2007

Hidden Type Variables and Conditional Extension for More Expressive Generic Programs

Hallett, Joseph J.

Boston University Computer Science Department

http://hdl.handle.net/2144/1689

Boston University
HIDDEN TYPE VARIABLES AND CONDITIONAL EXTENSION FOR MORE EXPRESSIVE GENERIC PROGRAMS

by

JOSEPH J. HALLETT

B.A., Colgate University, 2002

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy 2008
Approved by

First Reader

Assaf J. Kfoury, Ph.D.
Professor of Computer Science
Boston University

Second Reader

Guy L. Steele Jr., Ph.D.
Sun Fellow
Sun Microsystems Laboratories

Third Reader

Sukyoung Ryu, Ph.D.
Member of Technical Staff
Sun Microsystems Laboratories
Acknowledgments

I thank Joseph Hallett, Susan Hallett, Ellen Hallett, Lindsay Hallett, and Kathryn Hallett for their love and support. The enduring support provided by Vanessa Kramer has helped me complete this dissertation. For this, I thank her. I thank my examining committee: Assaf Kfoury, Guy Steele, Sukyoung Ryu, Hongwei Xi, and Eric Allen for the encouragement and technical assistance that they provided me. I thank Victor Luchangco for helping me develop early versions of this work and develop my technical abilities in general. I thank Ronald Cohen, Donna Kramer, and Lola Kramer-Cohen for helping me escape my work.
ABSTRACT

Generic object-oriented programming languages combine parametric polymorphism and nominal subtype polymorphism, thereby providing better data abstraction, greater code reuse, and fewer run-time errors. However, most generic object-oriented languages provide a straightforward combination of the two kinds of polymorphism, which prevents the expression of advanced type relationships. Furthermore, most generic object-oriented languages have a type-erasure semantics: instantiations of type parameters are not available at run time, and thus may not be used by type-dependent operations.

This dissertation shows that two features, which allow the expression of many advanced type relationships, can be added to a generic object-oriented programming language without type erasure:

1. type variables that are not parameters of the class that declares them, and

2. extension that is dependent on the satisfiability of one or more constraints.

We refer to the first feature as hidden type variables and the second feature as conditional extension. Hidden type variables allow: covariance and contravariance without variance annotations or special type arguments such as wildcards; a single type to extend, and inherit methods from, infinitely many instantiations of another type; a limited capacity to augment the set of superclasses after that class is
defined; and the omission of redundant type arguments. Conditional extension allows the properties of a collection type to be dependent on the properties of its element type.

This dissertation describes the semantics and implementation of hidden type variables and conditional extension. A sound type system is presented. In addition, a sound and terminating type checking algorithm is presented.

Although designed for the Fortress programming language, hidden type variables and conditional extension can be incorporated into other generic object-oriented languages. Many of the same problems would arise, and solutions analogous to those we present would apply.
# Contents

1 Background  \hspace{1em} 1  
1.1 Generics in Object-Oriented Programming  \hspace{1em} 1  
1.2 Preserving Types at Run Time  \hspace{1em} 3  

2 Pointwise Subtyping  \hspace{1em} 7  
2.1 Limitations of Pointwise Subtyping  \hspace{1em} 7  
\hspace{1em} 2.1.1 Inexpressible Type Relationships  \hspace{1em} 8  
\hspace{1em} 2.1.2 Extraneous Type Parameters  \hspace{1em} 9  
\hspace{1em} 2.1.3 Fixed Extension  \hspace{1em} 10  
2.2 Extending Pointwise Subtyping  \hspace{1em} 11  

3 Hidden Type Variables  \hspace{1em} 12  
3.1 An Introductory Example  \hspace{1em} 12  
3.2 The Dual Purposes of Hidden Type Variables  \hspace{1em} 15  
3.3 Relation to Ordinary Type Parameters  \hspace{1em} 15  
3.4 Occurrence of Hidden Type Variables  \hspace{1em} 18  
\hspace{1em} 3.4.1 Hidden Type Variables in Methods  \hspace{1em} 18  
\hspace{1em} 3.4.2 Hidden Type Variables in Field Types  \hspace{1em} 20  
3.5 Relation to Type Inference  \hspace{1em} 21  
3.6 Challenges of Hidden Type Variables  \hspace{1em} 21  

4 Applications of Hidden Type Variables  \hspace{1em} 23  
4.1 Infinitely Broad Extensions  \hspace{1em} 23  
4.2 Variance Types  \hspace{1em} 24  
4.3 Variance Relations between Different Types  \hspace{1em} 26  
4.4 Selective Extensions  \hspace{1em} 27  
4.5 Open Extensions  \hspace{1em} 28  
4.6 Elimination of Extraneous Parameters  \hspace{1em} 28  

5 Conditional Extension  \hspace{1em} 30  
5.1 An Introductory Example  \hspace{1em} 30  
5.2 Hidden Type Variables or Extra Constraints  \hspace{1em} 31  
5.3 Challenges of Conditional Extension  \hspace{1em} 31
6 Applications of Conditional Extension
   6.1 *able Extensions ..................................... 32
   6.2 Algebraic Properties .................................. 33

7 The Languages: HTV0, HTV1 and CE
   7.1 HTV0 ........................................ 36
      7.1.1 Type System .................................. 38
   7.2 HTV1 ........................................ 39
   7.3 CE ....................................... 40
   7.4 Intermediate Languages ............................... 41

8 Semantics of Hidden Type Variables
   8.1 Witnesses for Hidden Type Variables .................... 42
      8.1.1 Witnesses for Subtype Checking .................... 43
      8.1.2 Witnesses for Type Checking Method Calls ............ 43
      8.1.3 Witnesses for Dispatching Method Calls ............... 44
   8.2 Selecting Witnesses for Hidden Type Variables ............ 44
      8.2.1 Unique Most-Specific Witnesses .................... 45
   8.3 Annotating Programs with Witnesses ...................... 46
      8.3.1 Path Annotations ................................ 47
      8.3.2 Multiple Extension and Ambiguous Paths ............... 49
      8.3.3 Additional Witness Annotations ...................... 49
   8.4 Formal Semantics .................................. 51
      8.4.1 Notational Conventions ............................ 51
      8.4.2 HTV0 .................................. 51
      8.4.3 HTV1 .................................. 67
   8.5 Type Soundness .................................. 73
   8.6 Preventing Cycles in the Type Hierarchy ................ 73

9 Implementing Hidden Type Variables
   9.1 Syntax of HTV1 in PLT Redex ........................ 75
   9.2 Evaluation Rules of HTV1 in PLT Redex .................. 78
   9.3 Type Checking HTV1 ................................ 84
      9.3.1 Subtype Constraint Solving ....................... 84
      9.3.2 Inferring Paths ................................ 104
      9.3.3 The Rest of the Type Checker .................... 104

10 Semantics of Conditional Extension
   10.1 Syntax of CE .................................. 106
   10.2 Static Semantics of CE ................................ 107
   10.3 Dynamic Semantics of CE .............................. 109
   10.4 Type Soundness .................................. 109
## List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Syntax of HTV0</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Difference between the Syntax of HTV1 and HTV0</td>
<td>39</td>
</tr>
<tr>
<td>7.3</td>
<td>Difference between the Syntax of CE and HTV1</td>
<td>40</td>
</tr>
<tr>
<td>8.1</td>
<td>Covariant List Example</td>
<td>43</td>
</tr>
<tr>
<td>8.2</td>
<td>Path Extension Example</td>
<td>47</td>
</tr>
<tr>
<td>8.3</td>
<td>Syntax of Intermediate Language for HTV0</td>
<td>52</td>
</tr>
<tr>
<td>8.4</td>
<td>Path Rules for HTV0</td>
<td>53</td>
</tr>
<tr>
<td>8.5</td>
<td>Subtyping Rules for HTV0</td>
<td>54</td>
</tr>
<tr>
<td>8.6</td>
<td>Valid Paths in HTV0</td>
<td>54</td>
</tr>
<tr>
<td>8.7</td>
<td>Type Well-Formedness Rules for HTV0</td>
<td>55</td>
</tr>
<tr>
<td>8.8</td>
<td>Method Dispatch for HTV0</td>
<td>56</td>
</tr>
<tr>
<td>8.9</td>
<td>Expression Typing Rules for HTV0</td>
<td>57</td>
</tr>
<tr>
<td>8.10</td>
<td>Program Typing Rule for HTV0</td>
<td>58</td>
</tr>
<tr>
<td>8.11</td>
<td>Inherited Methods for HTV0</td>
<td>58</td>
</tr>
<tr>
<td>8.12</td>
<td>Trait and Object Definition Typing Rules for HTV0</td>
<td>59</td>
</tr>
<tr>
<td>8.13</td>
<td>Method Typing Rule for HTV0</td>
<td>59</td>
</tr>
<tr>
<td>8.14</td>
<td>Overriding Rules for HTV0</td>
<td>60</td>
</tr>
<tr>
<td>8.15</td>
<td>Path Comparison Relation for HTV0</td>
<td>62</td>
</tr>
<tr>
<td>8.16</td>
<td>Path Extension for HTV0</td>
<td>63</td>
</tr>
<tr>
<td>8.17</td>
<td>Path Update for HTV0</td>
<td>65</td>
</tr>
<tr>
<td>8.18</td>
<td>Values, Evaluation Contexts and Redexes for HTV0</td>
<td>66</td>
</tr>
<tr>
<td>8.19</td>
<td>Redex Evaluation Rules for HTV0</td>
<td>66</td>
</tr>
<tr>
<td>8.20</td>
<td>Context Evaluation Rule for HTV0</td>
<td>67</td>
</tr>
<tr>
<td>8.21</td>
<td>Evaluation Rules for HTV0</td>
<td>67</td>
</tr>
<tr>
<td>8.22</td>
<td>Difference between the Syntax of Intermediate Languages for HTV1 and HTV0</td>
<td>68</td>
</tr>
<tr>
<td>8.23</td>
<td>Bounds Conversion for HTV1</td>
<td>68</td>
</tr>
<tr>
<td>8.24</td>
<td>Difference between the Path Rules for HTV1 and HTV0</td>
<td>68</td>
</tr>
<tr>
<td>8.25</td>
<td>Difference between the Type Well-Formedness Rules for HTV1 and HTV0</td>
<td>69</td>
</tr>
<tr>
<td>8.26</td>
<td>Method Dispatch for HTV1</td>
<td>69</td>
</tr>
<tr>
<td>8.27</td>
<td>Difference between the Expression Typing Rules for HTV0 and HTV1</td>
<td>70</td>
</tr>
<tr>
<td>8.28</td>
<td>Trait and Object Definition Typing Rules for HTV1</td>
<td>70</td>
</tr>
<tr>
<td>8.29</td>
<td>Method Typing Rule for HTV1</td>
<td>71</td>
</tr>
<tr>
<td>8.30</td>
<td>Overriding Rules for HTV1</td>
<td>72</td>
</tr>
<tr>
<td>8.31</td>
<td>Difference between Path Update for HTV1 and HTV0</td>
<td>72</td>
</tr>
<tr>
<td>8.32</td>
<td>Difference between the Evaluation Contexts and Redexes for HTV1 and HTV0</td>
<td>72</td>
</tr>
</tbody>
</table>
8.33 Difference between the Redex Evaluation Rules for HTV1 and HTV0

9.1 Implementation of Syntax of Intermediate Language for HTV1
9.2 Implementation of Values and Evaluation Contexts for HTV1
9.3 Implementation of Path Update for HTV1
9.4 Implementation of Evaluation Rules for HTV1 (I)
9.5 Implementation of Evaluation Rules for HTV1 (II)
9.6 High-Level Diagram of the Constraint Solving Algorithm
9.7 Auxiliary Functions for Constraint Solving Pseudocode
9.8 Pseudocode: Skeleton of the Type Hierarchy Search
9.9 Pseudocode: Bound Checking
9.10 Pseudocode: Constraint Simplification
9.11 Pseudocode: Witness Construction
9.12 Pseudocode: Witness Checking
9.13 Pseudocode: Constraint Solving

10.1 Difference between the Syntax of Intermediate Languages for CE and HTV1
10.2 Difference between the Path Rules for CE and HTV1
10.3 Trait and Object Definition Typing Rules for CE
10.4 Inherited Methods for CE
10.5 Path Comparison Relation for CE

11.1 Difference between the Implementation of Syntax of Intermediate Languages for CE and HTV1
List of Abbreviations

BNF    Backus-Naur form
J2SDK5.0 Java 2 Standard Edition Development Kit version 5.0
VPTs   variant parametric types
JLS    The Java Language Specification
GADTs  generalized algebraic data types
LR parsing parsing by reading input Left-to-right and producing a Rightmost derivation
Chapter 1

Background

1.1 Generics in Object-Oriented Programming

Type systems that allow a single program definition to be used with multiple different types of data are called *polymorphic type systems*. Polymorphic type systems allow definitions to be safely reused. For example, a single function may be applied to arguments of different types without causing a run-time error. Similarly, a function may evaluate to values of different types. In addition to functions, data type definitions can be reused. A data type definition, such as a list, may be used several times; each use may contain elements of different types.

Modern programming languages contain many varieties of polymorphic type systems. One of the cornerstones of object-oriented programming languages is the ability to reuse code via *nominal subtype polymorphism*. With subtype polymorphism, types are arranged in a *type hierarchy*, and values of a type $T$ can be used wherever values of any supertype of $T$ are expected. In a nominal subtyping system, the subtype relation is explicitly defined by the programmer. For example, the following defines boxed integers and boxed strings, which are both subtypes of $\texttt{Object}$, in the Java™ Programming Language [22].
class Integer extends Object {
    int i;
    Integer(int i) {this.i=i;}
}

class String extends Object {
    string s;
    String(string s) {this.s=s;}
}

In addition, the following defines a container, which holds elements of all types that are a subtype of Object.

class Container {
    Object element;
    void setElement(Object e) {this.element=e;}
    Object getElement() {return this.element;}
}

Therefore, both:

    Container c = new Container();
    c.setElement(new Integer(5));

and:

    Container c = new Container();
    c.setElement(new String("myElement"));

are valid uses of the class. However, an undesirable consequence of this code reuse is a loss of type information. In our example, the type checker is unable to distinguish between a container of integers and a container of strings. This forces the programmer to downcast elements that are extracted from the container:

    Integer x = (Integer) c.getElement();

Such downcasts are potentially unsafe and can cause the program to get stuck at run time.

An alternative form of code reuse, called parametric polymorphism or generics, avoids this
problem. With parametric polymorphism, a function or data type may be parameterized by types. Type parameters can be used as the types of value parameters, as the bounds of or to instantiate other type parameters, and by type-dependent operations in the body of the parameterized data type or function. Using parametric polymorphism, our container class can be parameterized by an element type:

```java
class Container<X> {
    X element;
    void setElement(X e) {this.element = e;}
    X getElement() {return this.element;}
}
```

This requires the type of the container to be specified at compile time:

```java
Container<Integer> c = new Container<Integer>();
c.setElement(new Integer(5));
```

but avoids run-time type casts when elements are extracted:

```java
Integer x = c.getElement();
```

Together parametric polymorphism and nominal subtype polymorphism provide better data and function abstraction, more code reuse, and reduce the need for run-time checks. However, naive integration of parametric polymorphism and nominal subtype polymorphism can result in an unsound type system [41], as evidenced by analyses of the Java programming language extended with generic types [39, 10, 1].

### 1.2 Preserving Types at Run Time

Many object-oriented languages that provide support for generic types, such as the Java programming language, erase generic type information at compile time. This semantics, called a type-erasure semantics, allows generic programs to be backward and forward compatible with non-generic programs. In essence, type erasure removes all type arguments from a program, that is:
\begin{verbatim}
new List<Integer>();
\end{verbatim}

would become:

\begin{verbatim}
new List();
\end{verbatim}

and type parameters are replaced by their bounds, for example:

\begin{verbatim}
class List<X extends Object> {
    void insert(X x) {...}
}
\end{verbatim}

would become:

\begin{verbatim}
class List {
    void insert(Object x) {...}
}
\end{verbatim}

In addition, casts are inserted into the program wherever the erased type information is required. In this way, type erasure translates generic programs into non-generic equivalents.

An undesirable consequence of type erasure is that instantiations of type parameters are used only to improve the precision of static (i.e., compile-time) type checking; they are not available at run time, and thus may not be used by type-dependent operations. In other words, type variables are relegated to a second-class status; they cannot be used in any operation that depends on type information at run time. For example, in the Java programming language, such operations include:

- new operations on type variables, such as new X() or new X[]
- instanceof tests on generic types
- static operations on generic classes
- catch operations that take generic exception types
- run-time casts to generic types

Unfortunately, many programming patterns require operations that depend on the instantiations of type parameters [2]. For example, consider the following code:
Although this code is a straightforward use of the singleton pattern in a generic tree data structure, it uses type variables in a static method signature. The implementation of the LEAF field must be suitable for all instantiations of X. But there is no concrete type that can be substituted for X (i.e., there is no type compatible with all other types)! Therefore a type-erasing compiler is forced to reject this program.

In order to overcome these limitations, programmers are forced to perform elaborate workarounds. Consider the toArray method defined in the java.util.Collection<E> interface. This method returns an array containing all of the elements in the collection. However, to allocate the array, an argument of the appropriate type must be passed to the method:

    <T> T[] toArray(T[] a)

Given this argument, the method can use reflection to allocate an array of type T[]:

    a = (T[]) java.lang.Array.newInstance
        (a.getClass().getComponentType(), size);

Fortunately, these difficulties can be avoided if generic type information is preserved at runtime. Cartwright and Steele have shown that it is possible to implement Generic Java without a
type-erasure semantics, while maintaining backward compatibility and good performance [13, 3]. By implementing a separate class for each distinct instantiation of a generic type, the compiler can preserve type information, which is then available at run time.
Chapter 2

Pointwise Subtyping

Conventional programming languages with nominal subtyping and generics employ a pointwise subtyping scheme. Pointwise subtyping is a straightforward combination of subtyping and generics in which a generic type extends another generic type with some of its type parameters. For example, the following program declares a list of $X$’s to be a subtype of a collection of $X$’s.

```java
class List<X> extends Collection<X>
```

Using this declaration, the type `List<String>` would be a subtype of the type `Collection<String>`.

### 2.1 Limitations of Pointwise Subtyping

Pointwise subtyping requires all type variables in a class declaration to be parameters of the class. In other words, given the following class declaration:

```java
class C₁<X₁ extends S₁,...,Xₘ extends Sₘ> extends C₂<T₁,...,Tₙ>
```

every type variable must be one of $X₁,...,Xₘ$. In addition, type variables are constrained up-front.

---

1Type variables may also be parameters of a method definition, but these variables can appear only in the method.
That is, the only constraints on type variables are $X_1 \text{ extends } S_1, \ldots, X_m \text{ extends } S_m$; it is not possible to further constrain a particular use of a type variable.

These restrictions prevent many useful type relationships such as non-parameterized types extending parameterized types and parameterized types that extend a different instantiation of themselves. In addition, these restrictions lead to redundant type parameterization and inflexible type extension. Next, we describe the effects of these restrictions in detail.

### 2.1.1 Inexpressible Type Relationships

Variance types are parameterized types with subtyping relationships between their different instantiations. If a type name $T$ has a covariant parameter $X$, and type $S$ is a subtype of $R$, then the instantiation $T<S>$ is a subtype of $T<R>$. If $X$ is contravariant, then $T<R>$ is a subtype of $T<S>$. These rules extend to multiple type parameters in the obvious way.

Expressing variance types naturally in a conventional object-oriented language is prohibited by the fact that all type variables must be parameters of the defined class. For example, the natural expression of covariant lists would look like:

```java
class List<X extends Y> extends List<Y>
```

But this declaration is rejected because $Y$ is an unbound type variable. To bind $Y$ it must be a parameter of $List$:

```java
class List<X extends Y, Y extends Object> extends List<Y, Y>
```

However, our $List$ declaration has become unwieldy; it is no longer clear which type variable corresponds to the element type of the list. As an alternative, annotations on type parameters, such as variant parametric types [25], and special type arguments, such as wildcards in the Java programming language [45], have been investigated.

Other type relationships besides variance types cannot be expressed in a pointwise-subtyping language. For example, in the following ML variant type:
datatype 'a option = None
    | Some of 'a

the variant None is not parametric with respect to 'a because it makes no reference to 'a; a single value None is shared by all instantiations of 'a option. This relationship may be represented via nominal subtyping in the Java programming language as follows:

```java
class Option<X>
    class None<X> extends Option<X>
    class Some<X> extends Option<X>
```

However, this encoding does not conform to the original because None is parametric with respect to X. In effect, each instantiation of Option has a separate None option. This distinction prevents code reuse: None cannot be used as both a boolean option and an integer option, for example. We cannot encode the desired relationship in conventional object-oriented languages:

```java
class None extends Option<X>
```

because X must be bound as a type parameter of None.

