Integrating Nominal and Structural Subtyping

Donna Malayeri and Jonathan Aldrich

May 2008
CMU-CS-08-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Nominal and structural subtyping each have their own strengths and weaknesses. Nominal subtyping allows programmers to explicitly express design intent, and, when types are associated with run time tags, enables run-time type tests and external method dispatch. On the other hand, structural subtyping is flexible and compositional, allowing unanticipated reuse. To date, nearly all object-oriented languages fully support one subtyping paradigm or the other.

In this paper, we describe a core calculus for a language that integrates the key aspects of nominal and structural subtyping in a unified framework. We have also merged the flexibility of structural subtyping with statically typechecked external methods, a novel combination. We prove type safety for this language and illustrate its practical utility through examples that are not easily expressed in other languages. Our work provides a clean foundation for the design of future languages that enjoy the benefits of both nominal and structural subtyping.

This research was supported in part by the U.S. Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," and NSF CAREER award CCF-0546550.

Keywords: Nominal subtyping, structural subtyping, external dispatch

1 Introduction

In a language with structural subtyping, a type *U* is a subtype of *T* if its methods and fields are a superset of *T*'s methods and fields. The interface of a class is simply its public fields and methods; there is no need to declare a separate interface type. In a language with nominal subtyping, on the other hand, *U* is a subtype of *T* if and only if it is *declared* to be. Accordingly, structural subtyping can be considered *intrinsic*, while nominal subtyping is *declarative*. Each kind of subtyping has its merits, but a formal model has not been developed for a language that integrates the two subtyping disciplines.

Structural subtyping offers a number of advantages. It is often more expressive than nominal subtyping, as subtyping relationships do not need to be declared ahead of time. It is compositional and intrinsic, existing outside of the mind of the programmer. This has the advantage of supporting unanticipated reuse—programmers don't have to plan for all possible scenarios. Additionally, structural subtyping is often more notationally succinct. Programmers can concisely express type requirements without having to define an entire subtyping hierarchy. In nominal systems, some situations may require multiple inheritance or an unnecessary proliferation of types; in a structural system, the desired subtyping properties just arise naturally from the base cases. Finally, structural subtyping is far superior in contexts where the structure of the data is of primary importance, such as in data persistent environments or distributed computing. In contrast, nominal subtyping can lead to unnecessary versioning problems: if some class *C* is modified to C' (perhaps to add a method m), C' objects cannot be serialized and sent to a distributed process with the original definition *C*, even if C' is a strict extension of *C*.

As an example of the reuse benefits of structural subtyping, suppose class *A* has methods `foo()`, `a()` and `b()`, and class *B* has methods `foo()`, `x()` and `y()`. Suppose also that the code for *A* and *B* cannot be modified. In a language with structural subtyping, *A* and *B* share an implicit common interface { `foo` } and a programmer can write code against this interface. But, in a language with nominal subtyping, since the programmer did not plan ahead and create an `IFoo` interface and make *A* and *B* its subtypes, there is no way to write code that takes advantage of this commonality (short of using reflection). In contrast, with structural subtyping, if a class *C* is later added that contains method `foo()`, it too will share this implicit interface. If a programmer adds new methods to *A*, *A*'s interface type will change automatically, without the programmer having to maintain the interface himself. If *B* or *C* also contain these new methods, the implicit combined interfaces will automatically exist.

Nominal subtyping also has its advantages. First, it allows the programmer to express and enforce design intent explicitly. A programmer's defined subtyping hierarchy serves as checked documentation that specifies how the various parts of a program are intended to work together. Second, explicit specification has the advantage of preventing "accidental" subtyping relationships, such as the standard example of `cowboy.draw()` and `circle.draw()`. Nominal subtyping also allows recursive types to be easily and transparently defined, since recursion can simply go through the declared names. Third, error messages are usually much more comprehensible, since, for the most part, every type in a type error is one that the programmer has defined explicitly. Finally, nominal subtyping enables efficient implementation of external dispatch.

External dispatch is provided by number of statically typed languages, such as Cecil [7, 8], MultiJava [9], among others. External methods increase the flexibility and evolvability of code because they do not fix in advance the set of methods of a class. Consider the example of a class hierarchy that represents AST nodes. (This motivating example is expanded further in Sec. 2.3.)

Suppose this is part of a larger system, which includes an IDE for editing elements represented by this AST. Now suppose a programmer wishes to add new functionality to the IDE but cannot modify the original source code for the AST nodes. The new function provides the capability to jump from one node to a node that it references; this differs depending on what type of node is selected. Clearly, this functionality cannot be written without code that somehow performs dispatch on the AST class hierarchy.

In a language without external dispatch, the developer has limited choices. Usually, she must resort to manually writing `instanceof` tests, which is tedious and error-prone. In particular, if a new element is added to the AST hierarchy, the implementation will not behave correctly.

If the developers of the original class hierarchy *anticipated* this need and implemented the Visitor design pattern, it would then be easy to add new operations to the hierarchy, but then it would be difficult to add new classes. At best, Visitor trades one problem for another.

On the other hand, in a language with external dispatch, a programmer simply writes an *external method* that dispatches on the AST class hierarchy (i.e., separate from its code). External dispatch makes it easy to adapt existing code to new interfaces, since new code can be added as an external method. Exhaustiveness checking rules for external methods ensure that when a new class is added to the hierarchy, in the absence of an inherited method, a new method must be added for that class.

Nominal subtyping enables efficient external dispatch since there is a name on which to tie the dispatch. Additionally, if external dispatch were allowed on structural types, the problem of accidental subtyping would be exacerbated, since overridden methods would apply wherever there was a structural match. Further, ambiguity problems could frequently arise, which would have to be manually resolved by the programmer. Consider, for example, a method *m* defined on objects with a `foo:int` field. If *m* is also later defined for objects with a `bar:char` field, *m* is now ambiguous—which method is called for an object with both fields?

In our language, Unity, we sidestep this issue—nominal and structural subtyping are integrated. This makes efficient external dispatch compatible with structural subtyping, but also gives programmers the benefits of both subtyping disciplines. Nominal subtyping gives programmers the ability to express explicit design intent, while structural subtyping makes interfaces easier to maintain and reuse.

Contributions. The contributions of this paper are as follows:

- A language design, Unity, that provides user-defined and structural subtyping relationships in a novel and uniform way. Unity combines the flexibility of external dispatch with the conceptual clarity of width and depth subtyping.
- A formalization of the design of Unity, along with proofs of type safety (Sec. 5).
- Examples that illustrate the expressiveness and flexibility of the language (Sec. 2), We contrast Unity with other languages in Sec. 2.1.
- A case study (Sec. 3) and an empirical study of several Java programs (Sec. 4).

2 Motivating Examples

We give, by example, the intuition behind Unity and illustrate combining structural subtyping with external methods. In Unity, an object type is a record type tagged with a *brand*. Brands induce the nominal subtyping relation, which we call "sub-branding."[1] Brands are nominal in that the user defines the sub-brand relationship, like the subclass relation in languages like Java, Eiffel, and C++.

When a brand β is defined, the programmer lists the fields that any objects tagged with β will include. In other words, if the user defines the brand `Point` as having the fieldss $\{$`x : int, y : int`$\}$, then any value tagged with `Point` must include at least the labels `x` and `y` (with `int` type)—but it may also contain additional fields, due to record subtyping. For instance, a programmer could create a colored point object with the expression `Point({x=0,y=1,color=blue})`. Subtyping takes into account both the nominal sub-brand relationship and the usual structural subtyping relationship (width and depth) on records.

To integrate these two relationships, brand extension is constrained: the associated field types must be subtypes. In other words, a brand β_1 can be declared as a sub-brand of β_2 only if β_1's field type is a structural subtype of β_2's field type. As an example, suppose the brand `3DPoint` is defined as `3DPoint({x:int, y:int, z:int})`. `3DPoint` can be declared as a sub-brand of `Point`, since `{x:int, y:int, z:int}` is a sub-record of `{x:int, y:int}`. However, a brand `1DPoint({x:int})` cannot be a sub-brand of `Point` (since it lacks the `y` field), nor can `FloatingPoint(x:float, y:float})` (assuming `float` is not a subtype of `int`).

2.1 Example 1: A Window Toolkit

Fig. 1 contains a code excerpt for a windowing system and illustrates the novel features of Unity. The built-in brand `Top` is the root of the brand hierarchy, like `Object` in Java. To simplify the presentation, we include only the field `title`. `ScrollBar` is defined as a type alias using the `type` syntax. By default, a window does not have a scrollbar. The brands `Textbox` and `StaticText` extend `Window`, and also do not scroll by default.[2]

To add scrolling functionality, we have defined the function `scroll`, which operates on any `Window` (or sub-brand thereof) that has a `getScrollBar()` method. The type `Window({getScrollBar():ScrollBar})` classifies such an object. (We suppose here that the implementation of `scroll` need only access the scrollbar field and the fields of `Window`.) Note that the structural component of this type refers to another structural type, `ScrollBar`; structural types may be arbitrarily nested.

Let us assume that in a non-scrolling textbox, the user can only enter a fixed number of characters. Consequently, we define the brand `ScrollingTextbox` in order to change textbox functionality—in particular, the behavior when inserting a character. The `scroll` function is applicable to `ScrollingTextbox` since it is automatically a subtype of `Window({getScrollBar():ScrollBar})`.

In Unity, methods can be either internal (defined within a brand), or external (defined outside the brand). To allow sound modular checking of external methods (see Sec. 5), only internal methods are permitted to be abstract; external methods must be concrete. The method `insertChar` has been defined as an external method. This method is applicable to a `Textbox` or `ScrollingTextbox`

[1] The name "brand" is borrowed from Strongtalk [4], which in turn borrowed it from Modula-3.

[2] Note that all fields must be listed by the subtypes of `Window`; this design decision is merely to simplify our core calculus.

```
abstract brand Window ({title : string}) extends Top
concrete brand Textbox ({title : string, text : string}) extends Window
concrete brand StaticText ({title : string, text : string}) extends Window
concrete brand ScrollingTextbox({title : string, text : string, s : ScrollBar};
               method getScrollbar() : ScrollBar = this.s)
               extends Textbox

type ScrollBar = Top(getMaximum():int, setMaximum(x:int) : unit,
                     getValue():int, setValue(x:int) : unit)


let scroll = λw : Window({getScrollBar() : ScrollBar}) .
    . . . // code that performs the scrolling operation


method insertChar Textbox({getCurrentPos() : int}) : unit =
         λc : char .  ...  // insert a character only if it will fit in the window

method insertChar ScrollingTextbox({getCurrentPos() : int}) : unit =
         λc : char .  ...  // insert the character, scrolling if necessary

method getCurrentPos(Textbox({pos:int})) : int = ...
```

Subtyping relationships

```
Window ({title : string, s : ScrollBar}) ≤ Window ({title : string})

Textbox ({...}) ≤ Window ({title : string})
ScrollingTextbox ({...}) ≤ Textbox ({...})
ScrollingTextbox({...}) ≤ Window({title : string, s : ScrollBar})

StaticText({...}) ≤ Window ({title : string})
StaticText({..., s : ScrollBar}) ≤ Window(title : string, s : ScrollBar)
```

Figure 1: Unity code for a windowing system. Nominal subtyping allows the brand `ScrollingTextbox` to change the behavior of `insertChar` through tag dispatch, while structural subtyping allows the `scroll` function to apply to any window with an `s : ScrollBar` field. `ScrollBar` is defined as a type alias using the `type` syntax. In the subtyping relationships, some field names are elided with "...."

that has a `getCurrentPos` method. `Textbox` does not have an internal `getCurrentPos` method, so it has been added as an external method. The method `getCurrentPos`, in turn, is only applicable to a `Textbox` that has a `pos:int` field. This illustrates the structural constraints that can be put on a method. For a method *m*, a programmer can specify a set of fields and methods that must be present in *m*'s receiver.

Since a textbox that scrolls allows the user to enter more text than the window size permits, a new sub-brand had to be defined so that its implementation of `insertChar` could be overridden. If other sub-brands of `Window` (such as `StaticText`) do not need to change their existing behavior when a scrollbar is added, no new sub-brands need be defined. Scrolling functionality can be added to these types by including a `ScrollBar` field and a `getScrollBar()` method, and the `scroll` function is then applicable.

This example demonstrates both the use of functions (i.e., lambda expressions), and methods. The difference between the two is that functions do not perform dispatch (that is, they cannot be overridden), but they can be defined at any scope. Methods can be overridden, but they can only be defined at the top-level.

These brand definitions induce subtyping relationships, shown below the code listing in Fig. 1. Interestingly, `ScrollingTextbox({...})` is a subtype of both `Window({getScrollBar():ScrollBar})` and of `Textbox({...})`, but we have avoided both multiple inheritance and the problems typically associated with it. The type `Window({getScrollBar():ScrollBar})` is a conceptual interface that exists without being named.

The example illustrates the two kinds of extensibility that Unity provides: structural extensibility and brand extensibility.

- **Structural types** can be used to create *structural method constraints*—methods that an object must have in order to conform to that type. They can be also be used to *create a new type that adds fields to an existing brand*, without defining new behavior for the resulting type. So, a `ScrollBar` can be added to a `StaticText` object without defining a new brand, as the existing functionality of the static text box does not need to change if a scrollbar is added.

- **Brand extension** creates a new brand that can be used in dispatch; as a consequence, programs can *define new behavior for the newly defined brand*. Here, `ScrollingTextbox` is defined as an extension of `Textbox` because the behavior of `insertChar` is different depending on whether or not the text box has a scrollbar attached to it. Design intent is preserved because whenever different behavior is required (such as with `Cowboy.draw()` and `Circle.draw()`), nominal subtyping must be used.

Additionally, we see here the synergy between structural subtyping and external dispatch. Structural subtyping can be used to *specify the constraints* of a method, and external methods can be used to make existing brands *conform* to those constraints.

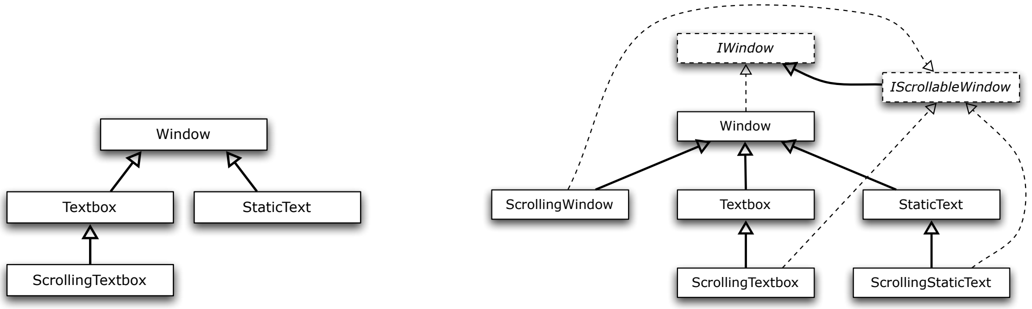
2.2 Comparison to Other Systems

Here we compare Unity to closely related systems. See Sec. 6 for other related work.

Java. In Java-like languages, expressing this example would be unwieldy. A common way to express the necessary constraints would involve first defining two interfaces, `IWindow` and `IScrollableWindow`. `ScrollBar` would also have to be an interface.

If a programmer wished to allow the possibility of adding a scrollbar to a window class, even without changing any other functionality, he would have to define a subclass that also implemented `IScrollableWindow`. In this example, we would define the classes `ScrollingWindow`, `ScrollingTextbox`, and `ScrollingStaticText`, though only `ScrollingTextbox` needs to change any functionality; see the class diagram Fig. 2(b). Contrast this with the brand structure of the Unity program depicted in Fig. 2(a). In Unity, only the types associated with dispatch need to be defined, in contrast to Java.

The Java equivalent of the `scroll` function could be a static function of some helper class and would take an object of type `IScrollableWindow`. Of course, if a programmer defined a



(a) The windowing example as implemented in Unity. Depicted here are the brands that must be defined in order to obtain the desired subtyping relationships.

(b) The same example implemented in Java. Dashed rectangles are interfaces; solid rectangles are classes. Dashed lines indicate the `implements` relationship and solid lines indicate `extends`.

Figure 2: For the windowing example, the user-defined subtyping relationships necessary in Unity vs. those necessary in Java.

new scrolling window class with the correct `getScrollBar()` method, but forgot to implement `IScrollableWindow`, the `scroll` function could not be used on objects of that class. (This situation often arises in Java programs, particularly when one wishes to use library code, the developers of which are not even aware of the interface that they should implement.)

There are other oddities in the Java version. The Java class `ScrollingWindow` is semantically analogous to the Unity type `Window(getScrollBar():ScrollBar)`, but `ScrollingTextbox` and `ScrollingStaticText` are *not* subclasses of `ScrollingWindow`, while the corresponding Unity types *are* subtypes of `Window(getScrollBar():ScrollBar)`. To have such subtyping relationships would require multiple inheritance in a language like Java, while the Unity code works with just single inheritance. This illustrates the lack of expressiveness that is inherent in languages that require the programmer to name all relevant subtyping relationships.

Traits. A language with traits [25] would provide a much cleaner solution than that of Java, but would still lack the expressiveness of Unity's structural subtyping. This is because traits are mainly designed to solve issues of implementation inheritance (especially multiple inheritance) that are largely orthogonal to the ones we are considering. In this example, the same subtyping hierarchy would have to be created as in the Java example, but the `scroll` function could be written for `IScrollableWindow` with the appropriate dispatch. (A static method could always be written in Java, but it would not perform dispatch on subtypes.) This would enable some code reuse, but would still require creating a number of types.

Mixins. In a language with mixins [3], the programmer would create a mixin class `ScrollableWindow` that consists of the fields of `Window` along with `s : ScrollBar` and the code for the function `scroll`. The code for the `ScrollableWindow` would then be mixed into `StaticText` to create a `ScrollingStaticText` and into `Textbox` to create `ScrollingTextbox`. The behavior of `insertChar` would then be specialized for `ScrollingTextbox`.

With mixins, the same number of eventual classes would be created as in Java, but creating them becomes easier because of mixin construction. In contrast with Unity, the code for `scroll` cannot be reused unless the mixin `ScrollableWindow` is used, which restricts its flexibility. This can pose a problem when interoperating with classes that were created in isolation from the mixin. Mixins also require up-front planning; methods and fields cannot be added after-the-fact.

Structural subtyping. Languages which support structural subtyping, such as Moby [11], O'Caml [15], and PolyTOIL [5], would elegantly express all of the desired subtyping relationships, but these languages allow only internal dispatch—that is, all methods must be defined inside the class definition. In our language, `insertChar` can be an external method; it need not reside inside the definitions of `Textbox` and `ScrollingTextbox`. It would be non-trivial to add support for external dispatch or multimethods to these types of languages.

Cecil. Cecil fully supports external and multimethod dispatch [7, 8]. Cecil's powerful, but very complex, type system can express most of the necessary relationships (though new classes do need to be defined for `ScrollingWindow` and `ScrollingStaticText`). To write the `scroll` function, a programmer would have to use bounded quantification and a "where" clause constraint, the latter being typechecked via a constraint solver. That is, in psuedo-code, the argument to `scroll` would have type:

for all T where T <: Window and signature getScrollBar(T) : ScrollBar

Here, the type `ScrollBar` would have to be a class, rather than a structural type as in Unity, due to two major shortcomings of `where` clauses: they cannot be nested and can only occur on top level methods. Additionally, `where` clauses cannot be used to constrain the method's receiver. In Unity, on the other hand, structural types are compositional and can therefore be nested within another type (e.g., `ScrollBar` in Fig. 1), can occur at any level in the program (e.g., the lambda expression `scroll`), and can be used to constrain a method's receiver (e.g., method `insertChar`).

Virtual classes. Some of the required relationships could be expressed elegantly using virtual classes [16] or nested inheritance [21], but only with advance planning. To express this example, the programmer would create a class `Base` containing the virtual classes `Window`, `Textbox`, and `StaticText`. Then, she would create a subclass of `Base`, called `Scroll`, that contained its own `Window`. This definition of `Window` would add a field for a scrollbar. Additionally, `Scroll` would have a virtual class `ScrollingTextbox` which would include the new definition of `insertChar`. The programmer would not to create a new class `ScrollingStaticText` since the new definition of `Window` would automatically apply to `StaticText` (i.e., `Scroll.StaticText` would automatically have a scrollbar).

The virtual classes solution is elegant, but if the programmer did not plan ahead and redefine `Window` in the `Scroll` class, there would not be a way to describe this type. Essentially, virtual classes make it very easy to reuse code across related classes (an advantage of virtual classes and nested inheritance over Unity), but cannot easily express the structural types of Unity.

2.3 Example 2: AST Nodes in an IDE

In this section, we describe another example to show other ways in which Unity can be used. Suppose we have an integrated development environment that includes an editor and a compiler.

```
abstract brand AstNode( ;{abstract method compile : () → unit}) extends Top

concrete brand PlusNode ({n1 : AstNode(), n2 : AstNode()};
   method compile() : unit = compilePlus(this); /* compile PlusNode */)
   extends AstNode
concrete brand Num ({val : int}; method compile() : unit = ... /* compile Num*/)
   extends AstNode
concrete brand Var ({s : Symbol}; method compile() : unit = ... /* compile Var */)
   extends AstNode

// highlight the text corresponding to 'node' in the text editor,
// using the location specified by the 'loc' field
let highlightNode = λ node : AstNode(loc : Location). ...

// AST nodes with debug information
concrete brand DebugPlusNode ({n1 : AstNode(), n2 : AstNode(), loc : Location};
   method compile() : unit = compilePlus(this); outputLocation(out, this.loc))
   extends PlusNode
concrete brand DebugNum ({val : int, loc : Location};
   method compile() : unit = ... /* compile DebugNum */) extends Num
concrete brand DebugVar ({s : Symbol, loc : Location, varName : string};
   method compile() : unit = ... /* compile DebugVar */) extends Var
```

Figure 3: Example 2: AST Nodes in an IDE. The top portion is the code before changes to add debug information to the AST. The function `highlightNode` makes use of structural information and the external dispatch in `compile` changes its behavior for the declared `Debug*` sub-brands.

