \mathtt{N}'} \text{[B-Refine]}$$

With $\Delta \vdash \mathtt{N}' <: \mathtt{N}$, letting $\mathtt{P} = \mathtt{N}'$ simply finishes the case.

2. By induction on the derivation of $\mathit{bound}_\Delta(\mathtt{U}) = \mathtt{P}$.

**Case** B-VAR: $\mathtt{U} = \mathtt{X}$.

By induction hypothesis, $\Delta \vdash \Delta(\mathtt{X}) <: \mathtt{P}$.

By the rule S-VAR, $\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$.

Then by the rule S-TRANS, $\Delta \vdash \mathtt{X} <: \mathtt{P}$.

**Case** B-NONVAR: Immediate by the rule S-REFL.

**Case** B-REFINE:

$$\frac{\mathtt{U} \lll: \mathtt{T} \in \Delta \qquad \mathit{bound}_\Delta(\mathtt{T}) = \mathtt{P}}{\mathit{bound}_\Delta(\mathtt{U}) = \mathtt{P}} \text{ [B-REFINE]}$$

By induction hypothesis, $\Delta \vdash \mathtt{T} <: \mathtt{P}$.

By the rule S-REFINE, $\Delta \vdash \mathtt{U} <: \mathtt{T}$.

By the rule S-TRANS, $\Delta \vdash \mathtt{U} <: \mathtt{P}$.

**LEMMA A.10 (Monotonicity of Bound)** If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{P}$ then $\mathit{bound}_\Delta(\mathtt{S}) = \mathtt{N}$ and $\Delta \vdash \mathtt{N} <: \mathtt{P}$ for some $\mathtt{N}$.

**Proof:** If $\mathtt{S} = \mathtt{N}'$ then it must be the case that $\mathtt{T} = \mathtt{P}'$.
Then $\mathit{bound}_\Delta(\mathtt{S}) = \mathtt{N}'$ and $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{P}'$ and thus $\mathtt{P}' = \mathtt{P}$.
So we have $\Delta \vdash \mathtt{N}' <: \mathtt{P}$. Letting $\mathtt{N} = \mathtt{N}'$ finishes the case.

If $\mathtt{S} = \mathtt{X}$ then we do induction on $\Delta \vdash \mathtt{X} <: \mathtt{T}$.

**Case** S-REFL: $\mathtt{T} = \mathtt{X}$. Trivial.

**Case** S-CLASS: Impossible.

**Case** S-VAR: $\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$ and $\Delta(\mathtt{X}) = \mathtt{T}$ and $\mathit{bound}_\Delta(\Delta(\mathtt{X})) = \mathtt{P}$.
By the rule B-VAR we have

$$\frac{\mathit{bound}_\Delta(\Delta(\mathtt{X})) = \mathtt{P}}{\mathit{bound}_\Delta(\mathtt{X}) = \mathtt{P}} \text{ [B-VAR]}$$

**Case** S-TRANS:

$$\frac{\Delta \vdash \mathtt{X} <: \mathtt{U} \qquad \Delta \vdash \mathtt{U} <: \mathtt{T}}{\Delta \vdash \mathtt{X} <: \mathtt{T}} \text{ [S-TRANS]}$$

Immediate by applying induction hypothesis twice and the rule S-TRANS.

**Case** S-REFINE: Immediate by applying the rule B-REFINE:

$$\frac{\mathtt{X} \lll: \mathtt{T} \in \Delta \qquad \mathit{bound}_\Delta(\mathtt{T}) = \mathtt{P}}{\mathit{bound}_\Delta(\mathtt{X}) = \mathtt{P}} \text{ [B-REFINE]}$$

**LEMMA A.11 (Strengthening)**

1. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta, \mathtt{S} \lll: \mathtt{T}; \varnothing \vdash \mathtt{e} \in \mathtt{U}$ then $\Delta; \varnothing \vdash \mathtt{e} \in \mathtt{U}' <: \mathtt{U}$.
2. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta, \mathtt{S} \lll: \mathtt{T} \vdash \mathtt{U} <: \mathtt{V}$ then $\Delta \vdash \mathtt{U} <: \mathtt{V}$
3. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta, \mathtt{S} \lll: \mathtt{T} \vdash \mathtt{U}$ ok then $\Delta \vdash \mathtt{U}$ ok

4. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta, \mathtt{S} \lll: \mathtt{T} \vdash \mathtt{U} \lll: \{\mathtt{V}\ \mathtt{f_x}\}$then $\Delta \vdash \mathtt{U} \lll: \{\mathtt{V}\ \mathtt{f_x}\}$
5. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta, \mathtt{S} \lll: \mathtt{T} \vdash \mathtt{U} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}$then $\Delta \vdash \mathtt{U} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}$
6. (Corollary of 1 and 2) If $\Delta, \mathtt{S} \lll: \mathtt{T}; \varnothing \vdash \mathtt{e} \in \mathtt{U} <: \mathtt{V}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$ then $\Delta; \varnothing \vdash \mathtt{e} \in \mathtt{U}' <: \mathtt{V}$

**Proof:**
Let $\Delta' = \mathtt{S} \lll: \mathtt{T}$.

1. By induction on $\Delta, \Delta'; \varnothing \vdash \mathtt{e} \in \mathtt{U}$.

<u>**Case** T-Var:</u> Impossible. (closed form)

<u>**Case** T-Cast:</u> Immediate from induction hypothesis and the rule T-Cast.

<u>**Case** T-Field:</u>

$$\frac{\Delta, \Delta'; \varnothing \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{bound}_{\Delta,\Delta'}(\mathtt{T}_0) = \mathtt{N} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \ \overline{\mathtt{f}}}{\Delta, \Delta'; \varnothing \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}_i}\ [\text{T-Field}]$$

By inversion and Lemma A.9 (Subtyping of Bound, part 3), we have

$$\Delta \vdash \mathtt{T}_0 <: \mathtt{N}$$

By induction hypothesis,

$$\Delta; \varnothing \vdash \mathtt{e}_0 \in \mathtt{T}_0' <: \mathtt{T}_0$$

So by the rule S-Trans, $\Delta \vdash \mathtt{T}_0' <: \mathtt{N}$.
By Lemma A.7 (Fields and Methods for Subtypes) with $\mathit{bound}_\Delta(\mathtt{N}) = \mathtt{N}$, we have

$$\mathit{bound}_\Delta(\mathtt{T}_0') = \mathtt{N}' \quad \text{with} \quad \mathit{fields}(\mathtt{N}') = \mathit{fields}(\mathtt{N}); \overline{\mathtt{T}'}\ \overline{\mathtt{f}'}$$

Applying the rule T-Field again finishes the case.

<u>**Case** T-New:</u> very similar to the case T-Field.