### 2.1.2 Extraneous Type Parameters

In conventional object-oriented languages, we must sometimes string along sequences of easily inferred type parameters just to instantiate bounds on “the ones that matter”. For example, suppose we wish to encode physical units and dimensions as types, to facilitate static checking of units. We want values to be measurements in a particular unit, such as 5 meters or 12.4 seconds. We define a type Quantity that contains all dimensional values. Each particular dimension, such as Length, is a subtype of Quantity:\(^2\)

```java
class Quantity
    class Length extends Quantity
```

\(^2\)Alternatively, a type Dimension could be defined as a metaclass whose elements are the dimensions such as Length [27, 5]. However, such an encoding requires support for metaclasses, which is out of the scope of this dissertation.
We might then encode units as follows:

```java
class Unit<Q extends Quantity>   
class Meter extends Unit<Length>
```

A measurement is a value with a particular unit. However, it must include two type parameters: one corresponding to its unit, and one corresponding to the dimension of its unit:

```java
class Measurement<Q extends Quantity,   
    U extends Unit<Q>>
```

The first type parameter is redundant: its value can be inferred from an instantiation of the second.

### 2.1.3 Fixed Extension

In conventional object-oriented languages, the same number of types are extended for every instantiation of a class. However, sometimes it is preferable to make extension dependent on the instantiation of a type variable. For example, consider the class `Printable`, which provides an abstract method that, when implemented, allows an object to print itself. For example, `PSFile` can print itself, but `Horse` cannot:

```java
class PSFile extends Printable
class Horse extends Object
```

Collections of `PSFile` can also print themselves. However, without knowing the element type of the collection, it is impossible to determine whether that collection is capable of printing itself. For example, given:

```java
class Vector<X> extends Printable
```

`Vector<PSFile>` is clearly capable of printing itself, however, printing `Vector<Horse>` is nonsensical. Therefore, a programmer is forced to choose between a non-printable vector, which
can hold both \texttt{PSFile} and \texttt{Horse}, and a printable vector, which can hold only \texttt{PSFile}. There is no middle ground.

### 2.2 Extending Pointwise Subtyping

This dissertation shows that two features, which allow the expression of many advanced type relationships, can be added to a generic object-oriented programming language without type erasure:

1. type variables that are not parameters of the class that declares them, and
2. extension that is dependent on the satisfiability of one or more constraints.

We refer to the first feature as \textit{hidden type variables} and the second feature as \textit{conditional extension}. Hidden type variables and conditional extension overcome the limitation of pointwise subtyping that were discussed in the previous section. The next four chapters will introduce these two features and describe their applications.
Chapter 3

Hidden Type Variables

The Fortress programming language [4] supports both parametric and subtype polymorphism without type erasure. In addition, it supports hidden type variables, which are generic type variables declared in a where clause that is separate from ordinary type parameter declarations. Hidden type variables are first-class types: they may be used in type-dependent operations. With hidden type variables, we can express covariance and contravariance without additional variance annotations, and a single type can extend, and inherit methods from, infinitely many instantiations of another type. Hidden type variables also enable the omission of redundant type parameters in type instantiations. Because hidden type variables are not explicitly instantiated as ordinary type parameters are, we must be able to construct instantiations. This construction may be needed at compile time or run time. This requirement significantly complicates the semantics, in return for added expressive power.

3.1 An Introductory Example

In statically typed languages with nominal subtyping, a parametric type hierarchy is often used to recursively define composite data types such as lists, trees, and graphs. In the Fortress programming language, we can model such data types with object and trait definitions [4]. An object definition defines either a single object or (if it is parametric) a set of objects with common structure. A trait definition defines a named trait with a set of methods in a multiple-inheritance hierarchy; every
object extending a trait inherits all the methods declared in that trait and all of its supertraits. Objects are leaves of the hierarchy; they cannot be extended. As a simple example, we can represent lists in Fortress as follows:

```fortress
trait List extends {Collection, Object} end
object Empty extends List end
object Cons(first: Object, rest: List) extends List end
```

We declare a trait `List`, extending two supertraits, `Collection` and `Object`. We also declare two subtypes of `List`: the singleton object `Empty` and the parametric object `Cons`. `Cons` includes a constructor with two parameters: `first` (of type `Object`) and `rest` (of type `List`). Each parameter of a constructor indicates the existence of a field in an instance of the object, whose value is determined by the corresponding argument in a constructor call. Fields that do not correspond to constructor parameters can be defined in the body of an object definition.

Unfortunately, this type hierarchy lacks precision: All we know about the types of the elements of lists is that they are subtypes of type `Object` (the top of the type hierarchy). We can increase the precision by augmenting each definition with a type parameter `X` (bound in white square brackets), which denotes the static type of the elements in a list. Each distinct instantiation of this parameter corresponds to a distinct type of list:

```fortress
trait List[X] extends {Collection, Object} end
object Empty[X] extends List[X] end
object Cons[X](first: X, rest: List[X]) extends List[X] end
```

However, the definition of `Empty` has changed in a significant and undesirable way. Whereas before, there was exactly one empty list object shared by all lists, now each instantiation of `Empty[X]` corresponds to a distinct value. Effectively, we have infinitely many empty lists, one for each instantiation of `Empty[X]`! This parameterization is misleading (it implies that the structure of `Empty` is dependent on type parameter `X`), wasteful of memory (multiple values are kept, with indistinguishable structure), and inhibits polymorphism (the value `Empty[String]` cannot be used in a context requiring a `List[Number]`, for example). Because an empty list contains no elements, it can be (and, in dynamically typed languages such as Scheme and Python, often is) used safely in a
context requiring a list of any element type. But the parameterization of Empty prevents such use.

In a conventional nominal subtyping hierarchy, there is no way to represent the intended relationship that a single base case can stand in for any of the parametrically defined inductive cases of a composite data type such as List: Subtypes must specify their supertypes, which must be well-formed in the enclosing type environment.\(^1\) Requiring subtypes to specify their supertypes is an essential property of object-oriented languages; it allows programmers to add new subtypes without having to edit or recompile supertypes. However, as demonstrated above, this cornerstone of object-oriented programming prevents the expression of certain relationships when combined with parametric polymorphism.

We can more clearly see the difficulty in capturing the desired type relationship by encoding subtyping relationships in a higher-order predicate calculus over a universe consisting of types as predicates and values as the objects of discourse. Then the Empty object definition above implies that Empty is a second-order function mapping predicates to predicates (as is List) that satisfies the following statement:\(^2\)

\[
\forall X. \forall x. \text{Empty}[X](x) \implies \text{List}[X](x)
\]

That is, whenever a value \(x\) satisfies the predicate \(\text{Empty}[X]\), it also satisfies the predicate \(\text{List}[X]\).

We can “de-Skolemize” \(^{[20]}\) this statement so that it contains an existential quantification instead of a function definition:

\[
\forall X. \exists E_X. \forall x. E_X(x) \implies \text{List}[X](x)
\]

Conceptually, there are two quantifications involving \(X\), one universal, the other existential. Because conventional parametric type systems do not separate these two quantifications, we cannot declare many important relationships among types. For example, we cannot state universally quantified relationships where there is no implicit existential quantification, as with empty lists, where all instantiations of type List apply to a single empty list:

\(^1\)The inclusion of a null value (as in the Java programming language) would not solve the problem with Empty: because null is a single built-in type shared by all types, method calls on null cannot be statically ensured to succeed. In contrast, we can statically check that the definition of Empty includes a definition for every method declared in supertrait List.

\(^2\) We use white square brackets [ ] to denote higher-order function application, and \(\implies\) to denote logical implication.
∀X.List[X](Empty)

To express this relation (and many other important type relationships), we propose to generalize
conventional nominal subtyping to allow the declaration of type variables in a where clause that is
separate from ordinary type parameter declarations. As with ordinary type parameters, the type vari-
ables in a where clause denote universal quantification. But unlike with ordinary type parameters,
the parameterized type is not a function of these hidden variables; that is, there is no accompanying
existential quantification over a parameterized type in de-Skolemized form. For example, we can
write the declaration:

object Empty extends List[X] where { X extends Object } end

This definition declares that a single object Empty has all the methods of every instantiation of
List[X].

3.2 The Dual Purposes of Hidden Type Variables

Hidden type variables serve two purposes: introducing new constraints on existing type variables
and introducing new type variables to be used elsewhere in the program. Often times these two
purposes are intertwined; the same hidden type variable serves both purposes. For example, consider
a covariant list in Fortress:

trait List[X extends Y] extends List[Y] where { Y extends Object } end

The newly introduced type variable Y provides an upper bound on X as well as instantiating the
supertype List[Y].

3.3 Relation to Ordinary Type Parameters

Logically, hidden type variables are interpreted the same as ordinary type parameters. To logi-
cally interpret hidden type variables, we model objects as records whose components are functions
representing methods [12, 11]. Therefore, the object:
object Cons(first : Object, rest : List) extends List
    append(l : List) : List = Cons(first, rest.append(l))
end

would be interpreted as the record:

Cons = { first : () → Object, rest : () → List, append : List → List }

Notice that data members of the object are encoded as getter methods. The special type () is used to represent the empty tuple. Extension is modeled in the straightforward way:

Cons <: List

In a language with parametric polymorphism, type parameters may be added to both the objects and methods:

object Cons[X extends Object](first : X, rest : List[X])
extends List[X]
    append(l : List[X]) : List[X] = Cons[X](first, rest.append(l))
end

In this case, our interpretation would be augmented as follows:

∀X . (X <: Object) ⇒
Cons[X] = { first : () → X, rest : () → List[X], append : List[X] → List[X] }

and:

∀X . (X <: Object) ⇒ (Cons[X] <: List[X])

Now, consider adding hidden type variables to our example:

object Cons[X extends Y](first : X, rest : List[X])
extends List[Y]
    where {Y extends Object}
    append(l : List[Y]) : List[Y] = Cons[Y](first, rest.append(l))
end
The logical interpretation would be:

$$\forall X . \forall Y . ((X <: Y) \land (Y <: \text{Object})) \implies \text{Cons}[X] = \{ \text{first} : () \to X, \text{rest} : () \to \text{List}[X], \text{append} : \text{List}[Y] \to \text{List}[Y] \}$$

and:

$$\forall X . \forall Y . ((X <: Y) \land (Y <: \text{Object})) \implies (\text{Cons}[X] <: \text{List}[Y])$$

Notice that, like ordinary type parameters, hidden type variables are universally quantified. The difference between hidden type variables and ordinary type parameters is when they are instantiated. Ordinary type parameters are instantiated when a type is created. However, hidden type variables are instantiated when a type is used.

Because a hidden type variable is not a parameter of a trait or object, the instantiation of a hidden type variable is not fixed for each type. A type fixes the instantiation of an ordinary type parameter by definition, but, a type may have several different instantiations of a hidden type variable depending on its context.

For example, consider an empty list written in Fortress:

```fortress
object Empty extends List[X] where \{ X extends \text{Object} \} end
```

Within the same program, an `Empty` object may be used as a list of floating-point numbers and as a list of strings. In other words, `X` may be instantiated to `Float` in one use of the object, and to `String` in another use of the `same` object.

As a result of this property, hidden type variables allow for code reuse beyond that provided by a straightforward combination of parametric polymorphism and nominal subtype polymorphism. For example, using the same empty list object as both a generic list of floating-point number and a generic list of strings is not possible in a language with pointwise subtyping.

For a more thorough investigation of hidden type variables see Chapter 8, which presents a formal semantics.
3.4 Occurrence of Hidden Type Variables

When designing a language with hidden type variables, one must determine where a hidden type variable is allowed to occur. Some occurrences of hidden type variables allow for greater expressivity, while others are problematic. Consider the following object definition:

```plaintext
object O[X extends A](x : B) extends C where {Y extends D}
  m[Z extends E](x : F) : G = e
end
```

where `e` is an arbitrary expression. Syntactically, a hidden type variable may occur in any of the following places:

- type variable bounds (`A, D`)
- field types (`B`)
- extended types (`C`)
- the types or body of a method (`E, F, G, e`)

This dissertation will develop languages that allow hidden type variables to occur in each location except field types. In the following, we examine the use of hidden type variables in method definitions and field types in more detail.

3.4.1 Hidden Type Variables in Methods

Allowing hidden type variables to occur in method definitions can provide greater expressivity. For example, consider the following covariant list:

```plaintext
trait List[X extends Y] extends List[Y] where {Y extends Object} end
```

Without allowing hidden type variables to occur in methods (and assuming no lower bounds on type variables), a purely-functional definition of the `cons` method would not be possible. A definition such as:

```plaintext
trait List[X extends Y] extends List[Y] where {Y extends Object}
  cons(x : X) : List[X] = Cons[X](x, self)
end
```
is invalid overriding because the type List[X] extends the following trait (where Y is renamed to Y' for exposition):

```
trait List[Y extends Y'] extends List[Y'] where {Y' extends Object}
cons(x : Y) : List[Y] = Cons[Y](x, self)
end
```

The parameter type of the inherited definition, Y, is a supertype of the overriding definition's parameter type, X. As a result, the argument of a call to cons may be a subtype of Y at compile time, but fail to be a subtype of X at run time. This would result in the method call “getting stuck” at run time. Therefore, this is an invalid overriding.

However, if a hidden type variable is allowed to appear in the method definition, then the following method can be defined:

```
trait List[X extends Y] extends List[Y] where {Y extends Object}
cons(x : Y) : List[Y] = Cons[Y](x, self)
end
```

The type List[X] provides infinitely many methods, one for each instantiation of the hidden type variable Y. In this case, the type List[X] extends the following trait (where Y is renamed to Y' for exposition):

```
trait List[Y extends Y'] extends List[Y'] where {Y' extends Object}
cons(x : Y') : List[Y'] = Cons[Y'](x, self)
end
```

The type List[Y] also provides infinitely many methods, one for each instantiation of the hidden type variable Y'. Notice that each instantiation of Y' is a valid instantiation of Y. Therefore, the type List[X] redefines all of the cons methods that are inherited from the type List[Y]. In this way, the overriding method “covers” the inherited method. Therefore, this definition is a valid overriding. A formal definition of valid overriding with hidden type variables is given in Chapter 8.

Notice that it is also possible to define the cons method as a generic definition. For example, consider the following:

```
trait List[X extends Y] extends List[Y] where {Y extends Object}
```
We use the keyword `bounds` to require that `Z` be a supertype of `X`. This definition defines infinitely many methods, one for each instantiation of `Z`. This definition overrides the following inherited method (where `Z` is renamed to `Z'` for exposition):

```
trait List[Z extends Y'] extends List[Y'] where {Y' extends Object}
  cons[Z' bounds Y'](x: Z'): List[Z'] = Cons[Z'](x, self)
end
```

Because `Y` is a supertype of `X`, each instantiation of `Z'` is a valid instantiation of `Z`. Therefore, this is a valid overriding.

This dissertation will develop a language that allows hidden type variables to occur in method definitions. In addition, an alternative language will be developed, which replaces method definitions containing hidden type variables with generic method definitions.

### 3.4.2 Hidden Type Variables in Field Types

Hidden type variables in field types are problematic. Intuitively, such objects would contain infinitely many fields with the same name, one for each instantiation of the hidden type variables:

```
object Full extends List[X] where {X extends Object}
  foo: X = ??????
end
```

We do not believe a sensible semantics can be provided to a field whose type contains a hidden type variable—how would one replace the right-hand side of the definition of `foo` to provide initializing values for all of these fields? Consider rewriting fields into getter methods. This rewrite, when applied to a field whose type contains a hidden type variable, would define an infinite number of method definitions, one for each instantiation of the hidden type variable. Because each getter method would have the same parameter type, namely `()`, there is no way to know which method is invoked at a call site.
For these reasons, the languages developed in this dissertation do not allow hidden type variables to occur in field types.

### 3.5 Relation to Type Inference

The instantiations for hidden type variables must be inferred by the compiler and run-time system. Although many object-oriented programming languages infer instantiations for type parameters, the apparent similarity of hidden type variables to conventional type inference is superficial; these two mechanisms serve entirely different purposes. Whereas type inference is used to add type annotations that a programmer could have expressed manually, hidden type variables allow programmers to define new type relationships that are otherwise inexpressible. Some of these new type relationships are discussed in Chapter 4.

Instantiations for hidden type variables are inferred when checking subtype relationships and dispatching methods. Therefore, the inference procedure may be used multiple times for a single value. On the other hand, type inference in a conventional programming language is used to assign a type to a value, which occurs only once per value. In this way, type inference plays a larger role in the semantics of a language with hidden type variables than in a conventional language. Chapter 8 discusses the role of inference in the semantics of hidden type variables in more detail. Chapter 9 discusses how this inference is implemented.

Note also that conventional languages require the programmer to instantiate either all or none of the type parameters. By separating the hidden type variables from the ordinary type parameters, a class (or trait) can be designed such that some of the type parameters are explicitly instantiated and used to infer the instantiations of hidden type variables.

### 3.6 Challenges of Hidden Type Variables

The expressive power of hidden type variables does not come for free: several complications arising from their introduction must be resolved to ensure a sound type system. For example, when checking subtyping relationships among types with hidden type variables, one type may be a subtype of
infinitely many other types. Therefore, determining whether one type is a subtype of another re-
quires determining whether there is a path between two types in a hierarchy that can have infinitely
broad branches.

Moreover, if hidden type variables are in scope over the entire type definition, they can impact
not only subtyping relationships, but also method definitions, inheritance, and overriding. Method
declarations that refer to hidden type variables declare infinitely many methods (one for each instan-
tiation of the hidden type variables). Similarly, a subtype of multiple instantiations of a single trait
inherits multiple methods with the same name. We must ensure it is possible to distinguish which
method is referred to at a call site.

These issues significantly complicate a formal treatment of hidden type variables. In the re-
mainder of this dissertation, we explore these problems, and show how to devise a type system that
captures the notion of hidden type variables. We prove the type system sound and present an al-
gorithm for checking subtyping in this system. However, this algorithm does not terminate on all
inputs; we discuss restrictions on the type system that ensure termination. We have mechanized the
semantics of hidden type variables in PLT Redex [35]. The mechanization is available online [23].
Chapter 4

Applications of Hidden Type Variables

Although simple, the idea of hidden type variables provides surprisingly rich expressive power. It enables us to elide redundant type parameters from definitions, encode many-one relationships among instantiations of multiple parametric types, and define types with infinitely many methods, each with a distinct type. It also allows us to express complex subtyping relationships among various instantiations of a single parametric type; in particular, we can encode covariance and contravariance without additional language primitives. The ability to universally quantify over subtyping definitions is quite powerful, as the following applications demonstrate.

4.1 Infinitely Broad Extensions

Hidden type variables allow a single type to extend infinite supertypes. A single type can be a subtype of another type instantiated with hidden type variables, which admits an infinite number of instantiations. For example, the None type of an option type need not be parameterized:

```plaintext
trait Option[X] end
object None extends Option[X] where {X extends Object} end
object Some[X](x: X) extends Option[X] end
```

The resulting program captures the desired specification and enjoys the performance benefits of sharing a single value None among all instantiations of Option[X].
In statically typed languages, inductively defined data structures are often parametric with respect to an element type. Common examples are the types \( \text{List}[X] \), \( \text{Tree}[X] \) and \( \text{Queue}[X] \). However, the base cases of these inductive definitions are sometimes independent of a particular element type. For example, no elements are contained in an empty list or an empty queue. And, often times, no elements are in a leaf of a tree. Therefore, parameterizing the base cases by an element type may be misleading. Even more, this parameterization prevents sharing base case values in a language that preserves types. For example, \( \text{Leaf}[\text{Float}] \) cannot be used interchangeably with \( \text{Leaf}[\text{String}] \), despite the fact that the two values are observably equivalent. Yet, these base cases must be parameterized in conventional generic object-oriented languages in order to extend a parameterized type:

```plaintext
object Empty[X extends Object] extends List[X] end
object Leaf[X extends Object] extends Tree[X] end
object EmptyQueue[X extends Object] extends Queue[X] end
```

Hidden type variables avoid these shortcomings by allowing the base case of a parametric inductive type to be defined without a type parameter. For example, we can define:

```plaintext
object Empty extends List[X] where \{ X extends Object \} end
object Leaf extends Tree[X] where \{ X extends Object \} end
object EmptyQueue extends Queue[X] where \{ X extends Object \} end
```

### 4.2 Variance Types

Hidden type variables allow us to express subtype relationships between various instantiations of a single parametric type directly in the language. The need to express such relationships is often useful, but is typically allowed only through special-purpose language features. For example, if we wish to define an immutable list type that is covariant in its element type, we need some way to express that some instantiations of type \( \text{List}[X] \) are subtypes of other instantiations. Essentially, we need to declare the following property:

\[
\forall X.\forall Y. (X <: Y) \implies (\text{List}[X] <: \text{List}[Y])
\]
where \(<\): denotes the subtype relation, again interpreting types as predicates.