The top portion of Fig. 3 contains an excerpt of a simplified version of the code for such a system. Here, the brands `PlusNode`, `Num` and `Var` define a simple abstract syntax tree. The internal method `compile` performs compilation on an `AstNode`.

One can use structural subtyping to create AST nodes with additional information, such as a node with a `loc` field specifying the file location of the code corresponding to the node. Additional functions are available for such nodes, such as the function `highlightNode` that highlights a node's source code in the text editor.

We did not have to define a new brand for AST nodes that include file location information. Whether or not a node contains file information, functions that operate over AST nodes need not change their behavior, so in this case structural subtyping suffices.

Suppose now that the programmer wishes to add "debug" versions of these AST nodes that contain additional output information for compiling in debug mode. For example, a `DebugNum` has a `Location` field, while `DebugVar` includes a `Location` field as well as a string representation of the variable name. The newly added code is listed in the bottom portion of Fig. 3.

Since each of these brands have been defined as extensions, they may also customize the behavior of `compile` to output this additional information when compiling. Additionally, since all of the `Debug*` brands have a `Location` field, the function `highlightNode` can be used on objects of this type.

This example again illustrates the expressiveness that achieved by combining nominal and structural subtyping; `highlightNode` makes use of additional structural information, while `compile` relies on nominal dispatch to behave differently in different situations.

2.4 Real-World Examples

The following real-world examples illustrate the gains in flexibility that could be achieved through structural subtyping.

2.4.1 Eclipse SWT.

In the Eclipse SWT (Simple Windowing Toolkit), many classes (such as `Button`, `Label`, `Link`, etc.) have the methods `getText` and `setText`, that set the main text for the control, such as a button's text, the text in a textbox, etc. However, there is no common `IText` interface. Many classes—13 in total—also support adding an image through the `getImage` and `setImage` method, but again there is no interface that captures this. A programmer may wish to write a method that sets the image of any control by retrieving the image from an image registry. Given the current API, such a method would have to rely on runtime reflection, with no guarantee of successful method invocation at compile time.

2.4.2 Eclipse JDT.

In the JDT (Java Development Tools), there are 8 classes (including `IMethod`, `IType`, `IField`) that have the method `getElementName`, but there is no `IElement` interface with this method. With structural subtyping, these classes implicitly share an interface, and code could be written that is polymorphic over the exact class type. For instance, a tree view of an AST may wish to display packages, methods, and fields in a uniform way. With the current hierarchy it is not possible to simply call the `getElementName` of the object, since these classes do not have an explicit interface with this method.

3 Case Study: Optional Methods in Java

In this section, we describe the tradeoffs that a library designer must make when using a language that has only nominal subtyping. The design of the Java collections library illustrates that designers would rather circumvent the type system than have a proliferation of types. We believe this situation often occurs with nominal subtyping, but because of structural subtyping, such a situation need not occur in Unity.

In the Java collections library, the interface `java.util.Collection` has several "optional" methods: `add`, `addAll`, `clear`, `remove`, `removeAll`, and `retainAll`. Many of the abstract classes implementing `Collection` (e.g., `AbstractCollection`, `AbstractList`, `AbstractSet`) throw an `UnsupportedOperationException` when those methods are called. There are a total of 30 optional methods in `java.util.*`, and `java.lang.Iterator` has an additional optional method. The methods were designed this way to avoid an explosion of interfaces such as `MutableCollection`, `ImmutableCollection`, etc., and a corresponding increase in the number of sub-interfaces (e.g., `MutableList`, `ImmutableList`, etc.) [18].

Let us consider a Java collections framework without the optional methods. Figure 4 shows a relevant portion of the current Java collections hierarchy. Figure 5 show new interfaces that capture the distinction of mutability directly in the hierarchy—doing away with optional operations. The interface `Collection<E>` represents a collection that is modifiable, while the new

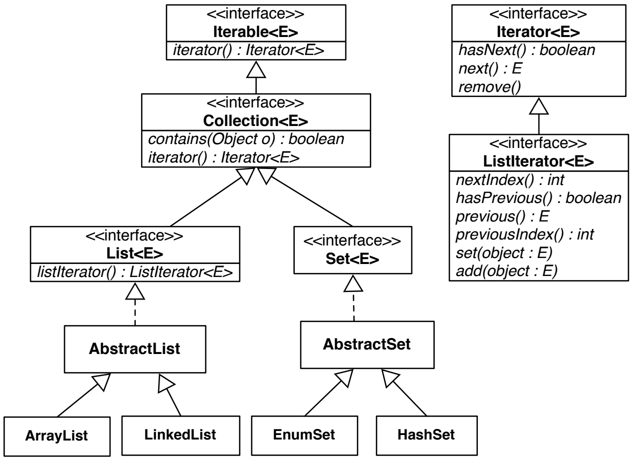


Figure 4: A portion of the Java collections framework. A few methods are highlighted in most interfaces. Type parameters are elided in classes.

interface `ReadableCollection`<E> represents a collection that only contains read operations. Accordingly, its `iterator()` method returns a `ReadIterator`, which is defined without a `remove()` operation. There are now two new `ListIterator` interfaces, for fixed-size lists, modifiable lists, read-only lists. These correspond to the `FixedSizeList`<E> and `ReadableList`<E> interfaces in Figure 5. (The interface `FixedSizeList`<E> has been added because selective overriding of methods in `AbstractList` would yield such a type, as noted in the documentation.) The hierarchy for `Set` is similar to that of `List` (though simpler, since there are no fixed-size sets, and no set-specific iterator).

In a language with structural subtyping, such as Unity, not all interesting combinations of structural types have to be declared in advance (though in a library setting they might be, for consistency's sake). For a language with type abbreviations, the key idea is that a type abbreviation would simply be syntactic sugar for a set of methods, which could be given an abbreviation with a different name in another part of the system. Additionally, the subtyping relationships between all the interfaces would not need to be defined in advance. Finally, as a side note, the notational overhead in defining type abbreviations would be potentially far lower than that of defining a Java interface, which has a relatively high notational cost (due, in part to the nominal nature of interfaces).

For this example, in Unity, the new interfaces (shown in gray), would not necessarily need to be defined by the library author, unless specifically needed. This eases the task of library development, as the library author does not need to anticipate which supertypes of the given interfaces would be useful for clients.

Thus, with a combination of nominal and structural subtyping, we need not sacrifice static type safety in order to overcome the shortcomings of a purely nominal type system.

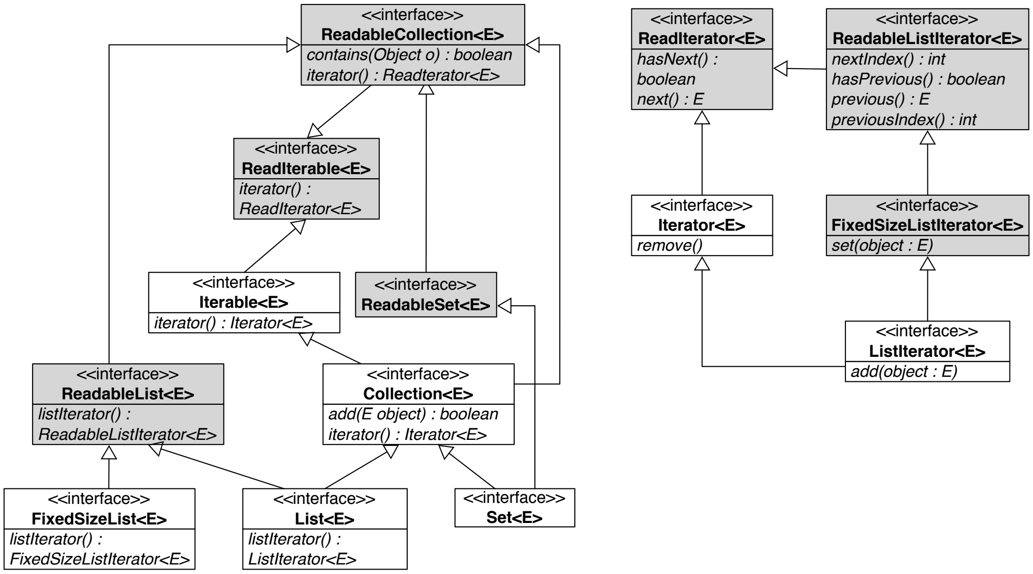


Figure 5: Refactored `AbstractList` and `AbstractSet` classes, along with new interfaces to remove optional methods. A few methods of most interfaces are highlighted for List and Set; for iterators, all methods, except for inherited methods, are shown. New interfaces have a gray background.

4 Empirical Analysis

To determine if there are potential cases where structural subtyping would be useful in a real system, we ran an analysis of 15 open-source Java programs. The analysis searches for common method signatures that are not related through inheritance. A "common method" is any method declaration where there exists in another class a method declaration with an identical name and the same signature, but the method is not present in any common supertype of the two.

For instance, in Apache Collections, four buffer classes had the methods `increment` and `decrement`, but these were not contained in a common superclass or super-interface. The results of the analysis are in Fig. 6. Tomcat, a servlet container, had the greatest percentage of common methods, 28.4%. Ant, the software build system, was close behind with 28.1%. Even the programs with the smallest number of common methods had a significant number of them; Areca, a backup program, had 11.9% common methods.

Inspecting the common methods, we found several cases where a structural type could be useful. For instance, in Smack, a Jabber client library, there were 6 classes with the common method `String getElementName()`. There were also 30 classes with the method `String toXML()` which might also be a method that clients might wish to call in a uniform manner. In JHotDraw, a GUI framework, there were 9 classes that had `addPropertyChangeListener` and `removePropertyChangeListener` methods. In Log4j, a logging library, there were 4 classes with the method `int getBufferSize()`, and 8 classes with the method `setOption`. In the Apache Collections library, nearly all the common methods appeared to be potentially useful. For instance, there were 5 iterator decorator classes with `getIterator` and `setIterator` methods. 4 bag classes had a method `getBag`, 4 buffer classes had a method `getBuffer`, and 4 classes had the method

Figure 6: Results of empirical analysis. For each program, the total number of methods, the number of common methods, the percentage of common methods compared to the total, the average number of methods in each common method group, and the number of common method groups are displayed. "Total methods" includes interface methods, abstract methods, and overriding implementations of the same method. The results suggest that many Java programs have potential uses of structural subtyping.

	Total methods	Common methods	% Common	Average methods/group	Total common groups
Tomcat	14678	4172	28.4%	3.2	1288
Ant	9178	2577	28.1%	3.5	727
JHotDraw	5149	1193	23.2%	2.8	428
Smack	3921	881	22.5%	3.3	270
Struts	3783	772	20.4%	2.7	291
Apache Forrest	164	28	17.1%	2.2	13
Cayenne	9243	1545	16.7%	2.8	553
Log4j	1950	312	16.0%	3.1	102
OpenFire	8135	1300	16.0%	2.8	470
Apache Collections	3762	584	15.5%	2.8	211
Derby	24521	3575	14.6%	2.5	1402
Lucene	2472	331	13.4%	2.5	134
jEdit	5845	699	12.0%	2.6	271
Apache HttpClient	1818	217	11.9%	2.6	83
Areca	3565	423	11.9%	2.6	163

`getComparator`. Additionally, 7 classes had the method `int size()` and 5 classes had the method `int indexOf(Object)`.

Note that we did not closely examine the implementations of these methods to determine if they were semantically performing similar actions, as we were unfamiliar with the codebase. So, it is possible that the methods coincidentally had similar names but were performing different actions. In future work, we plan to study one application in depth to see how it can benefit from structural subtyping.

Overall, however, we found the results to be promising, and they suggest that Java programs could indeed benefit from structural subtyping. If a programmer wished to write code that called a common method, he could easily do so by using a type—which exists implicitly—consisting of that method. In contrast, in Java and other languages with nominal subtyping, programmers would have to explicitly create interfaces. And, in some cases, the interface would contain only one method, which seems an unnecessary overhead.

5 Formal System

The Unity grammar is presented in Fig. 7. The language is a lambda calculus extended with values tagged with brands. Methods can be defined on a brand and the usual dispatch semantics apply. Brand and method declarations are top-level. To define brands, the `brand` construct is used. A brand can be either `abstract` or `concrete`. Objects cannot be created from `abstract` brands (similar

Programs	p	$::= \mathit{decl} \ \textsf{in} \ p \mid e \mid e; p$
Declarations	decl	$::= \mathit{brand\text{-}decl} \mid \mathit{ext\text{-}decl}$
Brand declaration	$\mathit{brand\text{-}decl}$	$::= \mathit{mod} \ \textsf{brand} \ \beta(\tau; \overline{\mathit{m\text{-}decl}}) \ \textsf{extends} \ \beta$
Modifiers	mod	$::= \textsf{abstract} \mid \textsf{concrete}$
Method declaration	$\mathit{m\text{-}decl}$	$::= \textsf{abstract method} \ m \ (\overline{m : \tau}) : \tau$
		$\mid \ \textsf{method} \ m \ (\overline{m : \tau}) : \tau = e$
External method	$\mathit{ext\text{-}decl}$	$::= \textsf{method} \ m \ \beta(\overline{m : \tau}) : \tau = e$
Expressions	e	$::= () \mid x \mid \lambda x{:}\tau.\, e \mid e\, e \mid \widehat{\beta}(e) \mid (\overline{\ell = e}) \mid e.\ell \mid e.m \mid \textsf{fold}_\tau \, e \mid \textsf{unfold}_\tau \, e$
Types	τ	$::= \textsf{unit} \mid \tau \to \tau \mid \tau \wedge \tau \mid \beta(\overline{m : \tau}) \mid \{\overline{\ell : \tau}\} \mid X \mid \mu X.\tau \mid \tau \Rightarrow \tau$
Values	v	$::= () \mid \lambda x{:}\tau.\, e \mid \widehat{\beta}(v) \mid (\overline{\ell = v}) \mid \textsf{fold}_\tau \, v$
Contexts	Γ	$::= \ \cdot \mid \Gamma, x : \tau$
	Σ	$::= \ \cdot \mid \Sigma, \mathit{mod} \ \beta(\tau; \overline{\mathit{mod} \ m : \tau}) \ \textsf{extends} \ \beta$
	Δ	$::= \ \cdot \mid \Delta, \widehat{\beta}(\overline{m = e}) \ \textsf{extends} \ \widehat{\beta}$

Conventions

$\widehat{\beta} \equiv$ tag value corresponding to β

$\text{req}_\Sigma(\beta) = \tau$ if $\beta(\tau; \overline{m : \tau}) \ \textsf{extends} \ \beta' \in \Sigma$

$\text{modifier}_\Sigma(\beta) = \mathit{mod}$ if $\mathit{mod} \ \beta(\tau; \overline{m : \tau}) \ \textsf{extends} \ \beta' \in \Sigma$

M ranges over $\overline{m : \tau}$

Figure 7: Unity grammar

to Java's abstract classes). We use the metavariables β and θ to range over brand names. The metavariable M ranges over a list of (method : type) pairs.

One of the valid expression forms for a program is an expression followed by a program ($e; p$). In this last construct, the expression is evaluated and will be type correct according to the definitions that preceded it.

When a brand is defined, a name is given for it, as well as the brand's *field type* (usually a record); this is the type of the fields of the brand. The programmer initializes the field value when an object is created.

In Unity, a method is either internal or external. In the former case, the method is defined along with the brand, like method declarations in Java-like languages. To allow modular exhaustiveness checking of external methods, external methods cannot be abstract; a method body must be provided for every external method. We have taken this rule from MultiJava [9].

When a method m is defined on a brand β, a set of methods is specified—the methods that must exist within β (either internal or external) before m can be invoked.[3] For example, in Fig. 1,

[3] For simplicity and to support information hiding, types cannot contain field constraints as in example 1, but this is not

the function `insertChar` required that its receiver have a `getCurrentPos` method.

To simplify the formal system, methods take only one argument: the this parameter. Additional parameters may be specified using lambda expressions.[4]

If β is a brand name, then $\widehat{\beta}$ is the tag value corresponding to β. In other words, $\beta(\overline{m : \tau})$ is a type, and $\widehat{\beta}$ is its run-time analogue.

To create objects, the expression form $\widehat{\beta}(e)$ is used. This creates an object that is tagged with $\widehat{\beta}$. Methods are called using $e.m$, while function application is written $e_1 e_2$.

Our language includes a limited form of intersection types. Our motivation for including these is to make external methods available to objects that were defined before the external method was created. Section 5.1.2 describes this in more detail.

Σ is the subtyping context; it stores the user-declared sub-branding relationships. Δ is the corresponding run-time context. Δ contains a strict subset of the information in Σ—it does not contain whether a brand is abstract or concrete, and it does not keep track of the field type or methods associated with each brand. We assume the existence of a special brand Top that is not defined in Σ or Δ, but that may be extended by user-defined brands. Since every brand must have a super-brand, the brand subtype hierarchy is a tree rooted at Top.

Like other object calculi, Unity is purely functional so as to simplify the system. State is orthogonal to the issues we are considering; our design should be easily adaptable to a language with imperative features.

5.1 Static Semantics

Here we describe the subtyping and typing judgements shown in Figures 9, 11 and 12. Auxiliary judgements are in Fig. 13.

5.1.1 Subtyping.

Subtyping comprises two parts: the sub-brand judgement ($\sqsubseteq$) and the subtype judgement ($\leq$), shown in Figures 8 and 9. The first judgement is not on types, but brands, which are a component of a type but not themselves a type. The sub-brand judgement is just the reflexive, transitive closure of the declared extends relation.

The subtype judgement ($\leq$) uses the sub-brand judgement in the third subtyping rule, which states that an object type $\beta_1(M_1)$ is a subtype of $\beta_2(M_2)$ when β_1 is a sub-brand of β_2 ($\beta_1 \sqsubseteq \beta_2$) and M_1 is a sub-record of M_2 ($M_1 \leq M_2$). There are additional conditions that $\beta_1(M_1)$ **type** and $\beta_2(M_2)$ **type**, which ensures that these are valid types. The relevant type formation rule here is BRAND-TYPE which checks that the given labels and types are a sub-record of the required fields for the brand. This ensures that a brand type always contains at least the labels it was defined to have. There is an additional check that the methods are valid overrides (*override* is defined in Fig. 13). The rest of the rules for the type formation judgement are straightforward; the full judgement appears in Fig. 10.

a fundamental limitation of the system.