<u>**Case** T-Invk:</u>

$$\frac{\begin{array}{c}\Delta, \Delta'; \varnothing \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta, \Delta'; \varnothing \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta, \Delta' \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \mathit{bound}_{\Delta,\Delta'}(\mathtt{T}_0) = \mathtt{N} \\ \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U} \qquad \Delta, \Delta' \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta, \Delta' \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}\end{array}}{\Delta, \Delta'; \varnothing \vdash \mathtt{e}_0.\mathtt{m<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}}\ [\text{T-Invk}]$$

By inversion and Lemma A.9 (Subtyping of Bound, part 3), we have

$$\Delta \vdash \mathtt{T}_0 <: \mathtt{N}$$

By induction hypothesis,

$$\Delta; \varnothing \vdash \mathtt{e}_0 \in \mathtt{T}_0' <: \mathtt{T}_0$$

So by the rule S-Trans, $\Delta \vdash \mathtt{T}_0' <: \mathtt{N}$.
By Lemma A.7 (Fields and Methods for Subtypes) with $\mathit{bound}_\Delta(\mathtt{N}) = \mathtt{N}$ and $\mathit{mtype}(\mathtt{m_i}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U}$, we have

$$\mathit{bound}_\Delta(\mathtt{T}_0') = \mathtt{N}' \quad \text{with} \quad \Delta \vdash \mathit{mtype}(\mathtt{m_i}, \mathtt{N}') <: \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}} \to \mathtt{U}$$

Then by Lemma A.6 (Transitivity of Method Type Subtyping), letting $\texttt{MT} = \mathit{mtype}(\texttt{m}_\texttt{i}, \texttt{N}')$, we have

$$\Delta \vdash \texttt{MT} <: \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{\texttt{U}} \to \texttt{U}$$

By inversion of the rule MTSub, it must be the case that

$$\texttt{MT} = \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{\texttt{U}} \to \texttt{U}' \quad \text{and} \quad \Delta, \overline{\texttt{Y}} <: \overline{\texttt{P}} \vdash \texttt{U}' <: \texttt{U}$$

From inversion of the rule T-Invk, we have

$$\Delta, \Delta'; \varnothing \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta, \Delta' \vdash \overline{\texttt{T}} \text{ ok} \qquad \Delta, \Delta' \vdash \overline{\texttt{T}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{P}} \qquad \Delta, \Delta' \vdash \overline{\texttt{S}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{U}}$$

Applying induction hypothesis, part 2 (subtyping) and part 3 (wellformedness) of this lemma, we get

$$\Delta; \varnothing \vdash \overline{\texttt{e}} \in \overline{\texttt{V}} <: \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{T}} \text{ ok} \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{P}} \qquad \Delta \vdash \overline{\texttt{S}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{U}}$$

By the rule S-Trans , we have

$$\Delta \vdash \overline{\texttt{V}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{U}}$$

Now apply the rule T-Invk again,

$$\frac{\begin{array}{c} \Delta; \varnothing \vdash \texttt{e}_0 \in \texttt{T}'_0 \qquad \Delta; \varnothing \vdash \overline{\texttt{e}} \in \overline{\texttt{V}} \qquad \Delta \vdash \overline{\texttt{T}} \text{ ok} \\ \mathit{bound}_\Delta(\texttt{T}'_0) = \texttt{N}' \qquad \mathit{mtype}(\texttt{m}, \texttt{N}') = \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{\texttt{U}} \to \texttt{U}' \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{P}} \qquad \Delta \vdash \overline{\texttt{V}} <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\overline{\texttt{U}} \end{array}}{\Delta; \varnothing \vdash \texttt{e}_0.\texttt{m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\texttt{U}'} \text{[T-Invk]}$$

Applying Lemma A.13 (Subtyping) to $\Delta, \overline{\texttt{Y}} <: \overline{\texttt{P}} \vdash \texttt{U}' <: \texttt{U}$, we have

$$\Delta \vdash [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\texttt{U}' <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\texttt{U}$$

and thus $\Delta; \varnothing \vdash \texttt{e}_0.\texttt{m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\texttt{U}' <: [\overline{\texttt{Y}} \mapsto \overline{\texttt{T}}]\texttt{U}$.

**Case** T-Refine, T-FieldFold, T-MethFold: Easy. Just apply induction hypothesis, and part 2 and 3 of this lemma and use the same rule again.

**Case** T-FieldVar: Easy. Just apply induction hypothesis, and part 4 of this lemma and the rule T-FieldVar again.

**Case** T-InvkVar: Easy. Just apply induction hypothesis, and part 5 of this lemma and the rule T-InvkVar again.

2. By straightforward induction on $\Delta, \Delta' \vdash \texttt{U} <: \texttt{V}$.

**Case** S-Refl: Trivial.

**Case** S-Class: Trivial.

**Case** S-Trans: Immediate from induction hypothesis and the rule S-Trans.

**Case** S-Var:

$$\Delta, \Delta' \vdash \texttt{X} <: (\Delta, \Delta')(\texttt{X}) = \Delta(\texttt{X})$$

By the rule S-Var again,

$$\Delta \vdash \texttt{U} = \texttt{X} <: \texttt{V} = (\Delta, \Delta')(\texttt{X}) = \Delta(\texttt{X})$$

**Case** S-REFINE:

$$\frac{\mathtt{U} \ll: \mathtt{V} \in \Delta, \Delta'}{\Delta, \Delta' \vdash \mathtt{U} <: \mathtt{V}} \text{ [S-Refine]}$$

**Subcase** $\mathtt{U} \ll: \mathtt{V} \in \Delta'$:

Then it must be the case that $\mathtt{U} = \mathtt{S}$ and $\mathtt{V} = \mathtt{T}$. Since we know $\Delta \vdash \mathtt{S} <: \mathtt{T}$, so $\Delta \vdash \mathtt{U} <: \mathtt{V}$.

**Subcase** $\mathtt{U} \ll: \mathtt{V} \in \Delta$: Then by the rule S-REFINE again, we have $\Delta \vdash \mathtt{U} <: \mathtt{V}$.

3. By straightforward induction on $\Delta, \Delta' \vdash \mathtt{U}$ ok.

**Case** WF-OBJECT: Trivial.

**Case** WF-VAR:

$$\frac{\mathtt{X} \in dom(\Delta, \Delta')}{\Delta, \Delta' \vdash \mathtt{X} \text{ ok}} \text{ [WF-Var]}$$

From $\mathtt{X} \in dom(\Delta, \Delta')$ we have $\mathtt{X} \in dom(\Delta)$. Applying this rule again gives $\Delta \vdash \mathtt{X}$ ok.

**Case** WF-CLASS: Immediate by induction hypothesis, part 2 of this lemma and the rule WF-CLASS again.

4. By induction on the derivation of $\Delta, \Delta' \vdash \mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\}$.

**Case** RF-HYP:

$$\frac{\mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\} \in \Delta, \Delta'}{\Delta, \Delta' \vdash \mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\}} \text{ [RF-Hyp]}$$

Since $\Delta' = \mathtt{S} \ll: \mathtt{T}$, it must be the caes that $\mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\} \in \Delta$. Applying the rule RF-VAR again finishes the case.

**Case** RF-TRANS:

$$\frac{\Delta, \Delta' \vdash \mathtt{U'} \ll: \{\mathtt{V}\ \mathtt{f_x}\} \qquad \Delta, \Delta' \vdash \mathtt{U} <: \mathtt{U'}}{\Delta, \Delta' \vdash \mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\}} \text{ [RF-Trans]}$$

By induction hypothesis,

$$\Delta \vdash \mathtt{U'} \ll: \{\mathtt{V}\ \mathtt{f_x}\}$$

And by part 2 of this lemma (subtyping), we have

$$\Delta \vdash \mathtt{U} <: \mathtt{U'}$$

So by the rule RF-TRANS again,

$$\Delta \vdash \mathtt{U} \ll: \{\mathtt{V}\ \mathtt{f_x}\}$$

5. nearly same as part 4, replacing the field-folding judgements with their method-folding counterparts.