This property cannot be expressed directly in most object-oriented languages; we cannot include an instantiation of type \(\text{List}[X]\) in its extends clause with type parameters other than those declared for trait \(\text{List}[X]\) itself. As the predicate calculus encoding makes clear, we need to quantify over an extra type parameter \(Y\) to instantiate the supertype. Languages that do allow expression of covariance typically do so via a mechanism designed specifically for this purpose, such as variance annotations in Scala [37] or wildcards in the Java programming language [45]. Using hidden type variables, we are able to express covariance without additional language features:

```scala
trait List[X extends Y]
    extends {List[Y], Collection, Object}
    where {Y extends Object}
    cons(y: Y): List[Y] = Cons[Y](y, self)
end
```

The definition of trait \(\text{List}[X]\) expresses that an instantiation \(\text{List}[X]\) is a subtype of every instantiation \(\text{List}[Y]\) such that \(X\) is a subtype of \(Y\), (i.e., \(\text{List}[X]\) is covariant). Note that the reserved word `extends` is used to denote both the subtyping relation when used as a bound on a type variable and the `extends` relation between a trait and its supertraits. Because the former relation is reflexive, this definition implies that an instantiation of \(\text{List}[X]\) extends itself. As we see in our formal treatment, such trivial self extensions are not problematic. Intuitively, we can think of a self extension as both inheriting and overriding methods from itself. We discuss the technical details of overriding in Section 8.4.

Moreover, our definition of covariant lists gives us a handle on the instantiation of the type parameter of the supertype (in this case, the type variable \(Y\)). This handle is a powerful tool. For example, suppose we have the following definitions of simple numeric types and of two subtypes of the trait \(\text{List}[X]\):

```scala
trait Number extends Object end
trait Z extends Number end
trait R extends Number end
object Cons[X](first: X, rest: List[X]) extends List[X] end
object Empty extends List[Y] where {Y extends Object} end
```
Consider the following method invocation:

\[ \text{Cons}[Z](3, \text{Empty}).\text{cons}(5.7) \]

where 3 has type \( Z \) and 5.7 has type \( \mathbb{R} \).

At first glance, this expression may appear ill typed. How can a real number be added to a list of integers? But note that the parameter type of method \( \text{cons} \) is not \( X \); it is \( Y \). Thus, \( \text{cons} \) can be called on a \( \text{List}[X] \) with any argument of a type \( Y \) that is a supertype of \( X \); the result of the operation is a \( \text{List}[Y] \). The value 5.7 is of type \( \mathbb{R} \), and also of type \( \text{Number} \), which is a supertype of \( Z \). Because \( \text{List}[Z] \) inherits the \( \text{cons} \) method from \( \text{List}[\text{Number}] \), this invocation is well typed with the result:

\[ \text{Cons}[	ext{Number}](5.7, \text{Cons}[Z](3, \text{Empty})) \]

Because the type system can statically generalize the type of the list, the list need not be copied before generalization. Expression of contravariant relationships is also possible.

### 4.3 Variance Relations between Different Types

Hidden type variables can also be used to express variance relations between different types. For example, \( \text{List}[X] \) can be declared a subtype of \( \text{Collection}[Y] \) where \( X \) is a subtype of \( Y \).

```scala
trait List[X extends Y] extends Collection[Y] where {Y extends Object} end
```

Specialized variance constructs, such as variant parametric types and wildcards, are not designed to express these relationships. However, because hidden type variables can be used as any other type would, such relationships naturally fall out of their design.

The following program demonstrates another example of variance between different types.

```scala
trait Iterable[X]
  iterator(): Iterator[X] = self.iterator()
end

trait Queue[X extends Y] extends Iterable[Y] where {Y extends Object}
... end
```
trait List[X extends Y] extends Iterable[Y] where {Y extends Object}

... end

These definitions allow greater code reuse. To see this, consider a list, l, of type List[Z] and two queues, q₁ and q₂, of types Queue[Z] and Queue[Float], respectively. Given the following two print functions:

\begin{align*}
print_1(x : \text{Iterable}[Z]) & : () \\
print_2(x : \text{Iterable}[\text{Number}]) & : ()
\end{align*}

we can print both l and q₁ using print₁ and all three of l, q₁ and q₂ using print₂. The polymorphic use of print₁ can be achieved by using a pointwise subtyping scheme when defining List[X] and Queue[X]. However, this is not the case for the polymorphic use of print₂. The polymorphic use of print₂ can only be achieved by the variance relations between List[X] and Iterable[Y], and Queue[X] and Iterable[Y].

4.4 Selective Extensions

Using hidden type variables, we can selectively extend only some instantiations of a supertype, as in the following reworking of a classic example from Shang [42]:

\begin{verbatim}
trait Eater[X extends Food]
  eat(x : X): ()
end

trait Herbivore[X extends Plant] extends Eater[X] end

trait Cow extends Herbivore[X] where {X extends Grass} end
\end{verbatim}

Here, Herbivore[X] requires an eat method that only accepts Plant, and Cow requires an eat method that only accepts Grass. Notice that Cow extends every instantiation Herbivore[X], where X is a subtype of Grass. This is preferable to a formulation where Cow extends Herbivore[Grass]. Our definition allows Cow to be used in a context requiring Herbivore[BlueGrass] (assuming BlueGrass is a subtype of Grass).
4.5 Open Extensions

Consider the following definition of Comparable, with abstract methods lessThan and greaterThan:

\[
\text{trait Comparable}\lfloor X \rfloor \\
\quad \text{lessThan}(x: X): \text{Boolean} \\
\quad \text{greaterThan}(x: X): \text{Boolean} \\
\text{end}
\]

Using hidden type variables, we can define a type for boxed values such that an instance of Box\lfloor Y \rfloor is comparable to anything that is comparable to an instance of Y:

\[
\text{object Box}\lfloor Y \rfloor(y: Y) \text{ extends Comparable}\lfloor X \rfloor \text{ where } \{ X \text{ extends Comparable}\lfloor Y \rfloor \} \\
\quad \text{lessThan}(x: X) = x.\text{greaterThan}(y) \\
\quad \text{greaterThan}(x: X) = x.\text{lessThan}(y) \\
\text{end}
\]

Given the following types:

\[
\text{trait Number end} \\
\text{trait Z extends } \{ \text{Number, Comparable}\lfloor Z \rfloor \} \text{ end}
\]

Box\lfloor Z \rfloor extends Comparable\lfloor Z \rfloor. The extends clause of Box is an example of an open extends clause that provides us with a limited capacity to augment the set of superclasses of a class after that class is defined. For example, if we later add the following type:\footnote{In Fortress, R128 denotes 128-bit precision floating point numbers.}

\[
\text{trait R128 extends } \{ \text{Number, Comparable}\lfloor \text{R128} \rfloor, \text{Comparable}\lfloor Z \rfloor \} \text{ end}
\]

then Box\lfloor \text{R128} \rfloor extends Comparable\lfloor \text{R128} \rfloor. In fact, both Box\lfloor \text{R128} \rfloor and Box\lfloor Z \rfloor extend Comparable\lfloor \text{R128} \rfloor.

Notice that, by using conditional extension, we do not require the element type of a box to be comparable to anything. For example, Box\lfloor \text{Number} \rfloor extends only Object.

4.6 Elimination of Extraneous Parameters

Recall the following measurement example from Section 2.1.2:
trait Quantity end
trait Length extends Quantity end

trait Unit[Q extends Quantity] end
object Meter extends Unit[Length] end

object Measurement[Q extends Quantity,
      U extends Unit[Q]]
end

Also, recall that the first type parameter is redundant: its value can be inferred from an instantiation of the second. By moving the type parameter into the where clause, we can eliminate the redundant type parameter:

object Measurement[U extends Unit[Q]]
      where { Q extends Quantity }
end

The where clause allows us to omit the dimension, with the intention that it will be derived from the unit parameter, making the resulting program more readable and requiring users of Measurement to instantiate fewer type parameters. For example, programmers can use “Measurement[Mile]” without the redundant dimension parameter instead of “Measurement[Length, Mile]”.

Chapter 5

Conditional Extension

*Conditional extension* provides even greater expressiveness (beyond that provided by hidden type variables) to a language with nominal subtyping and generics. Attaching a where clause to an individual type in the extends clause allows the type to be conditionally extended. That is, the type in the extends clause is a valid extension if and only if the constraints in the where clause are satisfied. We call these constraints the *conditional constraints*. By building on the infrastructure developed for hidden type variables, we can add conditional extension into a generic object-oriented language without much difficulty.

5.1 An Introductory Example

Recall the following example from Section 2.1.3.

```plaintext
trait PSFile extends Printable end
trait Horse extends Object end
trait Vector[X] extends Printable end
```

Also, recall that `Vector[PSFile]` is clearly capable of printing itself, however, printing `Vector[Horse]` is nonsensical.

What is needed is a way to declare vectors as printable if and only if the elements of the vector are, themselves, printable. Conditioning an extended type on the satisfiability of a constraint provides this ability. Using conditional extension, we can define vectors as follows:
trait Vector[[X]] extends {Printable where {X extends Printable}} end

Now, Vector[[PSFile]] is subtype of Printable, but Vector[[Horse]] is not a subtype of Printable.

As an alternative, one could require all elements of a vector to be printable, such as:

trait Vector[[X extends Printable]] extends {Printable} end

This would prevent printing inappropriate types, like Vector[[Horse]], but it also disallows vectors of Horse altogether. Conditional extension separates type well-formedness from type extension. That is, using conditional extension, vectors can contain elements of any type whatsoever, and the extended types, such as Printable, are determined by the element type.

5.2 Hidden Type Variables or Extra Constraints

The language designer must decide whether conditional constraints are used to constrain existing type variables, introduce hidden type variables, or both. If a hidden type variable is introduced then the conditional constraint is satisfied as long as there exists a type that meets the bounds on the hidden type variable. Otherwise, conditional constraint satisfiability amounts to subtype checking.

For the remainder of this dissertation, we assume that conditional constraints do not introduce hidden type variables. However, the semantics and algorithms developed in this dissertation can be extended to allow conditional constraints to introduce hidden type variables. Essentially, a hidden type variable introduced by a conditional constraint must be treated as if it were declared in the trait or object’s outer where clause.

5.3 Challenges of Conditional Extension

Adding conditional extension to a language with hidden type variables requires processing additional constraints whenever the type hierarchy is traversed. For example, when checking subtype relations or dispatching to an inherited method, the conditional constraints must be checked. This dissertation develops a semantics and type checking algorithm for a language with conditional extension.
Chapter 6

Applications of Conditional Extension

Conditionally extending a type is useful when properties of a trait or object are not shared by all instantiations of the trait or object. We show several such examples in this chapter.

6.1 *able Extensions

Chapter 5 discussed the following example:

```scala
trait Vector[X] extends {Printable where {X extends Printable}} end
```

Notice that this example can be altered to fit any collection type, i.e., we could have just as easily defined a set or map to conditionally extend Printable. The following shows a definition of maps:

```scala
trait Map[K, V] extends {Printable where {K extends Printable, V extends Printable}} end
```

Similarly, it is useful to conditionally extend other types besides Printable. In fact, most of the Java interfaces ending in “able” are candidates for conditional extension. We call these *able extensions. For example, consider the following definition:

```scala
trait Vector[X] extends {Serializable where {X extends Serializable}} end
```

Other types such as Comparable, Iterable and Clonable can be conditionally extended in the same way. Certain other types that do not end in “able” such as RandomAccess, which designates
that a collection type supports fast random access to its elements, can also be conditionally extended.

As a side note, consider that many of these *able extensions, like Comparable, define abstract methods, thereby placing an obligation on the extending type. If Comparable is conditionally extended then how are its abstract methods implemented? The methods should be implemented only if the conditional constraints are satisfied. By placing where clauses on methods, we can conditionally implement methods in the same way that we conditionally extend types. For example, consider the following definition:

```scala
trait Vector[X]
  extends {Comparable[X] where {X extends Comparable[X]}}
  compareTo(v: Vector[X]): Boolean where {X extends Comparable[X]} = ...
end
```

In this example, compareTo is defined if and only if X is a subtype of Comparable[X]. Providing where clauses on methods is beyond the scope of this dissertation and will not be investigated further.

### 6.2 Algebraic Properties

For another application of conditional extension, consider the task of encoding algebraic structures into a type hierarchy. A ring can be defined as:

```scala
trait Ring[X extends Ring[X, ⊕, ⊗], opr ⊕, opr ⊗]
  extends {AbelianGroup[X, ⊕], SemiRing[X, ⊕, ⊗]}
end
```

where X is the underlying set of the ring and ⊕ and ⊗ are two binary operations on elements of X called addition and multiplication, respectively. Notice that we include opr ⊕ and opr ⊗ in, what has been up to this point, the type parameter list. Fortress allows traits and objects to contain static parameters beyond types. For the purposes of exposition, we include the operator parameters ⊕ and ⊗. Operator parameters are instantiated by operator names at compile time. Unlike type parameters, they can be used in both type and value contexts. That is, operator parameters can instantiate other operator parameters or be applied to operands of the appropriate type. Fortress also
allows natural number and integer parameters, boolean parameters, dimension parameters, and unit parameters. See the Fortress specification for a detailed discussion of these static parameters [4].

Notice that Ring\([X, \oplus, \otimes]\) extends AbelianGroup\([X, \oplus]\) and SemiRing\([X, \oplus, \otimes]\). These extensions guarantee that the properties of a ring actually hold.

Recall that all square matrices over rings are themselves rings, with matrix addition and matrix multiplication as the operations. Using conditional extension, we can encode this logic in the definition of square matrices as follows:

```fortress
trait SquareMatrix[X extends Object, opr +, opr -]
   extends {Ring[X, +, \cdot] where {X extends Ring[X, +, .]}}
end
```

In this example, the operators + and - are overloaded; they are defined on arguments of type X as well as arguments of type SquareMatrix\([X, +, \cdot]\). By making use of conditional extension, we can treat a square matrix as a ring if we can statically verify that the elements of the matrix are themselves a ring.

A final example of conditional extension for algebraic properties comes from the Fortress standard library. Recall the standard types for rational numbers: \(\mathbb{Q}\), \(\mathbb{Q}^*\) and \(\mathbb{Q}^\#\). The type \(\mathbb{Q}\) contains all finite rational numbers (i.e., the result of dividing any integer by any nonzero integer). The type \(\mathbb{Q}^*\) is \(\mathbb{Q}\) extended with +\(\infty\) and -\(\infty\). The type \(\mathbb{Q}^\#\) is \(\mathbb{Q}^*\) extended with the indefinite rational (written 0/0), which is the result of dividing zero by zero or of adding -\(\infty\) to +\(\infty\).

In Fortress, the types \(\mathbb{Q}\), \(\mathbb{Q}^*\) and \(\mathbb{Q}^\#\) are defined in terms of a single trait RationalQuantity. This trait takes seven static parameters. The first is a dimensional unit parameter and the remaining six are boolean parameters, which specify whether an instance of the trait can be -\(\infty\), a finite rational less than zero, zero, a finite rational greater than zero, +\(\infty\), or the indefinite rational. As it is not relevant to our current discussion, we ignore the first parameter for the remainder of the chapter. Some of the rational types that can be defined using this trait are:

```fortress
type Q = RationalQuantity[false, true, true, true, false, false]
type Q* = RationalQuantity[true, true, true, true, false]
type Q# = RationalQuantity[true, true, true, true, true]
```
The trait `RationalQuantity` is defined using conditional extension. To see why, notice that the three rational types above have different properties: \( \mathbb{Q} \) is both a total order and a field, \( \mathbb{Q}^\star \) is a total order but is not a field, and \( \mathbb{Q}^\# \) is neither a total order nor a field. A simplified version of the `RationalQuantity` definition follows:

```scala
trait RationalQuantity[bool ninf, bool lt, bool eq, bool gt, bool pinf, bool nan]
  extends { Field[RationalQuantity[ninf, lt, eq, gt, pinf, nan], +,-,*,/] 
    where \{lt \& eq \& gt \& \neg ninf \& \neg pinf \& \neg nan\}, 
    TotalOrder[RationalQuantity[ninf, lt, eq, gt, pinf, nan], <,\leq,>,\geq] 
    where \{\neg nan\}}
```

The conditional constraints used in the `RationalQuantity` trait are boolean constraints. Although this dissertation deals with subtype constraints only, the semantics of conditional extension developed here can be applied to other kinds of constraints, for example, boolean constraints or numeric constraints.\(^1\) Similarly, the subtype constraint solver in our implementation of conditional extension can be removed and an arbitrary constraint solver can be plugged in.

---

\(^1\)Notice that [P-Ext], [T-TraitDef], and [T-ObjectDef] are the only rules (defined in Chapter 8) that need to be altered in order to allow conditional extension with other kinds of constraints. And, these rules require only small tweaks. Rule [P-Ext] would require the premise that checks the conditional constraint to be replaced by another premise for checking the satisfiability of the new constraints. Rules [T-TraitDef] and [T-ObjectDef] would require new premises to check the well-formedness of the new conditional constraints. These are relatively small changes. Mostly the semantics can be reused.
Chapter 7

The Languages: HTV0, HTV1 and CE

This dissertation is done in the context of the Fortress programming language. To explore the semantics and implementation of hidden type variables and conditional extension, we develop three languages based on the Fortress language: HTV0, HTV1 and CE. In this chapter, we present the syntax of these languages and informally describe the semantics. First, we present HTV0, which provides support for hidden type variables and type-dependent operations in a multiple extension type system. Next, we augment HTV0 with the ability to write lower bounds on type variables. We call this language HTV1. We describe how HTV1 provides an alternative to the semantics of HTV0. Lastly, we introduce CE, which is an extension of HTV1. In addition to hidden type variables, CE provides the ability to conditionally extend a type.

7.1 HTV0

Figure 7.1 shows the syntax of HTV0. For conciseness, we use $\overrightarrow{A}$ to denote a possibly empty sequence $A_1, \ldots, A_n$. A program consists of a sequence of trait and object definitions followed by a single expression. Trait and object definitions may include method definitions. Object definitions may include field declarations, which are shown as value parameters. A trait or object definition may include ordinary type parameters along with hidden type variables, which are given in a where clause. A method definition may also include ordinary type parameters.

A method body consists of a single expression. Valid expressions are variable references, ref-
trait name \( T \)

object name \( O \)

type variable \( X \)

type \( A \) ::= \( O[A] \)  
non-object type \( K \) ::= \( X \)  
trait type \( M \) ::= \( T[A] \)  

method name \( m \)  
parameter \( x \)  
expression \( e \) ::= \( x \)  
| self  
| \( O[A](e') \)  
| \( e.x \)  
| \( e.m[A](e') \)  
typecase \( x = e \) of \( A \Rightarrow e \) else \( e \) end  
| \( e \) as \( A \)  

bound \( bnd \) ::= \( X \) extends \( K \)  
method \( md \) ::= \( m[bnd](x:A):A = e \)  
trait definition \( td \) ::= \( \text{trait } T[bnd](x:A) \text{ extends } \{ M \} \text{ where } \{ bnd \} \ 
object definition \( od \) ::= \( \text{object } O[bnd](x:A) \text{ extends } \{ M \} \text{ where } \{ bnd \} \ 
definition \( d \) ::= \( od \)  
| \( p \) ::= \( d \ e \)  

Figure 7.1: Syntax of HTV0
erences to the special identifier `self`, constructor calls, field accesses, method invocations, type annotations, and `typecase` expressions. `typecase` expressions are type-dependent operations that evaluate the test expression, assigning the result to the specified variable, and then evaluates the first clause whose guarding type is a supertype of the test expression’s type. Note that the static type of the variable in the evaluated clause is the guarding type of that clause. We sometimes parenthesize expressions for clarity.

In HTV0, types, including hidden type variables, are retained at run time and type-dependent operations allow types to influence the run-time semantics of valid programs. For example, to provide the effect of a casting operator (equivalent to casts in the Java programming language), the following method `cast` can be defined:

```plaintext
trait Tree[X] extends Object
  cast[Y]():Y =
  typecase x = self of
    Y ⇒ x
  else ⇒ error()
end
end
```

The `typecase` expression evaluates its test expression `self` and compares the type of the test expression with the types in its clauses from top to bottom.

### 7.1.1 Type System

**Traits and Objects**

The type hierarchy produced by a HTV0 program is a graph in which the only cycles are produced by `self` extensions. That is, the only cycles are created by a trait extending another instantiation of itself. Trait definitions specify the internal nodes of the hierarchy and object definitions specify the leaves of the hierarchy. In other words, only traits can be extended.

For simplicity, abstract methods are not included in HTV0. In examples where it is natural to use abstract methods, we simulate their use by defining a method to recursively call itself with the same arguments. Also for simplicity, recursive bounds are not allowed on type variables.
Multiple Extension

Both traits and objects may extend multiple traits; they inherit the methods provided by the extended traits. A trait or object provides a method if it contains a definition for the method or it inherits a definition. For simplicity, we allow a trait or object to inherit only a single method definition for each method name. In other words, method overloading is forbidden. However, a trait or object can override the definition of an inherited method. As a result of this choice, methods are dispatched using a single dispatch semantics. The most specific method definition provided by the dynamic type of the receiver is invoked at run time. Note that Fortress allows method overloading and uses a multiple dispatch semantics.

7.2 HTV1

The syntax of HTV1 is identical to the syntax of HTV0 with the exception of the definition of bounds. HTV1 allows lower bounds to be assigned to type variables. Figure 7.2 shows the difference between the syntax of HTV1 and HTV0.