[4] Note that if the body of a method is a lambda expression, it does not perform dispatch. To perform dispatch, the body of the method should be another method call. In this way, asymmetric multimethods (multimethods where the order of parameters is used in dispatch) can easily be encoded in our system. To encode a method m with body e that dispatches on β_1 and β_2, method m in β_1 dispatches to method m in β_2, the body of which is e.

$$\boxed{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}$$

$$\frac{\mathit{mod}\ \beta_1(\tau; \overline{m : \tau})\ \textsf{extends}\ \beta_2 \in \Sigma}{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}\ (\textsc{Sub-Brand-Decl}) \qquad \frac{}{\Sigma \vdash \beta \sqsubseteq \beta}\ (\textsc{Sub-Brand-Refl})$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash \beta_2 \sqsubseteq \beta_3}{\Sigma \vdash \beta_1 \sqsubseteq \beta_3}\ (\textsc{Sub-Brand-Trans})$$

Figure 8: Unity sub-branding judgement

$$\boxed{\Sigma \vdash \tau_1 \leq \tau_2}$$

$$\frac{}{\Sigma \vdash \tau \leq \tau}\ (\textsc{Sub-Refl}) \qquad \frac{\Sigma \vdash \tau_1 \leq \tau_2 \qquad \Sigma \vdash \tau_2 \leq \tau_3}{\Sigma \vdash \tau_1 \leq \tau_3}\ (\textsc{Sub-Trans})$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash M_1 <: M_2 \qquad \Sigma \vdash \beta_1(M_1)\ \textbf{type} \qquad \Sigma \vdash \beta_2(M_2)\ \textbf{type}}{\Sigma \vdash \beta_1(M_1) \leq \beta_2(M_2)}\ (\textsc{Sub-Name})$$

$$\frac{\Sigma, X \leq Y \vdash \tau_1 \leq \tau_2}{\Sigma \vdash \mu X.\tau_1 \leq \mu Y.\tau_2}\ (\textsc{Sub-Rec}) \qquad \frac{\Sigma \vdash \sigma_1 \leq \tau_1 \qquad \Sigma \vdash \tau_2 \leq \sigma_2}{\Sigma \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ (\textsc{Sub-Func})$$

$$\frac{\Sigma \vdash \tau \leq \sigma_1 \qquad \Sigma \vdash \tau \leq \sigma_2}{\Sigma \vdash \tau \leq \sigma_1 \wedge \sigma_2}\ (\textsc{Sub-}\wedge R) \qquad \frac{\Sigma \vdash \tau_1 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_1) \qquad \frac{\Sigma \vdash \tau_2 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_2)$$

$$\frac{\{\ell_i : \tau_i\ ^{i \in 1..n}\}\ \text{is a permutation of}\ \{\ell_j : \tau_j\ ^{j \in 1..n}\}}{\Sigma \vdash \{\ell_i : \tau_i\ ^{i \in 1..n}\} \leq \{\ell_j : \tau_j\ ^{j \in 1..n}\}}\ (\textsc{Sub-Rec-Perm})$$

$$\frac{n > m}{\Sigma \vdash \{\ell_i : \tau_i\ ^{i \in 1..n}\}\ \leq\ \{\ell_j : \tau_j\ ^{j \in 1..m}\}}\ (\textsc{Sub-Rec-Width})$$

$$\frac{\Sigma \vdash \tau_i \leq \sigma_i\ {\scriptstyle (i \in 1..n)}}{\Sigma \vdash \{\ell_i : \tau_i\}\ ^{i \in 1..n} \leq \{\ell_i : \sigma_i\}\ ^{i \in 1..n}}\ (\textsc{Sub-Rec-Depth})$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}{\Sigma \vdash \beta_1(\tau_1) \wedge \beta_2(\tau_2) \leq \beta_1(\tau_1 \wedge \tau_2)}\ (\textsc{Sub-Brand-}\wedge L)$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash M_2 <: M_1 \qquad \Sigma \vdash \sigma_1 \leq \sigma_2}{\Sigma \vdash \beta_1(M_1) \Rightarrow \sigma_1 \leq \beta_2(M_2) \Rightarrow \sigma_2}\ (\textsc{Sub-Method})$$

Figure 9: Unity subtyping judgement

Our language includes a limited form of intersection types, à la Davies and Pfenning; the rules for intersection types are borrowed from their work [10].

$$\boxed{\Gamma \mid \Sigma \vdash \tau \ \mathbf{type}}$$

$$\frac{}{\Gamma \mid \Sigma \vdash \mathsf{unit}\ \mathbf{type}}\ (\textsc{Unit-Type}) \qquad \frac{\Gamma \mid \Sigma \vdash \tau_1\ \mathbf{type} \qquad \Gamma \mid \Sigma \vdash \tau_2\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \tau_1 \to \tau_2\ \mathbf{type}}\ (\textsc{Fun-Type})$$

$$\frac{\Gamma \mid \Sigma \vdash \tau_1\ \mathbf{type} \qquad \Gamma \mid \Sigma \vdash \tau_2\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \tau_1 \wedge \tau_2\ \mathbf{type}}\ (\wedge\textsc{-Type})$$

$$\frac{\begin{array}{cc} \overline{m}\ \text{distinct} & \Gamma \mid \Sigma \vdash \overline{\sigma}\ \mathbf{type} \\ \Sigma \vdash \beta\ \mathsf{extends}\ \beta_2 & \Sigma \vdash \mathit{override}(\overline{m : \sigma}, \beta_2) \end{array}}{\Gamma \mid \Sigma \vdash \beta(\overline{m : \sigma})\ \mathbf{type}}\ (\textsc{Brand-Type})$$

$$\frac{\overline{\ell}\ \text{distinct} \qquad \Gamma \mid \Sigma \vdash \overline{\tau}\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \{\overline{\ell : \tau}\}\ \mathbf{type}}\ (\textsc{Record-Type}) \qquad \frac{\Gamma, X\ \mathsf{type} \mid \Sigma \vdash \tau\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \mu X.\tau\ \mathbf{type}}\ (\textsc{Mu-Type})$$

$$\frac{T\ \mathsf{type} \in \Gamma}{\Gamma \mid \Sigma \vdash T\ \mathbf{type}}\ (\textsc{Var-Type})$$

$$\frac{\Gamma \mid \Sigma \vdash \beta(m_i : \theta_i(\overline{n_i : \sigma_i}) \Rightarrow \tau_i'\ ^{i \in 1..n})\ \mathbf{type} \qquad \Sigma \vdash \beta \sqsubseteq \theta_i\ \ (i \in 1..n) \qquad \Gamma \mid \Sigma \vdash \tau_2\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \beta(m_i : \theta_i(\overline{n_i : \sigma_i}) \Rightarrow \tau_i'\ ^{i \in 1..n}) \Rightarrow \tau_2\ \mathbf{type}}\ (\textsc{Method-Type})$$

Figure 10: Unity type formation judgement

There is also a subtyping rule for a list of (method : type) pairs; it simply applies the record subtyping rule. The remaining subtyping rules are the standard reflexivity, transitivity, and function subtyping rules.

5.1.2 Typing rules.

Full typing rules for typechecking programs and expressions appear in Figs. 11 and 12, respectively. Auxiliary judgements are defined in Fig. 13. The interesting rules are Tp-Brand, Tp-Ext-Method, Tp-New-Obj and Tp-Invoke; the others are standard.

The rule Tp-Brand (Fig. 11) ensures that a brand declaration is well-formed. The newly defined brand must contain at least the labels and fields of the supertype (possibly with refined types); this is checked via the condition $\tau \leq \mathrm{req}_\Sigma\,(\beta')$. Note that if a field type is a record, then subtypes must list all the labels of the parent. Aside from simplifying the calculus, this sidesteps issues of variable shadowing while allowing subtypes to refine the type of a particular label. The rule also checks that the methods given are valid overrides of the methods of the super-brand, and, in the case of concrete classes, that all methods are concrete.

This rule and the type formation rule for brands described above illustrate the need for both a sub-brand and subtype judgement. The context Σ stores information about the fields and methods of a brand; these are retrieved via req_Σ and $\mathit{methods}_\Sigma$ (called by $\mathit{override}_\Sigma$), respectively. Additionally, without a runtime component to the nominal hierarchy, there would not be a way to perform dispatch, which we describe in Sect. 5.2.

$$\boxed{\Sigma \vdash p\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\beta_1 \notin \Sigma \qquad \tau \leq \mathrm{req}_{\Sigma}\,(\beta') \qquad \Sigma \vdash \beta.\overline{\mathit{m\text{-}decl}} : (\overline{\mathit{mod}_m m : \tau}) \\ \Sigma' = \Sigma, \mathit{mod}\ \beta(\tau; \overline{\mathit{mod}_m\ m : \tau})\ \textsf{extends}\ \beta' \qquad \Sigma' \vdash \beta.(\tau; \overline{\mathit{m\text{-}decl}})\ \mathbf{ok} \\ \mathit{mod} = \textsf{concrete}\ \text{implies}\ \mathit{abstractCover}(\overline{\mathit{mod}_m m : \tau}, \beta') \qquad \mathit{override}(\overline{m : \tau}, \beta') \\ \textsf{abstract method}\ m \in \overline{\mathit{m\text{-}decl}}\ \text{implies}\ \mathit{mod} = \textsf{abstract} \qquad \Sigma' \vdash p\ \mathbf{ok}\end{array}}{\Sigma \vdash \mathit{mod}\ \textsf{brand}\ \beta(\tau; \overline{\mathit{m\text{-}decl}})\ \textsf{extends}\ \beta'\ \textsf{in}\ p\ \mathbf{ok}}\ (\text{Tp-Brand-Intro})$$

$$\frac{\begin{array}{c}\Sigma = \{\mathit{mod}\ \beta_1(\sigma; M')\ \textsf{extends}\ \beta_2\}, \Sigma_0 \\ m \notin M' \qquad \Sigma' = \{\mathit{mod}\ \beta_1(\sigma; M', m : \beta_1(M) \Rightarrow \tau)\ \textsf{extends}\ \beta_2\}, \Sigma_0 \\ \mathit{override}(\beta_1(M) \Rightarrow \tau, \beta_2) \qquad \textsf{this} : \beta_1(M), \textsf{fields} : \sigma \mid \Sigma' \vdash e : \tau \qquad \Sigma' \vdash p\ \mathbf{ok}\end{array}}{\Sigma \vdash \textsf{method}\ m\ \beta_1(M) : \tau = e\ \textsf{in}\ p\ \mathbf{ok}}\ (\text{Tp-Ext-Method})$$

$$\frac{\cdot \mid \Sigma \vdash e : \tau}{\Sigma \vdash e\ \mathbf{ok}}\ (\text{Tp-Expr1}) \qquad \frac{\cdot \mid \Sigma \vdash e : \tau \qquad \Sigma \vdash p\ \mathbf{ok}}{\Sigma \vdash e; p\ \mathbf{ok}}\ (\text{Tp-Expr2})$$

Figure 11: Unity typing judgement for programs

The rule Tp-Ext-Method checks external method definitions. The existing brand definitions are updated by adding the new external method via the new context Σ'. The rule also checks that the method types of the external method defined on sub-brands are in the subtype relation, which ensures that the context Σ' is well-formed.

The rule Tp-New-Obj (Fig. 12) checks the correctness of the object creation expression. The rule checks that the brand has been defined as concrete, and that the given record labels are a subtype of the required record labels.

The rule Tp-Invoke typechecks method invocations. The method being called must be contained in either the set of methods in the brand's type, M, or in the set of methods of the brand ($\mathit{methods}_{\Sigma}(\beta)$). Additionally, the methods in the brand's type, combined with the methods of the brand (via intersection) must be a subtype of the method's required methods. Adding the intersection condition increases expressiveness over having the rule just consider the methods of the brand, since the type might have methods defined on a sub-brand. For example, within the body of the function $\lambda x : \textsf{Top}(\texttt{toString : ()} \rightarrow \textsf{string}).\, e$, the type of x contains the method `toString`. If we suppose that `toString` is not defined for the brand Top, then x's type contains methods that are not defined in the brand itself.

Unity includes standard iso-recursive μ types to the language, along with a fold and unfold operation. In this system, it is possible to express types such as:

$$\mu X.\textsf{Top}(\texttt{clone()} : X)$$

which specifies that the result of the clone function is the type itself being defined. The advantage to structural recursive types is that structural object interfaces, such as `ScrollBar` in Example 1, can be specified as pure structural types (using the Top brand) while still being self-referential.

$$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash x : \tau}\ (\textsc{Tp-Var}) \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}\ (\textsc{Tp-Unit}) \qquad \frac{\Sigma \vdash \tau_1\ \mathbf{type} \qquad \Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}\ (\textsc{Tp-Fun})$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid \Sigma \vdash e_2 : \tau_1}{\Gamma \mid \Sigma \vdash e_1\, e_2 : \tau_2}\ (\textsc{Tp-App}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \sigma \qquad \Sigma \vdash \sigma \leq \tau}{\Gamma \mid \Sigma \vdash e : \tau}\ (\textsc{Tp-Subs})$$

$$\frac{\mathsf{concrete}\ \beta(\tau) \in \Sigma \qquad \Gamma \mid \Sigma \vdash e : \tau' \qquad \Sigma \vdash \tau' \leq \tau \qquad \mathit{methods}_\Sigma(\beta) = \overline{\mathit{mod}_m\ m : \sigma}}{\Gamma \mid \Sigma \vdash \widehat{\beta}(e) : \beta(\overline{m : \sigma})}\ (\textsc{Tp-New-Obj})$$

$$\frac{\Gamma \mid \Sigma \vdash \overline{e} : \overline{\tau}}{\Gamma \mid \Sigma \vdash (\overline{\ell = e}) : \{\overline{\ell : \tau}\}}\ (\textsc{Tp-New-Record}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \{\ell_i : \tau_i\ ^{i \in 1..n}\}}{\Gamma \mid \Sigma \vdash e.\ell_k : \tau_k}\ (\textsc{Tp-Proj})$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e : \beta(M) \qquad m_k : \tau_{m_k} \in (M \wedge \mathit{methods}_\Sigma(\beta)) \\ \tau_{m_k} = \beta'(\overline{n : \sigma}) \Rightarrow \tau \qquad \beta(M \wedge \mathit{methods}_\Sigma(\beta)) \leq \beta'(\overline{n : \sigma})\end{array}}{\Gamma \mid \Sigma \vdash e.m_k : \tau}\ (\textsc{Tp-Invoke})$$

$$\frac{\Gamma \mid \Sigma \vdash e : [\mu X.\tau / X]\tau}{\Gamma \mid \Sigma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \mu X.\tau}\ (\textsc{Tp-Fold}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \mu X.\tau}{\Gamma \mid \Sigma \vdash \mathsf{unfold}_{\mu X.\tau}\, e : [\mu X.\tau / X]\tau}\ (\textsc{Tp-Unfold})$$

Figure 12: Unity typing judgement for expressions

5.2 Dynamic Semantics

Most of the evaluation rules for Unity are standard; the evaluation judgement is in Fig. 14 and auxiliary judgements are in Fig. 15.

The interesting evaluation rules are E-BRAND-DECL and E-EXT-DECL, which evaluate brand definitions and external method definitions, respectively. To evaluate a brand definition, the method definitions are evaluated to the method body and the rest of the program is evaluated with the extended context. Similarly, E-EXT-DECL updates the context with new method definitions for the brand, then evaluates the rest of the program with the new context.

The auxiliary function $\mathit{mbody}_\Delta(m, \widehat{\beta})$ finds the appropriate method body for a method m, starting at the tag $\widehat{\beta}$. This function is used by the rule E-INVOKE, which within the method body returned by mbody_Δ, substitutes the object for `this` and the field value of the object for `fields`. Method declarations are evaluated in a straightforward manner; all of the type information is discarded (so in the case of abstract methods, the entire declaration is discarded), leaving just the method body.

5.3 Type Safety

The full proof of type safety is provided in a companion technical report [1]. We summarize the main results here. First, we provide the definition of a well-formed context:

Definition 5.1 (Well-formed context)**.**

$$\boxed{\Sigma \vdash \beta.\mathit{m\text{-}decl} : \tau}$$

$$\frac{\Sigma \vdash \beta(\overline{m : \sigma}) \Rightarrow \tau \ \textbf{type}}{\Sigma \vdash \beta.\mathit{mod}\ \textsf{method}\ m_1(\overline{m : \sigma}) : \tau = e \quad : \quad \mathit{mod}\ m_1 : \beta(\overline{m : \sigma}) \Rightarrow \tau}$$

$$\boxed{\Sigma \vdash \beta.(m; \mathit{m\text{-}decl})\ \textbf{ok}}$$

$$\frac{\textsf{this} : \beta(\overline{m : \sigma}), \textsf{fields} : \tau \mid \Sigma \vdash e : \tau'}{\Sigma \vdash \beta.\textsf{method}\ m_1(\overline{m : \sigma}) : \tau' = e\ \textbf{ok}}$$

$$\boxed{\mathit{methods}_\Sigma(\beta) = \overline{\mathit{mod}\ m : \tau}}$$

$$\frac{\begin{array}{c}\Sigma \vdash \beta_1(\tau; \mathit{mod}_i\ m_i : \tau_{m_i}{}^{i \in 1..n})\ \textsf{extends}\ \beta_2 \\ \mathit{methods}_\Sigma(\beta_2) = \mathit{mod}'_j\ m_j : \sigma_{m_j}{}^{j \in 1..k}, \overline{\mathit{mod}_2\ n : \sigma'_m}\end{array}}{\mathit{methods}_\Sigma(\beta_1) = \overline{\mathit{mod}\ m : \tau}, \overline{\mathit{mod}_2\ n : \sigma'_m}} \qquad \frac{}{\mathit{methods}_\Sigma(\textsf{Top}) = \cdot}$$

$$\boxed{\Sigma \vdash \mathit{override}(m : \tau, \beta)}$$

$$\frac{\mathit{methods}_\Sigma(\beta) = \mathit{mod}\ m : \sigma, \overline{\mathit{mod}_n n : \sigma'_m} \qquad \tau \leq \sigma}{\Sigma \vdash \mathit{override}(m : \tau, \beta)} \qquad \frac{m \notin \mathit{methods}_\Sigma(\beta)}{\Sigma \vdash \mathit{override}(m : \tau, \beta)}$$

$$\boxed{\mathit{abstractCover}_\Sigma(\overline{\textsf{concrete}\ m : \tau}, \beta)}$$

$$\frac{\mathit{methods}_\Sigma(\beta) = \textsf{abstract}\ n_i : \sigma_i{}^{i \in 1..n}, \overline{\textsf{concrete}\ n' : \sigma'} \qquad n_i \in \overline{m}^{\,i \in 1..n}}{\mathit{abstractCover}_\Sigma(\overline{\textsf{concrete}\ m : \tau})}$$

Figure 13: Unity typechecking auxiliary judgements

The context Σ is *well-formed*, iff the following conditions hold:

1. there is exactly one entry for each brand β.
2. if $\mathit{mod}\ \beta_1(\tau; M)\ \textsf{extends}\ \beta_2 \in \Sigma,$ then
   (a) $\beta_2(M)$ **type**
   (b) $\tau \leq \mathrm{req}_\Sigma\ (\beta_2)$
   (c) if mod = concrete, then $\mathit{methods}_\Sigma(\beta_1) = \textsf{concrete}\ \overline{n : \tau}$.

Our theorems on type safety assume a correspondence between the static brand definition context Σ and the runtime context Δ. This ensures that the runtime context, which does not contain type information, is consistent with the static typing context. Formally, this correspondence is defined as follows:

$$\boxed{e \longmapsto_\Delta e'}$$

$$\frac{e_1 \longmapsto_\Delta e_1'}{e_1\, e_2 \longmapsto_\Delta e_1'\, e_2}\ (\text{E-}\textsc{App1}) \qquad \frac{e_2 \longmapsto_\Delta e_2'}{v_1\, e_2 \longmapsto_\Delta v_1\, e_2'}\ (\text{E-}\textsc{App2})$$

$$\frac{}{(\lambda x{:}\tau.\, e)\, v \longmapsto_\Delta [\, v/x\,]\, e}\ (\text{E-}\textsc{App-Abs})$$

$$\frac{e_k \longmapsto_\Delta e_k'}{(\ell_1 = v_1, \ldots, \ell_{k-1} = v_{k-1}, \ell_k = e_k, \ldots) \longmapsto_\Delta (\ldots, \ell_k = e_k', \ldots)}\ (\text{E-}\textsc{Record})$$

$$\frac{e \longmapsto_\Delta e'}{e.\ell \longmapsto_\Delta e'.\ell}\ (\text{E-}\textsc{Proj1}) \qquad \frac{}{(\ell_i = v_i\ ^{i\in 1..n}).\ell_k \longmapsto_\Delta v_k}\ (\text{E-}\textsc{Proj2})$$

$$\frac{e \longmapsto_\Delta e'}{\widehat{\beta}(e) \longmapsto_\Delta \widehat{\beta}(e')}\ (\text{E-}\textsc{Brand-Cons}) \qquad \frac{e \longmapsto_\Delta e'}{e.m \longmapsto_\Delta e'.m}\ (\text{E-}\textsc{Invoke1})$$

$$\frac{mbody_\Delta(m, \widehat{\beta}) = e}{\widehat{\beta}(v).m \longmapsto_\Delta \{\widehat{\beta}(v)/\mathsf{this}, v/\mathsf{fields}\}\, e}\ (\text{E-}\textsc{Invoke2}) \qquad \frac{e \longmapsto_\Delta e'}{\mathsf{fold}_\tau\, e \longmapsto_\Delta \mathsf{fold}_\tau\, e'}\ (\text{E-}\textsc{Fold})$$

$$\frac{e \longmapsto_\Delta e'}{\mathsf{unfold}_\tau\, e \longmapsto_\Delta \mathsf{unfold}_\tau\, e'}\ (\text{E-}\textsc{Unfold}) \qquad \frac{}{\mathsf{unfold}_\tau\, (\mathsf{fold}_\tau\, v) \longmapsto_\Delta v}\ (\text{E-}\textsc{Unfold-Fold})$$

Figure 14: Unity evaluation judgement

$$\boxed{mbody_\Delta(m, \widehat{\beta}) = e}$$

$$\frac{\widehat{\beta}_1(m_0 = e_0, \overline{m' = e_m'})\ \mathsf{extends}\ \widehat{\beta}_2 \in \Delta}{mbody_\Delta(m_0, \widehat{\beta}_1) = e_0} \qquad \frac{\begin{array}{c}\widehat{\beta}_1(\overline{m = e})\ \mathsf{extends}\ \widehat{\beta}_2 \in \Delta \\ m_0 \notin \overline{m} \qquad mbody_\Delta(m_0, \widehat{\beta}_2) = e_0\end{array}}{mbody_\Delta(m_0, \widehat{\beta}_1) = e_0}$$

$$\boxed{m\text{-}decl \longmapsto m = e}$$

$$\frac{}{\mathsf{abstract\ method}\ m(\overline{m : \sigma_m}) : \tau \quad \longmapsto \quad \cdot} \qquad \frac{}{\mathsf{method}\ m(\overline{m : \sigma_m}) : \tau = e \quad \longmapsto \quad m = e}$$

Figure 15: Unity evaluation auxiliary judgements

Definition 5.2 (Models relation on contexts)**.** The definition of $\Sigma \vdash \Delta$ (Σ *models* Δ) is given by the following inference rules:

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\begin{array}{c}\Sigma \vdash \Delta \qquad \Sigma' = \Sigma, mod\ \beta_1(\tau; \{\mathsf{concrete}\ m_i : \beta_1(M_i) \Rightarrow \tau_i'\ ^{i\in 1..n}\}, \overline{\mathsf{abstract}\ n : \sigma})\ \mathsf{extends}\ \beta_2 \\ \mathsf{this} : \beta_1(M_i), \mathsf{fields} : \tau \mid \Sigma' \vdash e_i : \tau_i\ (i \in 1..n)\end{array}}{\Sigma' \vdash \Delta, \widehat{\beta}_1(m_i = e_i\ ^{i\in 1..n})\ \mathsf{extends}\ \widehat{\beta}_2}$$

Type safety is proved using the standard progress and preservation theorems. For progress, we prove a lemma that states that if we have a well-typed value whose type contains a method m_k, then a runtime context consistent with the static context must contain a method body for m_k:

Lemma 5.1. If $\Gamma \mid \Sigma \vdash \widehat{\beta}(v) : \tau$ and $\Sigma \vdash \tau \leq \beta'(M)$, where $\Sigma \vdash \Delta$ and $m_k \in M$, then $mbody_{\Delta}(m_k, \widehat{\beta})$ is defined.