**<u>LEMMA</u> A.12 (Type Substitution Preserves Typing)** If $\Delta_1, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Delta_1 \vdash \overline{\mathtt{U}}$ ok and $\Delta_1 \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{N}}$ and none of $\overline{\mathtt{X}}$ appears in $\Delta_1$ then $\Delta_1, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\Delta_2; [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\Gamma \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{e} \in \mathtt{S} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T}$

**Note:** by our definition of $\Sigma$, we can rewrite this lemma in the following form:

$$\left.\begin{array}{l} \Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \\ \Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T} \\ dom(\Sigma) = dom(\Delta') \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) <: \Sigma(\Delta'(\overline{\mathtt{X}})) \\ FV(\Delta_1) \cap dom(\Sigma) = \varnothing \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) \text{ ok} \end{array}\right\} \Longrightarrow \Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{S} <: \Sigma(\mathtt{T}).$$

**Proof:**
We just show the new cases here. Other cases can be found in [22].

<u>**Case** T-Refine:</u>

$$\frac{\begin{array}{c}\Delta_1, \Delta', \Delta_2 \vdash \mathtt{T} \text{ ok} \qquad \Delta_1, \Delta', \Delta_2 \vdash \mathtt{U} \text{ ok} \qquad \Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \\ (1 \le i \le |\overline{\mathtt{T}}|) \quad \Delta_i \vdash \mathtt{T}_i \text{ minok} \quad \Delta_1, \Delta', \Delta_2, \Delta_i, \mathtt{T} \lll: \mathtt{T}_i; \Gamma \vdash \mathtt{e}_i \in \mathtt{U}_i <: \mathtt{U}\end{array}}{\Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{typematch}\ \mathtt{T}\ \mathtt{with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \in \mathtt{U}} \text{[T-Refine]}$$

By Lemma A.14 (Wellformedness),

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T}) \text{ ok and } \Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{U}) \text{ ok}$$

By induction hypothesis and the rule S-Trans,

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}') \in \mathtt{U}'' <: \Sigma(\mathtt{U})$$

$$\Delta_1, \Sigma(\Delta_2), \Sigma(\Delta_i), \Sigma(\mathtt{T}) \lll: \Sigma(\mathtt{T}_i); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}_i) \in \mathtt{U}_i' <: \Sigma(\mathtt{U})$$

We assume $dom(\Delta') \cap dom(\Delta_i) = \varnothing$ by writing $\Delta_1, \Delta', \Delta_2, \Delta_i, \mathtt{T} \lll: \mathtt{T}_i$. With $dom(\Sigma) = dom(\Delta')$ we know

$$dom(\Sigma) \cap dom(\Delta_i) = \varnothing$$

By Lemma A.15 (Minimal Context), we have $\Sigma(\Delta_i) \vdash \Sigma(\mathtt{T}_i)$ minok.
Finally by applpying the rule T-Refine again we conclude with

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \mathtt{typematch}\ \Sigma(\mathtt{T})\ \mathtt{with}\ \Sigma(\overline{\mathtt{T}}) : \Sigma(\overline{\mathtt{e}})\ \ \mathtt{default} : \Sigma(\mathtt{e}') \in \mathtt{S} <: \Sigma(\mathtt{U})$$

<u>**Case** T-FieldFold:</u>

$$\frac{\begin{array}{c} i > 0 \qquad \Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \qquad \Delta_1, \Delta', \Delta_2 \vdash \mathtt{T}' \text{ ok} \\ \Delta'' \vdash \mathtt{T} \text{ minok} \qquad \Delta_1, \Delta', \Delta_2, \Delta', \mathtt{T}' \lll: \{\mathtt{T}\ \mathtt{f_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U}\end{array}}{\Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{T}')\ \mathtt{e}' \in \mathtt{U}} \text{[T-FieldFold]}$$

By induction hypothesis and the rule S-Trans,

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \Sigma(\mathtt{U}'') <: \Sigma(\mathtt{U})$$

$$\Delta_1, \Sigma(\Delta_2), \Sigma(\Delta''), \Sigma(\mathtt{T}') \lll: \{\Sigma(\mathtt{T})\ \mathtt{f_x}\}; \Sigma(\Gamma), x : \Sigma(\mathtt{U}) \vdash \Sigma(\mathtt{e}') \in \Sigma(\mathtt{U}') <: \Sigma(\mathtt{U})$$

By Lemma A.14 (Wellformedness), we have

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T}) \text{ ok}$$

Lemma A.15 (Minimal Context), we have

$$\Sigma(\Delta'') \vdash \Sigma(\mathtt{T}) \text{ minok}$$

Finally by applying the rule T-FieldFold again we have

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \mathtt{fieldfold}_i(\mathtt{x} = \Sigma(\mathtt{e}); \Sigma(\mathtt{T})\ \mathtt{f_x} \in \Sigma(\mathtt{T'}))\ \Sigma(\mathtt{e'}) \in \mathtt{S} <: \Sigma(\mathtt{U})$$

**Case** T-MethFold:

$$\frac{\begin{array}{c} i > 0 \qquad \Delta'' \vdash \mathtt{MT} \text{ minok} \qquad \Delta_1, \Delta', \Delta_2 \vdash \mathtt{T} \text{ ok} \\ \Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{U''} <: \mathtt{U} \qquad \Delta_1, \Delta', \Delta_2, \Delta'', \mathtt{T} \ll: \{\mathtt{MT}\ \mathtt{m_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e'} \in \mathtt{U'} <: \mathtt{U} \end{array}}{\Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e'} \in \mathtt{U}} \text{[T-MethFold]}$$

By induction hypothesis and the rule S-Trans,

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \Sigma(\mathtt{U''}) <: \Sigma(\mathtt{U})$$

$$\Delta_1, \Sigma(\Delta_2), \Sigma(\Delta''), \Sigma(\mathtt{T}) \ll: \{\Sigma(\mathtt{MT})\ \mathtt{f_x}\}; \Sigma(\Gamma), x : \Sigma(\mathtt{U}) \vdash \Sigma(\mathtt{e'}) \in \Sigma(\mathtt{U'}) <: \Sigma(\mathtt{U})$$

By Lemma A.14 (Wellformedness), we have

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T}) \text{ ok}$$

By Lemma A.15 (Minimal Context), we have

$$\Sigma(\Delta'') \vdash \Sigma(\mathtt{MT}) \text{ minok}$$

Finally by applying the rule T-MethFold again we have

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \mathtt{methfold}_i(\mathtt{x} = \Sigma(\mathtt{e}); \Sigma(\mathtt{MT})\ \mathtt{m_x} \in \Sigma(\mathtt{T}))\ \Sigma(\mathtt{e'}) \in \mathtt{S} <: \Sigma(\mathtt{U})$$

**Case** T-FieldVar:

$$\frac{\Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T_0} \qquad \Delta_1, \Delta', \Delta_2 \vdash \mathtt{T_0} \ll: \{\mathtt{T}\ \mathtt{f_x}\}}{\Delta_1, \Delta', \Delta_2; \Gamma \vdash \mathtt{e}.\mathtt{f_x} \in \mathtt{T}} \text{[T-FieldVar]}$$

By induction hypothesis,

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{T}'_0 <: \Sigma(\mathtt{T_0})$$

By Lemma A.16 (Structural Refinement Preservation),

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T_0}) \ll: \{\Sigma(\mathtt{T})\ \mathtt{f_x}\}$$

And by the rule RF-Trans,

$$\Delta_1, \Sigma(\Delta_2) \vdash \mathtt{T}'_0 \ll: \{\Sigma(\mathtt{T})\ \mathtt{f_x}\}$$

Applying the rule T-FieldVar again, we get

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}).\mathtt{f_x} \in \Sigma(\mathtt{T})$$

By Lemma A.13 (Subtyping), we have $\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T}) <: \mathtt{T}$. So

$$\Delta_1, \Sigma(\Delta_2); \Sigma(\Gamma) \vdash \Sigma(\mathtt{e}).\mathtt{f_x} \in \Sigma(\mathtt{T}) <: \mathtt{T}$$

**Case** T-InvkVar: Similar to the above case.

**LEMMA A.13 (Type Substitution Preserves Subtyping)**

$$\left.\begin{array}{l}\Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \\ \Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} <: \mathtt{T} \\ \mathit{dom}(\Sigma) = \mathit{dom}(\Delta') \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) <: \Sigma(\Delta'(\overline{\mathtt{X}})) \\ \mathit{FV}(\Delta_1) \cap \mathit{dom}(\Sigma) = \varnothing \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) \text{ ok}\end{array}\right\} \Longrightarrow \Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{S}) <: \Sigma(\mathtt{T}).$$

**Proof:** By induction on the derivation of $\Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} <: \mathtt{T}$.
We show the only new case below. For other cases, see [22].