Lower bounds are introduced in HTV1 to provide an alternate semantics for hidden type variables, while maintaining the same expressive power as HTV0. Recall from Section 3.3 that hidden type variables in methods are interpreted as defining infinitely many methods. To make this interpretation more explicit, methods containing hidden type variables can be rewritten with method type parameters replacing the hidden type variables. For example, the $\text{cons}$ method of a covariant list can be defined using hidden type variables as:

```
trait List[X extends Y] extends List[Y] where {Y extends Object}
cons(y : Y) : List[Y] = Cons[Y](y, self)
end
```
... trait definition

\[
\text{td} ::= \text{trait } T[bnd] \text{ extends } \{ M \text{ where } \{ bnd \} \} \text{ where } \{ bnd \} \rightarrow \text{md end}
\]

object definition

\[
\text{od} ::= \text{object } O[bnd](x:A) \text{ extends } \{ M \text{ where } \{ bnd \} \} \text{ where } \{ bnd \} \rightarrow \text{md end}
\]

... Figure 7.3: Difference between the Syntax of CE and HTV1

Alternatively, using lower bounds, it can be defined as:

\[
\text{trait } \text{List}[X \text{ extends } Y] \text{ extends } \text{List}[Y] \text{ where } \{ Y \text{ extends } \text{Object} \}
\]

\[
\text{cons}[Z \text{ bounds } X](z:Z):\text{List}[Z] = \text{Cons}[Z](z,\text{self})
\]

each

Here, \( Z \) can be substituted with any supertype of \( X \). Notice that both traits define an infinite number of \textit{cons} methods. And, semantically, both method definitions have the same type:

\[
\forall X.\forall Y.((X <: Y) \land (Y <: \text{Object})) \implies (Y \rightarrow \text{List}[Y])
\]

HTV1 prevents hidden type variables from occurring in methods. This prevents the programmer from having to reason about hidden type variables in methods. Programmers who have not encountered hidden type variables before may find this formulation of hidden type variables easier to understand. However, programs that contain hidden type variables in methods (in HTV0) must be rewritten with method type parameters (in HTV1). For this reason, programmers who are familiar with hidden type variables may prefer the semantics of HTV0.

### 7.3 CE

The syntax of CE extends that of HTV1. In addition to lower bounds on type variables, CE allows where clauses on extended types. Figure 7.3 shows the differences between the syntax of CE and HTV1.
7.4 Intermediate Languages

The formal semantics of HTV0, HTV1 and CE require method calls to be annotated with instantiations for hidden type variables. We discuss the reason for this in Chapter 8. As a result, the semantics are defined for variants of these languages. In an implementation, these variants serve as intermediate languages, which HTV0, HTV1 and CE are compiled into. The intermediate representation of HTV0 is identical to HTV0 except for the syntax of method calls, which are annotated as follows:

\[ e_{\text{path}}(\overrightarrow{A}, \overrightarrow{A}).m[\overrightarrow{A}]() \]

Similarly, method calls in the intermediate representations of HTV1 and CE are written as:

\[ e_{\text{path}}(\overrightarrow{A}).m[\overrightarrow{A}]() \]

Chapter 8 describes the meaning of these annotations and how they are used. Chapter 9 describes how the annotations are inferred.
Chapter 8

Semantics of Hidden Type Variables

In this chapter, we discuss the semantics of hidden type variables. The major challenge when defining the semantics is finding witnesses for hidden type variables. We discuss the need for witnesses in Section 8.1. In Section 8.2, we describe which witnesses are selected when more than one are available. To make use of witnesses at run time, witnesses are added to a program in the form of annotations. This is discussed in Section 8.3. In Section 8.4, we turn to the formal semantics of HTV0 and HTV1. The formal semantics of HTV1 is used to prove that the type system is sound in Section 8.5. Lastly, restrictions to prevent cycles in the type hierarchy of HTV0 and HTV1 programs are presented in Section 8.6.

8.1 Witnesses for Hidden Type Variables

Unlike ordinary type parameters, it is not possible for the programmer to explicitly instantiate hidden type variables. Therefore, the compiler must find instantiations, or witnesses, for hidden type variables. Finding witnesses for hidden type variables is similar to type inference, where instantiations are found for trait, object, and method type parameters. These instantiations are needed to determine concrete trait, object, and method types. Witnesses, on the other hand, are needed to determine concrete subtyping relationships between types. Concrete subtyping relationships are needed when traversing the type hierarchy (i.e., subtype checking and method dispatch). Because subtype checking and method dispatch occur at both compile time and run time, witnesses must
trait Collection extends Object
    clear() : Collection = self.clear()
end

trait List[X extends Y] extends {List[Y], Collection} where {Y extends Object}
    clear() : List[X] = Empty
    cons(y:Y) : List[Y] = Cons[Y](y, self)
end

object Cons[Z extends Object] (first:Z, rest:List[Z]) extends List[Z] end

object Empty extends List[V] where {V extends Object} end

Figure 8.1: Covariant List Example

be found at both compile time and run time. The following subsections elaborate on the need for witnesses.

8.1.1 Witnesses for Subtype Checking

The occurrence of a hidden type variable in a type in an extends clause indicates that infinitely many types are extended. Therefore, during subtype checking, it is necessary to verify an extension by finding a specific witness for the hidden type variable. For example, Figure 8.1 shows a collection trait, a list trait, and two objects of the list trait. Consider checking whether Empty is a subtype of List[Number]. A witness needs to be found for V. In particular, Number must be a valid witness for V.

8.1.2 Witnesses for Type Checking Method Calls

If a hidden type variable occurs in a type in an extends clause then the hidden type variable may be substituted into the type of an inherited method. In this case, a witness must be found for the hidden type variable in order to determine the concrete type of a call to the method. For example, the type Empty extends List[V], where V is a hidden type variable. Therefore, when type checking the method invocation:

    Empty.clear(),

a witness must be found for V. This witness will determine the concrete type of the method call.
In HTV0, a hidden type variable may occur directly in a method definition. To determine the concrete type of a call to such a method, a witnesses must be found for the hidden type variable. For example, consider the following method call:

```
Empty.cons(5)
```

The `cons` method inherited from type `List[V]` contains a hidden type variable in its definition. Notice that, unlike the previous example, the hidden type variable is defined in the method, rather than entering the method via a type hierarchy traversal. To assign a concrete type to the method call, a witness must be found for the hidden type variable.

### 8.1.3 Witnesses for Dispatching Method Calls

There are programs in which hidden type variables are not statically visible, but appear during dynamic dispatch. In this case, witnesses must be found at run time. For example, consider the following method call on a variable `v` of type `Collection`:

```
v.clear()
```

The static type of this call is `Collection`. However, if `v` reduces to the value `Empty`, we must find a witness for `V` in the definition of `clear` (inherited from `List[V]`) at run time.

### 8.2 Selecting Witnesses for Hidden Type Variables

Witnesses for hidden type variables are automatically inferred by the compiler and run-time system. Substituting any witness that satisfies the bounds on the corresponding hidden type variable will produce a well-typed program. However, in many cases there are several such witnesses. For example, consider the definitions in Figure 8.1 and the following method call:

```
Cons[Z](3, Empty).cons(5)
```

The type of the call depends on the witness that is chosen for the hidden type variable `Y`. Therefore, the type could be `List[Z]`, `List[Number]`, or `List[Object]`. 

The lack of unique witnesses presents a problem for the semantics of hidden type variables. If witnesses are non-deterministically chosen then the programmer will not be able to determine the type of a program. In a language with type-dependent operations, the witnesses for hidden type variables can influence the result of the program. In this case, if witnesses are non-deterministically chosen then the programmer will not be able to determine the result of a program.

To address this issue, HTV0, HTV1 and CE infer most-specific witnesses for hidden type variables. For example, the most-specific witness for the hidden type variable \( Y \) in the method call:

\[
\text{Cons}[Z](3, \text{Empty}).\text{cons}(5)
\]

is \( Z \). Therefore, the type of the call is \( \text{List}[Z] \).

Most-specific witnesses provide the most information about a hidden type variable that is possible. Because a most-specific witness is a subtype of every witness for the hidden type variable, an expression that is assigned a most-specific witness can be used in a context requiring any witness for the hidden type variable whatsoever. In other words, the increase in type information provided by a most-specific witness enlarges the set of contexts in which it is known that it is safe to use an expression whose type is the witness.

### 8.2.1 Unique Most-Specific Witnesses

Most-specific witnesses do not always exist. If a hidden type variable has no lower bounds then a most-specific witness exists only if the programmer defined a most-specific trait or object. In case the programmer did not define such a trait or object, we introduce the type \( \text{Bottom} \), which is a subtype of every other type. For example, the most-specific type of:

\[
\text{Empty}.\text{clear}()
\]

is \( \text{List}[\text{Bottom}] \).

Even with the type \( \text{Bottom} \), most-specific witnesses for HTV0, HTV1 and CE programs do not necessarily exist. Consider the following program:

```
trait Printable extends Object end
trait Serializable extends Object end
```
object PSFile(name: string) extends {Printable, Serializable} end
object PDFFile(name: string) extends {Printable, Serializable} end
trait List[X extends Y] extends {List[Y], Object} where {Y extends Object}
    cons(y: Y): List[Y] = Cons[Y](y, self)
end
object Cons[X](first: X, rest: List[X]) extends List[X] end
object Empty extends List[X] where {X extends Object} end
Cons[PSFile](PSFile("myPSFile"), Empty).cons(PDFFile("myPDFFile"))

Notice that the type of the method call is List[Y], for some instantiation of Y. In fact, Y must be a supertype of both PSFile and PDFFile. However, both Printable and Serializable are supertypes of both PSFile and PDFFile, and neither is more specific than the other. In other words, the least upper bound of PSFile and PDFFile is not unique. This is a consequence of multiple extension.

To ensure that least upper bounds are always unique, we introduce union types into our intermediate languages. In the above example, the witness PSFile ∪ PDFFile would be inferred for Y. Informally, a value of this type either has type PSFile or has type PDFFile. Section 8.4 discusses how the subtype relation for union types is determined.

The types PSFile ∪ PDFFile and PDFFile ∪ PSFile are considered to be the same types. More specifically, we define type A ∪ B and type C to be equivalent if and only if A ∪ B is a subtype of C and C is a subtype of A ∪ B. Equivalence between two non-union types is determined by syntactic equivalence. Using union types and Bottom, we can be assured that the most-specific witness for any hidden type variable is unique up to type equivalence.

8.3 Annotating Programs with Witnesses

Witnesses for method dispatch are added to a program as annotations. The run-time system makes use of the annotations during the execution of the program. Not only does this strategy save the run-time system from re-computing the witnesses, but it is necessary for a sound type system. Figure 8.2 gives an example that helps to illustrate this point.

The type OutputStream[X] represents write-only streams that take data of type X, where X is a contravariant type parameter. The type Process[X] represents processes that have the ability
trait OutputStream[X]
  extends {OutputStream[Y], Object}
  where {Y extends X}
  write(data: X) : () = self.write(data)
end

trait Process[X] extends Object
  getOutputStream() : OutputStream[X] = ...
end

object Win32Process[X extends Y]
  extends Process[Y]
  where {Y extends Object}
end

(Win32Process[Z] as Process[Number]).getOutputStream()

---

Figure 8.2: Path Extension Example

to output data of type $X$. The object Win32Process is an implementation of the Process trait.

Consider the method call in Figure 8.2. Statically, the receiver has type Process[Number]. Therefore, the method call has type OutputStream[Number]. But, at run time the call is reduced to:

Win32Process[Z].getOutputStream()

If at run time a most-specific witnesses is chosen for $Y$ irrespective of its compile-time witness, then at run time the method getOutputStream is inherited from Process[Z]. Therefore, at run time the method call has type OutputStream[Z], which is a supertype of its static type (and thus would be a counter example to the type soundness theorem).

When the program is annotated with witnesses for method dispatch, as is done in the system we present, the method getOutputStream is inherited from Process[Number] at run time. Therefore, the method call has type OutputStream[Number] at run time (thus satisfying the type soundness theorem). The remainder of this section describes the witness annotations for method dispatch.

### 8.3.1 Path Annotations

To annotate a program with the witnesses discussed in Sections 8.1.2 and 8.1.3, we have developed the notion of a path in the type hierarchy. A path is a list of types such that each type is an immediate
subtype of its successor. That is, a path is a step-by-step traversal between two types in the type hierarchy. As an example:

```
Empty List[Number] Collection
```

is a valid path in the type hierarchy created by the declarations in Figure 8.1.

If each method call is annotated with a path from the type of the receiver to the type defining the method, then the run-time semantics can simply follow this path when dispatching to a method. Because each type in the path will be fully instantiated at run time, the path will provide witnesses for hidden type variables. At compile time, a path from the static type of the receiver to the type defining the method is inferred. Therefore, witnesses that are found statically are included in the path at compile time. At run time, the path is extended from the dynamic type of the receiver to the static type of the receiver. So, witnesses for hidden type variables that are visible only at run time will be included in the extended path and utilized in the same way as static witnesses.

For example, in the following method invocation:

```
(Empty as Collection).clear()
```

the receiver expression has type `Collection` (which is a result of upcasting `Empty`). In addition, `Collection` defines the method `clear`. Thus, at compile time, the method invocation is annotated with the single element path:

```
Collection as follows:
```

```
(Empty as Collection) path Collection.clear()
```

At run time, before the method is dispatched, the receiver is fully evaluated:

```
Empty path Collection.clear()
```

Notice that the current path does not begin with the type of the receiver. In particular, the type of the receiver, `Empty`, is a strict subtype of the first (and only) type in the path, `Collection`. For this reason, if the current path is used to dispatch the method then the definition of the method in trait
Collection would be invoked instead of the definition in trait List. Therefore, the path annotation must be extended to the dynamic type of the receiver:

\[(\text{Empty as Collection}) \text{ path Empty List}[\text{Bottom}] \text{ Collection.clear()}\]

At this point, the method can be dispatched. Notice that the type List[Bottom] provides a witness, namely Bottom, for the hidden type variable \(V\).

Chapter 9 describes the procedure for inferring and extending paths. In particular, Section 9.3.2 describes the procedure for path inference and Section 9.2 describes the procedure for extending paths.

### 8.3.2 Multiple Extension and Ambiguous Paths

The languages that are considered in this dissertation allow multiple extension. In such languages, the path from one type to another is not necessarily unique. For the purpose of subtype checking, any valid path is sufficient. However, when inferring path annotations for method calls, different paths can lead to different method types, or even different method definitions. To avoid this situation, HTV0, HTV1 and CE disallow method overloading. If overloading is forbidden then there exists at most one path from a type that inherits a method to the type that defines the method.

### 8.3.3 Additional Witness Annotations

If hidden type variables can occur in method definitions (as in HTV0) then path annotations do not provide all necessary witnesses. For example, given the definition of lists in Figure 8.1, what is the type of the following method call?

\[
\text{Cons}[\text{Z}](3, \text{Empty}) \text{ path Cons}[\text{Z}] \text{ List}[\text{Z}].\text{cons}(5)
\]

According to the definition of List, its type is List[Y], where \(Y\) is a supertype of \(Z\). But, no witness for \(Y\) is given by the path annotation. In order to provide witnesses for these hidden type variables, the compiler annotates method calls with additional witnesses. The above method call becomes:
Cons[Z](3, Empty) path (Cons[Z] List[Z], Z).cons(5)

where the type Z is substituted for the hidden type variable of the trait defining cons (i.e., Y).

Unlike path annotations, the additional witness annotations are not extended at run time. These annotations are inferred at compile time and remain fixed at run time. To ensure that the additional witness annotations are sufficient for an overriding method when the method call is dynamically dispatched, restrictions must be placed on overriding methods in HTV0.

For example, consider the consequences of allowing the following program to pass type checking:

```latex
trait List[X] extends Object
  append(l: List[X]): List[X] = self.append(l)
end
object Cons[Y](first : Y, rest : List[Y]) extends List[Y]
  append(l: List[Y]): List[Y] = Cons[Y](first, rest.append(l))
end
object Empty extends List[Z] where {Z extends Object}
  append(l: List[Z]): List[Z] = l
end
(Empty as List[Number]) path (List[Number], ·).append(Cons[Number](5, Empty))
```

The method call requires no additional witness annotations at compile time (as shown by the empty sequence ·) because List[Number] defines no hidden type variables. However, at run time the append method of Empty requires a witness for the hidden type variable Z. Since no witness is given by the program syntax, the program will get stuck.\footnote{The astute reader will notice that an appropriate witness for Z, namely Number, can be found in the path annotation. This follows from the fact that Z occurs in an extended type of Empty. In general, there is no guarantee that hidden type variables will occur in extended types. Therefore, it is not possible, in general, to extract the additional witness annotation from the path annotation.}

To explain the overriding restrictions in HTV0, we define the following two terms. We call a trait or object defining an overriding method an \textit{overriding trait}. Similarly, we call a trait defining an overridden method an \textit{overridden trait}.

The overriding restrictions of HTV0 require overriding traits to define the same number of hidden type variables as the overridden trait. In addition, \textit{any} types meeting the bounds on the
hidden type variables of the overridden trait must also meet the bounds on the hidden type variables of the overriding trait. These restrictions are formalized in the following section.

### 8.4 Formal Semantics

This section describes the semantics of hidden type variables in HTV0 and HTV1. The formal semantics are defined in terms of the intermediate languages for HTV0 and HTV1. Chapter 9 describes how HTV0 and HTV1 programs are translated into their intermediate representations.

#### 8.4.1 Notational Conventions

As before, we use $\overline{A}$ as short-hand notation for $A_1 \ldots A_n$. We write the empty sequence as $\cdot$. We denote the concatenation of two sequences by juxtaposition, such as $\overline{A} \overline{B}$. The notation $|\overline{A}|$ is used to denote the size of the sequence $\overline{A}$. We extend this notation to other entities beyond types.

We use $\_\_\_$ to denote some parts of the syntax which do not have key roles in a rule. We assume that $\_\_\_$ matches all sequences of characters, including the empty sequence.

For the most part, the rules in the formal semantics abide by the follow conventions:

- Subscripts are used to distinguishing between different elements of a vector.
- Different alphabetic characters are used to distinguishing between otherwise unrelated entities of the same kind in the same rule.
- Primes are used to distinguishing between corresponding entities before and after a transformation of some sort.

However, occasionally we will use primes and subscripts to distinguish between entities of the same kind.

#### 8.4.2 HTV0

**Syntax**

The syntax of the intermediate language for HTV0 is given in Figure 8.3. This syntax is mostly identical to the syntax of HTV0 given in Figure 7.1. The only differences are the definitions of
trait name $T$
object name $O$
trait or object name $S::=T|O$
type variable $X,Y,Z$
type $A,B,C,$
$D,E,F,$
$G,H,I,$
$J::=O[A]|K|\text{Bottom}|A\cup A$
non-object type $K,L::=X|M$
trait type $M,N::=T[A]|\text{Object}$
method name $m$
parameter $x,y::=x$
expression $e,f,g,h::=x$
bound $bnd::=X\text{ extends }K$
method $md::=m[bnd]\langle x:A \rangle:A=e$
trait definition $td::=\text{trait } T[bnd]\langle x:A \rangle \text{ extends } \{ \overrightarrow{M} \} \text{ where } \{ \overrightarrow{bnd} \} \overrightarrow{md} \text{ end}$
object definition $od::=\text{object } O[bnd]\langle x:A \rangle \text{ extends } \{ \overrightarrow{M} \} \text{ where } \{ \overrightarrow{bnd} \} \overrightarrow{md} \text{ end}$
definition $d::=od$
program $p::=d\ e$

Figure 8.3: Syntax of Intermediate Language for HTV0
Paths: $p; \Delta \vdash A <:: A$

\[
\begin{align*}
\text{[P-OBJ]} & \quad p; \Delta \vdash A <:: \text{Object} \\
\text{[P-BOT]} & \quad p; \Delta \vdash \text{Bottom} <:: A \\
\text{[P-VAR]} & \quad \frac{(X <: K) \in \Delta}{p; \Delta \vdash X <:: K} \\
\text{[P-EXT]} & \quad \frac{\begin{array}{c}
\text{S[}\overline{X}\text{ extends }K]\text{ extends }\{\overline{M}\}\text{ where }\{\overline{Y}\text{ extends }L\}\text{ end }\in p \\
1 \leq i \leq |\overline{M}| \\
p; \Delta \vdash \overline{A} \text{ ok} \\
p; \Delta \vdash \overline{A} <:: \begin{cases}
\overline{A}/\overline{X}/\overline{B}/\overline{Y}/K
\end{cases}
\end{array}}{p; \Delta \vdash S\bigl[\overline{A}\bigr] <:: \begin{cases}
\overline{A}/\overline{X}/\overline{B}/\overline{Y}/M_i
\end{cases}} \\
\text{[P-UNIONSUB]} & \quad \frac{p; \Delta \vdash A <:: C}{p; \Delta \vdash B <:: C} \\
\text{[P-UNIONSUPER]} & \quad \frac{A \in \{B, C\}}{p; \Delta \vdash A <:: B \cup C}
\end{align*}
\]

Figure 8.4: Path Rules for HTV0

The syntax of types is augmented to include union types and Bottom. The syntax of expressions requires witness annotations on method calls.