The lemma is stated in this way so that the subsumption case is easy to prove. The lemma is proved by induction on the typing derivation. The interesting case is that of Tp-New-Obj, which uses the definition of a well-formed context and that of $\Sigma \vdash \Delta$.

Theorem 5.1 (Progress [programs])**.** If $\cdot \mid \Sigma \vdash p$ **ok**, for some Σ, then either p is a value or, for Δ such that $\Sigma \vdash \Delta$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

This theorem is proved by appealing to an auxiliary lemma that proves progress for expressions and a standard canonical forms lemma. The interesting case is that of method invocation, which is proved using Lemma 5.1.

Preservation is slightly more difficult to prove. We first prove the following lemma by induction on the typing derivation. The lemma states that the body of a method is well-typed if the static context Σ models the runtime context Δ.

Lemma 5.2. If $\Gamma \mid \Sigma \vdash \widehat{\theta}(v) : \sigma$ and $\sigma \leq \beta(m_0 : \beta'(M_0) \Rightarrow \tau, M)$ and $\Sigma \vdash \Delta$ and $mbody_{\Delta}(m_0, \widehat{\theta}) = e_0$, then $\mathsf{this} : \beta'(M_0), \mathsf{fields} : \mathrm{req}_{\Sigma}\, \theta \mid \Sigma \vdash e_0 : \tau$.

Theorem 5.2 (Preservation [programs])**.** If $\Gamma \mid \Sigma \vdash p$ **ok** and $\Sigma \vdash \Delta$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that $\Sigma' \vdash \Delta'$ where $\Gamma \mid \Sigma' \vdash p'$ **ok**.

We prove this theorem using of a preservation theorem on expressions, a standard substitution lemma, and Lemma 5.2 above.

5.4 Modularity

Our typechecking rules are modular; each rule relies only on information in the context up to the current program point, rather than requiring a global dictionary of brand definitions. Our exhaustiveness checks are modular because external method definitions cannot be abstract (enforced by the grammar); otherwise, information about all brand definitions would be required.

Since our language does not include modules, our ambiguity checks are not modular in the strictest sense of the term, as they depend on all definitions up to the current program point. However, our system could be easily extended with additional rules to support modular ambiguity checking. Millstein and Chambers have developed such rules and have also defined several levels of modular typechecking [19]. Our current system is compatible with their broadest notion of modular typechecking, the so-called "most-extending module" approach, exemplified by their language System E. To perform the most modular form of typechecking, however, we would require that all implementations of an external method be in the same module. Further, external methods would be forbidden from overriding internal methods (currently permitted in our system). These checks correspond to the restriction $M1$ in Dubious [19] and restriction $R3$ in MultiJava [9].

A related issue is that of information hiding, a form of which our language supports. A brand's field value can only be accessed by the brand's methods, effectively making them private. It would be possible to extend this further and disallow external methods from accessing fields, or allow marking some internal methods as private.

5.5 Polymorphism

We have designed an extension Unity$_\alpha$ with polymorphism (presented in Appendix B), but we have omitted this feature in this version of Unity since we discovered that polymorphism was orthogonal to the issues surrounding nominal and structural subtyping. In Unity$_\alpha$, the syntax is extended as follows:

$$\begin{aligned}
\textit{brand-decl} &::= \textit{mod}\ \textsf{brand}\ \forall\overline{T}.\, \beta\langle\overline{T}\rangle(\tau; \overline{\textit{m-decl}})\ \textsf{extends}\ \beta\langle\overline{\tau}\rangle \\
\textit{ext-decl} &::= \textsf{method}\ m\ \forall\overline{T'}.\, \beta\langle\overline{T'}\rangle(\overline{m : \tau}) : \tau = e \\
e &::= \ \ldots \mid \widehat{\beta}[\overline{\tau}] \mid \Lambda T.\, e \mid e[e] \\
\tau &::= \ \ldots \mid X \mid \beta\langle\overline{\tau}\rangle(\overline{m : \tau}) \mid \forall T.\, \tau
\end{aligned}$$

The sub-brand judgement is on parameterized brands (i.e. $\beta\langle\overline{\tau}\rangle$) and, aside from a new rule for $\forall T.\, \tau$ types, the subtype judgement is essentially the same.

Note that in Unity$_\alpha$, methods are not polymorphic. However, since the language includes the $\Lambda T.\, e$ construct (type abstraction functions), many examples can be written using these.

6 Related Work

Type Systems. At the FOOL/WOOD '07 workshop, we presented the predecessor of this version of Unity [17]. Here, we have extended that work by adding methods and information hiding to our core calculus, providing additional examples, and including a case study.

Researchers have recently considered the problem of integrating nominal and structural subtyping. Reppy and Turon have addressed the problem in the context of typechecking traits [24]. Their resulting type system is a hybrid of nominal and structural subtyping. However, in their system, structural types are second-class; they apply to trait functions but not to expressions or ordinary functions. Consequently, there is less expressiveness as compared with Unity: it is not possible to constrain the argument of a function to have particular members, for example.

After our initial workshop proposal, Odersky et al. independently implemented a similar language feature, validating the practical importance of our work. In Scala, type refinements allow a nominal type to include additional structural information [22]. Scala type refinements have many similarities with the language Whiteoak, an extension of Java with structural types [13]. Like Scala, in Whiteoak, by using intersection types, a type can include both structural and nominal aspects.

Scala and Whiteoak differ from Unity in that they do not have external methods, nor do they allow structural constraints to be placed on a method's receiver. Also, the language designs have neither been formalized nor proved sound.

Ostermann has designed a language that seeks to enhance the expressiveness of nominal subtyping to gain some of the benefits of structural subtyping [23]. Ostermann has identified an additional important benefit of nominal subtyping—that of blame assignment: i.e., who accepts responsibility for maintaining a subtype relation, the user or the designer of a component? The language design is much more expressive than a purely nominal system; it is possible to, for example, create subtypes of a class type without inheriting its implementation, and declare supertypes of an existing type. But, this comes at the cost of a subtyping relation that is not transitive, which may prove counter-intuitive to programmers. The programmer must manually provide a set of "witness" types so that the typechecker can apply subsumption. Therefore, it is unclear whether this approach is practical.

Bono et al. have also proposed a type system that includes both nominal and structural aspects, but their system does not fully integrate the two disciplines [2]. The system only uses structural typing when typechecking uses of the `this` variable, making their system considerably less expressive than ours.

The language MOBY is in many ways similar to Unity, as it supports structural subtyping and a form of tag subtyping through its inheritance-based subtyping mechanism, which is similar to our sub-branding [11, 12]. This allows expressing many useful subtyping constraints, but MOBY's class types are not integrated with object types in the same way as in Unity. For instance, in MOBY, it is not possible to express the constraint that an object should have a particular class *and* should have some particular methods (that are not defined in the class itself). Additionally, the object-oriented core of MOBY supports only internal dispatch. MOBY does include "tagtypes" that are very similar to our brands. These can be used to support downcasts or to encode multimethods, but they are disjoint from the object-oriented core of the system.

Strongtalk presents a structural type system for Smalltalk and also supports named subtyping relationships through its "brand" mechanism [4]. However, it is not possible to define subtyping on brands. Additionally, since it is a type system for Smalltalk, it supports only the single dispatch model.

Modula-3's type system has structural types with branding, but not structural sub*typing* [20]. That is, its type system will treat two record types as equivalent if they have the same structure but different type aliases, but does not recognize one as a subtype of the other if it has additional fields. The object-oriented part of the language uses nominal subtyping.

In the C++ concepts proposal, concepts can be either nominal or structural [14]. However, concepts apply only to template constraints, not to the subtyping relation.

External and Multimethod Dispatch. External and multimethod dispatch has been extensively studied, but in the context of either dynamically typed languages, or languages with a purely nominal type system. Cecil is one of the first languages to include statically checked multimethods, but performs a whole-program analysis to ensure that multimethods are exhaustive and unambiguous [7, 8]. As previously mentioned (Sect. 2.2), Cecil contains "where" clauses that can model some aspects of structural types, but they can only appear on top-level methods and cannot be nested, in contrast to Unity.

More recent work has focused on modular typechecking of external methods and multimethods, as well as the problem of integrating external methods into existing languages; this includes the Dubious calculus (System M) and MultiJava [19, 9]. We have built on these existing techniques for modular typechecking of external methods.

The language $\lambda\&$ [6] includes multimethod dispatch and includes structural subtyping on methods, similar to Unity. However, the subtyping hierarchy on classes uses only nominal subtyping, in contrast to Unity.

Acknowledgements

We would like to thank Karl Crary and William Lovas for helpful discussion and feedback on our language, and Kevin Bierhoff for comments on an earier version of this paper.

References

[1] Technical report.

[2] Viviana Bono, Ferruccio Damiani, and Elena Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP'07 (http://www.cs.ru.nl/ftfjp/)*, 2007.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP '90*, 1990.

[4] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93*, pages 215–230, 1993.

[5] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[6] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Inf. Comput.*, 117(1):115–135, 1995.

[7] Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, 1992.

[8] Craig Chambers and the Cecil Group. The Cecil language: specification and rationale, version 3.2. Available at `http://www.cs.washington.edu/research/projects/cecil/`, February 2004.

[9] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[10] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP '00*, pages 198–208, 2000.

[11] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *PLDI '99*, pages 37–49, 1999.

[12] Kathleen Fisher and John Reppy. Inheritance-based subtyping. *Inf. Comput.*, 177(1):28–55, 2002.

[13] Joseph Gil and Itay Maman. Whiteoak. Available at `http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Whiteoak`, 2008.

[14] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of OOPSLA '06*, pages 291–310. ACM Press, October 2006.

[15] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, release 3.09. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, 2004.

[16] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*, pages 397–406, 1989.

[17] Donna Malayeri and Jonathan Aldrich. Combining structural subtyping and external dispatch. In *FOOL/WOOD'07*, January 2007. Available at `http://foolwood07.cs.uchicago.edu/program.html`.

[18] Sun Microsystems. Java collections API design FAQ. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html`, 2003.

[19] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. *Inf. Comput.*, 175(1):76–118, 2002.

[20] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[21] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04*, pages 99–115, 2004.

[22] Martin Odersky. The Scala language specification. Available at `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2007.

[23] K. Ostermann. Nominal and Structural Subtyping in Component-Based Programming. *Journal of Object Technology*, 7(1), 2008.

[24] John Reppy and Aaron Turon. Metaprogramming with traits. In *ECOOP 2007*, July-August 2007.

[25] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP '03*, 2003.

A Unity Type Safety

A.1 Definitions

Definition A.1 (Well-formed context)**.**
The context Σ is *well-formed*, iff the following conditions hold:

1. there is exactly one entry for each brand β.
2. if $\mathit{mod}\ \beta_1(\tau; M)$ $\mathsf{extends}\ \beta_2 \in \Sigma$, then
   (a) $\beta_2(M)$ **type**
   (b) $\tau \leq \mathrm{req}_\Sigma\, \beta_2$
   (c) if *mod* = concrete, then $\mathit{methods}_\Sigma \beta_1 = \mathsf{concrete}\ \overline{n : \tau}$.

Definition A.2 (Models relation on contexts)**.**
The definition of $\Sigma \vdash \Delta$ (Σ *models* Δ) is given by the following inference rules.

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\Sigma \vdash \Delta \qquad \Sigma' = \Sigma, \mathit{mod}\ \beta_1(\tau; \{\mathsf{concrete}\ m_i : \beta_1(M_i) \Rightarrow \tau_i'\ ^{i \in 1..n}\}, \overline{\mathsf{abstract}\ n : \sigma})\ \mathsf{extends}\ \beta_2 \quad \mathsf{this} : \beta_1(M_i), \mathsf{fields} : \tau \mid \Sigma' \vdash e_i : \tau_i\ (i \in 1..n)}{\Sigma' \vdash \Delta, \widehat{\beta}_1(m_i = e_i\ ^{i \in 1..n})\ \mathsf{extends}\ \widehat{\beta}_2}$$

A.2 Inversion Lemmas

Lemma A.1 (Inversion of subtyping)**.**

1. If $\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$, then $\Sigma \vdash \sigma_1 \leq \tau_1$ and $\Sigma \vdash \tau_2 \leq \sigma_2$.
2. If $\Sigma \vdash \beta_1(M_1) \leq \beta_2(M_2)$, then $\Sigma \vdash \beta_1 \sqsubseteq \beta_2$ and $\Sigma \vdash M_1 <: M_2$ and $\Sigma \vdash \beta_1(M_1)$ **type** and $\Sigma \vdash \beta_2(M_2)$ **type**.
3. If $\Sigma \vdash (\ell_i : \tau_i\ ^{i \in 1..n}) \leq (k_j : \sigma_j\ ^{j \in 1..m})$, then $\{k_j\ ^{j \in 1..m}\} \subseteq \{\ell_i\ ^{i \in 1..n}\}$ ($\overline{\ell}$ includes at least the labels in $\overline{k}$) and $\Sigma \vdash \tau_i \leq \sigma_j$ for each common label $\ell_i = k_j$.
4. If $\Sigma \vdash \beta_1(M_1) \Rightarrow \tau_1 \leq \beta_2(M_2) \Rightarrow \tau_2$ then $\Sigma \vdash \beta_1 \sqsubseteq \beta_2$ and $\Sigma \vdash M_2 <: M_1$ and $\Sigma \vdash \tau_1 \leq \tau_2$.

Proof. By induction on the subtyping derivation, with case analysis of the final rule used. Vacuous cases have been omitted.

1. Function type.

   case Sub-Refl. Result is immediate.

   case Sub-Trans. We have $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ and $\tau_1' \to \tau_2' \leq \sigma_1 \to \sigma_2$. By the induction hypothesis, $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$ and $\sigma_1 \leq \tau_1'$ and $\tau_2' \leq \sigma_2$. The result then follows from Sub-Trans.

   case Sub-Fun. Result is immediate.

2. Brand type.

case Sub-Refl. Result is immediate from Sub-Brand-Refl and Sub-Rec-Refl.

case Sub-Trans. We have $\beta_1(L_1; M_1) \leq \beta_1'(L_1'; M_1')$ and $\beta_1'(L_1'; M_1') \leq \beta_2(L_2; M_2)$. By the induction hypothesis, $\beta_1 \sqsubseteq \beta_1'$ and $\beta_1' \sqsubseteq \beta_2$ and $(L_1; M_1) <: (L_1'; M_1')$ and $(L_1'; M_1') <: (L_2; M_2)$ and $\beta_1(L_1; M_1)$ **type** and $\beta_2(L_2; M_2)$ **type**. The result then follows from Sub-Brand-Trans, Sub-Rec-Trans and Sub-Name.

case Sub-Name. Result is immediate.

3. Record type. Straightforward.

4. Function type. Straightforward.

□

Lemma A.2 (Inversion of the typing judgement)**.**

1. If $\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \sigma$ and $\Sigma \vdash \sigma \leq \sigma_1 \to \sigma_2$ then $\Sigma \vdash \sigma_1 \leq \tau_1$ and $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \sigma_2$.

2. If $\Gamma \mid \Sigma \vdash \widehat{\theta}(e) : \tau$ and $\Sigma \vdash \tau \leq \beta(M)$ then for some σ we have:

   (a) $\Gamma \mid \Sigma \vdash e : \sigma$

   (b) $\sigma \leq \mathrm{req}_\Sigma\, \theta$

   (c) $\Gamma \mid \Sigma \vdash \widehat{\theta}(e) : \theta(\mathit{methods}_\Sigma \theta)$

   (d) $\Sigma \vdash \mathit{methods}_\Sigma \theta <: M$

   (e) $\Sigma \vdash \theta \sqsubseteq \beta$

3. If $\Gamma \mid \Sigma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \sigma$ and $\sigma \leq \mu X.\tau$, then $\Gamma \mid \Sigma \vdash e : [\mu X.\tau / X]\, \tau$.

Proof. For each part, we proceed by induction on the typing derivation, with case analysis of the final rule used. Vacuous cases have been omitted.

1. $\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \sigma$

   case Tp-Fun. $\sigma = \tau_1 \to \tau_2$
   By Sub-Trans, $\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$; by subtype inversion (Lemma A.1), $\sigma_1 \leq \tau_1$ and $\tau_2 \leq \sigma_2$. By the rule's premise, $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2$, and the result follows from Tp-Subs.

   case Tp-Subs. We have $\lambda x{:}\tau_1.\, e : \tau$ and $\tau \leq \sigma$. By Sub-Trans, $\tau \leq \sigma_1 \to \sigma_2$ and the result follows from the induction hypothesis.

2. $\Gamma \mid \Sigma \vdash \widehat{\theta}(e) : \sigma$

   case Tp-New-Obj. $\tau = \theta(\mathit{methods}_\Sigma \theta)$.
   Conclusions (a), (b) and (c) follow from the premises of Tp-New-Obj. By subtype inversion (Lemma A.1), $\mathit{methods}_\Sigma \theta <: M$ and $\theta \sqsubseteq \beta$, which proves conclusions (d) and (e).

   case Tp-Subs. Result follows from Sub-Trans and the induction hypothesis.

3. $\Gamma \mid \Sigma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \sigma$. Straightforward.

□

A.3 Type safety theorems and lemmas

Lemma A.3. If $\Sigma \vdash \Delta$ then $\Sigma \vdash \beta_1 \sqsubseteq \beta_2$ iff $\Delta \vdash \widehat{\beta}_1 \sqsubseteq \widehat{\beta}_2$.

Proof. Straightforward induction on $\Sigma \vdash \Delta$. □

Lemma A.4 (Canonical forms)**.** Suppose $\cdot \mid \Sigma \vdash v : \sigma$ and $\Sigma \vdash \sigma \leq \tau$.

1. If $\tau = \mathsf{unit}$ then $v = \mathsf{()}$.
2. If $\tau = \tau_1 \rightarrow \tau_2$ then v is of the form $\lambda x{:}\tau_{11}.\, e$.
3. If $\tau = \beta(\overline{m : \tau})$ then v is of the form $\widehat{\beta}'(v)$.
4. If $\tau = \{\overline{\ell : \tau}\}$ then v is of the form $(\overline{k = v})$.
5. If $\tau = \mu X.\tau$ then v is of the form $\mathsf{fold}_\sigma\, v$.

Proof. Straightforward induction on typing derivations. □

Lemma A.5. If $\Sigma \vdash \beta_1 \sqsubseteq \beta_2$ and $\mathit{mbody}_\Delta(m, \beta_2) = e$ then $\mathit{mbody}_\Delta(m, \beta_1) = e$.

Proof. By induction on $\beta_1 \sqsubseteq \beta_2$.

case SUB-BRAND-DECL. If either the first or second case of *mbody* applies, we have $\mathit{mbody}(m, \widehat{\beta}_2) = e$. By the second rule of mbody_Δ, $\mathit{mbody}(m, \widehat{\beta}_1) = e$.

case SUB-BRAND-REFL. Immediate.

case SUB-BRAND-TRANS. We have $\beta_1 \sqsubseteq \beta_1'$ and $\beta_1' \sqsubseteq \beta_2$. Applying the induction hypothesis to $\beta_1' \sqsubseteq \beta_1$ gives $\mathit{mbody}_\Delta(m, \beta_1') = e$. Applying the induction hypothesis to $\beta_1 \sqsubseteq \beta_1'$ gives the required result.

□

Lemma A.6. If $\Gamma \mid \Sigma \vdash \widehat{\beta}(v) : \tau$ and $\Sigma \vdash \tau \leq \beta'(M)$, where $\Sigma \vdash \Delta$ and $m_k \in M$, then $\mathit{mbody}_\Delta(m_k, \widehat{\beta})$ is defined.

Proof. By induction on $\widehat{\beta}(v) : \tau$.

case TP-SUBS. Immediate from the induction hypothesis.

case TP-NEW-OBJ. We have $\beta = \beta'$. From the definition of a well-formed context Σ, $\overline{\mathit{mod}_m} =$ concrete. From the definition of $\mathit{methods}_\Sigma$, either m_k is defined in β or some super-brand θ (i.e., some θ where $\beta \sqsubseteq \theta$). If it is defined in β, then mbody_Δ is defined, by the definition of $\Sigma \vdash \Delta$. Otherwise, from the definition of $\Sigma \vdash \Delta$, we have $\theta(m_k = e_k; \overline{m' = e'})\ \mathsf{extends}\ \theta' \in \Delta$. By the definition of mbody_Δ, we have $\mathit{mbody}_\Delta(m_k, \widehat{\theta}) = e_k$. By Lemma A.5, $\mathit{mbody}_\Delta(m_k, \widehat{\beta}) = e_k$, which is the required result.

□

Lemma A.7 (Progress [expressions])**.** If $\cdot \mid \Sigma \vdash e : \tau$ then either e is a value, or for some Δ such that $\Sigma \vdash \Delta$, there is an e' with $e \longmapsto_\Delta e'$.