**Case** S-Refine:

$$\frac{\mathtt{S} \lll: \mathtt{T} \in \Delta_1, \Delta', \Delta_2}{\Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} <: \mathtt{T}} \text{ [S-Refine]}$$

**Subcase** $\mathtt{S} \lll: \mathtt{T} \in \Delta_1$:
Since $\mathit{FV}(\Delta_1) \cap \mathit{dom}(\Sigma) = \varnothing$,

$$\Sigma(\mathtt{S}) = \mathtt{S} \text{ and } \Sigma(\mathtt{T}) = \mathtt{T}$$

So $\Sigma(\mathtt{S}) \lll: \Sigma(\mathtt{T}) \in \Delta_1$ and thus $\Sigma(\mathtt{S}) \lll: \Sigma(\mathtt{T}) \in \Delta_1, \Sigma(\Delta_2)$.
Then by the rule S-Refine again,

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{S}) <: \Sigma(\mathtt{T})$$

**Subcase** $\mathtt{S} \lll: \mathtt{T} \in \Delta'$: Impossible since $\Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}}$.

**Subcase** $\mathtt{S} \lll: \mathtt{T} \in \Delta_2$:
So $\Sigma(\mathtt{S} \lll: \mathtt{T}) \in \Sigma(\Delta_2)$ and thus $\Sigma(\mathtt{S}) \lll: \Sigma(\mathtt{T}) \in \Sigma(\Delta_2)$ and also $\Sigma(\mathtt{S}) \lll: \Sigma(\mathtt{T}) \in \Delta_1, \Sigma(\Delta_2)$.
Then by the rule S-Refine again,

$$\Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{S}) <: \Sigma(\mathtt{T})$$

**LEMMA A.14 (Type Substitution Preserves Wellformedness)**

$$\left.\begin{array}{l}\Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \\ \Delta_1, \Delta', \Delta_2 \vdash \mathtt{T} \text{ ok} \\ \mathit{dom}(\Sigma) = \mathit{dom}(\Delta') \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) <: \Sigma(\Delta'(\overline{\mathtt{X}})) \\ \mathit{FV}(\Delta_1) \cap \mathit{dom}(\Sigma) = \varnothing \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) \text{ ok}\end{array}\right\} \Longrightarrow \Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{T}) \text{ ok}.$$

**Proof:** See [22].

**LEMMA A.15 (Type Substitution Preserves Minimal Context)**

1. If $\Delta \vdash \mathtt{T}$ minok and $\mathit{dom}(\Sigma) \cap \mathit{dom}(\Delta) = \varnothing$ then $\Sigma(\Delta) \vdash \Sigma(\mathtt{T})$ minok.
2. If $\Delta \vdash \mathtt{MT}$ minok and $\mathit{dom}(\Sigma) \cap \mathit{dom}(\Delta) = \varnothing$ then $\Sigma(\Delta) \vdash \Sigma(\mathtt{MT})$ minok.

**Proof:**

1. By the definition of $\Delta \vdash \mathtt{T}$ minok we have

$$FV(\mathtt{T}) = dom(\Delta) \ \text{ and } \ \Delta(\mathtt{X}) = \mathtt{Object}$$

Since $dom(\Sigma) \cap dom(\Delta) = \varnothing$ we have

$$\Sigma(\Delta) = \Delta \ \text{ and } \ \Sigma(\mathtt{T}) = \mathtt{T}$$

So still $\Sigma(\Delta) \vdash \Sigma(\mathtt{T})$ minok.

2. Similar to the above.

**LEMMA A.16 (Type Substitution Preserves Structural Refinement)**

1.

$$\left.\begin{array}{l} \Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \\ \Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} \lll: \{\mathtt{T}\ \mathtt{f}_{\mathtt{x}}\} \\ dom(\Sigma) = dom(\Delta') \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) <: \Sigma(\Delta'(\overline{\mathtt{X}})) \\ FV(\Delta_1) \cap dom(\Sigma) = \varnothing \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) \text{ ok} \end{array}\right\} \Longrightarrow \Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{S}) \lll: \{\Sigma(\mathtt{T})\ \mathtt{f}_{\mathtt{x}}\}$$

2.

$$\left.\begin{array}{l} \Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \\ \Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} \lll: \{\mathtt{MT}\ \mathtt{m}_{\mathtt{x}}\} \\ dom(\Sigma) = dom(\Delta') \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) <: \Sigma(\Delta'(\overline{\mathtt{X}})) \\ FV(\Delta_1) \cap dom(\Sigma) = \varnothing \\ \Delta_1 \vdash \Sigma(\overline{\mathtt{X}}) \text{ ok} \end{array}\right\} \Longrightarrow \Delta_1, \Sigma(\Delta_2) \vdash \Sigma(\mathtt{S}) \lll: \{\Sigma(\mathtt{MT})\ \mathtt{m}_{\mathtt{x}}\}$$

**Proof:**

1. By straightforward induction on $\Delta_1, \Delta', \Delta_2 \vdash \mathtt{S} \lll: \{\mathtt{T}\ \mathtt{f}_{\mathtt{x}}\}$.

   **Case** RF-Hyp: Immediate.

   **Case** RF-Trans: Easy, by induction hypothesis and Lemma A.13 (Subtyping) and the rule RF-Trans again.

2. Similar to the above.

**LEMMA A.17 (Term Substitution)** If $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}$ and $\Delta; \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}} <: \overline{\mathtt{T}}$, then

$$\Delta; \Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} \in \mathtt{S} <: \mathtt{T}.$$

We just show the 5 new cases here.

**Case** T-REFINE:

$$\frac{\Delta;\Gamma \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \qquad (1 \leq i \leq |\overline{\mathtt{T}}|) \quad \begin{array}{c}\Delta \vdash \mathtt{T}\ \text{ok} \qquad \Delta \vdash \mathtt{U}\ \text{ok} \\ \Delta_i \vdash \mathtt{T}_i\ \text{minok} \quad \Delta, \Delta_i, \mathtt{T} \lll: \mathtt{T}_i; \Gamma \vdash \mathtt{e}_i \in \mathtt{U}_i <: \mathtt{U}\end{array}}{\Delta;\Gamma \vdash \mathtt{typematch\ T\ with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \in \mathtt{U}} \text{[T-REFINE]}$$

By induction hypothesis and the rule S-TRANS,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e}' \in \mathtt{U}'' <: \mathtt{U}$$

Similarly,

$$\Delta, \Delta_i, \mathtt{T} \lll: \mathtt{T}_i; \Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e}_i \in \mathtt{U}_i' <: \mathtt{U}$$

By applying the rule T-REFINE again, we have

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}](\mathtt{typematch\ T\ with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}') \in \mathtt{U} <: \mathtt{U}$$

**Case** T-FIELDFOLD:

$$\frac{\Delta \vdash \mathtt{T}'\ \text{ok} \qquad \Delta' \vdash \mathtt{T}\ \text{minok} \qquad \begin{array}{c} i > 0 \qquad \Delta;\Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \\ \Delta, \Delta', \mathtt{T}' \lll: \{\mathtt{T\ f_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U}\end{array}}{\Delta;\Gamma \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{T\ f_x} \in \mathtt{T}')\ \mathtt{e}' \in \mathtt{U}} \text{[T-FIELDFOLD]}$$

By induction hypothesis and the rule S-TRANS,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} \in \mathtt{S}'' <: \mathtt{U}$$

Similarly,

$$\Delta, \Delta', \mathtt{T}' \lll: \{\mathtt{T\ f_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e}' \in \mathtt{S}' <: \mathtt{U}'$$

Applying the rule T-FIELDFOLD again finishes the case.