The grammar of Figure 8.3 defines five terminal symbols: trait names, object names, type variables, method names, and parameters. Each terminal ranges over an infinite set of names. For the purposes of the formal semantics, we assume that each concrete instance of a terminal symbol in a program is unique.

**Static Semantics: Type Relations**

The rules to judge valid paths are given in Figure 8.4. The relation $<::$ is used to show that two types form a path, whereas $<:$ indicates a subtype relation between two types. We describe subtyping shortly. We use $\Delta$ as a metavariable for type variable environments. A type variable environment is a set of bounds on type variables. A bound (written $X <: K$) assigns a supertype to a type variable. A substitution, which replaces $\overline{X}$ with $\overline{A}$, is written $[\overline{A}/\overline{X}]$. We use $td \in p$ and $od \in p$ to denote a class-table lookup.

Recall that paths are used to dispatch method calls. To serve this purpose, a path must provide
Subtyping: \[ p; \Delta \vdash A \ll B \]

\begin{align*}
[S\text{-}REFL] & & p; \Delta \vdash A \ll A \\
[S\text{-}TRANS] & & p; \Delta \vdash A \ll B \\
 & & p; \Delta \vdash B \ll C \\
 & & p; \Delta \vdash A \ll C \\
[S\text{-}PATH] & & p; \Delta \vdash A \ll B \\
 & & p; \Delta \vdash A \ll B
\end{align*}

Figure 8.5: Subtyping Rules for HTV0

Well-formed paths: \[ p; \Delta \vdash A \text{ path-ok} \]

\begin{align*}
[S\text{-}SINGLE]\text{-PATH} & & p; \Delta \vdash A \text{ path-ok} \\
[M\text{-}MULTIPLE]\text{-PATH} & & |A| = n \\
 & & 1 < n \\
 & & p; \Delta \vdash A_1 \ll A_2 \cdots A_{n-1} \ll A_n \\
 & & p; \Delta \vdash A \text{ path-ok}
\end{align*}

Figure 8.6: Valid Paths in HTV0

a step-by-step traversal of the type hierarchy. The path rules determine when any two types form a path. We will make use of this relation to judge the well-formedness of longer paths shortly.

For the most part, the path rules are straightforward. Any type followed by \texttt{Object} is a valid path. Similarly, \texttt{Bottom} followed by any type is a valid path. A type variable followed by its bound is a valid path. A trait or object type followed by a type that it extends is also a valid path. A union type followed by a single type that is a supertype of each element of the union is a valid path. Lastly, any type followed by a union type, which includes the preceding type, is a valid path.

Figure 8.5 shows the subtyping rules of HTV0. The relation \ll is the reflexive transitive closure of the relation \ll. Notice that the subtyping rules and the path rules are mutually recursive (i.e., both reference the other). Also notice that the path rules rely on the type-well-formedness rules of Figure 8.7.

Valid paths of an arbitrary size are determined by applying the path rules to each contiguous pair in the path. This idea is formalized in Figure 8.6.
Well-formed types: \( p; \Delta \vdash A \text{ ok} \)

[W-OBJ] \( p; \Delta \vdash \text{Object ok} \)  
[W-BOT] \( p; \Delta \vdash \text{Bottom ok} \)

[W-VAR] \( \frac{X \in tv(\Delta)}{p; \Delta \vdash X \text{ ok}} \)

[W-UNION] \( \frac{p; \Delta \vdash A \text{ ok}}{p; \Delta \vdash A \cup B \text{ ok}} \)

\[ S \llbracket X \text{ extends } K \rrbracket \text{ where } \{ Y \text{ extends } L \} \text{ end } \in p \]

\[ p; \Delta \vdash A \text{ ok} \]
\[ p; \Delta \vdash B \text{ ok} \]
\[ p; \Delta \vdash A < : \llbracket A \llbracket X \rrbracket \llbracket B \llbracket Y \rrbracket K \]
\[ p; \Delta \vdash B < : \llbracket A \llbracket X \rrbracket \llbracket B \llbracket Y \rrbracket L \]

\( p; \Delta \vdash S \llbracket A \rrbracket \text{ ok} \)

Figure 8.7: Type Well-Formedness Rules for HTV0

Well-formed types are determined by the rules in Figure 8.7. We define \( tv(\Delta) \) to be the type variables bound by \( \Delta \). For example, \( tv(\{ X <: K, L <: Y \}) = \{ X, Y \} \). The types Object and Bottom are well formed. Type variables are well formed if they are bound in the type environment. A union type is well formed if each of its component types is well formed. Lastly, a trait or object type is well formed if the type arguments satisfy the bounds on the corresponding type variables and there exist witnesses for all hidden type variables.

**Static Semantics: Expression Typing**

Figure 8.8 shows the auxiliary function that dispatches method calls. We define \( name(md) \) to be the name of method \( md \). The function is passed the name of the method, a path, and additional witness annotations. The function returns the first method definition on the path, as well as the type variables (and bounds) of the enclosing trait or object. Type arguments and the additional witness annotations are substituted into the result of the function. If the path is empty (i.e., the method definition is not found on the path) then the empty set is returned.

Figure 8.9 shows the expression typing rules for HTV0. We use \( \Gamma \) as a metavariable for parameter environments (i.e., mappings from parameters to types). Parameters and the reserved word self are assigned the types that are provided by the parameter environment. A constructor call is
Method dispatch:  
\[
\text{method}_p(m, \overrightarrow{A}, \overrightarrow{A}) = \{(md_1, \overrightarrow{A} : : A)\}
\]

\[
\text{method}_p(m, \overrightarrow{A}, \overrightarrow{B}) =
\]

\[
\emptyset \quad \text{if } \overrightarrow{A} = \cdot
\]

\[
\{(\overrightarrow{C} / \overrightarrow{X})[\overrightarrow{B} / \overrightarrow{Y}]md, (\overrightarrow{C} / \overrightarrow{X})[\overrightarrow{B} / \overrightarrow{Y}]\Delta\}
\quad \text{if } \overrightarrow{A} = S[\overrightarrow{C}] \overrightarrow{D}
\]

\[
\quad \quad \text{and } S[X\text{ extends }K] \quad \text{where } \{Y\text{ extends }L\} \overrightarrow{md} \text{ end } \in p
\]

\[
\quad \quad \text{and } md \in \{md\}
\]

\[
\quad \quad \text{and } \text{name}(md) = m
\]

\[
\quad \quad \text{and } \Delta = \overrightarrow{X} : : \overrightarrow{K} \quad \overrightarrow{Y} : : \overrightarrow{L}
\]

\[
\text{method}_p(m, \overrightarrow{D}, \overrightarrow{B})
\quad \text{otherwise } \overrightarrow{A} = C \overrightarrow{D}
\]

Figure 8.8: Method Dispatch for HTV0

assigned the corresponding object type, if the type arguments meet the necessary bounds and the value arguments have subtypes of the constructor’s corresponding parameter types. A field access is assigned the type of the field, which is found by looking up the type of the receiver.

A method call is assigned the return type of the method definition, with type arguments substituted. The receiver’s type must be the initial type in the path annotation. Both witness annotations must be well formed. The auxiliary function \textit{method}_p returns the first method definition on the path and the type variables (and their bounds) of the enclosing trait or object. The type variables and bounds are substituted with type arguments and witnesses. To guarantee that the witnesses are valid, the type variables and bounds must be in the subtype relation. Lastly, the type and value arguments must meet the required bounds.

The type assigned to a typecase expression is a supertype of the type of each clause. Each clause is typed under the assumption that the variable assigned to the test expression has the guarding type. The else clause assumes this variable has the type of the test expression.

A type ascription is assigned the ascribed type, assuming that the underlying expression is as-
Expression typing: $p; \Delta; \Gamma \vdash e : A$

**[T-VAR]** $\quad p; \Delta; \Gamma \vdash x : \Gamma(x)$

**[T-SELF]** $\quad p; \Delta; \Gamma \vdash \text{self} : \Gamma(\text{self})$

**[T-OBJECT]**
- object $O[X] \quad \text{end} \in p$
- $p; \Delta \vdash O[A] \quad \text{ok}$
- $p; \Delta; \Gamma \vdash c : C$
- $p; \Delta \vdash C \prec \overline{[A/X]B}$
- $p; \Delta; \Gamma \vdash O[A][c] : O[A]$

**[T-FIELD]**
- object $O[X] \quad \text{end} \in p$
- $1 \leq i \leq |X|$
- $p; \Delta; \Gamma \vdash e : O[A]$
- $p; \Delta; \Gamma \vdash e.x_i : \overline{[A/X]B_i}$

**[T-METHOD]**
- $method_p(m, AB, C) = \{ (m[X \text{ extends } K][x : E] : F = \overline{G} \prec \overline{H}) \}$
- $p; \Delta; \Gamma \vdash f : I$
- $p; \Delta \vdash I \prec \overline{[D/X]E}$
- $p; \Delta \vdash D \quad \text{ok}$
- $p; \Delta \vdash \overline{D} \prec \overline{[D/X]K}$
- $p; \Delta \vdash \overline{G} \prec \overline{H}$

**[T-METHOD]**
- $p; \Delta; \Gamma \vdash e \quad \text{path} \ (AB, C).m[D](f) : \overline{[D/X]F}$

**[T-TYPECASE]**
- $p; \Delta; \Gamma \vdash e : C$
- $p; \Delta; \Gamma \vdash f_i : D_i \quad p; \Delta \vdash D_i \prec B$
- $1 \leq i \leq |A|$
- $p; \Delta; \Gamma \vdash g : E \quad p; \Delta \vdash E \prec B$
- $p; \Delta; \Gamma \vdash \text{typecase } x = e \text{ of } A \Rightarrow f \text{ else } g \text{ end : } B$

**[T-ASCRIPTION]**
- $p; \Delta; \Gamma \vdash e : B$
- $p; \Delta \vdash B \prec A$
- $p; \Delta; \Gamma \vdash e \quad \text{as } A : A$

Figure 8.9: Expression Typing Rules for HTV0
Program typing: \[ \vdash p : A \]

\[
\begin{array}{c}
{T\text{-}PROGRAM} \quad p = \overrightarrow{d} e \quad p \vdash \overrightarrow{d} \text{ ok } p; \emptyset; \emptyset \vdash e : A \\
\hline
\vdash p : A
\end{array}
\]

Figure 8.10: Program Typing Rule for HTV0

Inherited methods: \[ \text{inherited}_p(M) = \{ \overrightarrow{m} \} \]

\[
inherited_p(M) = \\
\begin{cases}
\emptyset & \text{if } M = \text{Object} \\
\{ \overrightarrow{m} \} \cup \inherited_p(M_1) \cup \ldots \cup \inherited_p(M_n) & \text{if } M = T[\overrightarrow{A}] \\
& \text{and } \text{trait } T \text{ extends } \{ \overrightarrow{M} \} \_ \overrightarrow{md} \text{ end } \in p \\
& \text{and } \overrightarrow{m} = \text{name}(md) \\
& \text{and } |\overrightarrow{M}| = n
\end{cases}
\]

Figure 8.11: Inherited Methods for HTV0

signed a subtype of the ascribed type.

**Static Semantics: Definition Typing**

Programs are typed according to the rule given in Figure 8.10. Trait and object definitions must be well formed. The expression must be well typed under empty environments. The type of the expression is also the type of the program.

Figure 8.11 defines an auxiliary function that is used to calculate the names of inherited methods. Figure 8.12 gives the trait and object definition typing rules for HTV0. A trait or object must not inherit more than one method definition with the same name. All types in the definition must be well formed. In the case of object definitions, the types of constructor parameters must not contain hidden type variables. Method definitions must be well formed under environments that include all type variables (including hidden type variables) and the reserved word `self`.
Definition typing: \( p \vdash d \ ok \)

\[\text{T-TraitDef}\]

\[\Delta = \overrightarrow{X} <: \overrightarrow{K} \overrightarrow{Y} <: \overrightarrow{L}\]

\[\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) = \emptyset \quad i \neq j \quad i, j \in \{1, \ldots, |\overrightarrow{M}|\}\]

\[p; \Delta \vdash \overrightarrow{K} ok \quad p; \Delta \vdash \overrightarrow{M} ok \quad p; \Delta \vdash \overrightarrow{L} ok\]

\[p; \Delta; \text{self} : T[\overrightarrow{X}]; T[\overrightarrow{X}]\[\overrightarrow{Y}] \vdash \text{md ok}\]

\( p \vdash \text{trait } T[\overrightarrow{X} \text{ extends } \overrightarrow{K}] \text{ extends } \{\overrightarrow{M}\} \text{ where } \{\overrightarrow{Y} \text{ extends } \overrightarrow{L}\} \text{ md end ok}\)

\[\text{T-ObjectDef}\]

\[\Delta = \overrightarrow{X} <: \overrightarrow{K} \overrightarrow{Y} <: \overrightarrow{L}\]

\[\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) = \emptyset \quad i \neq j \quad i, j \in \{1, \ldots, |\overrightarrow{M}|\}\]

\[p; \Delta \vdash \overrightarrow{K} ok \quad p; \Delta \vdash \overrightarrow{A} ok\]

\[p; \Delta \vdash \overrightarrow{M} ok \quad p; \Delta \vdash \overrightarrow{L} ok\]

\[p; \Delta; \text{self} : O[\overrightarrow{X}] x : A; O[\overrightarrow{X}]\[\overrightarrow{Y}] \vdash \text{md ok}\]

\( p \vdash \text{object } O[\overrightarrow{X} \text{ extends } \overrightarrow{K}](x; A) \text{ extends } \{\overrightarrow{M}\} \text{ where } \{\overrightarrow{Y} \text{ extends } \overrightarrow{L}\} \text{ md end ok}\)

Figure 8.12: Trait and Object Definition Typing Rules for HTV0

Method typing: \( p; \Delta; \Gamma; S[\overrightarrow{X}]\[\overrightarrow{X}] \vdash \text{md ok}\)

\[\Delta' = \Delta \quad \overrightarrow{X} <: \overrightarrow{K}\]

\[p; \Delta; S[\overrightarrow{Y}]\[\overrightarrow{Z}] \vdash \text{override}(m[\overrightarrow{X} \text{ extends } \overrightarrow{K}](x : A); B = e)\]

\[p; \Delta' \vdash \overrightarrow{K} ok \quad p; \Delta' \vdash \overrightarrow{A} ok \quad p; \Delta' \vdash B ok\]

\[p; \Delta'; \Gamma x : A \vdash e : C \quad p; \Delta' \vdash C <: B\]

\[p; \Delta; \Gamma; S[\overrightarrow{Y}]\[\overrightarrow{Z}] \vdash m[\overrightarrow{X} \text{ extends } \overrightarrow{K}](x : A); B = e \ ok\]

Figure 8.13: Method Typing Rule for HTV0
Method overriding: \[ p;\Delta;S[\overrightarrow{X}] \vdash \text{override}(md) \]

\[
\begin{align*}
\text{method}_p(m, \overrightarrow{C}, \overrightarrow{W}) &= \{(m[\overrightarrow{X} \text{ extends } K'](\overrightarrow{x}:A')\vdash B' = \overrightarrow{D'} <: \overrightarrow{E'})\} \\
\Delta &= \overrightarrow{D} <: \overrightarrow{E} & \Delta' &= \Delta \quad \overrightarrow{D'} <: \overrightarrow{E'} \\
p;\Delta \vdash [\overrightarrow{X} / \overrightarrow{X}'] \overrightarrow{A'} <: \overrightarrow{A} & \quad p;\Delta \vdash [\overrightarrow{X} / \overrightarrow{X}'] \overrightarrow{B'} <: [\overrightarrow{W} / \overrightarrow{Z}] \overrightarrow{D} <: [\overrightarrow{W} / \overrightarrow{Z}] \overrightarrow{E} \\
p;\Delta;S[\overrightarrow{Y}] [\overrightarrow{Z}] \vdash \text{override}(m[\overrightarrow{X} \text{ extends } K'](x : \overrightarrow{A})\vdash B = e)
\end{align*}
\]

\[
\begin{align*}
\text{method}_p(m, \overrightarrow{C}, \overrightarrow{W}) &= \emptyset \\
p;\Delta;S[\overrightarrow{Y}] [\overrightarrow{Z}] \vdash \text{override}(m[\overrightarrow{bnd}](x : \overrightarrow{A})\vdash B = e)
\end{align*}
\]

Figure 8.14: Overriding Rules for HTV0

Figure 8.13 shows the method typing rules for HTV0. All types in the definition must be well formed. In addition, the method must be a valid overriding, which we define next. The type of the method body must be a subtype of the return type of the method.

The rules to judge valid overriding for HTV0 are given in Figure 8.14. We write “\(\overrightarrow{X} \text{ fresh}\)” to denote that \(\overrightarrow{X}\) is a sequence of fresh type variables. A valid overriding is a method definition such that for every path starting from the trait or object enclosing the method, either no methods of the same name are found on the path, or the first method found satisfies the following criteria:

1. bounds on type parameters are equal,
2. parameter types are contravariantly related,
3. the return types are covariantly related, and
4. the hidden type variables of the overridden trait must be appropriately related to the hidden type variables of the overriding trait (see below for more detail).

These criteria are verified after renaming method type parameters and hidden type variables. The renaming replaces the method type parameters of the overridden trait with the method type parameters of the overriding trait. The hidden type variables of the overriding trait are replaced with the
hidden type variables of the overridden trait. This allows the type variables to be compared to one another.

The fourth criterion ensures that additional witness annotations, which are guaranteed to be valid witnesses at compile time, continue to be valid witnesses when the method call is dynamically dispatched. More specifically, this criterion requires any type that satisfies the bounds on the hidden type variables of the overridden trait to also satisfy the bounds on the hidden type variables of the overriding trait.

The fourth criterion is formalized by the last subtyping judgment in rule \[\text{OVERRIDE}\]. In this judgment, \(\overrightarrow{D}\) and \(\overrightarrow{E}\) are the type variables and their bounds (respectively) of the overriding trait. Notice that the fresh type variables \(\overrightarrow{W}\) are provided as the third argument of the call to \(\text{method}_p\). This means that \(\text{method}_p\) will use the type variables \(\overrightarrow{W}\) as witnesses for the hidden type variables of the overridden trait. Therefore, the type variables \(\overrightarrow{W}\) “act” as the hidden type variables of the overridden trait in this rule. The substitution \([\overrightarrow{W}/\overrightarrow{Z}]\) replaces the hidden type variables of the overriding trait \(\overrightarrow{Z}\) with type variables \(\overrightarrow{W}\). Therefore, this judgment asserts that the hidden type variables of the overridden trait (i.e., \(\overrightarrow{W}\)) satisfy the bounds on the hidden type variables of the overriding trait (i.e., \(\overrightarrow{Z}\)).

As a side note, consider that valid variance relations “fall out” of the overriding rules. In other words, the methods of covariant or contravariant types are judged in the same way as any other methods. Special treatment for these types is unnecessary.

**Dynamic Semantics: Path Extension**

Recall that path annotations are extended during the evaluation of a program. This ensures that a path always begins with the dynamic type of the method call’s receiver. In many programs, a path can be extended in more than one way. For example, given the type hierarchy created by the following definitions:

```plaintext
trait A end
trait B extends A end
trait C extends A end
trait D extends C end
```
Path comparison: \[
\vec{A} \prec \vec{A}
\]

\[
\vec{A} \prec \vec{B}
\]

if \( p; \emptyset \vdash \vec{A} \) path-ok
and \( p; \emptyset \vdash \vec{B} \) path-ok
and either \( |\vec{A}| < |\vec{B}| \)

or \( |\vec{A}| = |\vec{B}| \) and \( p; \emptyset \vdash \vec{E} <: \vec{F} \)

where \( A_i = S[\vec{C}] \) and \( B_i = S[\vec{D}] \)
and \( 1 \leq i \leq |\vec{A}| - 1 \)

and \( S[\text{bind}_1] \) extends \( \{ \text{bind} \} \) where \( \{ \text{bind}_2 \} \) end \( \in p \)
and \( \text{bind}_1 = X \) and \( \text{bind}_2 = Y \)
and \( \vec{C}/X][\vec{E}/Y]M_j = A_{i+1} \) and \( \vec{D}/X][\vec{F}/Y]M_j = B_{i+1} \)
and \( 1 \leq j \leq |\vec{M}| \)

Figure 8.15: Path Comparison Relation for HTV0

trait \( E \) extends \( \{ B, D \} \) end

there are two ways to extend the one element path \( A \) to begin with type \( E \). In addition, there may be several witnesses for a hidden type variable in a path extension. For example, given the type hierarchy created by the following definitions:

\[
\begin{align*}
\text{trait } & A \text{ end} \\
\text{trait } & B[X] \text{ extends } A \text{ end} \\
\text{trait } & C \text{ extends } B[Y] \text{ where } \{ Y \text{ extends } \text{Object} \} \text{ end}
\end{align*}
\]

there are many ways to extend the one element path \( A \) to begin with type \( C \). In fact, each distinct witness for \( Y \) yields a different path.