Proof. By induction on $e : \tau$, with case analysis of final rule used.

case TP-UNIT, TP-FUN. Immediate.

case TP-APP. Straightforward.

case TP-SUBS. Result follows from induction hypothesis.

case TP-NEW-OBJ. $e = \widehat{\beta}(e_1)$
By the induction hypothesis, e_1 is a value or it steps to some e_1'. If it takes a step, then E-BRAND-CONS applies. If it is a value, then then e is also a value.

case TP-NEW-RECORD $e = (\overline{\ell = e})$
By the induction hypothesis, each e_i is a value or it steps to some e_i'. If any e_i steps, then the rule E-RECORD applies. Otherwise, the entire expression is a value.

case TP-PROJ. $e = e_1.\ell_k \quad e_1 : \{\overline{\ell : \tau}\}$
By the induction hypothesis, either e_1 is a value or it steps to some e'. If it is a value, then by canonical forms it has the form $(\overline{k = v})$ and E-PROJ2 applies. If it steps to e', then E-PROJ1 applies.

case TP-INVOKE. $e = e_1.m_k \quad e_1 : \beta(\overline{m : \tau})$
By the induction hypothesis, either e_1 is a value or it steps to some e_1'. If e_1 evaluates to e_1', E-INVOKE1 applies. Otherwise, by canonical forms, e_1 has the form $\widehat{\beta'}(v)$. By Lemma A.6, $mbody_\Delta(m, \widehat{\beta'})$ is defined; the rule E-INVOKE2 then applies.

case TP-FOLD. $e = \mathsf{fold}_\tau\, e_1$
By the induction hypothesis, either e_1 is a value or it takes a step. If it takes a step, then the rule E-FOLD applies. Otherwise, e is a value.

case TP-UNFOLD $e = \mathsf{unfold}_{\mu X.\tau}\, e_1$
By the induction hypothesis, either e_1 is a value or it takes a step. If it takes a step, then the rule E-UNFOLD applies. Otherwise, it is a value v of type $\mu X.\tau$. By canonical forms, v has form $\mathsf{fold}_{\mu X.\tau}\, v_1$, so the rule E-UNFOLD-FOLD applies.

□

Theorem A.1 (Progress [programs]). If $\cdot \mid \Sigma \vdash p$ **ok**, for some Σ, then one of the following cases holds:

1. p is a value
2. for Δ such that $\Sigma \vdash \Delta$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

Proof. By induction on p **ok**.

case TP-BRAND-INTRO. The rule E-BRAND-DECL applies.

case TP-EXT-METHOD. The rule E-EXT-DECL applies. Δ has the appropriate form because $\Sigma \vdash \Delta$ and $\Sigma = \{mod\ \beta_1(\sigma, M)\ \mathsf{extends}\ \beta_2\}, \Sigma_0$.

case Tp-Expr1. The result follows from the progress lemma for expressions (Lemma A.7).

case Tp-Expr2. By Lemma A.7, $e \longmapsto_{\Delta} e'$. By the induction hypothesis, $p_2 \mid \Delta \longmapsto p_2' \mid \Delta'$. Then the rule E-Expr2 applies.

□

Lemma A.8 (Substitution)**.**
If $\Gamma, x : \sigma \mid \Sigma \vdash e_1 : \tau$ and $\Gamma \mid \Sigma \vdash e_2 : \sigma$ then $\Gamma \mid \Sigma \vdash [\, e_2/x \,]\, e_1 : \tau$.

Proof. Straightforward induction on typing derivations. □

Lemma A.9. If $\Gamma, x : \tau, \Gamma' \mid \Sigma \vdash e : \sigma$ and $\tau' \leq \tau$, then $\Gamma, x : \tau', \Gamma' \mid \Sigma \vdash e : \sigma$.

Proof. Straightforward induction on typing derivations. □

Lemma A.10.
If $\Gamma \mid \Sigma \vdash \widehat{\theta}(v) : \sigma$ and $\sigma \leq \beta(m_0 : \beta'(M_0) \Rightarrow \tau, M)$ and $\Sigma \vdash \Delta$ and $mbody_{\Delta}(m_0, \widehat{\theta}) = e_0$, then $\mathsf{this} : \beta'(M_0), \mathsf{fields} : \mathsf{req}_{\Sigma}\, \theta \mid \Sigma \vdash e_0 : \tau$.

Proof. By induction on $\widehat{\theta}(v) : \sigma$.

case Tp-Subs. Result follows from the induction hypothesis and Sub-Trans.

case Tp-New-Obj. Let $\tau_0 = \beta'(M_0) \Rightarrow \tau$. We have $\widehat{\beta}(v) : \beta(m_0 : \tau_0, M)$. Since $\text{modifier}_{\Sigma}\beta = \mathsf{concrete}$, $methods_{\Sigma}\beta = \mathsf{concrete}\ \overline{m : \tau}$. There are two possible rules that apply for $mbody_{\Delta}$. In the first case, m_0 is defined in β. From the definition of $\Sigma \vdash \Delta$, we can conclude that $\mathsf{this} : \beta(M_0), \mathsf{fields} : \mathsf{req}_{\Sigma}\, \beta \mid \Sigma \vdash e_0 : \tau$. This context is well-formed, since by a straightforward typing inversion, $\beta(M_0)$ is a well-formed type.

Otherwise, from the definition of $methods_{\Sigma}$, there exists some β_2 where m_0 with type τ_0 is defined in β_2 and $\beta \sqsubseteq \beta_2$. From the definition of $\Sigma \vdash \Delta$, and the fact that $mbody_{\Delta}(m, \beta_2) = mbody_{\Delta}(m, B)$, we know that $\mathsf{this} : \beta_2(M_0), \mathsf{fields} : \mathsf{req}_{\Sigma}\, \beta_2 \mid \Sigma \vdash e_0 : \tau$. The result then follows from Lemma A.9.

□

Lemma A.11 (Preservation [expressions])**.**
If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Sigma \vdash \Delta$ and $e \longmapsto_{\Delta} e'$, then $\Gamma \mid \Sigma \vdash e' : \tau$.

Proof. By induction on $e : \tau$.

case Tp-Var, Tp-Unit, Tp-Fun. Vacuous; e does not evaluate.

case Tp-App. Straightforward.

case Tp-Subs. $e : \sigma \quad \sigma \leq \tau$
By the induction hypothesis, $e' : \sigma$ and the result follows from Tp-Subs.

case Tp-New-Obj. $e = \widehat{\beta}(e_1) \;\; e_1 : \tau'$
The only evaluation rule that applies is E-Brand-Cons. By the induction hypothesis, $e_1' : \tau'$. The result then follows from Tp-New-Obj.

case Tp-New-Record. The only evaluation rule that applies is E-Record. We have $e_k \longmapsto_\Delta e'_k$. By the induction hypothesis, $e_k : \tau_k$. The result then follows from Tp-New-Record.

case Tp-Proj. $e : \{\ell_i : \tau_i \ ^{i \in 1..n}\}$
There are two possible evaluation rules that apply:

case E-Proj1. Result follows from the induction hypothesis and Tp-Proj.

case E-Proj2. $(\ell_j = v_j \ ^{j \in 1..m}).\ell_k \longmapsto_\Delta v_k$
By typing inversion (Lemma A.2), we have $\{\ell_j : \tau_j \ ^{j \in 1..m}\} \leq \{\ell_i : \tau_i \ ^{i \in 1..n}\}$ and $v_k : \tau_k$, which is the required result.

case Tp-Invoke.

$$\frac{m_k : \tau_{m_k} \in (M \wedge \mathit{methods}_\Sigma(\beta)) \qquad \begin{array}{c}\Gamma \mid \Sigma \vdash e : \beta(M) \\ \tau_{m_k} = \beta'(\overline{n : \sigma}) \Rightarrow \tau\end{array} \qquad \beta(M \wedge \mathit{methods}_\Sigma(\beta)) \leq \beta'(\overline{n : \sigma})}{\Gamma \mid \Sigma \vdash e.m_k : \tau}$$

There are two possible evaluation rules that apply.

case E-Invoke1. Result follows from the induction hypothesis and Tp-Invoke.

case E-Invoke2.

$$\frac{\mathit{mbody}_\Delta(m, \widehat{\beta}) = e}{\widehat{\beta}(v).m \longmapsto_\Delta \{\widehat{\beta}(v)/\mathsf{this}, v/\mathsf{fields}\}\, e}$$

We have $\widehat{\theta}(v) : \beta(M)$, where $m : t_k \in (M \wedge \mathit{methods}_\Sigma(\theta))$
By typing inversion (Lemma A.2), $\widehat{\theta}(v) : \beta(\mathit{methods}_\Sigma(\theta))$ and $\mathit{methods}_\Sigma(\theta) <: M$. Therefore, $(M \wedge \mathit{methods}_\Sigma(\theta)) = \mathit{methods}_\Sigma(\theta)$. By Lemma A.10, $\mathsf{this} : \beta'(\overline{n : \sigma}), \mathsf{fields} : \mathrm{req}_\Sigma\, \theta \mid \Sigma \vdash e_m : \tau'$. The result follows from the substitution lemma (Lemma A.8).

case Tp-Fold. The only evaluation rule that applies is E-Fold. The result follows from the induction hypothesis and Tp-Fold.

case Tp-Unfold. There are two possible evaluation rules that apply.

case E-Unfold. The result follows from the induction hypothesis and Tp-Unfold.

case E-Unfold-Fold. We have $e = \mathsf{unfold}_{\mu X.\tau}\,(\mathsf{fold}_{\mu X.\tau'}\, v) : \tau, \mathsf{fold}_{\mu X.\tau'} : \mu X.\tau$. By typing inversion (Lemma A.2), $\tau = \tau'$ and $v : \tau$. But this is just what the expression evaluates to, so this is the required result.

□

Theorem A.2 (Preservation [programs])**.**
If $\Gamma \mid \Sigma \vdash p$ **ok** and $\Sigma \vdash \Delta$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that $\Sigma' \vdash \Delta'$ where $\Gamma \mid \Sigma' \vdash p'$ **ok**.

Proof. By induction on p **ok**.

case TP-BRAND-INTRO.

$$\begin{array}{c}
\beta_1 \notin \Sigma \qquad \tau \leq \mathrm{req}_{\Sigma}\,(\beta_2) \\
\Sigma \vdash \beta_1.\overline{\mathit{m\text{-}decl}} : (\overline{\mathit{mod}_m m : \tau}) \qquad \Sigma' = \Sigma, \mathit{mod}\ \beta_1(\tau; \overline{\mathit{mod}_m\ m : \tau})\ \mathsf{extends}\ \beta_2 \\
\Sigma' \vdash \beta_1.(\tau; \overline{\mathit{m\text{-}decl}})\ \mathbf{ok} \qquad \mathit{mod} = \mathsf{concrete}\ \text{implies}\ \mathit{abstractCover}(\overline{\mathit{mod}_m m : \tau}, \beta_2) \\
\mathit{override}(\overline{m : \tau}, \beta_2) \qquad \mathsf{abstract\ method}\ m \in \overline{\mathit{m\text{-}decl}}\ \text{implies}\ \mathit{mod} = \mathsf{abstract} \qquad \Sigma' \vdash p\ \mathbf{ok} \\
\hline
\Sigma \vdash \mathit{mod}\ \mathsf{brand}\ \beta_1(\tau; \overline{\mathit{m\text{-}decl}})\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p\ \mathbf{ok}
\end{array}$$

The rule E-BRAND-DECL applies.

$$\begin{array}{c}
\overline{\mathit{m\text{-}decl}} \longmapsto \overline{m = e} \\
\hline
\mathit{mod}\ \mathsf{brand}\ \beta_1(\tau; \overline{\mathit{m\text{-}decl}})\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p \mid \Delta \longmapsto \\
p \mid \Delta, (\beta_1(\overline{m = e})\ \mathsf{extends}\ \beta_2)
\end{array}$$

Take $\Delta_2 = \Delta, (\beta_1(\overline{m = e})\ \mathsf{extends}\ \beta_2)$. It remains to show that $\Sigma' \vdash \Delta_2$. This result follows from the definition of $\Sigma' \vdash \beta_1.(\tau; \overline{\mathit{m\text{-}decl}})\ \mathbf{ok}$ and the definition of $\Sigma \vdash \Delta$.

case TP-EXT-METHOD.

$$\begin{array}{c}
\Sigma = \{\mathit{mod}\ \beta_1(\sigma; M')\ \mathsf{extends}\ \beta_2\}, \Sigma_0 \\
m \notin M' \qquad \Sigma' = \{\mathit{mod}\ \beta_1(\sigma; M', m : \beta_1(M) \Rightarrow \tau)\ \mathsf{extends}\ \beta_2\}, \Sigma_0 \\
\mathit{override}(\beta_1(M) \Rightarrow \tau, \beta_2) \qquad \mathsf{this} : \beta_1(M), \mathsf{fields} : \sigma \mid \Sigma' \vdash e : \tau \qquad \Sigma' \vdash p_1\ \mathbf{ok} \\
\hline
\Sigma \vdash \mathsf{method}\ m\ \beta(M) : \tau = e\ \mathsf{in}\ p_1\ \mathbf{ok}
\end{array}$$

The rule E-EXT-DECL applies.

$$\begin{array}{c}
\Delta = \{\beta(\overline{m = e})\ \mathsf{extends}\ \beta'\}, \Delta_0 \\
\hline
\mathsf{method}\ m_1\ \beta(\overline{m : \tau}) : \tau' = e_1\ \mathsf{in}\ p_1 \mid \Delta \longmapsto \\
p_1 \mid \{\beta(\overline{m = e}, m_1 = e_1)\ \mathsf{extends}\ \beta'\}, \Delta_0
\end{array}$$

From the definition of $\Sigma \vdash \Delta$, Δ has the form $\beta(\overline{m = e})\ \mathsf{extends}\ \beta', \Delta_0$, where $\Sigma_0 \vdash \Delta_0$. Take $\Delta_2 = \beta(\overline{m = e}, m_1 = e_1)\ \mathsf{extends}\ \beta', \Delta_0$. We have $\Sigma' \vdash p_1$ **ok**. It remains to show that $\Sigma' \vdash \Delta_2$. This result follows from the premise $\mathsf{this} : \beta(\overline{n : \sigma}), \mathsf{fields} : \sigma \mid \Sigma' \vdash e_i : \tau_i'$ and the definition of $\Sigma \vdash \Delta$.

case TP-EXPR1. The rule E-EXPR1 applies. The result then follows from the preservation lemma for expressions (Lemma A.11).

case TP-EXPR2. The rule E-EXPR2 applies. The result then follows from the induction hypothesis and the preservation lemma for expressions (Lemma A.11).

□

A.4 Validity theorems

Lemma A.12. If $\Sigma \vdash \sigma$ **type** and $\Sigma \vdash \sigma \leq \tau$, then $\Sigma \vdash \tau$ **type**.

Proof. Straightforward induction on $\Sigma \vdash \sigma \leq \tau$. □

Lemma A.13. If $\Gamma, T\ \mathsf{type}, \Gamma' \mid \Sigma \vdash \tau$ **type** and $\Gamma, \Gamma' \mid \Sigma \vdash \sigma$ **type**, then $\Gamma, [\sigma/T]\,\Gamma' \mid \Sigma \vdash [\sigma/T]\,\tau$ **type**.

Proof. Straightforward induction on τ **type**. □

Lemma A.14. If X appears in τ and $\Gamma \mid \Sigma \vdash [\tau'/X]\,\tau$ **type** then $\Gamma \mid \Sigma \vdash \tau'$ **type**.

Proof. By induction on $\Gamma \mid \Sigma \vdash [\tau'/X]\,\tau$ **type**.

case FUN-TYPE. We have $[\tau'/X]\,(\tau_1 \rightarrow \tau_2) = [\tau'/X]\,\tau_1 \rightarrow [\tau'/X]\,\tau_2$. If X appears in τ, then either X appears in τ_1 or τ_2, or both. In either case, the result follows from the induction hypothesis.

case $\wedge$-TYPE. Similar to above.

case BRAND-TYPE. We have $[\tau'/X]\,\beta(\overline{m : \sigma}) = \beta(\overline{m : [\tau'/X]\,\sigma})$. The result then follows from the induction hypothesis.

case RECORD-TYPE. Similar to above.

case MU-TYPE. Result follows from the induction hypothesis.

case VAR-TYPE. Immediate.

case METHOD-TYPE. Similar to case for FUN-TYPE.

□

Theorem A.3. If $\Gamma \mid \Sigma \vdash e : \tau$, and Γ and Σ are well-formed, then $\Sigma \vdash \tau$ **type**.

Proof. By induction on $e : \tau$.

case TP-VAR. Result follows from the fact that Γ is well-formed.

case TP-UNIT. Immediate.

case TP-FUN. Since τ_1 **type**, $\Gamma, x : \tau_1$ is well-formed. The result then follows from the induction hypothesis and FUN-TYPE.

case TP-APP. By the induction hypothesis, $\tau_1 \rightarrow \tau_2$ **type**. By a straightforward inversion on the type formation judgement, τ_2 **type**, which is the required result.

case TP-SUBS. Result follows from the induction hypothesis and Lemma A.12.

case TP-NEW-OBJ. Result follows from the induction hypothesis and BRAND-TYPE.

case TP-NEW-RECORD. Result follows from the induction hypothesis and RECORD-TYPE.

case TP-PROJ. By the induction hypothesis, $\{\ell_i : \tau_i{}^{i \in 1..n}\}$ **type**. By a straightforward inversion of the type formation judgement, τ_k **type**, which is the required result.

case TP-INVOKE. By the induction hypothesis, $\beta(M)$ **type**. Let $M = \overline{\ell : \tau}$. By a straightforward inversion of the type formation judgement, $\overline{\tau}$ **type** and therefore $\tau_{m_k} = \beta'(M') \Rightarrow \tau'$ **type**. By another inversion of the type formation judgement, τ' **type**, which is the required result.

case TP-FOLD. By the induction hypothesis, $[\mu X.\tau / X]\,\tau$ **type**. Either τ contains X or it does not. If it does not contain X, then the result of the substitution is simply τ and the result follows by MU-TYPE. Otherwise, by Lemma A.14, we have that $\mu X.\tau$ **type**, which is the required result.

case TP-UNFOLD. By the induction hypothesis, $\mu X.\tau$ **type**. By inversion of the type formation judgement, Γ, X **type** $\mid \Sigma \vdash \tau$ **type**. The result follows from the fact that type substitution preserves well-formed types (Lemma A.13).

□

B Unity$_\alpha$ formal system

B.1 Grammar

Programs	$p ::=$	$decl$ in $p \mid e \mid e; p$
Declarations	$decl ::=$	$brand\text{-}decl \mid ext\text{-}decl$
Brand declaration	$brand\text{-}decl ::=$	mod brand $\forall\overline{T}.\,\beta\langle\overline{T}\rangle(\tau; \overline{m\text{-}decl})$ extends $\beta\langle\overline{\tau}\rangle$
Modifiers	$mod ::=$	abstract $\mid$ concrete
Method declaration	$m\text{-}decl ::=$	abstract method $m\ (\overline{m : \tau}) : \tau$
		$\mid$ method $m\ (\overline{m : \tau}) : \tau = e$
External method	$ext\text{-}decl ::=$	method $m\ \forall\overline{T}.\,\beta\langle\overline{T}\rangle(\overline{m : \tau}) : \tau = e$
Expressions	$e ::=$	$() \mid x \mid \lambda x{:}\tau.\,e \mid e\,e \mid \widehat{\beta}[\overline{\tau}](e) \mid (\overline{\ell = e})$
		$\mid e.\ell \mid e.m \mid \mathsf{fold}_\tau\, e \mid \mathsf{unfold}_\tau\, e \mid \Lambda T.\,e \mid e[e]$
Types	$\tau, \sigma ::=$	$\mathsf{unit} \mid \tau \to \tau \mid \tau \wedge \tau \mid \beta\langle\overline{\tau}\rangle(\overline{m : \tau}) \mid \{\overline{\ell : \tau}\} \mid X \mid T$
		$\mid \mu X.\tau \mid \forall T.\,\tau \mid \tau \Rightarrow \tau$
Values	$v ::=$	$() \mid \lambda x{:}\tau.\,e \mid \widehat{\beta}[\tau](v) \mid (\overline{\ell = v}) \mid \mathsf{fold}_\tau\, v \mid \Lambda T.\,e$
Contexts	$\Gamma ::=$	$\cdot \mid \Gamma, x : \tau \mid \Gamma, T$ type $\mid \Gamma, X$ type
	$\Sigma ::=$	$\cdot \mid \Sigma, mod\ \beta\langle\overline{T}\rangle(\tau; \overline{mod\ m : \tau})$ extends $\beta\langle\overline{\tau}\rangle$
	$\Delta ::=$	$\cdot \mid \Delta, \widehat{\beta}[\overline{T}](\overline{m = e})$ extends $\widehat{\beta}[\overline{\tau}]$

Conventions

$$
\begin{aligned}
&\widehat{\beta} \equiv \text{ tag value corresponding to } \beta \\
&\mathrm{req}_{\Sigma}\,(\beta\langle\overline{\sigma}\rangle) = \{\overline{\sigma}/\overline{T}\}\tau \;\text{ if }\; \beta\langle\overline{T}\rangle(\tau; \overline{m : \tau}) \in \Sigma \\
&\mathrm{modifier}_{\Sigma}(\beta) = \mathit{mod} \text{ if } \mathit{mod}\ \beta\langle\overline{T}\rangle(\tau; \overline{m : \tau}) \in \Sigma \\
&\mathrm{typevar}_{\Sigma}(\beta) = \overline{T} \text{ if } \beta\langle\overline{T}\rangle(\tau; \overline{m : \tau}) \in \Sigma \\
&M \text{ ranges over } \overline{m : \tau}
\end{aligned}
$$