**Case** T-METHFOLD:

$$\frac{\Delta \vdash \mathtt{T}\ \text{ok} \qquad \Delta;\Gamma \vdash \mathtt{e} \in \mathtt{U}'' <: \mathtt{U} \qquad \begin{array}{c} i > 0 \qquad \Delta' \vdash \mathtt{MT}\ \text{minok} \\ \Delta, \Delta', \mathtt{T} \lll: \{\mathtt{MT\ m_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U}\end{array}}{\Delta;\Gamma \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}; \mathtt{MT\ m_x} \in \mathtt{T})\ \mathtt{e}' \in \mathtt{U}} \text{[T-METHFOLD]}$$

By induction hypothesis and the rule S-TRANS,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} \in \mathtt{S}'' <: \mathtt{U}$$

Similarly,

$$\Delta, \Delta', \mathtt{T} \lll: \{\mathtt{MT\ m_x}\}; \Gamma, \mathtt{x} : \mathtt{U} \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e}' \in \mathtt{S}' <: \mathtt{U}'$$

Applying the rule T-METHFOLD again finishes the case.

**Case** T-FIELDVAR:

$$\frac{\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \lll: \{\mathtt{T\ f_x}\}}{\Delta;\Gamma \vdash \mathtt{e.f_x} \in \mathtt{T}} \text{[T-FIELDVAR]}$$

By induction hypothesis,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} \in \mathtt{T}_0' <: \mathtt{T}_0$$

Then by the rule RF-TRANS (Structural Refinement, See Fig. 12),

$$\Delta \vdash \mathtt{T}_0' \lll: \{\mathtt{T\ f_x}\}$$

Finally by applying the rule T-FieldVar, we have

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}](\mathtt{e.f_x}) = ([\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e.f_x}) \in \mathtt{T} <: \mathtt{T}$$

**Case** T-InvkVar:

$$\frac{\Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \begin{array}{c}\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T_0} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \\ \Delta \vdash \mathtt{T_0} \lll: \{(\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}\})\mathtt{m_x}\end{array} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{e.m_x<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}}\ [\text{T-InvkVar}]$$

By induction hypothesis,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} \in \mathtt{T_0'} <: \mathtt{T_0}$$

Then by the rule RM-Trans (Structural Refinement, See Fig. 12),

$$\Delta \vdash \mathtt{T_0'} \lll: \{(\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>U} \to \overline{\mathtt{U}}\})\mathtt{m_x}$$

By induction hypothesis and the rule S-Trans,

$$\Delta;\Gamma \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\overline{\mathtt{e}} \in \overline{\mathtt{V}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}$$

Finally by applying the rule T-MethVar, we have

$$\Delta;\Gamma \vdash ([\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e}).\mathtt{m_x<}\overline{\mathtt{T}}\mathtt{>}([\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\overline{\mathtt{e}}) = [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}](\mathtt{e.m_x<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}})) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}$$

**LEMMA A.18 (Substitution from Matching)**

1. If $\varnothing \vdash \mathtt{N}$ ok and *matches*$(\mathtt{N},\mathtt{T}) = \Sigma$ and $\Delta \vdash \mathtt{T}$ minok and $\Delta,\Delta';\Gamma \vdash \mathtt{e} \in \mathtt{U} <: \mathtt{S}$ then $\Sigma(\Delta');\Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{U}' <: \Sigma(\mathtt{S})$
2. If $\varnothing \vdash \mathtt{MT}$ ok and *matches*$(\mathtt{MT},\mathtt{MT}') = \Sigma$ and $\Delta \vdash \mathtt{MT}'$ minok and $\Delta,\Delta';\Gamma \vdash \mathtt{e} \in \mathtt{U} <: \mathtt{S}$ then $\Sigma(\Delta');\Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{U}' <: \Sigma(\mathtt{S})$

**Proof:**

1. By *matches*$(\mathtt{N},\mathtt{T}) = \Sigma$ and $\varnothing \vdash \mathtt{N}$ ok and Lemma A.3 (Matches), we have

$$dom(\Sigma) = FV(\mathtt{T}) \ \text{ and } \ \varnothing \vdash \Sigma(\mathtt{T}) \text{ ok.}$$

By $\Delta \vdash \mathtt{T}$ minok, we have

$$dom(\Delta) = FV(\mathtt{T}) \ \text{ and } \ \Delta(\mathtt{X}) = \mathtt{Object}.$$

Therefore dom$(\Sigma) = dom(\Delta)$ and $\varnothing \vdash \Sigma(\mathtt{X}) <: \Delta(\mathtt{X}) = \mathtt{Object}$ and thus $\varnothing \vdash \Sigma(\mathtt{X}) <: \Sigma(\Delta(\mathtt{X}))$.
We also have the trivial $dom(\Sigma) \cap \varnothing = \varnothing$.

Now we can apply Lemma A.12 (Type-Substituion) to $\Delta,\Delta';\Gamma \vdash \mathtt{e} \in \mathtt{U}$, getting

$$\Sigma(\Delta');\Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{U}' <: \Sigma(\mathtt{U})$$

We can also apply Lemma A.13 (Subtyping) to $\Delta,\Delta' \vdash \mathtt{U} <: \mathtt{S}$, getting

$$\Sigma(\Delta') \vdash \Sigma(\mathtt{U}) <: \Sigma(\mathtt{S})$$

And then by the rule S-Trans,

$$\Sigma(\Delta');\Sigma(\Gamma) \vdash \Sigma(\mathtt{e}) \in \mathtt{U}' <: \Sigma(\mathtt{S})$$

2. By $\mathit{matches}(\texttt{MT}, \texttt{MT}') = \Sigma$ and $\varnothing \vdash \texttt{MT}$ ok and Lemma A.3 (Matches), we have

$$\mathit{dom}(\Sigma) = \mathit{FV}(\texttt{MT}') \quad \text{and} \quad \varnothing \vdash \Sigma(\texttt{MT}') \text{ ok}.$$

By $\Delta \vdash \texttt{MT}'$ minok, we have

$$\mathit{dom}(\Delta) = \mathit{FV}(\texttt{MT}') \quad \text{and} \quad \Delta(\texttt{X}) = \texttt{Object}.$$

Therefore $\mathrm{dom}(\Sigma) = \mathit{dom}(\Delta)$ and $\varnothing \vdash \Sigma(\texttt{X}) <: \Delta(\texttt{X}) = \texttt{Object}$ and thus $\varnothing \vdash \Sigma(\texttt{X}) <: \Sigma(\Delta(\texttt{X}))$. We also have the trivial $\mathit{dom}(\Sigma) \cap \varnothing = \varnothing$.