The dynamic semantics of HTV0 selects minimal paths when performing path extension. Figure 8.15 defines the path comparison relation that is used to determine minimal paths. The relation \( \vec{A} \prec \vec{B} \) holds if one of two conditions holds. The first condition requires the length of path \( \vec{A} \) to be shorter than the length of path \( \vec{B} \). The second condition requires the two paths to differ only in the witnesses for hidden type variables. In other words, the lengths of the two paths must be equal and each element \( A_i \) of \( \vec{A} \) must have the same outer-most constructor as \( B_i \). The second condition also requires every witness for a hidden type variable in \( \vec{A} \) to be a subtype of the corresponding
Path extension: \( \min\text{-path}_p(\overrightarrow{A}) = \overrightarrow{A} \)

\[
\min\text{-path}_p(\overrightarrow{A}) = A_1 \overrightarrow{B} A_2 \overrightarrow{B} A_3 \ldots A_{n-1} \overrightarrow{B} A_n
\]

such that \( n = |\overrightarrow{A}| \)

and \( p; \emptyset \vdash A_i \overrightarrow{B} A_{i+1} \) path-ok

and \( 1 \leq i \leq |\overrightarrow{A}| - 1 \)

and if \( p; \emptyset \vdash A_i \overrightarrow{C} A_{i+1} \) path-ok then \( \overrightarrow{B} \prec \overrightarrow{C} \)

Figure 8.16: Path Extension for HTV0

witness in \( \overrightarrow{B} \).

Figure 8.16 gives the specification of a function that performs path extension. Note that this definition is not an algorithm; it is non-constructive in form. We discuss an implementation of this function in the next chapter. This function is passed a sequence of types, which are not necessarily a path. The function returns a \( \prec \)-minimal path that contains each of the input types. More specifically, for each contiguous pair of types in the input sequence, the path extension function computes a \( \prec \)-minimal path between the two types. The result of the function is constructed by adjoining each of these paths.

Notice that for some type hierarchies there may not exists a \( \prec \)-minimal path between two types. For example, there is no \( \prec \)-minimal path between types \( A \) and \( D \) in the type hierarchy produced by the following definitions:

```
trait A end
trait B extends A end
trait C extends A end
trait D extends \{B, C\} end
```

If a \( \prec \)-minimal path does not exist then the path extension function is undefined. However, within the formal semantics of HTV0 the path extension function is invoked only on types for which it is defined. To see why, notice that all paths in the semantics are between two types that provide the same method name. We call these inheritance paths. Consider the set of inheritance paths between two fixed types. Each inheritance path in this set must pass through the same sequence of trait and object definitions. This follows from the restriction that each trait and object definition can
inherit at most one method definition of a given name. Therefore, the only differences between two inheritance paths in the set are the witnesses substituted for hidden type variables. As discussed earlier, Bottom and union types guarantee the existence of most-specific witnesses for all hidden type variables. Therefore, there must exist a ≺-minimal path in the set of inheritance paths.

The path extension function is invoked by the path update function. The path update function, which is shown in Figure 8.17, extends all the paths in an expression by recursively processing its subexpressions. When a method call is found, the type of the receiver and the existing path are passed to the path extension function. The resulting path becomes the new path annotation.

**Dynamic Semantics: Evaluation Rules**

The evaluation rules of HTV0 are defined in terms of evaluation contexts and redexes. Figure 8.18 defines the values, evaluation contexts and redexes of HTV0. The only values in HTV0 are fully evaluated constructor calls. The evaluation contexts impose a left-to-right call-by-value reduction order. We use the notation $EC[e]$ to denote the substitution of expression $e$ for the hole $□$ of the evaluation context $EC$.

The evaluation rules of HTV0 are defined in terms of three reduction relations given in Figures 8.19, 8.20 and 8.21. An expression is evaluated by first applying rule [R-EVAL]. This rule applies the path update function to the result of the context evaluation rule. The context evaluation rule partitions an expression into an evaluation context and a redex. The result of applying a redex evaluation rule to the redex is substituted into the hole of the evaluation context.

The redex evaluation rules are defined as follows. The result of evaluating a field access is the corresponding constructor argument. A method call is evaluated by replacing it with the method’s body and applying the necessary substitutions. The evaluation of a typecase expression is the first clause whose guarding type is a supertype of the test expression’s type. If no such guarding type exists then the result is the else branch. In each case, the test expression is substituted for the variable it is bound to. Lastly, a type ascription is evaluated by removing the type annotation.
Path update: \[ \text{path}_p(e) = e \]

\[
\begin{align*}
\text{path}_p(e) &= \\
&= f'.x \\
&= \begin{cases} 
  f'.x & \text{if } e = f.x \\
  &\text{and } f' = \text{path}_p(f) \\
  f' \text{ path } (E, B).m\{C\}(g') & \text{if } e = f \text{ path } (A, B).m\{C\}(g) \\
  &\text{and } f' = \text{path}_p(f) \\
  &\text{and } p; \emptyset; \emptyset \vdash f' : D \\
  &\text{and } E = \text{min-path}_p(DA) \\
  &\text{and } g' = \text{path}_p(g) \\
\end{cases}
\end{align*}
\]

\[
\text{typecase } x = f' \text{ of } A \Rightarrow g' \text{ else } h' \text{ end} \\
\begin{cases} 
  \text{if } e = \text{typecase } x = f \text{ of } A \Rightarrow g \text{ else } h \text{ end} \\
  &\text{and } f' = \text{path}_p(f) \\
  &\text{and } g' = \text{path}_p(g) \\
  &\text{and } h' = \text{path}_p(h) \\
\end{cases}
\]

\[
\begin{align*}
\text{f' as A} \\
&= \begin{cases} 
  f' & \text{if } e = f \text{ as A} \\
  &\text{and } f' = \text{path}_p(f) \\
  \text{O}[A](f') & \text{if } e = \text{O}[A](f') \\
  &\text{and } f' = \text{path}_p(f) \\
\end{cases}
\end{align*}
\]

\[
e \text{ otherwise}
\]

Figure 8.17: Path Update for HTV0
value \( v, w \) ::= \( O[\bar{A}]() \)

evaluation context \( EC \) ::= □

\n\n| \( O[\bar{A}]() EC \) |
| \( EC .x \) |
| \( v path (\bar{A}, \bar{A}).m[\bar{A}]() \) |
| \( \text{typecase } x = EC \text{ of } A \Rightarrow e \text{ else } \Rightarrow e \text{ end} \) |

redex \( R \) ::= \( v.x \)

\n\n| \( v path (\bar{A}, \bar{A}).m[\bar{A}]() \) |
| \( \text{typecase } x = v \text{ of } A \Rightarrow e \text{ else } \Rightarrow e \text{ end} \) |
| \( e \text{ as } A \) |

Figure 8.18: Values, Evaluation Contexts and Redexes for HTV0

Redex Evaluation rules: \( p \vdash R \leadsto e \)

[R-FIELD]

\[ \text{object } O (x: ) \text{ end } \in p \]
\[ 1 \leq i \leq |x| \]
\[ p \vdash O[\bar{A}]().x \leadsto v_i \]

[R-METHOD]

\[ \text{method}_p(m, \bar{B}, \bar{C}) = \{(m[\bar{X}]/\bar{y} = e, )\} \]
\[ p \vdash O[\bar{A}]() \text{ path } (\bar{B}, \bar{C}).m[\bar{D}]() \leadsto [\bar{v} / \bar{x}] |O[\bar{A}]()/\bar{v})/\bar{x} / D / X | e \]

[R-TYPECASE]

\[ p; \emptyset \vdash O[\bar{A}] <: B_i \quad 1 \leq i \leq |B| \]
\[ p; \emptyset \vdash O[\bar{A}] <: C \]
\[ p \vdash \text{typecase } x = O[\bar{A}]() \text{ of } B \Rightarrow e C \Rightarrow f D \Rightarrow g \text{ else } \Rightarrow h \text{ end } \leadsto [O[\bar{A}]()/\bar{x})/f \]

[R-TYPECASEELSE]

\[ p; \emptyset \vdash O[\bar{A}] <: B_i \quad 1 \leq i \leq |B| \]
\[ p \vdash \text{typecase } x = O[\bar{A}]() \text{ of } B \Rightarrow e \text{ else } \Rightarrow f \text{ end } \leadsto [O[\bar{A}]()/\bar{x})/f \]

[R-ASCRIPITON]

\[ p \vdash e \text{ as } A \leadsto e \]

Figure 8.19: Redex Evaluation Rules for HTV0
Context Evaluation rule: $p \vdash EC[R] \Rightarrow EC[e]$

$$\frac{p \vdash R \sim e}{p \vdash EC[R] \Rightarrow EC[e]}$$  

[R-CONTEXT]  

Figure 8.20: Context Evaluation Rule for HTV0

Evaluation rule: $p \vdash e \rightarrow e$

$$\frac{p \vdash e \Rightarrow f}{p \vdash e \rightarrow \text{path}_p(f)}$$  

[R-EVAL]  

Figure 8.21: Evaluation Rules for HTV0

8.4.3 HTV1

Syntax

The syntax of the intermediate language for HTV1 is identical to the intermediate language for HTV0 with two exceptions. Figure 8.22 shows the differences. Method calls contain path annotations, but no additional witness annotations. Also, type variables can be assigned lower bounds.

Static Semantics

The static semantics of HTV1 is similar to that of HTV0. In the following, we highlight the differences. To account for lower bounds on type variables, we extend the definition of type variable environments. A type variable environments is a set of bounds on type variables. There are two kinds of bounds: upper bounds on type variables (written $X <: K$) and lower bounds on type variables (written $K <= X$). Figure 8.23 defines an auxiliary function, which converts a bound into a type environment. This function is used by the path rules of HTV1. Figure 8.24 shows the two differences between the path rules for HTV1 and HTV0. Rule [P-VAR] is redefined with metavariables that account for lower bounds. Rule [P-EXT] is similar in spirit to the corresponding rule in HTV0, but lower bounds on type variables are taken into account. The same is true of the well-formedness rules for HTV1. Figure 8.25 shows the difference between the well-formedness
expression \( e, f, g, h \) ::= ...
    | \( e \) path \( \overrightarrow{A}.m[\overrightarrow{A}](\overrightarrow{e}) \)
    | ...

bound \( bnd \) ::= \( X \) extends \( K \)
    | \( X \) bounds \( K \)

Figure 8.22: Difference between the Syntax of Intermediate Languages for HTV1 and HTV0

Bounds Conversion:

\[
convert(bnd) = K <: K
\]

\[
convert(bnd) = \begin{cases} 
X <: K & \text{if } bnd = X \text{ extends } K \\
K <: X & \text{if } bnd = X \text{ bounds } K
\end{cases}
\]

Figure 8.23: Bounds Conversion for HTV1

Paths:

\[
p; \Delta \vdash A <: A
\]

\[
[P-VAR] \quad \frac{(K <: L) \in \Delta}{p; \Delta \vdash K <: L}
\]

\[ 
\text{S}[bnd_1] \text{ extends } \{M\} \text{ where } \{bnd_2\} \text{ end } \in p \\
\]

\[
\begin{align*}
\overrightarrow{bnd_1} &= X \\
\overrightarrow{bnd_2} &= Y \\
convert(bnd_1) &= \overrightarrow{K}_1 <: \overrightarrow{K}_2 \\
convert(bnd_2) &= \overrightarrow{L}_1 <: \overrightarrow{L}_2
\end{align*}
\]

\[
P; \Delta \vdash A \text{ ok} \quad p; \Delta \vdash [A/X][B/Y][\overrightarrow{K}_1] <: [A/X][B/Y][\overrightarrow{K}_2] \\
P; \Delta \vdash B \text{ ok} \quad p; \Delta \vdash [A/X][B/Y][\overrightarrow{L}_1] <: [A/X][B/Y][\overrightarrow{L}_2]
\]

\[
[P-EXT] \quad \frac{p; \Delta \vdash S[A]}{p; \Delta \vdash S[A] <: [A/X][B/Y]M_i}
\]

Figure 8.24: Difference between the Path Rules for HTV1 and HTV0
Well-formed types: $p; \Delta \vdash A \text{ ok}$

\[
\begin{align*}
S[\text{end}] & \quad \text{where } \{ \text{end} \} \quad \text{end} \in p \\
\text{bd1} & = X \quad \text{convert}(\text{bd1}) = K_1 <: K_2 \\
\text{bd2} & = Y \quad \text{convert}(\text{bd2}) = L_1 <: L_2 \\
\end{align*}
\]

\[
p; \Delta \vdash \text{bd1} \text{ ok} \quad p; \Delta \vdash \text{bd2} \text{ ok}
\]

\[
p; \Delta \vdash S[\text{A}] \text{ ok}
\]

Figure 8.25: Difference between the Type Well-Formedness Rules for HTV1 and HTV0

Method dispatch:

\[
\text{method}_p(m, \overrightarrow{A}) = \{ \text{md} \}
\]

\[
\begin{align*}
\text{method}_p(m, \overrightarrow{A}) = \\
\emptyset & \quad \text{if } \overrightarrow{A} = \cdot \\
\{ \overrightarrow{B}/X\} \text{md} & \quad \text{if } \overrightarrow{A} = S[\overrightarrow{B}] \overrightarrow{C} \\
& \quad \text{and } S[\overrightarrow{X}] \text{ md end } \in p \\
& \quad \text{and } \text{md } \in \{ \text{md} \} \\
& \quad \text{and } \text{name}(\text{md}) = m \\
\{ \text{method}_p(m, \overrightarrow{C}) & \quad \text{otherwise } \overrightarrow{A} = B \overrightarrow{C}
\end{align*}
\]

Figure 8.26: Method Dispatch for HTV1

rules for HTV1 and HTV0.

Figure 8.26 defines method dispatching for HTV1. Notice that a path annotation is sufficient to dispatch methods in HTV1. This follows from the fact that hidden type variables are not allowed in method definitions. Figure 8.27 defines the method call typing rule for HTV1. This rule is adjusted to include lower bounds on type variables and dispatch using only path annotations.

The trait and object definition typing rules of HTV1 are shown in Figure 8.28. Besides allowing for lower bounds on type variables, notice that the judgments for valid method definitions include two type variable environments instead of one. The first environment does not include hidden type variables, but the second does. Figure 8.29 gives the method typing rule for HTV1. The first
Expression typing: $\vdash p; \Delta ; \Gamma \vdash e : A$

Definition typing: $\vdash d$ ok

[T-METHOD] $\vdash e : A$ $\vdash A \rightarrow B$ path-ok

method\(_p\)(\(m, A \rightarrow B\)) = \{m[\(\text{bnd}\)[D]:E = _]\}

\(\text{bnd} = \bar{X}\) \(\text{convert}(\text{bnd}) = \bar{K}_1 \rightarrow \bar{K}_2\)

\(\vdash e : A\)

\(\vdash e : A\)

\[
\begin{align*}
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2) \\
inherited\_p(M_i) \cap inherited\_p(M_j) = \emptyset & \quad i \neq j \quad i, j \in \{1 \ldots |M|\}
\end{align*}
\]

\(\vdash A \rightarrow M\) ok $\vdash M$ ok $\vdash L$ ok

\(\vdash \Delta'; \self : (\text{T}[\bar{X}]; T[\bar{X}]) \rightarrow \text{md}\) ok

\(\vdash \text{trait T}[\text{bnd}_1] \text{ extends } \{M\} \text{ where } \{\text{bnd}_2\} \text{ md end}\) ok

\[
\begin{align*}
\text{bnd}_1 &= \bar{X}\bar{K} \quad \text{bnd}_2 = \bar{Y}\bar{L} \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2)
\end{align*}
\]

[T-OBJECTDEF] $\vdash \text{object O}[\text{bnd}_1](\text{x:A}) \text{ extends } \{M\} \text{ where } \{\text{bnd}_2\} \text{ md end}\) ok

\[
\begin{align*}
\text{bnd}_1 &= \bar{X}\bar{K} \quad \text{bnd}_2 = \bar{Y}\bar{L} \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2)
\end{align*}
\]

\(\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) = \emptyset & \quad i \neq j \quad i, j \in \{1 \ldots |M|\}
\end{align*}
\]

\(\vdash \Delta' \rightarrow K\) ok $\vdash A \rightarrow K$ ok $\vdash M$ ok $\vdash L$ ok

\(\vdash \Delta'; \self : (\text{O}[\bar{X}]; \text{O}[\bar{X}]) \rightarrow \text{md}\) ok

\[
\begin{align*}
\text{bnd}_1 &= \bar{X}\bar{K} \quad \text{bnd}_2 = \bar{Y}\bar{L} \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2)
\end{align*}
\]

\(\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) = \emptyset & \quad i \neq j \quad i, j \in \{1 \ldots |M|\}
\end{align*}
\]

\(\vdash \Delta' \rightarrow K\) ok $\vdash \Delta' \rightarrow M\) ok $\vdash \Delta' \rightarrow L\) ok

\[
\begin{align*}
\text{bnd}_1 &= \bar{X}\bar{K} \quad \text{bnd}_2 = \bar{Y}\bar{L} \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2)
\end{align*}
\]

\(\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) = \emptyset & \quad i \neq j \quad i, j \in \{1 \ldots |M|\}
\end{align*}
\]

\(\vdash \Delta' \rightarrow K\) ok $\vdash \Delta' \rightarrow M\) ok $\vdash \Delta' \rightarrow L\) ok

\[
\begin{align*}
\text{bnd}_1 &= \bar{X}\bar{K} \quad \text{bnd}_2 = \bar{Y}\bar{L} \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \text{ convert}(\text{bnd}_2)
\end{align*}
\]

Figure 8.27: Difference between the Expression Typing Rules for HTV0 and HTV1

Figure 8.28: Trait and Object Definition Typing Rules for HTV1
environment is used to check that the types in the definition are well formed and that the body’s type is a subtype of the declared return type. The second environment is used to check for valid overriding.

Figure 8.30 gives the overriding rules of HTV1. As for HTV0, a valid overriding is a method definition such that for every path starting from the trait or object enclosing the method, either no methods of the same name are found on the path, or the type of the first method found is in the appropriate relationship with the type of the overriding method (as described below). However, the relationship between the two types is slightly different that for HTV0. As for HTV0, the parameter types must be contravariantly related and the return types must be covariantly related. But, unlike HTV0, the bounds on the type parameters do not need to be equivalent. Instead, the type parameters of the overridden method must satisfy the bounds on the type parameters of the overriding method. This requirement ensures that all type arguments that satisfy the bounds on the type parameters of the overridden method will also satisfy the bounds on the type parameters of the overriding method.

Dynamic Semantics

The differences between the dynamic semantics of HTV1 and HTV0 are straightforward. Figure 8.31 shows the adjustment to the path update function for HTV1. Figure 8.32 shows the difference between the evaluation contexts and redexes for HTV1 and HTV0. Lastly, Figure 8.33 shows the differences between in the evaluation rules for HTV1 and HTV0. In each case, the definitions are altered to account for the new method call syntax. In addition, the method call evaluation rule
Method overriding: \[ p; \Delta; S[\overline{X}] \vdash \text{override}(md) \]

\[ p; \Delta \vdash S[\overline{Y}] \overline{C} \text{ path-ok} \]
\[ \text{method}_p(m, \overline{C}) = \{m[\overline{bnd}']((\overline{C};B') = \_ \} \]
\[ \overline{bnd} = X \overline{bnd}' = X' \overline{bnd} \]
\[ \text{convert}(\overline{bnd}) = K_1' <: K_2' \]
\[ \Delta' = \Delta \text{ convert}(\overline{bnd}') \]
\[ p; \Delta' \vdash [X'/X]K_1' <: [X'/X]K_2' \]
\[ p; \Delta' \vdash [X'/X] \overline{A} p; \Delta' \vdash [X'/X] B <: B' \]
\[ p; \Delta; S[\overline{Y}] \vdash \text{override}(m[\overline{bnd}][(x : A);B = e]) \]

[ OVERRIDE ]

\[ p; \Delta; S[\overline{Y}] \vdash \text{override}(m[\overline{bnd}][(x : A);B = e]) \]

[ NON OVERRIDE ]

\[ p; \Delta \vdash S[\overline{Y}] \overline{C} \text{ path-ok} \]
\[ \text{method}_p(m, \overline{C}) = \emptyset \]
\[ p; \Delta; S[\overline{Y}] \vdash \text{override}(m[\overline{bnd}][(x : A);B = e]) \]

Path update: \[ path_p(e) = e \]

\[ path_p(e) = \]
\[ \left\{ \begin{array}{l}
  f' \text{ path } \overline{D}.m[\overline{B}](\overline{g'}) \quad \text{if } e = f \text{ path } \overline{A}.m[\overline{B}](\overline{g}) \\
  \quad \text{and } f' = path_p(f) \\
  \quad \text{and } p; \emptyset; \emptyset \vdash f' : C \\
  \quad \text{and } \overline{D} = \text{min-path}_p(C \overline{A}) \\
  \quad \text{and } \overline{g'} = path_p(\overline{g}) \\
\end{array} \right. \]

Figure 8.31: Difference between Path Update for HTV1 and HTV0

\[ \text{evaluation context } \quad EC \quad ::= \quad \ldots \]
\[ \quad | \quad EC \quad \text{ path } \overline{A}.m[\overline{A}](\overline{e'}) \]
\[ \quad | \quad v \text{ path } \overline{A}.m[\overline{A}](\overline{v} \quad EC \quad \overline{e'}) \]
\[ \ldots \]

\[ \text{redex } \quad R \quad ::= \quad \ldots \]
\[ \quad | \quad v \text{ path } \overline{A}.m[\overline{A}](\overline{v}) \]
\[ \ldots \]

Figure 8.32: Difference between the Evaluation Contexts and Redexes for HTV1 and HTV0
Redex Evaluation rules: \( p \vdash R \leadsto e \)

\[ \text{[R-METHOD]} \]

\[
\begin{align*}
\text{object } O & \text{ (} \overrightarrow{x} \text{) end } \in p \\
\text{method } p(m, B) & = \{ m[\overrightarrow{X}](\overrightarrow{y}) = e \} \\
p \vdash O[\overrightarrow{A}] \text{ path } B.m[\overrightarrow{C}](\overrightarrow{w}) \leadsto [\overrightarrow{v} / \overrightarrow{x}] [O[\overrightarrow{A}]][\overrightarrow{w} / \overrightarrow{y}][\overrightarrow{C} / \overrightarrow{X}][e]
\end{align*}
\]

Figure 8.33: Difference between the Redex Evaluation Rules for HTV1 and HTV0

makes use of the method dispatch function for HTV1.