B.2 Static Semantics

B.2.1 Well-formed types

$$\boxed{\Gamma \mid \Sigma \vdash \tau\ \textbf{type}}$$

$$\frac{}{\Gamma \mid \Sigma \vdash \mathsf{unit}\ \textbf{type}}\ (\textsc{Unit-Type}) \qquad \frac{T\ \mathsf{type} \in \Gamma}{\Gamma \mid \Sigma \vdash T\ \textbf{type}}\ (\textsc{TypeVar-Type})$$

$$\frac{\Gamma \mid \Sigma \vdash \tau_1\ \textbf{type} \qquad \Gamma \mid \Sigma \vdash \tau_2\ \textbf{type}}{\Gamma \mid \Sigma \vdash \tau_1 \to \tau_2\ \textbf{type}}\ (\textsc{Fun-Type}) \qquad \frac{\Gamma \mid \Sigma \vdash \tau_1\ \textbf{type} \qquad \Gamma \mid \Sigma \vdash \tau_2\ \textbf{type}}{\Gamma \mid \Sigma \vdash \tau_1 \wedge \tau_2\ \textbf{type}}\ (\wedge\textsc{-Type})$$

$$\frac{\begin{array}{c}\beta\langle\overline{T}\rangle\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle \in \Sigma \qquad |\overline{T}| = |\overline{\tau}| \qquad \Gamma \mid \Sigma \vdash \overline{\tau}\ \textbf{type} \\ \overline{m}\ \text{distinct} \qquad \Gamma \mid \Sigma \vdash \overline{\sigma}_m\ \textbf{type} \qquad \Gamma \mid \Sigma \vdash \mathit{override}(\overline{m : \sigma_m}, \beta_2\langle[\overline{\tau}/\overline{T}]\overline{\sigma}\rangle)\end{array}}{\Gamma \mid \Sigma \vdash \beta\langle\overline{\tau}\rangle(\overline{m : \sigma_m})\ \textbf{type}}\ (\textsc{Brand-Type})$$

$$\frac{\overline{\ell}\ \text{distinct} \qquad \Gamma \mid \Sigma \vdash \overline{\tau}\ \textbf{type}}{\Gamma \mid \Sigma \vdash \{\overline{\ell : \tau}\}\ \textbf{type}}\ (\textsc{Record-Type}) \qquad \frac{\Gamma, X\ \mathsf{type} \mid \Sigma \vdash \tau\ \textbf{type}}{\Gamma \mid \Sigma \vdash \mu X.\, \tau\ \textbf{type}}\ (\textsc{Mu-Type})$$

$$\frac{\Gamma, T\ \mathsf{type} \mid \Sigma \vdash \tau\ \textbf{type}}{\Gamma \mid \Sigma \vdash \forall T.\, \tau\ \textbf{type}}\ (\forall\textsc{-Type})$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash \beta\langle\overline{\sigma}\rangle(m_i : \tau_i\ ^{i \in 1..n})\ \textbf{type} \\ \tau_i = \theta_i\langle\overline{\sigma}'_i\rangle(M_i) \Rightarrow \tau'_i \qquad \Sigma \vdash \beta\langle\overline{\sigma}\rangle \sqsubseteq \theta_i\langle\overline{\sigma}'_i\rangle\ \ (i \in 1..n) \qquad \Gamma \mid \Sigma \vdash \tau_2\ \textbf{type}\end{array}}{\Gamma \mid \Sigma \vdash \beta\langle\overline{\sigma}\rangle(m_i : \tau_i\ ^{i \in 1..n}) \Rightarrow \tau_2\ \textbf{type}}\ (\textsc{Method-Type})$$

B.2.2 Other auxillary judgements

$$\boxed{\Sigma \vdash \beta\langle\overline{T}\rangle.\mathit{m\text{-}decl} : \mathit{mod}\ m : \tau}$$

$$\frac{\Sigma \vdash \beta\langle\overline{T}\rangle(\overline{m : \sigma}) \Rightarrow \tau\ \textbf{type}}{\Sigma \vdash (\mathit{mod}\ \mathsf{method}\ \beta\langle\overline{T}\rangle.m(\overline{m : \sigma}) : \tau[= e]) \quad : \quad (\mathit{mod}\ m : \beta\langle\overline{T}\rangle(\overline{m : \sigma}) \Rightarrow \tau)}$$

$\boxed{\Sigma \vdash \beta\langle\overline{T}\rangle.(\tau; \mathit{m\text{-}decl})\ \mathbf{ok}}$

$$\frac{\overline{T}\ \mathsf{type}, \mathsf{this} : \beta\langle\overline{T}\rangle(\overline{m : \sigma}), \mathsf{fields} : \tau \mid \Sigma \vdash e : \tau'}{\Sigma \vdash \forall\overline{T}.\ \beta\langle\overline{T}\rangle.\mathsf{method}\ m_1(\overline{m : \sigma}) : \tau' = e\ \mathbf{ok}}$$

$\boxed{\mathit{methods}_\Sigma(\beta\langle\overline{\tau}\rangle) = \overline{\mathit{mod}\ m : \tau}}$

$$\frac{\begin{array}{c}\Sigma \vdash \beta_1\langle\overline{T}\rangle(\tau; \mathit{mod}_i\ m_i : \tau_m\ {}^{i\in 1..n})\ \mathsf{extends}\ \beta_2\langle\overline{\tau}\rangle \\ \mathit{methods}_\Sigma(\beta_2\langle\overline{\tau}\rangle) = \mathit{mod}'_j\ m_j : \sigma_j\ {}^{j\in 1..k}, \overline{\mathit{mod}_2\ n : \sigma'_m}\end{array}}{\mathit{methods}_\Sigma(\beta_1\langle\overline{\sigma}\rangle) = \{\overline{\sigma}/\overline{T}\}(\overline{\mathit{mod}\ m : \tau}, \overline{\mathit{mod}_2\ n : \sigma'_m})} \qquad \frac{}{\mathit{methods}_\Sigma(\mathsf{Top}\langle\overline{\tau}\rangle) = \cdot}$$

$\boxed{\Sigma \vdash \mathit{override}(m : \tau, \beta\langle\overline{\tau}\rangle)}$

$$\frac{m : \sigma \in \mathit{methods}_\Sigma(\beta\langle\overline{\tau}\rangle) \qquad \Sigma \vdash \tau \leq \sigma}{\Sigma \vdash \mathit{override}(m : \tau, \beta\langle\overline{\tau}\rangle)} \qquad \frac{m \notin \mathit{methods}_\Sigma(\beta\langle\overline{\tau}\rangle)}{\Sigma \vdash \mathit{override}(m : \tau, \beta\langle\overline{\tau}\rangle)}$$

$\boxed{\mathit{abstractCover}_\Sigma(\overline{\mathsf{concrete}\ m : \tau}, \beta\langle\overline{\tau}\rangle)}$

$$\frac{\mathit{methods}_\Sigma(\beta\langle\overline{\tau}\rangle) = \mathsf{abstract}\ n_i : \sigma_i\ {}^{i\in 1..n}, \overline{\mathsf{concrete}\ n' : \sigma'} \qquad n_i \in \overline{m}\ {}^{i\in 1..n}}{\mathit{abstractCover}_\Sigma(\overline{\mathsf{concrete}\ m : \tau}, \beta\langle\overline{\tau}\rangle)}$$

Definition B.1 (Intersection on M)**.**
Intersection on lists of method types is defined as intersecting types for methods, and concatenating additional methods. Formally:

$$(m_i : \tau_i\ {}^{i\in 1..n}, M) \wedge (m_i : \tau'_i\ {}^{i\in 1..n}, M') \overset{\text{def}}{=} \left(m_i : (\tau_i \wedge \tau'_i)\ {}^{i\in 1..n}, M, M'\right)$$

where the method names m_i, M and M' are mutually exclusive.

B.2.3 Subtyping

Sub-brand judgement. $\boxed{\Gamma \mid \Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle \sqsubseteq \beta_2\langle\overline{\tau}_2\rangle}$

$$\frac{\mathit{mod}\ \beta_1\langle\overline{T}\rangle\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle \in \Sigma \qquad \Gamma \mid \Sigma \vdash \overline{\tau}\ \mathbf{type} \qquad |\mathrm{typevar}_\Sigma(\beta_1)| = |\overline{\tau}|}{\Gamma \mid \Sigma \vdash \beta_1\langle\overline{\tau}\rangle \sqsubseteq \beta_2\langle\{\overline{\tau}/\overline{T}\}\,\overline{\sigma}\rangle}\ (\textsc{Sub-Brand-Decl})$$

$$\frac{|\mathrm{typevar}_\Sigma\beta| = |\overline{\tau}| \qquad \Gamma \mid \Sigma \vdash \overline{\tau}\ \mathbf{type}}{\Gamma \mid \Sigma \vdash \beta\langle\overline{\tau}\rangle \sqsubseteq \beta\langle\overline{\tau}\rangle}\ (\textsc{Sub-Brand-Refl})$$

$$\frac{\Gamma \mid \Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle \sqsubseteq \beta_2\langle\overline{\tau}_2\rangle \qquad \Gamma \mid \Sigma \vdash \beta_2\langle\overline{\tau}_2\rangle \sqsubseteq \beta_3\langle\overline{\tau}_3\rangle}{\Gamma \mid \Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle \sqsubseteq \beta_3\langle\overline{\tau}_3\rangle}\ (\textsc{Sub-Brand-Trans})$$

Subtype judgement. $\boxed{\Sigma \vdash \tau_1 \leq \tau_2}$

$$\frac{}{\Sigma \vdash \tau \leq \tau}\ \textsc{Sub-Refl} \qquad\qquad \frac{\Sigma \vdash \tau_1 \leq \tau_2 \qquad \Sigma \vdash \tau_2 \leq \tau_3}{\Sigma \vdash \tau_1 \leq \tau_3}\ \textsc{Sub-Trans}$$

$$\frac{\begin{array}{c}\Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle \sqsubseteq \beta_2\langle\overline{\tau}_2\rangle \qquad \Sigma \vdash \overline{\tau}_1 \leq \overline{\tau}_2 \\ \Sigma \vdash M_1 <: M_2 \qquad \Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle(M_1)\ \textbf{type} \qquad \Sigma \vdash \beta_2\langle\overline{\tau}_2\rangle(M_2)\ \textbf{type}\end{array}}{\Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle(M_1) \leq \beta_2\langle\overline{\tau}_2\rangle(M_2)}\ (\textsc{Sub-Name})$$

$$\frac{\Sigma, X \leq Y \vdash \tau_1 \leq \tau_2}{\Sigma \vdash \mu X.\tau_1 \leq \mu Y.\tau_2}\ \textsc{Sub-Rec} \qquad\qquad \frac{\Sigma \vdash \sigma_1 \leq \tau_1 \qquad \Sigma \vdash \tau_2 \leq \sigma_2}{\Sigma \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ (\textsc{Sub-Func})$$

$$\frac{\Sigma \vdash \tau \leq \sigma_1 \qquad \Sigma \vdash \tau \leq \sigma_2}{\Sigma \vdash \tau \leq \sigma_1 \wedge \sigma_2}\ (\textsc{Sub-}\wedge R) \qquad\qquad \frac{}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \tau_1}\ (\textsc{Sub-}\wedge L_1)$$

$$\frac{}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \tau_2}\ (\textsc{Sub-}\wedge L_2) \qquad\qquad \frac{\{\ell_i : \tau_i{}^{\,i\in 1..n}\} \text{ is a permutation of } \{\ell_j : \tau_j{}^{\,j\in 1..n}\}}{\Sigma \vdash \{\ell_i : \tau_i{}^{\,i\in 1..n}\} \leq \{\ell_j : \tau_j{}^{\,j\in 1..n}\}}\ (\textsc{Sub-Rec-Perm})$$

$$\frac{n > m}{\Sigma \vdash \{\ell_i : \tau_i{}^{\,i\in 1..n}\}\ \leq\ \{\ell_j : \tau_j{}^{\,j\in 1..m}\}}\ (\textsc{Sub-Rec-Width})$$

$$\frac{\Sigma \vdash \tau_i \leq \sigma_i\ (i\in 1..n)}{\Sigma \vdash \{\ell_i : \tau_i\}^{\,i\in 1..n} \leq \{\ell_i : \sigma_i\}^{\,i\in 1..n}}\ (\textsc{Sub-Rec-Depth}) \qquad\qquad \frac{T\ \mathsf{type} \mid \Sigma \vdash \tau_1 \leq \tau_2}{\Sigma \vdash \forall T.\,\tau_1 \leq \forall T.\,\tau_2}\ (\textsc{Sub-}\forall)$$

$$\frac{\Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle \sqsubseteq \beta_2\langle\overline{\tau}_2\rangle \qquad \Sigma \vdash M_2 <: M_1 \qquad \Sigma \vdash \sigma_1 \leq \sigma_2}{\Sigma \vdash \beta_1\langle\overline{\tau}_1\rangle(M_1) \Rightarrow \sigma_1 \leq \beta_2\langle\overline{\tau}_2\rangle(M_2) \Rightarrow \sigma_2}\ (\textsc{Sub-Method})$$

Subtyping on method records.

$$\frac{\Sigma \vdash \{\overline{m : \tau}\} \leq \{\overline{n : \sigma}\}}{\Sigma \vdash \overline{m : \tau} <: \overline{n : \sigma}}\ (\textsc{Sub-Method-Rec})$$

B.2.4 Typing rules

$$\boxed{\Sigma \vdash p\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\beta_1 \notin \Sigma \qquad \overline{T}\ \mathsf{type} \mid \Sigma \vdash \tau \leq \mathrm{req}_{\Sigma}\,(\beta_2\langle\overline{\sigma}\rangle) \qquad \Sigma \vdash \beta_1.\overline{m\text{-}decl} : M \\ \Sigma' = \Sigma, \mathit{mod}\ \beta_1\langle\overline{T}\rangle(\tau; M)\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle \qquad \Sigma' \vdash \beta_1\langle\overline{T}\rangle.(\tau; \overline{m\text{-}decl})\ \mathbf{ok} \\ \mathit{mod} = \mathsf{concrete}\ \text{implies}\ \mathit{abstractCover}(M, \beta_2\langle\overline{\sigma}\rangle) \qquad \mathit{override}(M, \beta_2\langle\overline{\sigma}\rangle) \\ \mathsf{abstract\ method}\ m \in M\ \text{implies}\ \mathit{mod} = \mathsf{abstract} \qquad \Sigma' \vdash p\ \mathbf{ok}\end{array}}{\Sigma \vdash \mathit{mod}\ \mathsf{brand}\ \forall\overline{T}.\,\beta_1\langle\overline{T}\rangle(\tau; \overline{m\text{-}decl})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle\ \ \mathsf{in}\ \ p\ \mathbf{ok}}\ (\textsc{Tp-Brand-Intro})$$

$$\frac{\begin{array}{c}\Sigma = \{\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\sigma; M')\ \mathsf{extends}\ \beta_2\langle\overline{\sigma'}\rangle\}, \Sigma_0 \\ m \notin M' \qquad \Sigma' = \{\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\sigma; M', m : \beta_1\langle\overline{T}\rangle(M) \Rightarrow \tau)\ \mathsf{extends}\ \beta_2\langle\overline{\sigma'}\rangle\}, \Sigma_0 \\ \mathit{override}(\beta_1\langle\overline{T}\rangle(M) \Rightarrow \tau, \beta_2\langle\overline{\sigma'}\rangle) \\ \overline{T}\ \mathsf{type}, \mathsf{this} : \beta_1\langle\overline{T}\rangle(M), \mathsf{fields} : \sigma \mid \Sigma' \vdash e : \tau \qquad \Sigma' \vdash p\ \mathbf{ok}\end{array}}{\Sigma \vdash \mathsf{method}\ m\ \forall\overline{T}.\,\beta_1\langle\overline{T}\rangle(M) : \tau = e\ \ \mathsf{in}\ \ p\ \mathbf{ok}}\ (\textsc{Tp-Ext-Method})$$

$$\frac{\cdot \mid \Sigma \vdash e : \tau}{\Sigma \vdash e\ \mathbf{ok}}\ (\textsc{Tp-Expr1}) \qquad\qquad \frac{\cdot \mid \Sigma \vdash e : \tau \qquad \Sigma \vdash p\ \mathbf{ok}}{\Sigma \vdash e; p\ \mathbf{ok}}\ (\textsc{Tp-Expr2})$$

$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash x : \tau}\ (\textsc{Tp-Var}) \qquad \frac{}{\Gamma \mid \Sigma \vdash () : \mathsf{unit}}\ (\textsc{Tp-Unit})$$

$$\frac{\Gamma \mid \Sigma \vdash \tau_1\ \textbf{type} \qquad \Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}\ (\textsc{Tp-Fun})$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid \Sigma \vdash e_2 : \tau_1}{\Gamma \mid \Sigma \vdash e_1\, e_2 : \tau_2}\ (\textsc{Tp-App}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \sigma \qquad \Sigma \vdash \sigma \leq \tau}{\Gamma \mid \Sigma \vdash e : \tau}\ (\textsc{Tp-Subs})$$

$$\frac{\begin{array}{c}\text{modifier}_\Sigma(\beta) = \mathsf{concrete} \qquad \Gamma \mid \Sigma \vdash e : \tau \qquad \Gamma \mid \Sigma \vdash \tau \leq \text{req}_\Sigma\,(\beta\langle\overline{\sigma}\rangle) \\ \mathit{methods}_\Sigma(\beta\langle\overline{\sigma}\rangle) = \overline{\mathit{mod}_m\ m : \tau_m} \qquad \Gamma \mid \Sigma \vdash \beta\langle\overline{\sigma}\rangle(\overline{m : \tau_m})\ \textbf{type}\end{array}}{\Gamma \mid \Sigma \vdash \widehat{\beta}[\overline{\sigma}](e) : \beta\langle\overline{\sigma}\rangle(\overline{m : \tau_m})}\ (\textsc{Tp-New-Obj})$$

$$\frac{\Gamma \mid \Sigma \vdash \overline{e} : \overline{\tau}}{\Gamma \mid \Sigma \vdash (\overline{\ell = e}) : \{\overline{\ell : \tau}\}}\ (\textsc{Tp-New-Record}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \{\ell_i : \tau_i\ ^{i \in 1..n}\}}{\Gamma \mid \Sigma \vdash e.\ell_k : \tau_k}\ (\textsc{Tp-Proj})$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e : \beta\langle\overline{\tau}\rangle(M) \\ m_k : \tau_k \in M \qquad \tau_k = \beta'\langle\overline{\sigma}\rangle(M') \Rightarrow \tau_1 \qquad \beta\langle\overline{\tau}\rangle(M) \leq \beta'\langle\overline{\sigma}\rangle(M')\end{array}}{\Gamma \mid \Sigma \vdash e.m_k : \tau_1}\ (\textsc{Tp-Invoke})$$

$$\frac{\Gamma \mid \Sigma \vdash e : [\mu X.\tau/X]\tau}{\Gamma \mid \Sigma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \mu X.\tau}\ (\textsc{Tp-Fold}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \mu X.\tau}{\Gamma \mid \Sigma \vdash \mathsf{unfold}_{\mu X.\tau}\, e : [\mu X.\tau/X]\tau}\ (\textsc{Tp-Unfold})$$

$$\frac{\Gamma, T\ \mathsf{type} \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \Lambda T.\, e : \forall T.\, \tau}\ (\textsc{Tp-TypeAbs}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \forall T.\, \tau \qquad \Gamma \mid \Sigma \vdash \sigma\ \textbf{type}}{\Gamma \mid \Sigma \vdash e[\sigma] : \{\sigma/T\}\,\tau}\ (\textsc{Tp-TypeApp})$$

B.3 Dynamic Semantics

$\boxed{\mathit{mbody}_\Delta(m, \widehat{\beta}[\overline{\tau}]) = e}$

$$\frac{\widehat{\beta}_1[\overline{T}](m_0 = e_0, \overline{m' = e'})\ \mathsf{extends}\ \widehat{\beta}_2[\overline{\sigma}] \in \Delta}{\mathit{mbody}_\Delta(m_0, \widehat{\beta}_1[\overline{\tau}]) = \{\overline{\tau}/\overline{T}\}\, e_0}$$

$$\frac{\widehat{\beta}_1[\overline{T}](\overline{m = e})\ \mathsf{extends}\ \widehat{\beta}_2[\overline{\tau}] \in \Delta \qquad m_0 \notin \overline{m} \qquad \mathit{mbody}_\Delta(m_0, \widehat{\beta}_2[\overline{\tau}]) = e_0}{\mathit{mbody}_\Delta(m_0, \widehat{\beta}_1[\overline{\sigma}]) = \{\overline{\sigma}/\overline{T}\}\, e_0}$$

$$\boxed{m\text{-}decl \longmapsto m = e}$$

$$\frac{}{\text{abstract method } m(\tau; \overline{m : \sigma_m}) : \tau \ \longmapsto\ \cdot}\ (\text{E-MD}\textsc{ecl}1)$$

$$\frac{}{\text{method } m(\tau; \overline{m : \sigma_m}) : \tau = e \ \longmapsto\ m = e}\ (\text{E-MD}\textsc{ecl}2)$$

$$\boxed{p \mid \Delta \longmapsto p' \mid \Delta'}$$

$$\frac{\overline{m\text{-}decl} \longmapsto \overline{m = e}}{\begin{array}{c} mod \text{ brand } \forall \overline{T}.\, \beta_1 \langle \overline{T} \rangle (\tau; \overline{m\text{-}decl}) \text{ extends } \beta_2 \langle \overline{\sigma} \rangle \text{ in } p \mid \Delta \longmapsto \\ p \mid \Delta, (\beta_1 \langle \overline{T} \rangle (\overline{m = e}) \text{ extends } \beta_2 \langle \overline{\sigma} \rangle) \end{array}}\ (\text{E-B}\textsc{rand}\text{-D}\textsc{ecl})$$

$$\frac{\Delta = \{\beta \langle \overline{T} \rangle (\overline{m = e_m}) \text{ extends } \beta' \langle \overline{\tau} \rangle\}, \Delta_0}{\begin{array}{c} \text{method } m_1\ \forall \overline{T}.\, \beta \langle \overline{T} \rangle (\overline{m : \tau}) : \sigma = e_1 \text{ in } p \mid \Delta \longmapsto \\ p \mid \{\beta \langle \overline{T} \rangle (\overline{m = e_m}, m_1 = e_{m_1}) \text{ extends } \beta' \langle \overline{\tau} \rangle\}, \Delta_0 \end{array}}\ (\text{E-E}\textsc{xt}\text{-D}\textsc{ecl})$$

$$\frac{e \longmapsto_{\Delta} e'}{e \mid \Delta \longmapsto e' \mid \Delta}\ (\text{E-E}\textsc{xpr}1)$$

$$\frac{e \longmapsto_{\Delta} e'}{e; p \mid \Delta \longmapsto e'; p \mid \Delta}\ (\text{E-E}\textsc{xpr}2)$$

$$\frac{}{v; p \mid \Delta \longmapsto p \mid \Delta}\ (\text{E-E}\textsc{xpr}3)$$