Now we can apply Lemma A.12 (Type-Substituion) to $\Delta, \Delta'; \Gamma \vdash \texttt{e} \in \texttt{U}$ , getting

$$\Sigma(\Delta'); \Sigma(\Gamma) \vdash \Sigma(\texttt{e}) \in \texttt{U}' <: \Sigma(\texttt{U})$$

We can also apply Lemma A.13 (Subtyping) to $\Delta, \Delta' \vdash \texttt{U} <: \texttt{S}$, getting

$$\Sigma(\Delta') \vdash \Sigma(\texttt{U}) <: \Sigma(\texttt{S})$$

And then by the rule S-Trans,

$$\Sigma(\Delta'); \Sigma(\Gamma) \vdash \Sigma(\texttt{e}) \in \texttt{U}' <: \Sigma(\texttt{S})$$

**LEMMA A.19 (Preservation)** If $\varnothing; \varnothing \vdash \texttt{e} \in \texttt{N}$ and $\texttt{e} \mapsto \texttt{e}'$, then $\varnothing; \varnothing \vdash \texttt{e}' \in \texttt{P} <: \texttt{N}$.

**Proof:** By induction on the typing derivation $\varnothing; \varnothing \vdash \texttt{e} \in \texttt{N}$, with a case analysis on the last rule used. We just show the new cases here. Other cases can be found in [22].

**Case** T-Refine:

$$\frac{\varnothing; \varnothing \vdash \texttt{e}' \in \texttt{U}' <: \texttt{U} \qquad \begin{array}{c} \varnothing \vdash \texttt{N} \text{ ok} \qquad \varnothing \vdash \texttt{U} \text{ ok} \\ (1 \le i \le |\overline{\texttt{T}}|) \quad \Delta_i \vdash \texttt{T}_i \text{ minok} \quad \Delta_i, \texttt{N} \lll: \texttt{T}_i; \varnothing \vdash \texttt{e}_i \in \texttt{U}_i <: \texttt{U} \end{array}}{\varnothing; \varnothing \vdash \texttt{typematch N with } \overline{\texttt{T}} : \overline{\texttt{e}} \texttt{ default} : \texttt{e}' \in \texttt{U}} \text{[T-Refine]}$$

By looking at the evaluation rules (Fig. 8), there are 3 possible cases that $\texttt{e}$ takes a step:

**Subcase** E-Match:

$$\frac{\mathit{matches}(\texttt{N}, \texttt{T}_1) = \Sigma_1}{\texttt{typematch N with } \texttt{T}_1 : \texttt{e}_1 \ \ \overline{\texttt{T}} : \overline{\texttt{e}} \texttt{ default} : \texttt{e}' \mapsto \Sigma_1(\texttt{e}_1)} \text{[E-Match]}$$

We know (from inversion) that

$$\Delta_1, \texttt{N} \lll: \texttt{T}_1; \varnothing \vdash \texttt{e}_1 \in \texttt{U}_1 <: \texttt{U}$$

From this and $\mathit{matches}(\texttt{N}, \texttt{T}_1) = \Sigma_1$ and $\varnothing \vdash \texttt{N}$ ok and $\Delta_1 \vdash \texttt{T}_1$ minok, we apply Lemma A.18 (Substitution from Matching),

$$\Sigma_1(\texttt{N} \lll: \texttt{T}_1); \varnothing \vdash \Sigma_1(\texttt{e}_1) \in \texttt{U}_1' <: \Sigma_1(\texttt{U})$$

and since $\varnothing \vdash \texttt{U}$ ok, we have $\Sigma_1(\texttt{U}) = \texttt{U}$ and therefore

$$\Sigma_1(\texttt{N} \lll: \texttt{T}_1); \varnothing \vdash \Sigma_1(\texttt{e}_1) \in \texttt{U}_1' <: \texttt{U}$$

By $matches(\mathtt{N}, \mathtt{T}_1) = \Sigma_1$ and Lemma A.3 (Matches), we have $\varnothing \vdash \mathtt{N} <: \Sigma_1(\mathtt{T}_1)$.
Since $\varnothing \vdash \Sigma_1(\mathtt{T}_1)$ ok by Lemma A.11 (Strengthening),

$$\varnothing; \varnothing \vdash \Sigma_1(\mathtt{e}_1) \in \mathtt{U}_1'' <: \mathtt{U}$$

**Subcase** E-NoMatch:

$$\frac{matches(\mathtt{N}, \mathtt{T}) \text{ is not defined}}{\mathtt{typematch\ N\ with\ T : e\ \ \overline{T} : \overline{e}\ \ default : e'} \mapsto \mathtt{typematch\ N\ with\ \overline{T} : \overline{e}\ \ default : e'}} \text{[E-NoMatch]}$$

immediate by the preconditions of the rule T-Refine and by applying the rule T-Refine again.

**Subcase** E-Default:

$$\frac{}{\mathtt{typematch\ N\ with\ \ \ default : e'} \mapsto \mathtt{e'}} \text{[R-Default]}$$

from the preconditions of the rule T-Refine we know $\varnothing; \varnothing \vdash \mathtt{e}' \in \mathtt{U}$. immediate.

**<u>Case</u>** <u>T-FieldFold:</u>

$$\frac{\begin{array}{c} i > 0 \qquad \varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{U}'' <: \mathtt{U} \\ \varnothing \vdash \mathtt{N} \text{ ok} \qquad \Delta' \vdash \mathtt{T} \text{ minok} \qquad \Delta', \mathtt{N} \lll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\}; \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\varnothing; \varnothing \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{N})\ \mathtt{e}' \in \mathtt{U}} \text{[T-FieldFold]}$$

By looking at the evaluation rules (Fig. 10), there are 4 possible rules that $\mathtt{e}$ takes a step (but only the first case is interesting):

**Subcase** E-FFMatch:

$$\frac{fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad 1 \le i \le |\overline{\mathtt{f}}| \qquad matches(\mathtt{T}_i, \mathtt{T}) = \Sigma}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{N})\ \mathtt{e}' \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{f}_\mathtt{x} \mapsto \mathtt{f}_i]\Sigma(\mathtt{e}'); \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{N})\ \mathtt{e}'} \text{[E-FFMatch]}$$

From inversion we know that

$$\Delta', \mathtt{N} \lll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\}; \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \tag{1}$$

By $\varnothing \vdash \mathtt{N}$ ok and $fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, we have $\varnothing \vdash \mathtt{T}_i$ ok.
From this and $matches(\mathtt{N}, \mathtt{T}) = \Sigma$ and $\Delta' \vdash \mathtt{T}$ minok, we apply Lemma A.18 (Substitution from Matching) to (1)

$$\Sigma(\mathtt{N} \lll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\}); \mathtt{x} : \Sigma(\mathtt{U}) \vdash \Sigma(\mathtt{e}') \in \mathtt{S} <: \Sigma(\mathtt{U})$$

Since $\varnothing \vdash \mathtt{N}$ ok and $\varnothing \vdash \mathtt{U}$ ok, this is equal to

$$\mathtt{N} \lll: \{\Sigma(\mathtt{T})\ \mathtt{f}_\mathtt{x}\}; \mathtt{x} : \mathtt{U} \vdash \Sigma(\mathtt{e}') \in \mathtt{S} <: \mathtt{U}$$

From $\varnothing \vdash \mathtt{T}_i$ ok and $matches(\mathtt{T}_i, \mathtt{T}) = \Sigma$ and Lemma A.3 (Matches) we know $\varnothing \vdash \mathtt{T}_i <: \Sigma(\mathtt{T})$. With $fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, by Lemma A.8 (Field Substitution) and the rule S-Trans, we have

$$\varnothing; \mathtt{x} : \mathtt{U} \vdash [\mathtt{f}_\mathtt{x} \mapsto \mathtt{f}_\mathtt{i}]\Sigma(\mathtt{e}') \in \mathtt{S}' <: \mathtt{U}$$

Since $\varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{U}'' <: \mathtt{U}$, by Lemma A.17 (Term substitution) and the rule S-Trans, we conclude with

$$\varnothing; \varnothing \vdash [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{f}_\mathtt{x} \mapsto \mathtt{f}_\mathtt{i}]\Sigma(\mathtt{e}') \in \mathtt{S}'' <: \mathtt{U}$$

Finally, by applying the rule T-FieldFold again, we have

$$\varnothing; \varnothing \vdash \mathtt{fieldfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{f}_\mathtt{x} \mapsto \mathtt{f}_i]\Sigma(\mathtt{e}'); \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{N})\ \mathtt{e}' \in \mathtt{U} <: \mathtt{U}$$

**Subcase** E-FFSkip:

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\;\overline{\mathtt{f}} \qquad 1 \le i \le |\overline{\mathtt{f}}| \qquad \mathit{matches}(\mathtt{T}_i, \mathtt{T}) \text{ is not defined}}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}}\ \text{[E-FFSkip]}$$

Immediate by simply applying the rule T-FieldFold again.