### 8.5 Type Soundness

The semantics of both HTV0 and HTV1 are type sound: if an expression is well typed then either it is a value or it reduces to an expression whose type is a subtype of the original expression. Appendix A provides a proof of type soundness for HTV1. The proof for HTV0 is very similar. Our proof structure follows the standard technique of establishing subject reduction and progress theorems [46].

**Theorem 1** (Progress). *If program \( p \) is well typed and \( p; \emptyset \vdash e : A \) for some expression \( e \) and type \( A \) then either \( e \) is a value or \( p \vdash e \leadsto f \) for some expression \( f \).*

**Theorem 2** (Subject Reduction). *If program \( p \) is well typed and \( p; \emptyset \vdash e : A \) for some expression \( e \) and type \( A \) and \( p \vdash e \leadsto f \) for some expression \( f \) then \( p; \emptyset \vdash f : B \) for some type \( B \) where \( p; \emptyset \vdash B <: A \).*

### 8.6 Preventing Cycles in the Type Hierarchy

Allowing a trait to extend other instantiations of itself raises the possibility of a cyclic type hierarchy. Consider:

```plaintext
trait T[X extends Object] extends T[Y]  
  where {Y extends Object} 
end
```

Then \( T[T[Object]] <: T[Object] <: T[T[Object]] \). As with mixin-style inheritance [1], we must prevent cyclic type hierarchies in order to keep the semantics of method inheritance and
overriding well defined. It is not particularly relevant to our system how such cycles are prevented, but they must be prevented in some way. Obviously, we could simply restrict cyclic hierarchies by fiat, but such a restriction is not useful unless there is an effective algorithm for enforcing it.

We say that a trait $T$ is self-extending if one of its supertypes $M$ is an instantiation of $T$; we also say that $M$ is a self-supertype of $T$. We place the following restrictions on a trait definition:

(R1) There are no cycles except through self-extension.

(R2) A trait definition has at most one self-supertype.

(R3) Each type parameter of a trait either bounds or is bounded by the corresponding type argument in the self-supertype.

These restrictions permit the class relationships we have discussed in this dissertation. A rigorous statement of these restrictions and a proof that they are sufficient to prevent cycles are presented in Appendix B.
Chapter 9

Implementing Hidden Type Variables

The semantics of HTV1 has been implemented using the PLT Redex term-rewriting system [35]. In Section 9.1, we present the implementation of the HTV1 syntax. In Section 9.2, we present the implementation of the HTV1 dynamic semantics. Implementation of the type checking algorithm and inference procedures are described in Section 9.3. The resulting system can be queried online [23].

9.1 Syntax of HTV1 in PLT Redex

We use PLT Redex to create a prototype implementation of HTV1. PLT Redex is a domain-specific language designed for specifying and debugging the operational semantics of a programming language. Given a grammar and a set of reduction rules, PLT Redex automatically generates the infrastructure necessary to evaluate programs of the specified language. First, we describe the syntax of our PLT Redex model. Second, we discuss the evaluation rules of our model. As HTV1 is implemented in MzScheme [21], the grammar and evaluation rules are defined using s-expression syntax.

A program written in the syntax of HTV1 is translated into the intermediate language of HTV1 by the parser. The intermediate language is used by the PLT Redex implementation. Before defining the implementation of the syntax of this language, we define the keywords:

```
(define keywords
  ' (object trait as self Object Bottom union extends bounds))
```
The keywords are used by the PLT Redex definition in Figure 9.1, which implements the syntax of the intermediate language for HTV1. The syntax is defined using the following PLT Redex function:

\[
(\text{language} \ (<\text{non-terminal-name}> \ <\text{rhs-pattern}> \ ... \ ) \ ...)
\]

This function specifies a BNF grammar for a regular tree language. The right-hand side of each grammar rule is defined using PLT Redex’s pattern language.

The metavariables in the implementation of the syntax are consistent with the specification in Figures 8.3 and 8.22. However, PLT Redex requires a single metavariable for each syntactic definition. Therefore, extra metavariables are omitted.

In addition, notice that the metavariables T and O match the same names. This causes the patterns \((O \ (A \ ...))\) and \((T \ (A \ ...))\) to match the same types. Therefore, choosing to represent object and trait types as \((O \ (A \ ...))\) and \((T \ (A \ ...))\) may cause the reduction relation to match an expression in more than one way. If this happens then PLT Redex will raise a non-unique results error. For this reason, the type \((O \ (A \ ...))\) is not included in the definition of A.

The rest of the implementation of the syntax is a straightforward translation of the specification, with one exception: the syntax of method calls. The implementation of expressions defines two forms of method calls:

1. \((e \ m \ (A \ ...) \ (e \ ...))\), and
2. \((e \ (A \ ...) \ m \ (A \ ...) \ (e \ ...))\).

The first form represents method calls without path annotations. The second form represents elaborated method calls, that is, method calls with path annotations. Both forms contain an expression corresponding to the receiver, a method name, a list of types corresponding to the type arguments, and a list of expressions corresponding to the value arguments. The second form contains an extra list of types corresponding to the path annotation. After parsing, but before type checking, all method calls have the first form. When type checking expressions, all method calls of the first form are elaborated into the second form.
Figure 9.1: Implementation of Syntax of Intermediate Language for HTV1
(define htv1-lang
  (extend-language
   htv1-core-lang
   ; value
   (v (O (A ...) (v ...)))
   ; evaluation context
   (EC hole
    (O (A ...) (v ... EC e ...))
    (EC x)
    (EC (A ...) m (A ...) (e ...))
    (v (A ...) m (A ...) (v ... EC e ...))
    ((x EC) ((A e) ...) e))))

Figure 9.2: Implementation of Values and Evaluation Contexts for HTV1

Figure 9.2 shows the PLT Redex definition that implements the values and evaluation contexts of HTV1. The specification is given in Figures 8.18 and 8.32. The implementation is a straightforward encoding of the specification.

9.2 Evaluation Rules of HTV1 in PLT Redex

The implementation of the path update function, as specified in Figures 8.17 and 8.31, is given in Figure 9.3. The metavariables in Figure 9.3 are consistent with those in the specification. However, in the implementation, the letter $p$ is appended to metavariable names that include primes in the specification.

The function $path$ takes a program and an expression whose paths will be updated. The form of the expression is matched against several patterns. As the specification of the function dictates, the actions associated with each pattern, except that of method calls, are either a straightforward recursive descent or a no-op.

If the expression is a method call then the expression must be an annotated method call. In this case, the receiver and value arguments are recursively processed. More interestingly, the receiver is typed using the following form:

(exp-type? <program>
     <type-variable-environment>
;; path update
;; In the formal semantics, path_p(e) = e
;; input: a program
;; an expression
;; output: a path-updated expression
(define (path p e)
  (match e
    ; field access
    [(f x)
     (let ((fp (path p f)))
      '(,fp ,x))]
    ; elaborated method call
    [(f (A ...) m (B ...) (g ...))
     (let* ((fp (path p f))
            (pair (exp-type? p () () fp))
            (C (second pair))
            (D (min-path p (cons C A)))
            (gp (map (lambda (g) (path p g)) g)))
      '(,fp ,D ,m ,B ,gp))]
    ; typecase
    [((x f) ((A g) ...) h)
     (let ((fp (path p f))
           (gp (map (lambda (g) (path p g)) g))
           (hp (path p h)))
      '((,x ,fp) ,(map (lambda (A gp) '(,A ,gp)) A gp) ,hp))]
    ; ascription
    [(f 'as A)
     (let ((fp (path p f)))
      '(,fp as ,A))]
    ; object instance
    [(O (A ...) (f ...))
     (let ((fp (map (lambda (f) (path p f)) f)))
      '(',O ,A ,fp))]
  [e e])

Figure 9.3: Implementation of Path Update for HTV1
This function returns a pair. The second element of the pair is the type of the expression. The receiver’s type is prepended onto the path annotation and passed to the following function:

(min-path <program> (<type> ...))

This function implements the path extension function from Figure 8.16. Each pair of contiguous types in the second argument of the function is passed to a constraint solving algorithm. The constraint solving algorithm returns a minimal path between these types. The result of the min-path function is the concatenation of these paths. The constraint solving algorithm is discussed in detail in Section 9.3.1. Finally, the path function adds the extended path to the method call and returns the result.

The PLT Redex definition of the evaluation rules is given in Figures 9.4 and 9.5. The evaluation rules are defined using the following PLT Redex function:

(reduction-relation
  <language>
  (〈arrow〉 <lhs-pattern> <rhs-exp> <extras>) ...
  where
  [(〈arrow〉 <var> <var>) (〈arrow〉 <lhs-pattern> <rhs-exp>)] ...)
  <extras> = (side-condition <guard> ...)
  | (where <var> <exp>)

This function defines a reduction relation using auxiliary reduction relations. The first argument is a language to which the relation applies. The second argument is a list of rules that make up a relation. The <lhs-pattern> refers to the <language>, and binds variables in the <rhs-exp>. The <rhs-exp> is the result of the reduction. The <extras> is either a side condition, whose argument is a boolean that is expected to hold, or a where clause, which binds a variable. Each clause after the where defines a new relation in terms of a previously defined relation. The main relation of the definition is denoted by the reserved arrow \rightarrow.

The first five rules implement the redex evaluation relation (written as \sim\sim> in the implementation) as specified in Figures 8.19 and 8.33. The last two rules implement the context evaluation
;; evaluation rules
;; In the formal semantics, p |- R `>` e
;; p |- EC[R] ==> EC[e]
;; p |- e ---> e
;; input: a program
;; an expression
;; output: a reduced expression
(define (htv1-reds p)
  (reduction-relation
   htv1-lang
   ; [R-Field]
   (`>` ((O_0 (A ...) (v_0 ...)) x_i)
     v_i
     (where obj-def
       ,(obj-lookup p (term O_0)))
     (where v_i
       ,(fld-lookup (term obj-def)
         (term (v_0 ...))
         (term x_i))))
   ; [R-Method]
   (`>` ((O_0 (A_0 ...) (v_0 ...)) (A_1 ...) m_0 (A_2 ...) (v_1 ...))
     ,(substs (term (v_0 ...))
       (term (x ...))
       (substs (term ((O_0 (A_0 ...) (v_0 ...))))
         '(self)
         (substs (term (v_1 ...))
           (term (y ...))
           (substs (term (A_2 ...))
             (term (X ...))
             (term e)))))
     (where ((bnd_0 ...) ((x A) ...) (M ...) (bnd_1 ...) (md ...))
       ,(obj-lookup p (term O_0)))
     (where (((X _ K) ...) ((y A) ...) B e)
       ,(md-lookup p (term m_0) (term (A_1 ...))))))

Figure 9.4: Implementation of Evaluation Rules for HTV1 (I)
Figure 9.5: Implementation of Evaluation Rules for HTV1 (II)
relation (written as \(\equiv\ldots\equiv\)) in the implementation) and the evaluation relation (written as \(\dashv\ldots\dashv\)) in the implementation), which are specified in Figures 8.20 and 8.21, respectively.

Unfortunately, the metavariables used in the implementation of these rules do not always match those in the specification. PLT Redex requires that variables bound in the \(<\text{lhs-pattern}>\) be metavariables of \(<\text{language}>\) followed by \(\_\) and an identifier. For example, the variables \(O_0\) and \(x_i\) are bound in implementation of rule \([\text{R-Field}]\), but \(A\) is not. Because of this restriction, the implementation must use the same metavariable for otherwise unrelated entities of the same kind in the same rule. The implementation differentiates metavariables of the same kind by the identifier that follows the \(\_\).

PLT Redex also requires the following function to be used when referencing a variable bound in the \(<\text{lhs-pattern}>\):

\[
\text{(term \(<\text{s-exp}>\))}
\]

This function is used for construction of new s-expressions in the \(<\text{rhs-exp}>\). It behaves similarly to quasiquote except that names bound in the \(<\text{lhs-pattern}>\) (and a few reserved names) are implicitly substituted with the values that those names are bound to.

The antecedents of the redex evaluation rules are implemented as either side conditions or where clauses. The implementation of rule \([\text{R-Field}]\) looks up the receiver object and the field name via the auxiliary functions \(\text{obj-lookup}\) and \(\text{fld-lookup}\), respectively. The corresponding value argument of the object instance (i.e., the field’s value) is returned.

The implementation of rule \([\text{R-Methode}]\) is defined for elaborated method calls only. This is because all method calls are elaborated prior to the program’s execution. The receiver object and the method name are looked up via the auxiliary functions \(\text{obj-lookup}\) and \(\text{md-lookup}\), respectively. The type and value arguments of the call are substituted into the body of the method using the function \(\text{substs}\).

The implementations of rules \([\text{R-Typecase}]\) and \([\text{R-TypecaseELSE}]\) make use of the functions \(\text{subtype}\) and \(\text{not-subtype}\) to check the appropriate subtype relationships. The value of the test expression is substituted into the result of the rule. The implementation of rule \([\text{R-Ascription}]\) simply removes the ascribed type.
9.3 Type Checking HTV1

The most interesting and novel aspect of HTV1 is its type system. For the most part, the implementation of the type system for HTV1 is a straightforward translation of its semantic specification. The major exception is subtyping. The semantic definition of subtyping does not provide a straightforward implementation strategy. In fact, there is no way to decide whether two arbitrary types are in the subtype relation of HTV1. Instead, we will investigate decision procedures that approximate the subtype relation.

The semantics of well-formed types and method overriding are also not immediately implementable. The rules for well-formed types require witnesses to be found for hidden type variables. Method overriding requires a path to be inferred. In addition, the semantics of HTV1 assumes that path annotations on method calls have been inferred. These issues are addressed in this section.

9.3.1 Subtype Constraint Solving

In the following, we call a pair of types a constraint. We write a constraint as $A <: B$. A constraint may contain free type variables (i.e., type variables not bound in the type variable environment). A constraint $A <: B$ with a type variable environment $\Delta$ is solved if there exists a substitution $\sigma$ for the free type variables in $A$ and $B$ such that the result of applying $\sigma$ to $A$ is a subtype of the result of applying $\sigma$ to $B$.

Notice that subtype checking is a special case of constraint solving, where the constraints contain no free type variables. Unlike subtype checking, constraint solving provides the ability to infer types. To infer a type, a fresh type variable is used as a placeholder for the type and the constraints on that variable are collected. The constraint solver then computes a substitution for that variable, which solves the associated constraints.

The constraint solving algorithm is the core of the type checker for HTV1. The constraint solver is used by virtually every component of the type checker. For example, the constraint solver is used to check subtyping, infer path annotations, extend path annotations, check the well-formedness of types, and check valid overriding. However, building a constraint solver for HTV1 is a difficult task.
In fact, even the simpler problem of subtype checking is undecidable.

**Undecidability of Subtype Checking in HTV1**

The problem of determining whether two types are in the subtype relation of HTV1 is undecidable. This follows from a result by Kennedy and Pierce [29], who show that subtype checking in a simple language with nominal inheritance and variance for generic types is undecidable.

The language of Kennedy and Pierce consists of class declarations of the form:

\[ C(\overrightarrow{vX}) \lll T_1 \ldots T_n \]

where:

- \( C \) is a type constructor,
- \( \overrightarrow{v} \) are variance annotations used to indicate covariant, invariant, and contravariant type parameters,
- \( \overrightarrow{X} \) are type variables, and
- \( T_1 \ldots T_n \) are types (i.e., either a type variable \( X \) or a constructed type \( D(\overrightarrow{U}) \)).

A subtyping judgment in this language is written:

\[ T_1 \lll T_2. \]

The types in a subtyping judgment do not contain type variables. In addition, Kennedy and Pierce use the following judgment to denote that a type is well-formed:

\[ \overrightarrow{vX} \vdash T \text{ ok}. \]

A subtyping judgment in the language of Kennedy and Pierce can be reduced to a subtyping judgment in HTV1. For the reduction, we assume a bijective encoding function \( E \) from the set of type variables and type constructors in the language of Kennedy and Pierce to the set of type
variables and trait and object names in HTV1. We write $\mathcal{E}(X)$ and $\mathcal{E}(C)$ for the encoding of type variable $X$ and constructor $C$.

We extend the encoding function to types and class declarations. A constructed type:

$$C\langle\vec{T}\rangle$$

in the language of Kennedy and Pierce is encoded as:

$$\mathcal{E}(C)\left[\mathcal{E}(\vec{T})\right]$$

in HTV1. A class:

$$C\langle v_1X_1\ldots v_mX_m\rangle <:: T_1\ldots T_n$$

in the language of Kennedy and Pierce is encoded as:

trait $S[\ bnd_1\ldots bnd_m\ ]$

extends $\{S[\ Z_1\ldots Z_m\ ]\ M_1\ldots M_n\}$

where $\{bnd'_1\ldots bnd'_m\}$

end

in HTV1, where:

- $S = \mathcal{E}(C)$,
- $Y_1 = \mathcal{E}(X_1)\ldots Y_m = \mathcal{E}(X_m)$,
- $bnd_i = \begin{cases} 
  Y_i\text{ extends } Z_i & \text{if } X_i \text{ is covariant} \\
  Y_i\text{ bounds } Z_i & \text{if } X_i \text{ is contravariant} \\
  Y_i\text{ extends } Z_i & \text{if } X_i \text{ is invariant} 
\end{cases}$
  where $1 \leq i \leq m$,
- $bnd'_i = \begin{cases} 
  Z_i\text{ extends } \text{Object} & \text{if } X_i \text{ is covariant} \\
  Z_i\text{ extends } Y_i & \text{if } X_i \text{ is contravariant} \\
  Z_i\text{ extends } Y_i & \text{if } X_i \text{ is invariant} 
\end{cases}$
  where $1 \leq i \leq m$, and
- $M_1 = \mathcal{E}(T_1)\ldots M_n = \mathcal{E}(T_n)$. 
The following lemma, which is proved in Appendix C, shows that $\mathcal{E}$ reduces subtype checking in the language of Kennedy and Pierce to subtype checking in HTV1.

**Lemma 13.** Let $CT$ be a class table in the language of Kennedy and Pierce. Let program $p$ (in the language of HTV1) be the result of encoding $CT$ with $\mathcal{E}$. Let $T_1$ and $T_2$ be two types in the language of Kennedy and Pierce such that:

\[ \emptyset \vdash T_1 \text{ ok and } \emptyset \vdash T_2 \text{ ok.} \]

Then:

\[ T_1 <: T_2 \]

if and only if:

\[ p; \emptyset \vdash \mathcal{E}(T_1) <: \mathcal{E}(T_2). \]

Lemma 13 (and the fact that subtype checking in the language of Kennedy and Pierce is undecidable) proves that subtype checking in HTV1 is undecidable (otherwise one could decide the subtype relation of Kennedy and Pierce by encoding the program with $\mathcal{E}$ and using the subtype checker for HTV1).

**Theorem 4** (Undecidability of Subtyping in HTV1). *Subtyping in HTV1 is undecidable.*

**Algorithm: High-Level Description**

Due to the undecidability of subtype checking, it is not possible to develop a constraint solving algorithm for HTV1 that terminates with the correct result on every input. Therefore, as an alternative, we present a constraint solving algorithm that yields the correct result when it terminates, but is not guaranteed to terminate on all inputs. Later, we discuss restrictions to guarantee termination.

Given a list of constraints, the constraint solving algorithm returns a list of paths in the type hierarchy and a substitution. The paths provide a proof that the substitution solves each constraint. In other words, for each input constraint $A <: B$, the constraint solver returns a path that begins with the substitution applied to $A$ and ends with the substitution applied to $B$.