$\boxed{e \longmapsto_{\Delta} e'}$

$$\frac{e_1 \longmapsto_{\Delta} e_1'}{e_1\,e_2 \longmapsto_{\Delta} e_1'\,e_2}\ (\textsc{E-App1}) \qquad \frac{e_2 \longmapsto_{\Delta} e_2'}{v_1\,e_2 \longmapsto_{\Delta} v_1\,e_2'}\ (\textsc{E-App2})$$

$$\frac{}{(\lambda x{:}\tau.\,e)\,v \longmapsto_{\Delta} \{v/x\}\,e}\ (\textsc{E-App-Abs})$$

$$\frac{e_k \longmapsto_{\Delta} e_k'}{(\ell_1 = v_1, \ldots, \ell_{k-1} = v_{k-1}, \ell_k = e_k, \ldots\,) \longmapsto_{\Delta} (\ldots, \ell_k = e_k', \ldots\,)}\ (\textsc{E-Record})$$

$$\frac{e \longmapsto_{\Delta} e'}{e.\ell \longmapsto_{\Delta} e'.\ell}\ (\textsc{E-Proj1}) \qquad \frac{}{(\ell_i = v_i{}^{\,i\in 1..n}).\ell_k \longmapsto_{\Delta} v_k}\ (\textsc{E-Proj2})$$

$$\frac{e \longmapsto_{\Delta} e'}{\widehat{\beta}[\overline{\tau}](e) \longmapsto_{\Delta} \widehat{\beta}[\overline{\tau}](e')}\ (\textsc{E-Brand-Cons}) \qquad \frac{e \longmapsto_{\Delta} e'}{e.m \longmapsto_{\Delta} e'.m}\ (\textsc{E-Invoke1})$$

$$\frac{\mathit{mbody}_{\Delta}(m, \widehat{\beta}[\overline{\tau}]) = e}{\widehat{\beta}[\overline{\tau}](v).m \longmapsto_{\Delta} \{\widehat{\beta}[\overline{\tau}](v)/\mathsf{this}, v/\mathsf{fields}\}\,e}\ (\textsc{E-Invoke2}) \qquad \frac{e \longmapsto_{\Delta} e'}{\mathsf{fold}_{\tau}\,e \longmapsto_{\Delta} \mathsf{fold}_{\tau}\,e'}\ (\textsc{E-Fold})$$

$$\frac{e \longmapsto_{\Delta} e'}{\mathsf{unfold}_{\tau}\,e \longmapsto_{\Delta} \mathsf{unfold}_{\tau}\,e'}\ (\textsc{E-Unfold}) \qquad \frac{}{\mathsf{unfold}_{\tau}\,(\mathsf{fold}_{\tau}\,v) \longmapsto_{\Delta} v}\ (\textsc{E-Unfold-Fold})$$

$$\frac{e \longmapsto_{\Delta} e'}{e[\tau] \longmapsto_{\Delta} e'[\tau]}\ \textsc{E-TApp} \qquad \frac{}{\Lambda T.e[\tau] \longmapsto_{\Delta} \{\tau/T\}\,e}\ \textsc{E-TApp-TAbs}$$

C Unity$_\alpha$ Type safety

C.1 Definitions

Definition C.1 (Well-formed context)**.**
The context Σ is *well-formed*, iff the following conditions hold:

1. there is exactly one entry for each brand β.
2. if $\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\tau; M)$ $\mathsf{extends}$ $\beta_2\langle\overline{\sigma}\rangle \in \Sigma$, then
   (a) $\overline{T}$ type $\mid \Sigma \vdash \beta_2\langle\overline{T}\rangle(M)$ **type**
   (b) $\overline{T}$ type $\mid \Sigma \vdash \tau \leq \mathrm{req}_\Sigma\, \beta_2\langle\overline{\sigma}\rangle$
   (c) if mod = $\mathsf{concrete}$, then $\mathit{methods}_\Sigma(\beta_1\langle\overline{\tau'}\rangle) = \mathsf{concrete}\ \overline{n : \tau}$.

Definition C.2 (Models relation on contexts)**.**
The context Σ *models* Δ, iff $\Sigma \vdash \Delta$. The definition of $\Sigma \vdash \Delta$ is given by the following inference rules.

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\begin{array}{c}\Sigma \vdash \Delta \\ \Sigma' = \Sigma, \mathit{mod}\ \beta_1\langle\overline{T}\rangle(\tau; \{\mathsf{concrete}\ m_i : \beta_1\langle\overline{T}\rangle(M_i) \Rightarrow \tau_i'\ ^{i\in 1..n}\}, \overline{\mathsf{abstract}\ n : \sigma_m})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle \\ \overline{T}\ \mathsf{type}, \mathsf{this} : \beta_1\langle\overline{T}\rangle(M_i), \mathsf{fields} : \tau \mid \Sigma' \vdash e_i : \tau_i\ (i\in 1..n)\end{array}}{\Sigma' \vdash \Delta, \widehat{\beta}_1[\overline{T}](m_i = e_i\ ^{i\in 1..n})\ \mathsf{extends}\ \widehat{\beta}_2[\overline{\sigma}]}$$

C.2 Inversion and Canonical Forms Lemmas

Lemma C.1 (Inversion of subtyping)**.**

1. If $\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$, then $\Sigma \vdash \sigma_1 \leq \tau_1$ and $\Sigma \vdash \tau_2 \leq \sigma_2$.
2. If $\Sigma \vdash \beta_1\langle\overline{\tau}\rangle(M_1) \leq \beta_2\langle\overline{\sigma}\rangle(M_2)$, then $\Sigma \vdash \beta_1\langle\overline{\tau}\rangle \sqsubseteq \beta_2\langle\overline{\sigma}\rangle$ and $\Sigma \vdash M_1 <: M_2$ and $\Sigma \vdash \beta_1\langle\overline{\tau}\rangle(M_1)$ **type** and $\Sigma \vdash \beta_2\langle\overline{\sigma}\rangle(M_2)$ **type**.
3. If $\Sigma \vdash (\ell_i : \tau_i\ ^{i\in 1..n}) \leq (k_j : \sigma_j\ ^{j\in 1..m})$, then $\{k_j\ ^{j\in 1..m}\} \subseteq \{\ell_i\ ^{i\in 1..n}\}$ ($\overline{\ell}$ includes at least the labels in $\overline{k}$) and $\Sigma \vdash \tau_i \leq \sigma_j$ for each common label $\ell_i = k_j$.
4. If $\Sigma \vdash \beta_1\langle\overline{\sigma}_1\rangle(M_1) \Rightarrow \tau_1 \leq \beta_2\langle\overline{\sigma}_2\rangle(M_2) \Rightarrow \tau_2$ then $\Sigma \vdash \beta_1\langle\overline{\sigma}_1\rangle \sqsubseteq \beta_2\langle\overline{\sigma}_2\rangle$ and $\Sigma \vdash M_2 <: M_1$ and $\Sigma \vdash \tau_1 \leq \tau_2$.
5. If $\Sigma \vdash \forall T.\, \tau_1 \leq \tau$ and $\Sigma \vdash \tau \leq \forall T.\, \tau_2$, then T **type** $\mid \Sigma \vdash \tau_1 \leq \tau_2$.

Proof. Straightforward induction on the subtyping derivation. □

Lemma C.2 (Inversion of the typing judgement)**.**

1. If $\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \sigma$ and $\Sigma \vdash \sigma \leq \sigma_1 \to \sigma_2$ then $\Sigma \vdash \sigma_1 \leq \tau_1$ and $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \sigma_2$.
2. If $\Gamma \mid \Sigma \vdash \widehat{\theta}[\overline{\sigma}](e) : \tau$ and $\Sigma \vdash \tau \leq \beta\langle\overline{\sigma}'\rangle(M)$ then for some τ' we have:
   (a) $\Gamma \mid \Sigma \vdash e : \tau'$

(b) $\tau' \leq \mathrm{req}_\Sigma\, \theta\langle\overline{\sigma}\rangle$

(c) $\Gamma \mid \Sigma \vdash \widehat{\theta}[\overline{\sigma}](e) : \theta\langle\overline{\sigma}\rangle(\mathit{methods}_\Sigma\theta\langle\overline{\sigma}\rangle)$

(d) $\Sigma \vdash \mathit{methods}_\Sigma\theta\langle\overline{\sigma}\rangle <: M$

(e) $\Sigma \vdash \theta\langle\overline{\sigma}\rangle \sqsubseteq \beta\langle\overline{\sigma}'\rangle$

3. If $\Gamma \mid \Sigma \vdash \Lambda T.\, e : \tau$ and $\Gamma \mid \Sigma \vdash \tau \leq \forall T.\, \sigma$, then Γ, T type $\mid \Sigma \vdash e : \sigma$.

Proof. By induction on the typing derivation, with case analysis of the final rule used. Vacuous cases have been omitted.

1. $\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \tau$

   case TP-FUN. $\tau = \tau_1 \to \tau_2$
   By SUB-TRANS, $\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$; by subtype inversion (Lemma C.1), $\sigma_1 \leq \tau_1$ and $\tau_2 \leq \sigma_2$. By the rule's premise, $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2$, and the result follows from TP-SUBS.

   case TP-SUBS. We have $\lambda x{:}\tau_1.\, e : \tau$ and $\tau \leq \sigma$. By SUB-TRANS, $\tau \leq \sigma_1 \to \sigma_2$ and the result follows from the induction hypothesis.

2. $\Gamma \mid \Sigma \vdash \widehat{\theta}[\overline{\sigma}](e) : \tau$

   case TP-NEW-OBJ. $\sigma = \theta\langle\overline{\sigma}\rangle(\overline{m : \tau})$.
   Conclusions (a), (b) and (c) follow from the premises of TP-NEW-OBJ. By subtype inversion (Lemma C.1), $\mathit{methods}_\Sigma\theta\langle\overline{\sigma}\rangle <: M$ and $\theta\langle\overline{\sigma}\rangle \sqsubseteq \beta\langle\overline{\sigma}'\rangle$, which proves conclusions (d) and (e).

   case TP-SUBS. Result follows from SUB-TRANS and the induction hypothesis.

3. $\Gamma \mid \Sigma \vdash \Lambda T.\, e : \tau$. Straightforward.

□

Lemma C.3 (Canonical forms)**.** Suppose $\cdot \mid \Sigma \vdash v : \sigma$ and $\Sigma \vdash \sigma \leq \tau$.

1. If $\tau = \mathsf{unit}$ then $v = \mathtt{()}$.

2. If $\tau = \tau_1 \to \tau_2$ then v is of the form $\lambda x{:}\tau_{11}.\, e$.

3. If $\tau = \beta\langle\sigma\rangle(\overline{m : \tau})$ then v is of the form $\widehat{\beta}'[\overline{\sigma}'](v)$.

4. If $\tau = \{\overline{\ell : \tau}\}$ then v is of the form $(\overline{k = v})$.

5. If $\tau = \mu X.\tau$ then v is of the form $\mathsf{fold}_\sigma\, v$.

6. If $\tau = \forall T.\, \tau$ then v is of the form $\Lambda T.\, e$.

Proof. Straightforward induction on typing derivations. □

C.3 Type substitution Lemmas

Lemma C.4 (Type substitution preserves well-formed types and subtyping)**.**

1. If $\Gamma, T\ \mathsf{type}, \Gamma' \mid \Sigma \vdash \tau$ **type** and $\Gamma, \Gamma' \mid \Sigma \vdash \sigma$ **type**, then $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\tau$ **type**.
2. If $\Gamma, T\ \mathsf{type}, \Gamma' \mid \Sigma \vdash \tau_1 \leq \tau_2$ and $\Gamma, \Gamma' \mid \Sigma \vdash \sigma$ **type**, then $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\tau_1 \leq \{\sigma/T\}\,\tau_2$.

Proof. By mutual induction on the derivations of the judgements τ **type** and $\tau_1 \leq \tau_2$.

1. **case** Unit-Type. Immediate.

   case TypeVar-Type. $T'\ \mathsf{type} \in \Gamma, T\ \mathsf{type}, \Gamma'$.
   There are three possible cases of the context that contains T'.

   subcase $T'\ \mathsf{type} \in \Gamma$. Because the context is well-formed, $\{\sigma/T\}\,T' = T'$. From this it follows that $\Gamma, \Gamma' \vdash T'$ **type**.

   subcase $T' = T$. Since $\{\sigma/T\}\,T = \sigma$, the result follows.

   subcase $T'\ \mathsf{type} \in \Gamma'$. We have $\{\sigma/T\}\,T' \in \{\sigma/T\}\,\Gamma'$. The result then follows from T-Type.

   case Fun-Type. We have $\Gamma, T\ \mathsf{type}, \Gamma' \vdash \tau_1$ **type** and $\Gamma, T\ \mathsf{type}, \Gamma' \vdash \tau_2$ **type**.
   By the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau_1$ **type** and $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau_2$ **type**. By Fun-Type, $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau_1 \rightarrow \{\sigma/T\}\,\tau_2$ **type**, which is equivalent to $\{\sigma/T\}\,(\tau_1 \rightarrow \tau_2)$.

   case $\wedge$-Type. Similar to above.

   case $\forall$-Type. By the induction hypothesis, and the fact that $\{\sigma/T\}\,T' = T'$, we have $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau$ **type**. The result then follows from $\forall$-Type.

   case Brand-Type. We are to show that $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\beta\langle\overline{\tau}\rangle(\overline{m : \sigma_m})$ **type**. This last expression is equivalent to $\beta\langle\{\sigma/T\}\,\overline{\tau}\rangle(\overline{m : \{\sigma/T\}\,\sigma_m})$.
   We have $\Gamma, T\ \mathsf{type}, \Gamma' \mid \Sigma \vdash \overline{\tau}$ **type**. By the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\overline{\tau}$ **type**. Similiarly, $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\overline{\sigma}_m$ **type**, which is the required result.

   case Record-Type. Result follows from the induction hypothesis.

   case Mu-Type. Result follows from the induction hypothesis.

   case Method-Type. We are to show that $\Gamma, \{\sigma/T\}, \Gamma' \mid \Sigma \vdash \{\sigma/T\}\,\beta\langle\overline{\tau}\rangle(m_i : \tau_{m_i}{}^{i\in 1..n}) \Rightarrow \{\sigma/T\}\,\tau_2$ **type**. This follows by applying the induction hypothesis to the premises $\beta\langle\overline{\tau}\rangle(m_i : \tau_i{}^{i\in 1..n})$ **type** and τ_2 **type**.

2. **case** Sub-Refl. Immediate.

   case Sub-Trans. Result follows from the induction hypothesis and Sub-Trans.

   case Sub-Name. By the induction hypothesis of (1), $\beta_1\langle\{\sigma/T\}\,\tau_1\rangle(\{\sigma/T\}\,M_1)$ **type** and $\beta_2\langle\{\sigma/T\}\,\tau_2\rangle(\{\sigma/T\}\,M_2)$ **type**. The result then follows from Sub-Name.

   case Sub-Func. Result follows from the induction hypothesis, Sub-Func, and the equality $\{\sigma/T\}\,(\tau_1 \rightarrow \tau_2) = \{\sigma/T\}\,\tau_1 \rightarrow \{\sigma/T\}\,\tau_2$.

   case Sub-$\forall$. Result follows from the induction hypothesis and Sub-$\forall$.

case Sub-$\wedge$R. By the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau \leq \{\sigma/T\}\,\sigma_1$ and $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,\tau \leq \{\sigma/T\}\,\sigma_2$. The result then follows from Sub-$\wedge$R and the equality $\{\sigma/T\}\,(\tau_1 \wedge \tau_2) = \{\sigma/T\}\,\tau_1 \wedge \{\sigma/T\}\,\tau_2$

case Sub-$\wedge L_1$, Sub-$\wedge L_2$. Similar to above.

case Sub-Brand-$\wedge L$. Similar to above.

□

Lemma C.5 (Type substitution preserves types)**.** If $\Gamma, T\ \text{type}, \Gamma' \mid \Sigma \vdash e : \tau$ and $\Gamma, \Gamma' \mid \Sigma \vdash \sigma$ **type**, then $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,e : \{\sigma/T\}\,\tau$.

Proof. By induction on $e : \tau$.

case Tp-Var. There are two possible subcases of the context that $x : \tau$ appears in.

subcase $x : \tau \in \Gamma$. In this case, because the context is well-formed and T does not appear in Γ', then $\{\sigma/T\}\,\tau = \tau$, which gives the required result.

subcase $x : \tau \in \Gamma'$. From this it follows that $x : \{\sigma/T\}\,\tau \in \{\sigma/T\}\,\Gamma'$. The result then follows from Tp-Var.

case Tp-Fun. It suffices to show that $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \lambda x{:}\{\sigma/T\}\,\tau_1.\,\{\sigma/T\}\,e_1 : \{\sigma/T\}\,\tau_1 \rightarrow \{\sigma/T\}\,\tau_2$, since $\{\sigma/T\}\,(\tau_1 \rightarrow \tau_2) = \{\sigma/T\}\,\tau_1 \rightarrow \{\sigma/T\}\,\tau_2$. From the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma', x : \{\sigma/T\}\,\tau_1 \mid \Sigma \vdash \{\sigma/T\}\,e_1 : \{\sigma/T\}\,\tau_2$. Since substitution preserves the well-typed property (Lemma C.4), the result follows from Tp-Fun.

case Tp-App. The result follows from the induction hypothesis, the equality $\{\sigma/T\}\,(\tau_1 \rightarrow \tau_2) = \{\sigma/T\}\,\tau_1 \rightarrow \{\sigma/T\}\,\tau_2$, and Tp-App.

case Tp-Subs. By the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma' \vdash \{\sigma/T\}\,e : \{\sigma/T\}\,\sigma'$. Since substitution preserves subtyping, $\{\sigma/T\}\,\sigma' \leq \{\sigma/T\}\,\tau$. The result then follows from Tp-Subs.

case Tp-New-Obj. By the induction hypothesis, $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,e : \{\sigma/T\}\,\tau$. Since substitution preserves the well-typed property (Lemma C.4), $\Gamma, \{\sigma/T\}\,\Gamma' \mid \Sigma \vdash \{\sigma/T\}\,(\beta\langle\overline{\sigma'}\rangle(\overline{m : \tau})$ **type**. We also have the equalities $\{\sigma/T\}\,(\widehat{\beta}[\overline{\sigma'}](e) = \widehat{\beta}[\{\sigma/T\}\,\overline{\sigma'}](\{\sigma/T\}\,e)$, $\{\sigma/T\}\,\mathrm{req}_\Sigma\,(\beta\langle\overline{\sigma}\rangle) = \mathrm{req}_\Sigma\,(\beta\langle\{\sigma/T\}\,\overline{\sigma}\rangle)$ and $\mathit{methods}_\Sigma(\beta\langle\{\sigma/T\}\,\overline{\sigma}\rangle) = \overline{m : \{\sigma/T\}\,\tau_m}$. From this, the result follows from Tp-New-Obj.

case Tp-New-Record. The result follows from the induction hypothesis, the equalities $\{\sigma/T\}\,(\overline{\ell = e}) = (\overline{\ell = \{\sigma/T\}\,e})$ and $\{\{\sigma/T\}\,\overline{\ell : \tau}\} = \{\overline{\ell : \{\sigma/T\}\,\tau}\}$ and Tp-New-Record.

case Tp-Proj. The result follows from the induction hypothesis, the equality $\{\{\sigma/T\}\,\overline{\ell : \tau}\} = \{\overline{\ell : \{\sigma/T\}\,\tau}\}$ and Tp-Proj.

case Tp-Invoke. The result follows from the induction hypothesis, the equality $\{\sigma/T\}\,(\beta'\langle\overline{\sigma}\rangle(M') \Rightarrow \tau_1) = \{\sigma/T\}\,(\beta'\langle\overline{\sigma}\rangle(M')) \Rightarrow \{\sigma/T\}\,\tau_1$ and Tp-Invoke.

case Tp-Fold, Tp-Unfold. Straightforward.

case Tp-TypeAbs. The result follows from the induction hypothesis, the equality $\{\sigma/T\}\,T' = T'$, and Tp-TypeAbs.

case Tp-TypeApp. The result follows from the induction hypothesis, the equality $\{\sigma/T\}\,(e[\sigma']) = \{\sigma/T\}\,e[\{\sigma/T\}\,\sigma']$, and Tp-TypeApp.