**Subcase** E-FFCong:

$$\frac{\mathtt{e}_0 \mapsto \mathtt{e}_0'}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}' \mapsto \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0'; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e}'}\ \text{[E-FFCong]}$$

Easy. By induction hypothesis, the rule S-Trans, and applying the rule T-FieldFold again.

**Subcase** E-FFBase:

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\;\overline{\mathtt{f}} \qquad i > |\overline{\mathtt{f}}|}{\mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}}\ \text{[E-FFBase]}$$

From inversion of the rule T-FieldFold, we $\varnothing; \varnothing \vdash \mathtt{v} \in \mathtt{U}'' <: \mathtt{U}$. Immediate.

<u>**Case** T-MethFold:</u>

$$\frac{\begin{array}{c} i > 0 \qquad \Delta' \vdash \mathtt{MT}\ \text{minok} \\ \varnothing \vdash \mathtt{N}\ \text{ok} \qquad \varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{U}'' <: \mathtt{U} \qquad \Delta', \mathtt{N} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}; \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \end{array}}{\varnothing; \varnothing \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}' \in \mathtt{U}}\ \text{[T-MethFold]}$$

By looking at the evaluation rules (Fig. 10), there are 4 possible rules that $\mathtt{e}$ takes a step (but only the first case is interesting):

**Subcase** E-MFMatch:

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad \mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) = \Sigma}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{m_x} \mapsto m_i]\Sigma(\mathtt{e}); \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}}\ \text{[E-MFMatch]}$$

From inversion we know that

$$\Delta', \mathtt{N} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}; \mathtt{x} : \mathtt{U} \vdash \mathtt{e}' \in \mathtt{U}' <: \mathtt{U} \tag{2}$$

By $\varnothing \vdash \mathtt{N}$ ok and $\mathit{mtype}(\mathtt{m_i}, \mathtt{N}) = \mathtt{MT}_i$, we have $\varnothing \vdash \mathtt{MT}_i$ ok.

From this and $\mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) = \Sigma$ and $\Delta' \vdash \mathtt{MT}$ minok, we apply Lemma A.18 (Substitution from Matching) to (2)

$$\Sigma(\mathtt{N} \lll: \{\mathtt{MT}\ \mathtt{m_x}\}); \mathtt{x} : \Sigma(\mathtt{U}) \vdash \Sigma(\mathtt{e}') \in \mathtt{S} <: \Sigma(\mathtt{U})$$

Since $\varnothing \vdash \mathtt{N}$ ok and $\varnothing \vdash \mathtt{U}$ ok, this is equal to

$$\mathtt{N} \lll: \{\Sigma(\mathtt{MT})\ \mathtt{m_x}\}; \mathtt{x} : \mathtt{U} \vdash \Sigma(\mathtt{e}') \in \mathtt{S} <: \mathtt{U}$$

From $\varnothing \vdash \mathtt{MT}_i$ ok and $\mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) = \Sigma$ and Lemma A.3 (Matches) we know $\varnothing \vdash \mathtt{MT}_i <: \Sigma(\mathtt{MT})$. With $\mathit{mtype}(\mathtt{m_i}, \mathtt{N}) = \mathtt{MT}_i$, by Lemma A.8 (Method Substitution) and the rule S-Trans, we have

$$\varnothing; \mathtt{x} : \mathtt{U} \vdash [\mathtt{m_x} \mapsto \mathtt{m_i}]\Sigma(\mathtt{e}') \in \mathtt{S}' <: \mathtt{U}$$

Since $\varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{U}'' <: \mathtt{U}$, by Lemma A.17 (Term substitution) and the rule S-Trans, we conclude with

$$\varnothing; \varnothing \vdash [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{m_x} \mapsto \mathtt{m_i}]\Sigma(\mathtt{e}') \in \mathtt{S}'' <: \mathtt{U}$$

Finally, by applying the rule T-MethFold again, we have

$$\varnothing;\varnothing \vdash \mathtt{methfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{e}_0, \mathtt{m_x} \mapsto \mathtt{m_i}]\Sigma(\mathtt{e}); \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \in \mathtt{U} <: \mathtt{U}$$

**Subcase** E-MFSkip:

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i \qquad \mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) \text{ is not defined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e}}\ \text{[E-MFSkip]}$$

Immediate by simply applying the rule M-FieldFold again.

**Subcase** E-MFCong:

$$\frac{\mathtt{e}_0 \mapsto \mathtt{e}_0'}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e} \mapsto \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0'; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{T})\ \mathtt{e}}\ \text{[E-MFCong]}$$

Easy. By induction hypothesis, the rule S-Trans and applying the rule M -FieldFold again.

**Subcase** E-MFBase:

$$\frac{\mathit{mtype}(\mathtt{m}_i, \mathtt{N})\ \mathit{is\ undefined}}{\mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m_x} \in \mathtt{N})\ \mathtt{e} \mapsto \mathtt{v}}\ \text{[E-MFBase]}$$

From inversion of the rule T-MethFold, we $\varnothing;\varnothing \vdash \mathtt{v} \in \mathtt{U}'' <: \mathtt{U}$. Immediate.

<u>**Case** T-FieldVar:</u>

$$\frac{\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \lll: \{\mathtt{T}\ \mathtt{f_x}\}}{\Delta;\Gamma \vdash \mathtt{e.f_x} \in \mathtt{T}}\ \text{[T-FieldVar]}$$

Can't happen in closed form.

<u>**Case** T-InvkVar:</u>

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}_0 \qquad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \\ \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{P}} \qquad \Delta \vdash \mathtt{T}_0 \lll: \{(\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}\})\mathtt{m_x} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} <: [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e.m_x}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}(\mathtt{e}) \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{T}}]\mathtt{U}}\ \text{[T-InvkVar]}$$

Can't happen in closed form.

<u>**LEMMA**</u> **A.20 (Progress)** If $\varnothing;\varnothing \vdash \mathtt{e} \in \mathtt{N}$, then one of the following holds:

- $\mathtt{e}$ is a value
- $\mathtt{e} \mapsto \mathtt{e}'$ for some $\mathtt{e}'$
- $\mathtt{e}$ contains a failed cast

**Proof:**

By induction on the derivation of $\varnothing;\varnothing \vdash \mathtt{e} \in \mathtt{N}$, with a case analysis on the last rule used. We just show the new cases here. Other cases can be found in the proof of [22].

<u>**Case** T-Refine:</u>

$$\frac{\begin{array}{c}\varnothing \vdash \mathtt{P}\ \text{ok} \qquad \varnothing \vdash \mathtt{N}\ \text{ok} \\ \varnothing;\varnothing \vdash \mathtt{e}' \in \mathtt{N}' <: \mathtt{N} \qquad (1 \le i \le |\overline{\mathtt{T}}|) \quad \Delta_i \vdash \mathtt{T}_i\ \text{minok} \quad \Delta_i, \mathtt{P} \lll: \mathtt{T}_i; \varnothing \vdash \mathtt{e}_i \in \mathtt{N}_i <: \mathtt{N}\end{array}}{\varnothing;\varnothing \vdash \mathtt{typematch}\ \mathtt{P}\ \mathtt{with}\ \overline{\mathtt{T}} : \overline{\mathtt{e}}\ \ \mathtt{default} : \mathtt{e}' \in \mathtt{N}}\ \text{[T-Refine]}$$

**Subcase** $\overline{\mathtt{T}} : \overline{\mathtt{e}} = \varnothing$:
Now the rule R-Default applies and $\mathtt{e} \mapsto \mathtt{e}'$.

**Subcase** $\overline{\mathtt{T}} : \overline{\mathtt{e}} = \mathtt{T}_1 : \mathtt{e}_1 \ \ \overline{\mathtt{T}'} : \overline{\mathtt{e}'}$:

**Subsubcase** $\mathit{matches}(\mathtt{P}, \mathtt{T}_1) = \Sigma$:
Then the rule R-Match applies and $\mathtt{e} \mapsto \Sigma(\mathtt{e}_1)$.

**Subsubcase** $\mathit{matches}(\mathtt{P}, \mathtt{T}_1)$ is not defined:
Then the rule R-NoMatch applies and $\mathtt{e} \mapsto \mathtt{typematch}\ \mathtt{P}\ \mathtt{with}\ \ \overline{\mathtt{T}'} : \overline{\mathtt{e}'}\ \ \mathtt{default} : \mathtt{e}'$

<u>**Case** T-FieldFold:</u>

$$\frac{\begin{array}{c} i > 0 \qquad \varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{N}'' <: \mathtt{N} \\ \varnothing \vdash \mathtt{P}\ \text{ok} \qquad \Delta' \vdash \mathtt{T}\ \text{minok} \qquad \Delta', \mathtt{P} \lll: \{\mathtt{T}\ \mathtt{f}_\mathtt{x}\}; \mathtt{x} : \mathtt{N} \vdash \mathtt{e}' \in \mathtt{N} <: \mathtt{N} \end{array}}{\varnothing; \varnothing \vdash \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}' \in \mathtt{N}} \text{[T-FieldFold]}$$

By induction hypothesis, one of the following holds for $\mathtt{e}_0$:

- $\mathtt{e}_0$ is a value
- $\mathtt{e}_0 \mapsto \mathtt{e}_0'$
- $\mathtt{e}_0$ contains a failed cast

**Subcase** $\mathtt{e}_0 \mapsto \mathtt{e}_0'$:
Simply the rule E-FFCong applies and $\mathtt{e} \mapsto \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{e}_0'; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

**Subcase** $\mathtt{e}_0$ contains a failed cast:
Then $\mathtt{e}$ contains a failed cast.

**Subcase** $\mathtt{e}_0 = \mathtt{v}$:
$\mathtt{e} = \mathtt{fieldfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$.
Let $\mathit{fields}(\mathtt{P}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$.

**Subsubcase** $i > |\overline{\mathtt{f}}|$:
Then the rule E-FFBase applies and $\mathtt{e} \mapsto \mathtt{v}$.

**Subsubcase** $1 \leq i \leq |\overline{\mathtt{f}}|$ and $\mathit{matches}(\mathtt{T}_i, \mathtt{T}) = \Sigma$:
Then the rule E-FFMatch applies and $\mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{f}_\mathtt{x} \mapsto \mathtt{f}_i]\Sigma(\mathtt{e}'); \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

**Subsubcase** $1 \leq i \leq |\overline{\mathtt{f}}|$ and $\mathit{matches}(\mathtt{T}_i, \mathtt{T})$ is not defined:
Now the rule E-FFSkip applies and $\mathtt{e} \mapsto \mathtt{fieldfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{T}\ \mathtt{f}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

<u>**Case** T-FieldVar:</u>
Impossible for closed form terms.

<u>**Case** T-InvkVar:</u>
Impossible for closed form terms.

<u>**Case** T-MethFold:</u>

$$\frac{\begin{array}{c} i > 0 \qquad \varnothing; \varnothing \vdash \mathtt{e}_0 \in \mathtt{N}'' <: \mathtt{N} \\ \varnothing \vdash \mathtt{P}\ \text{ok} \qquad \Delta' \vdash \mathtt{T}\ \text{minok} \qquad \Delta', \mathtt{P} \lll: \{\mathtt{MT}\ \mathtt{m}_\mathtt{x}\}; \mathtt{x} : \mathtt{N} \vdash \mathtt{e}' \in \mathtt{N} <: \mathtt{N} \end{array}}{\varnothing; \varnothing \vdash \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0; \mathtt{MT}\ \mathtt{m}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}' \in \mathtt{N}} \text{[T-MethFold]}$$

By induction hypothesis, one of the following holds for $\mathtt{e}_0$:

- $\mathtt{e}_0$ is a value

- $\mathtt{e}_0 \mapsto \mathtt{e}_0'$
- $\mathtt{e}_0$ contains a failed cast

**Subcase** $\mathtt{e}_0 \mapsto \mathtt{e}_0'$:
Simply the rule E-MFCONG applies and $\mathtt{e} \mapsto \mathtt{methfold}_i(\mathtt{x} = \mathtt{e}_0'; \mathtt{MT}\ \mathtt{m}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

**Subcase** $\mathtt{e}_0$ CONTAINS A FAILED CAST:
Then $\mathtt{e}$ contains a failed cast.

**Subcase** $\mathtt{e}_0 = \mathtt{v}$:
$\mathtt{e} = \mathtt{methfold}_i(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$.

**Subsubcase** $\mathit{mtype}(\mathtt{m}_i, \mathtt{N})$ UNDEFINED:
Then the rule E-MFBASE applies and $\mathtt{e} \mapsto \mathtt{v}$.

**Subsubcase** $\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i$ AND $\mathit{matches}(\mathtt{MT}_i, \mathtt{MT}) = \Sigma$:
Then the rule E-MFMATCH applies and $\mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = [\mathtt{x} \mapsto \mathtt{v}, \mathtt{m}_\mathtt{x} \mapsto \mathtt{m}_\mathtt{i}]\Sigma(\mathtt{e}'); \mathtt{MT}\ \mathtt{m}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

**Subsubcase** $\mathit{mtype}(\mathtt{m}_i, \mathtt{N}) = \mathtt{MT}_i$ AND $\mathit{matches}(\mathtt{MT}_i, \mathtt{MT})$ UNDEFINED:
Now the rule E-MFSKIP applies and $\mathtt{e} \mapsto \mathtt{methfold}_{i+1}(\mathtt{x} = \mathtt{v}; \mathtt{MT}\ \mathtt{m}_\mathtt{x} \in \mathtt{P})\ \mathtt{e}'$

**Theorem A.21 (Type Soundness)** If $\varnothing; \varnothing \vdash \mathtt{e} \in \mathtt{N}$ then one of the following holds:

- $\mathtt{e}$ evaluates to a value.
- $\mathtt{e}$ diverges.
- $\mathtt{e}$ contains a failed cast.

**Proof:** The proof of type soundness is immediate from Preservation (A.19) and Progress (A.20).