□

C.4 Progress Lemmas and Theorem

Lemma C.6. If $\Sigma \vdash \Delta$ then $\Sigma \vdash \beta_1\langle\overline{\tau}\rangle \sqsubseteq \beta_2\langle\overline{\sigma}\rangle$ iff $\Delta \vdash \widehat{\beta}_1[\overline{\tau}] \sqsubseteq \widehat{\beta}_2[\overline{\sigma}]$.

Proof. Straightforward induction on $\Sigma \vdash \Delta$. □

Lemma C.7. If $\Sigma \vdash \beta_1[\overline{\tau}] \sqsubseteq \beta_2[\overline{\sigma}]$ and $mbody_{\Delta}(m, \beta_2[\overline{\sigma}]) = e$ then $mbody_{\Delta}(m, \beta_1[\overline{\tau}]) = e'$, for some e'.

Proof. By induction on $\beta_1[\overline{\tau}] \sqsubseteq \beta_2[\overline{\sigma}]$.

case Sub-Brand-Decl. If either the first or second case of *mbody* applies, we have $mbody_{\Delta}(m, \widehat{\beta}_2[\overline{\sigma}]) = e$. By the second rule of $mbody_{\Delta}$, $mbody_{\Delta}(m, \widehat{\beta}_1[\overline{\tau}]) = \{\overline{\tau}/\overline{T}\}\,e$.

case Sub-Brand-Refl. Immediate.

case Sub-Brand-Trans. We have $\beta_1[\overline{\tau}] \sqsubseteq \beta_1'[\overline{\tau}']$ and $\beta_1'[\overline{\tau}'] \sqsubseteq \beta_2[\overline{\sigma}]$. Applying the induction hypothesis to $\beta_1'[\overline{\tau}'] \sqsubseteq \beta_2[\overline{\sigma}]$ gives $mbody_{\Delta}(m, \beta_1'[\overline{\tau}']) = e'$. Applying the induction hypothesis to $\beta_1[\overline{\tau}] \sqsubseteq \beta_1'[\overline{\tau}']$ gives the required result.

□

Lemma C.8. If $\Gamma \mid \Sigma \vdash \widehat{\beta}[\sigma](v) : \tau$ and $\Sigma \vdash \tau \leq \beta'[\sigma'](M)$, where $\Sigma \vdash \Delta$ and $m_k \in M$, then $mbody_{\Delta}(m_k, \widehat{\beta}[\sigma])$ is defined.

Proof. By induction on $\widehat{\beta}[\sigma](v) : \tau$.

case Tp-Subs. Immediate from the induction hypothesis.

case Tp-New-Obj. We have $\beta = \beta'$. From the definition of a well-formed context Σ, $\overline{mod_m} =$ concrete. From the definition of $methods_{\Sigma}$, either m_k is defined in β or some proper superbrand θ (i.e., some $\theta \neq \beta$ where $\beta \sqsubseteq \theta$). If it is defined in β, then $mbody_{\Delta}$ is defined, by the definition of $\Sigma \vdash \Delta$. Otherwise, from the definition of $\Sigma \vdash \Delta$, we have $\widehat{\theta}[\overline{T}](m_k = e_k; \overline{m' = e'})$ extends $\widehat{\theta}'[\tau'] \in \Delta$. Suppose we have $\widehat{\theta}_0[\overline{T}_0](...)$ extends $\widehat{\theta}[\overline{\tau}]$. By the definition of *mbody*, we have $mbody_{\Delta}(m_k, \widehat{\theta}[\overline{\tau}]) = \{\overline{\tau}/\overline{T}\}\,e_k$. Since $\widehat{\theta}[\tau] \sqsubseteq \widehat{\beta}[\overline{\sigma}]$, by Lemma C.7, $mbody_{\Delta}(m_k, \widehat{\beta}[\overline{\sigma}])$ is defined, which is the required result.

□

Lemma C.9 (Progress [expressions]). If $\cdot \mid \Sigma \vdash e : \tau$ then either e is a value, or for some Δ such that $\Sigma \vdash \Delta$, there is an e' with $e \longmapsto_{\Delta} e'$.

Proof. By induction on $e : \tau$, with case analysis of final rule used.

case Tp-Unit, Tp-Fun. Immediate.

case TP-APP. Straightforward.

case TP-SUBS. Result follows from induction hypothesis.

case TP-NEW-OBJ. $e = \widehat{\beta}[\overline{\tau}](e_1)$
By the induction hypothesis, e_1 is a value or it steps to some e_1'. If it takes a step, then E-BRAND-CONS applies. If it is a value, then then e is also a value.

case TP-NEW-RECORD $e = (\overline{\ell = e})$
By the induction hypothesis, each e_i is a value or it steps to some e_i'. If any e_i steps, then the rule E-RECORD applies. Otherwise, the entire expression is a value.

case TP-PROJ. $e = e_1.\ell_k \quad e_1 : \{\overline{\ell : \tau}\}$
By the induction hypothesis, either e_1 is a value or it steps to some e'. If it is a value, then by canonical forms it has the form $(\overline{k = v})$ and E-PROJ2 applies. If it steps to e', then E-PROJ1 applies.

case TP-INVOKE. $e = e_1.m_k \quad e_1 : \beta\langle\overline{\sigma}\rangle(\overline{m : \tau})$
By the induction hypothesis, either e_1 is a value or it steps to some e_1'. If e_1 evaluates to e_1', E-INVOKE1 applies. Otherwise, by canonical forms, e_1 has the form $\widehat{\beta}'[\overline{\sigma}'](v)$. By Lemma C.8, $mbody_\Delta(m, \widehat{\beta}'[\overline{\sigma}'])$ is defined; the rule E-INVOKE2 then applies.

case TP-FOLD. $e = \mathsf{fold}_\tau\, e_1$
By the induction hypothesis, either e_1 is a value or it takes a step. If it takes a step, then the rule E-FOLD applies. Otherwise, e is a value.

case TP-UNFOLD $e = \mathsf{unfold}_{\mu X.\tau}\, e_1$
By the induction hypothesis, either e_1 is a value or it takes a step. If it takes a step, then the rule E-UNFOLD applies. Otherwise, it is a value v of type $\mu X.\tau$. By canonical forms, v has form $\mathsf{fold}_{\mu X.\tau}\, v_1$, so the rule E-UNFOLD-FOLD applies.

case TP-TYPEABS. Immediate.

case TP-TYPEAPP. $e = e_1[\sigma] \quad e : \forall T.\, \tau.$
By the induction hypothesis, either e_1 is a value or $e_1 \longmapsto e_1'$, for some e_1'. If e_1 is a value, then E-TAPPTABS applies, since by canonical forms, e_1 has the form $\Lambda T.\, e'$. Otherwise, the rule E-TAPP applies.

□

Theorem C.1 (Progress [programs]). If $\cdot \mid \Sigma \vdash p$ **ok**, for some Σ, then one of the following cases holds:

1. p is a value
2. for Δ such that $\Sigma \vdash \Delta$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

Proof. By induction on p **ok**.

case TP-BRAND-INTRO. The rule E-BRAND-DECL applies.

case Tp-Ext-Method. The rule E-Ext-Decl applies. Δ has the appropriate form because $\Sigma \vdash \Delta$ and $\Sigma = \{\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\sigma; M)\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle\}, \Sigma_0$.

case Tp-Expr1. The result follows from the progress lemma for expressions (Lemma C.9).

case Tp-Expr2. $p = (e; p_2)$
By Lemma C.9, either $e \longmapsto_\Delta e'$ or e is a value. In the first case, E-Expr2 applies; in the second, E-Expr3 applies.

□

C.5 Preservation Lemmas and Theorem

Lemma C.10 (Substitution)**.**
If $\Gamma, x : \sigma \mid \Sigma \vdash e_1 : \tau$ and $\Gamma \mid \Sigma \vdash e_2 : \sigma$ then $\Gamma \mid \Sigma \vdash \{e_2/x\}\, e_1 : \tau$.

Proof. Straightforward induction on typing derivations. □

Lemma C.11. If $\Gamma, x : \tau, \Gamma' \mid \Sigma \vdash e : \sigma$ and $\tau' \leq \tau$, then $\Gamma, x : \tau', \Gamma' \mid \Sigma \vdash e : \sigma$.

Proof. Straightforward induction on typing derivations. □

Lemma C.12.
If $\Gamma \mid \Sigma \vdash \widehat{\theta}[\overline{\tau}](v) : \sigma$ and $\sigma \leq \beta\langle\overline{\tau}'\rangle(m_0 : \beta'\langle\overline{\tau}''\rangle(M_0) \Rightarrow \tau, M)$ and $\Sigma \vdash \Delta$ and $\mathit{mbody}_\Delta(m_0, \widehat{\theta}[\overline{\tau}]) = e_0$, then $\mathsf{this} : \beta'\langle\overline{\tau}''\rangle(M_0), \mathsf{fields} : \mathsf{req}_\Sigma\, \theta\langle\overline{\tau}\rangle \mid \Sigma \vdash e_0 : \tau$.

Proof. By induction on $\widehat{\theta}(v) : \sigma$.

case Tp-Subs. Result follows from the induction hypothesis and Sub-Trans.

case Tp-New-Obj. Let $\tau_0 = \beta'\langle\overline{\tau}''\rangle(M_0) \Rightarrow \tau$. We have $\widehat{\beta}[\overline{\tau}](v) : \beta\langle\overline{\tau}\rangle(m_0 : \tau_0, M)$. Since $\mathrm{modifier}(\beta) = \mathsf{concrete}$, $\mathit{methods}_\Sigma(\beta\langle\overline{\tau}\rangle) = \mathsf{concrete}\ \overline{m : \tau}$. There are two possible rules that apply for mbody_Δ. In the first case, m_0 is defined in β. Therefore, $\beta'' = \beta$ and $\overline{\tau}'' = \overline{\tau}$. From the definition of $\Sigma \vdash \Delta$, we can conclude that $\mathsf{this} : \beta\langle\overline{\tau}\rangle(M_0), \mathsf{fields} : \mathsf{req}_\Sigma\, \beta\langle\overline{\tau}\rangle \mid \Sigma \vdash e_0 : \tau$.

Otherwise, from the definition of $\mathit{methods}_\Sigma$, there exists some β_2 where m_0 with type τ_0' is defined in β_2 and $\beta\langle\overline{\tau}\rangle \sqsubseteq \beta_2\langle\overline{\tau}_2\rangle$. From the definition of $\Sigma \vdash \Delta$, we know that $\mathsf{this} : \beta_2(M_0), \mathsf{fields} : \mathsf{req}_\Sigma\, \beta_2 \mid \Sigma \vdash e_0 : \tau$. The result then follows from Lemma C.11.

□

Lemma C.13 (Preservation [expressions])**.**
If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Sigma \vdash \Delta$ and $e \longmapsto_\Delta e'$, then $\Gamma \mid \Sigma \vdash e' : \tau$.

Proof. By induction on $e : \tau$.

case Tp-Var, Tp-Unit, Tp-Fun. Vacuous; e does not evaluate.

case Tp-App. Straightforward.

case Tp-Subs. $e : \sigma \quad \sigma \leq \tau$
By the induction hypothesis, $e' : \sigma$ and the result follows from Tp-Subs.

case TP-NEW-OBJ. $e = \widehat{\beta}[\overline{\sigma}](e_1)\ \ e_1 : \tau'$
The only evaluation rule that applies is E-BRAND-CONS. By the induction hypothesis, $e_1' : \tau'$. The result then follows from TP-NEW-OBJ.

case TP-NEW-RECORD. The only evaluation rule that applies is E-RECORD. We have $e_k \longmapsto_\Delta e_k'$. By the induction hypothesis, $e_k : \tau_k$. The result then follows from TP-NEW-RECORD.

case TP-PROJ. $\ e : \{k_i : \tau_i\ ^{i \in 1..n}\}$
There are two possible evaluation rules that apply:

case E-PROJ1. Result follows from the induction hypothesis and TP-PROJ.

case E-PROJ2. $(\ell_j = v_j\ ^{j \in 1..m}).\ell_k \longmapsto_\Delta v_k$
By typing inversion, we have $\{\ell_j : \tau_j\ ^{j \in 1..m}\} \leq \{k_i : \tau_i\ ^{i \in 1..n}\}$ and $v_k : \tau_k$, which is the required result.

case TP-INVOKE.

$$\frac{\Gamma \mid \Sigma \vdash e : \beta\langle\overline{\tau}\rangle(M) \qquad m_k : \tau_k \in M \qquad \tau_k = \beta'\langle\overline{\sigma}\rangle(M') \Rightarrow \tau_1 \qquad \beta\langle\overline{\tau}\rangle(M) \leq \beta'\langle\overline{\sigma}\rangle(M')}{\Gamma \mid \Sigma \vdash e.m_k : \tau_1}$$

There are two possible evaluation rules that apply.

case E-INVOKE1. Result follows from the induction hypothesis and TP-INVOKE.

case E-INVOKE2.

$$\frac{mbody_\Delta(m, \widehat{\theta}[\overline{\sigma}']) = e_0}{\widehat{\theta}[\overline{\sigma}'](v).m \longmapsto_\Delta \{\widehat{\theta}[\overline{\sigma}'](v)/\mathsf{this}, v/\mathsf{fields}\}\, e_0}$$

We have $\widehat{\theta}[\overline{\sigma}'](v) : \beta\langle\overline{\tau}\rangle(M)$, where $m : t_k \in M$
By Lemma C.12, $\mathsf{this} : \beta'\langle\overline{\sigma}\rangle(M'), \mathsf{fields} : \mathrm{req}_\Sigma\, \theta\langle\overline{\sigma}'\rangle \mid \Sigma \vdash e_0 : \tau_1$. By typing inversion (Lemma C.2), and TP-SUBS, $v : \mathrm{req}_\Sigma\, \theta\langle\overline{\sigma}'\rangle$. By TP-SUBS, $\widehat{\theta}[\overline{\sigma}'](v) : \beta'\langle\overline{\sigma}\rangle(M')$. The result follows from the substitution lemma (Lemma C.10).

case TP-FOLD. The only evaluation rule that applies is E-FOLD. The result follows from the induction hypothesis and TP-FOLD.

case TP-UNFOLD. There are two possible evaluation rules that apply.

case E-UNFOLD. The result follows from the induction hypothesis and TP-UNFOLD.

case E-UNFOLD-FOLD. We have $e = \mathsf{unfold}_{\mu X.\tau}\,(\mathsf{fold}_{\mu X.\tau'}\, v) : \tau, \mathsf{fold}_{\mu X.\tau'} : \mu X.\tau$. By typing inversion, $\tau = \tau'$ and $v : \tau$. But this is just what the expression evaluates to, so this is the required result.

case TP-TYPEABS. Vacuous, e is a value.

case TP-TYPEAPP. There are two possible evaluation rules that apply:

subcase E-TAPP. $e = e'[\sigma] \quad e' : \forall T.\,\tau$.
By the induction hypothesis, if $e' \longmapsto_\Delta e''$ then $e'' : \forall T.\,\tau$. The result then follows from TP-TYPEAPP.

subcase E-TAPP-TABS. $(\Lambda T.\, e_1)[\sigma] \longmapsto_{\Delta} \{\sigma/T\}\, e_1 \quad \Gamma \mid \Sigma \vdash \Lambda T.\, e_1 : \forall T.\, \tau$

By typing inversion, $\Gamma, T \text{ type} \mid \Sigma \vdash e_1 : \tau$. The result follows from the fact that type substitution preserves types (Lemma C.5).

□

Theorem C.2 (Preservation [programs]).
If $\Gamma \mid \Sigma \vdash p$ **ok** and $\Sigma \vdash \Delta$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that $\Sigma' \vdash \Delta'$ where $\Gamma \mid \Sigma' \vdash p'$ **ok**.

Proof. By induction on p **ok**.

case TP-BRAND-INTRO.

$$\frac{\begin{array}{c} \beta_1 \notin \Sigma \qquad \overline{T}\ \mathsf{type} \mid \Sigma \vdash \tau \leq \mathrm{req}_{\Sigma}\,(\beta_2\langle\overline{\sigma}\rangle) \\ \Sigma \vdash \beta_1.\overline{\mathit{m\text{-}decl}} : M \qquad \Sigma' = \Sigma, \mathit{mod}\ \beta_1\langle\overline{T}\rangle(\tau; M)\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle \\ \Sigma' \vdash \beta_1\langle\overline{T}\rangle.(\tau; \overline{\mathit{m\text{-}decl}})\ \mathbf{ok} \qquad \mathit{mod} = \mathsf{concrete}\ \text{implies}\ \mathit{abstractCover}(M, \beta_2\langle\overline{\sigma}\rangle) \\ \mathit{override}(M, \beta_2\langle\overline{\sigma}\rangle) \qquad \mathsf{abstract\ method}\ m \in M\ \text{implies}\ \mathit{mod} = \mathsf{abstract} \qquad \Sigma' \vdash p\ \mathbf{ok} \end{array}}{\Sigma \vdash \mathit{mod}\ \mathsf{brand}\ \forall\overline{T}.\, \beta_1\langle\overline{T}\rangle(\tau; \overline{\mathit{m\text{-}decl}})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle\ \mathsf{in}\ p\ \mathbf{ok}}$$

The rule E-BRAND-DECL applies.

$$\frac{\overline{\mathit{m\text{-}decl}} \longmapsto \overline{m = e}}{\begin{array}{c} \mathit{mod}\ \mathsf{brand}\ \forall\overline{T}.\, \beta_1\langle\overline{T}\rangle(\tau; \overline{\mathit{m\text{-}decl}})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle\ \mathsf{in}\ p \mid \Delta \longmapsto \\ p \mid \Delta, (\beta_1\langle\overline{T}\rangle(\overline{m = e})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle) \end{array}}$$

Take $\Delta_2 = \Delta, (\beta_1\langle\overline{T}\rangle(\overline{m = e})\ \mathsf{extends}\ \beta_2\langle\overline{\sigma}\rangle)$. It remains to show that $\Sigma' \vdash \Delta_2$. This result follows from the definition of $\Sigma' \vdash \beta_1\langle\overline{T}\rangle.(\tau; \mathit{m\text{-}decl})$ **ok** and the definition of $\Sigma \vdash \Delta$.

case TP-EXT-METHOD.

$$\frac{\begin{array}{c} \Sigma = \{\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\sigma; M')\ \mathsf{extends}\ \beta_2\langle\overline{\sigma'}\rangle\}, \Sigma_0 \\ m \notin M' \qquad \Sigma' = \{\mathit{mod}\ \beta_1\langle\overline{T}\rangle(\sigma; M', m : \beta_1\langle\overline{T}\rangle(M) \Rightarrow \tau)\ \mathsf{extends}\ \beta_2\langle\overline{\sigma'}\rangle\}, \Sigma_0 \\ \mathit{override}(\beta_1\langle\overline{T}\rangle(M) \Rightarrow \tau, \beta_2\langle\overline{\sigma'}\rangle) \\ \overline{T}\ \mathsf{type}, \mathsf{this} : \beta_1\langle\overline{T}\rangle(M), \mathsf{fields} : \sigma \mid \Sigma' \vdash e_1 : \tau \qquad \Sigma' \vdash p\ \mathbf{ok} \end{array}}{\Sigma \vdash \mathsf{method}\ m\ \forall\overline{T}.\, \beta\langle\overline{T}\rangle(M) : \tau = e_1\ \mathsf{in}\ p\ \mathbf{ok}}$$

The rule E-EXT-DECL applies.

$$\frac{\Delta = \{\beta\langle\overline{T}\rangle(\overline{m = e})\ \mathsf{extends}\ \beta'\langle\overline{\tau}\rangle\}, \Delta_0}{\begin{array}{c} \mathsf{method}\ m_1\ \forall\overline{T}.\, \beta\langle\overline{T}\rangle(\overline{m : \tau}) : \tau = e_1\ \mathsf{in}\ p \mid \Delta \longmapsto \\ p \mid \{\beta\langle\overline{T}\rangle(\overline{m = e}, m_1 = e_1)\ \mathsf{extends}\ \beta'\langle\overline{\tau}\rangle\}, \Delta_0 \end{array}}$$

From the definition of $\Sigma \vdash \Delta$, Δ has the form $\beta\langle\overline{T}\rangle(\overline{m = e})$ $\mathsf{extends}$ $\beta'\langle\overline{\tau}\rangle, \Delta_0$, where $\Sigma_0 \vdash \Delta_0$. Take $\Delta_2 = \beta\langle\overline{T}\rangle(\overline{m = e}, m_1 = e_1)$ $\mathsf{extends}$ $\beta'\langle\overline{\tau}\rangle, \Delta_0$. We have $\Sigma' \vdash p_1$ **ok**. It remains to show that $\Sigma' \vdash \Delta_2$. This result follows from the premise $\overline{T}$ $\mathsf{type}, \mathsf{this} : \beta\langle\overline{T}\rangle(M), \mathsf{fields} : \sigma \mid \Sigma' \vdash e_1 : \tau$ and the definition of $\Sigma \vdash \Delta$.

case TP-EXPR1. The rule E-EXPR1 applies. The result then follows from the preservation lemma for expressions (Lemma C.13).

case TP-EXPR2. The rule E-EXPR2 applies. The result then follows from the induction hypothesis and the preservation lemma for expressions (Lemma C.13).

□