Figure 9.6 shows a diagram of the constraint solving algorithm. This diagram shows the relationship between the five main functions of the algorithm. Solid arrows denote function calls. The function that the arrow points from calls the function that the arrow points to. These arrows are labeled with the arguments of the call. Dashed arrows indicate the return of a function call. These
Figure 9.6: High-Level Diagram of the Constraint Solving Algorithm
arrows are labeled with the return values.

The driver of the constraint solving algorithm is a function called SOLVE. This function takes a list of constraints as input and returns a pair of a list of paths in type hierarchy and a substitution. The paths provide a proof that the substitution solves each constraint.

The function SOLVE passes the input constraints to the constraint simplification algorithm, called SIMPLIFY. The function SIMPLIFY takes a list of constraints as input and returns a triple of a list of paths, a list of constraints, and a substitution. The paths and the substitution are eventually returned by SOLVE. Therefore, the paths will eventually provide a proof that the substitution solves each input constraint. However, the paths may contain free type variables. The list of constraints returned by SIMPLIFY provides bounds on those free type variables.

In fact, the constraints returned by SIMPLIFY are simplified constraints. A constraint in which one of the types is a free type variable is a simplified constraint. Such a constraint places a bound on the type variable. This bound will be used to infer a witness for the type variable.

The function SOLVE passes the constraints returned by SIMPLIFY to the witness construction algorithm, called CONSTRUCT-WITNESSES. This function takes a list of simplified constraints as input and returns a substitution, which replaces each free type variable in the input constraints with the union of its lower bounds.

To ensure that the witnesses returned by CONSTRUCT-WITNESSES are valid, SOLVE applies the substitution returned by CONSTRUCT-WITNESSES to the constraints returned by SIMPLIFY and passes the result to the witness checking algorithm, called CHECK-WITNESSES. This function takes a list of constraints as input and returns true if the witnesses are valid. Otherwise, false is returned.

SOLVE applies the substitution returned by CONSTRUCT-WITNESSES to the paths returned by SIMPLIFY. Also, the substitution returned SIMPLIFY is combined with the substitution returned by CONSTRUCT-WITNESSES. Lastly, the new paths and the combined substitution are returned by SOLVE.

The constraint simplification algorithm SIMPLIFY passes each input constraint to the type hierarchy search algorithm, called SEARCH. This function takes a single constraint as input and returns a triple of a path in the type hierarchy, a list of constraints, and a substitution. The path proves that
the substitution solves the input constraint. However, the path may contain free type variables. The list of constraints returned by SEARCH provides bounds on those free type variables. These constraints are in simplified form. SIMPLIFY combines the results of each call to SEARCH and returns the combined result.

The witness checking algorithm CHECK-WITNESSES also passes each input constraint to SEARCH. If any call to SEARCH returns the special value error then the input constraint is not a valid subtyping. In this case, false is returned by CHECK-WITNESSES. Otherwise, true is returned by CHECK-WITNESSES.

For each constraint $A <: B$, the type hierarchy search algorithm SEARCH performs a breadth-first search of the type hierarchy for a path from $A$ to $B$. There may be several paths from $A$ to $B$, but the breadth-first search ensures that the chosen path is at least as short as every other path. Among the set of shortest possible paths, the chosen path is selected blindly.

The type hierarchy search may yield a path that contains hidden type variables. To find witnesses for the hidden type variables, the constraints on the hidden type variables are collected during the search. Once the search has finished, the constraints on the hidden type variables are passed to the constraint simplification algorithm SIMPLIFY. SEARCH returns the simplified constraints that are returned by SIMPLIFY.

In the process of searching the type hierarchy, witnesses may be found for the free type variables occurring in path. These witnesses are added to the substitution returned by SIMPLIFY and the extended substitution is returned by SEARCH.

**Algorithm: Example Invocation**

As an example invocation, consider solving the constraint Empty $<$: List[Number] in the context of the following definitions:

```plaintext
trait Number extends Object end
trait List[X extends Object] extends Object end
object Empty extends List[Y] where {Y extends Object} end
```
NIL An empty list, queue, or substitution.
CONS(element, list) The result of adding element onto the front of list.
LAST(list) The last element of list.
APPEND(list, list') A new list containing the elements of list followed by those of list'.
SIZE(list) The number of elements in the given list.
ENQUEUE(element, queue) The result of adding element to the back of queue.
DEQUEUE(queue) The first element of queue and the remainder of queue.
NOT-EMPTY(queue) If queue is empty then false. Otherwise true.
APPLY-SUB(sub, element) The result of applying the substitution sub to element. The argument element may be either a type, a path, a list of paths, a constraint, or a list of constraints.
DOMAIN(sub) The set of type variables in the domain of substitution sub.
COMPOSE(sub, sub') The composition of substitutions sub and sub'.
UNIFY\(\Delta\) (A, B) A substitution sub such that \(\text{DOMAIN}(\text{sub}) \cap tv(\Delta) = \emptyset\) and \(\text{APPLY-SUB}(\text{sub}, A) = \text{APPLY-SUB}(\text{sub}, B)\). If no such substitution exists then false.
FRESH-VARS(\(\vec{X}\)) A list of length \(|\vec{X}|\) containing that many fresh type variables.
FREE-VARS_{\Delta}(A) The set of free type variables in A (i.e., the set of type variables in A that are not bound in \(\Delta\)).
FREE-VARS_{\Delta}(constraints) The set of free type variables in each type of each constraint in constraints.
UPPER-BOUNDS_{\Delta}(K) A list of upper bounds of K in \(\Delta\).

Figure 9.7: Auxiliary Functions for Constraint Solving Pseudocode

Initially, the type hierarchy is traversed from Empty to List\(\llbracket Y\rrbracket\). While doing so, the following path is constructed:

Empty List\(\llbracket Y\rrbracket\)

and the following constraint on the hidden type variable Y is collected:

\(Y <: \text{Object}\)

In order for the path to end at List\(\llbracket \text{Number}\rrbracket\), Number is substituted for Y. Finally, the constraint:

\(\text{Number} <: \text{Object}\)

is solved to ensure that Number is a valid witness for Y.
Algorithm: Type Hierarchy Search Pseudocode

The implementation of the constraint solving algorithm is presented using pseudocode in the style of Cormen, Leiserson, Rivest, and Stein [17]. The pseudocode makes use of several auxiliary functions. These are listed in Figure 9.7. The auxiliary functions $tv(\Delta)$ and $convert(bnd)$, which were defined in Chapter 8, are also used. The notation $list[i]$ is used to select the $i^{th}$ element of $list$.

To indicate that the algorithm has failed to solve a set of constraints, the pseudocode makes use of the special value $error$. Once this value is returned by a function, it is automatically propagated through the call stack, like an exception. For some function calls, the pseudocode will specify whether $error$ is returned by the call. This is analogous to code that catches an exception thrown by the function.

Figure 9.8 shows the pseudocode for the constraint solver’s type hierarchy search. Due to the size of the pseudocode, parts of the algorithm are replaced by ellipses. These parts are discussed shortly.

An invocation of the search algorithm, such as $\text{SEARCH}_{p, \Delta}(A <: B)$, executes a breadth-first search of the trait and object hierarchy starting from $A$. To aid this search, a queue is constructed. Each element of the queue is a position in the hierarchy traversal. An element of the queue is a pair of a path and a list of constraints. The path is the list of types traversed, and is initially set to the singleton list $\text{CONS}(A, \text{NIL})$. The list of constraints is the collection of constraints on hidden type variables, and is initially empty.

The algorithm continually processes the first element of the queue. If the queue is empty then the algorithm has failed to find type $B$. In this case, $error$ is returned. Otherwise, the conditional shown in the body of the while loop in Figure 9.8 is executed.

This conditional consists of seven branches. The body of each branch is replaced by an ellipsis. The complete pseudocode and description of each branch follows.

The first branch is executed if the most recently visited type in the path can be unified with $B$. In this case, the algorithm has found a path to type $B$. However, the collected constraints must be solved to ensure that the path is valid. The type hierarchy search algorithm does not solve these constraints. Instead, the constraints are simplified and returned by the search algorithm to be
SEARCH\textsubscript{$p,\Delta$}$(A <: B) =$

\begin{verbatim}
path ← CONS(A, NIL)
constraints ← NIL
queue ← ENQUEUE((path, constraints), NIL)

while NOT-EMPTY(queue)

(path, constraints, queue) ← DEQUEUE(queue)
C ← LAST(path)

if UNIFY$\Delta$(C, B) returns sub
and sub \neq false
and SIMPLIFY$\Delta$($p$,$\Delta$)(APPLY-SUB(sub, constraints)) does not return error
... 
else if ($B = \text{Object}$ or $C = \text{Bottom}$)
and SIMPLIFY$\Delta$($p$,$\Delta$)(constraints) does not return error
... 
else if $B$ is of the form $D \cup E$
and SIMPLIFY$\Delta$($p$,$\Delta$)(CONS($C$ $<$: $D$, constraints)) does not return error
... 
else if $B$ is of the form $D \cup E$
and SIMPLIFY$\Delta$($p$,$\Delta$)(CONS($C$ $<$: $E$, constraints)) does not return error
... 
else if $C$ is of the form $D$\llbracket \rightarrow \rrbracket$
... 
else
... 
return error
\end{verbatim}

Figure 9.8: Pseudocode: Skeleton of the Type Hierarchy Search
solved at a later time by the driver algorithm SOLVE. The simplification is done by the constraint simplification algorithm SIMPLIFY, which we discuss shortly. The pseudocode for the first branch follows.

\[
\text{if UNIFY}_{\Delta}(C, B) \text{ returns } \text{sub}
\]
\[
\text{and } \text{sub} \neq \text{false}
\]
\[
\text{and SIMPLIFY}_{p,\Delta}(\text{APPLY-SUB}(\text{sub}, \text{constraints})) \text{ does not return error}
\]
\[
(\_ \_ \text{simplified}, \text{sub}') \leftarrow \text{SIMPLIFY}_{p,\Delta}(\text{APPLY-SUB}(\text{sub}, \text{constraints}))
\]
\[
\text{sub} \leftarrow \text{COMPOSE}(\text{sub}, \text{sub}')
\]
\[
\text{path} \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path})
\]
\[
\text{return } (\text{path}, \text{simplified, sub})
\]

The constraint simplification algorithm returns a list of paths, a list of simplified constraints, and a substitution. A triple of the path, the simplified constraints, and the combined substitutions is returned by the first branch.

The second branch is executed if either \( B \) is Object or the path has reached Bottom. In either case, the algorithm has found a valid path between \( A \) and \( B \). Again, the collected constraints must be simplified. The pseudocode for the second branch follows.

\[
\text{else if } (B = \text{Object} \text{ or } C = \text{Bottom})
\]
\[
\text{and SIMPLIFY}_{p,\Delta}(\text{constraints}) \text{ does not return error}
\]
\[
(\_ \_ \text{simplified, sub}) \leftarrow \text{SIMPLIFY}_{p,\Delta}(\text{constraints})
\]
\[
\text{path} \leftarrow \text{APPEND}(\text{path}, \text{CONS}(B, \text{NIL}))
\]
\[
\text{path} \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path})
\]
\[
\text{return } (\text{path}, \text{simplified, sub})
\]

After the constraints are simplified, the appropriate triple is returned.

The third or fourth branch is executed if \( B \) is a union type (i.e., \( D \cup E \)). By rule [P-UNIONSUPER], if the path reaches either element of the union type then the algorithm has constructed a valid path between \( A \) and \( B \). The third branch requires the constraint \( C \prec D \) to be simplified. This ensures that the path will reach \( D \). The fourth branch does the same for \( E \). The pseudocode for the third and fourth branches follows.
else if $B$ is of the form $D \cup E$
and SIMPLIFY$_{p,\Delta}(\text{CONS}(C <: D, \text{constraints}))$ does not return error
\begin{align*}
  & (\text{paths}, \text{simplified}, \text{sub}) \leftarrow \text{SIMPLIFY}_{p,\Delta}(\text{CONS}(C <: D, \text{constraints})) \\
  & \text{path} \leftarrow \text{APPEND}(\text{APPEND}(\text{path}, \text{paths}[1]), \text{CONS}(B, \text{NIL})) \\
  & \text{path} \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
  & \text{return } (\text{path}, \text{simplified}, \text{sub})
\end{align*}

Recall that the simplification algorithm returns a list of paths, a list of constraints, and a substitution. Assuming that the simplified constraints are solvable, the paths show that the substitution solves the input constraints. Therefore, each path returned by SIMPLIFY corresponds to an input constraint. In particular, the path $\text{paths}[1]$ corresponds to the first input constraint (i.e., $C <: D$ or $C <: E$). This path is appended to the current path and the appropriate triple is returned.

The fifth branch is executed if the path has reached a union type (i.e., $D \cup E$). In this case, rule [P-UNIONSUB] dictates that both $D$ and $E$ be subtypes of $B$. Therefore, the constraints $D <: B$ and $E <: B$ are passed to the constraint simplification algorithm. The pseudocode for the fifth branch follows.

else if $C$ is of the form $D \cup E$
and SIMPLIFY$_{p,\Delta}(\text{CONS}(D <: B, \text{CONS}(E <: B, \text{constraints})))$ does not return error
\begin{align*}
  & (\_, \text{simplified}, \text{sub}) \leftarrow \text{SIMPLIFY}_{p,\Delta}(\text{CONS}(D <: B, \text{CONS}(E <: B, \text{constraints}))) \\
  & \text{path} \leftarrow \text{APPEND}(\text{path}, \text{CONS}(B, \text{NIL})) \\
  & \text{path} \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
  & \text{return } (\text{path}, \text{simplified}, \text{sub})
\end{align*}

According to rule [P-UNIONSUB], adding $B$ to the current path is sufficient to show that the constraint is solved. The simplified constraints and the substitution returned from SIMPLIFY are also returned by the fifth branch.

The sixth branch is executed if the most recently visited type in the path is a trait or object type. Unlike the previous cases, this case does not simplify the collected constraints. This is because the collected constraints are simplified immediately before the search algorithm returns, and this...
case does not return from the search algorithm. Instead, the breadth-first search continues. The pseudocode for the sixth branch follows.

```plaintext
else if C is of the form S[[D]]
    if p contains a declaration of the form
    - S' [[X −→ D]] extends {M} where {Y −→ L} end
    where S' = S
        Z ← FRESH-VARS(Y)
        sub ← COMPOSE([D/X], [Z/Y])
        constraints' ← APPEND(convert(X −→ K), convert(Y −→ L))
        constraints' ← APPEND(constraints, constraints')
        constraints' ← APPLY-SUB(sub, constraints')
        for i ← 1 to SIZE(M)
            path' ← APPEND(path, CONS(APPLY-SUB(sub, M[i]), NIL))
            queue ← ENQUEUE((path', constraints'), queue)
        end
        uppers ← UPPER-BOUNDS(Δ(C))
        for i ← 1 to SIZE(uppers)
            path' ← APPEND(path, CONS(uppers[i], NIL))
            queue ← ENQUEUE((path', constraints), queue)
        end
else
    return error
```

In this case, a new element is added to the queue for each supertype. Supertypes include the types in the extends clause of the trait or object, and the upper bounds in the type variable environment. The path for the new queue element is the result of adding the supertype to the current path. For extended types, the constraints for the new queue element are the combination of the current constraints and the constraints on the type variables of the trait or object. For upper bounds, the constraints for the new queue element are the current constraints.

If none of the other cases are executed then the seventh branch is executed. In this case, the most recent type in the path must be either `Object` or a type variable. The pseudocode for the seventh branch follows.

```plaintext
else
    uppers ← UPPER-BOUNDS(Δ(C))
    for i ← 1 to SIZE(uppers)
        path' ← APPEND(path, CONS(uppers[i], NIL))
        queue ← ENQUEUE((path', constraints), queue)
    end
```

If this branch is executed then the breadth-first search continues to the upper bounds in the type
\[
\text{BOUND}_\Delta(A <: B) = \\
\text{if } A \text{ is a type variable } X \text{ and } X \notin tv(\Delta) \\
\text{ return true} \\
\text{ else if } B \text{ is a type variable } X \text{ and } X \notin tv(\Delta) \\
\text{ return true} \\
\text{ else} \\
\text{ return false}
\]

Figure 9.9: Pseudocode: Bound Checking

variable environment. A new element in the queue is constructed for each upper bound in the type variable environment. The path of the new element is the result of adding the upper bound to the current path. The constraints of the new element are the current constraints. Notice that if no upper bounds exist then the queue is not extended. In this case, the path has reached a dead end and the queue element is discarded.

Algorithm: Constraint Simplification Pseudocode

Figure 9.9 defines the bound checking algorithm, which is an auxiliary algorithm used by the constraint simplification procedure. This algorithm tests whether a constraint can be further simplified. A constraint in which one of the types is a free type variable cannot be further simplified. Instead, the constraint places a bound on the type variable. This bound will be used to infer a witness for the type variable. We call a constraint, in which one of the types is a free type variable, a simplified constraint.

Figure 9.10 defines the constraint simplification algorithm. This algorithm simplifies constraints to find the bounds on the free type variables of the constraints. Given a list of constraints, this algorithm returns a list of paths, a list of simplified constraints, and a substitution. Assuming the simplified constraints are solvable, the paths show that the substitution solves the input constraints. Each input constraint is passed to the bound checking algorithm to see if it can be further broken down. If not, then the constraint is already in simplified form and becomes part of the accumulating list of simplified constraints that will be returned. Otherwise, the constraint is passed to the type hierarchy search algorithm. The results of the type hierarchy search algorithm are combined with the results of simplifying other constraints. Notice that the search algorithm and the simplification
SIMPLIFY\(_{p,\Delta}(\text{constraints}) = \)
paths ← NIL
simplified ← NIL
sub ← NIL
for \(i \leftarrow 1\) to \(\text{SIZE(constraints)}\)
  constraint ← \(\text{constraints}[i]\)
  if BOUND\(_\Delta\)(constraint)
    simplified ← CONS(constraint, simplified)
    paths ← APPEND(paths, CONS(false, NIL))
  else
    \((\text{path}', \text{simplified}', \text{sub}')\) ← SEARCH\(_{p,\Delta}(\text{constraint})\)
    paths ← APPEND(paths, CONS(path', NIL))
    simplified ← APPEND(simplified, simplified')
    constraints ← APPLY-SUB(sub', constraints)
    sub ← COMPOSE(sub, sub')
    paths ← APPLY-SUB(sub, paths)
    simplified ← APPLY-SUB(sub, simplified)
  if for all \(i\) such that \(1 \leq i \leq \text{SIZE(simplified)}\) we have BOUND\(_\Delta\)(simplified\([i]\))
    return \((\text{paths}, \text{simplified}, \text{sub})\)
  else
    \((\_ , \text{simplified}'', \text{sub}'')\) ← SIMPLIFY\(_{p,\Delta}(\text{simplified})\)
    sub ← COMPOSE(sub, sub'')
    paths ← APPLY-SUB(sub, paths)
    return \((\text{paths, simplified}'', \text{sub})\)

Figure 9.10: Pseudocode: Constraint Simplification
**CONSTRUCT-WITNESSES**$_{\Delta}(constraints) =$

$\begin{align*}
\text{sub} & \leftarrow \text{NIL} \\
\text{for } i & \leftarrow 1 \text{ to } \text{SIZE}(constraints) \\
A & \leftarrow B \leftarrow \text{constraints}[i] \\
\text{if } A \text{ is a type variable } X \text{ and } X \notin \text{tv}(\Delta) \\
\quad & \text{if } \text{sub}[X] = \text{undefined} \\
\quad & \text{sub}[X] \leftarrow \text{Bottom} \\
\text{if } B \text{ is a type variable } X \text{ and } X \notin \text{tv}(\Delta) \\
\quad & \text{if } \text{sub}[X] = \text{undefined} \\
\quad & \text{sub}[X] \leftarrow A \\
\text{else} \\
\quad & \text{sub}[X] \leftarrow A \cup \text{sub}[X] \\
\text{for each } X & \in \text{DOMAIN}(\text{sub}) \\
\text{if } X & \in \text{FREE-VARS}_{\Delta}(\text{sub}[X]) \\
\quad & \text{return error} \\
\text{for each } Y & \in \text{DOMAIN}(\text{sub}) \\
\quad & \text{sub}[Y] \leftarrow \text{APPLY-SUB}([\text{sub}[X]/X], \text{sub}[Y]) \\
\text{if for each } X & \in \text{DOMAIN}(\text{sub}) \text{ we have } \text{FREE-VARS}_{\Delta}(\text{sub}[X]) = \emptyset \\
\quad & \text{return sub} \\
\text{else} \\
\quad & \text{return error}
\end{align*}$

Figure 9.11: Pseudocode: Witness Construction

Algorithm are mutually recursive.

After each input constraint has been processed in this way, the constraint simplification algorithm applies the accumulated substitution to the accumulated list of paths and the accumulated list of simplified constraints. Notice that this may cause a simplified constraint to no longer be in simplified form. Therefore, the constraint simplification algorithm tests whether the accumulated list of simplified constraints are in simplified form. If so, then the triple of the accumulated list of paths, the accumulate list of simplified constraints, and the accumulated substitution is returned. Otherwise, a recursive call is made and the accumulated list of simplified constraints are passed to the constraint simplification algorithm. The results of the recursive call are combined with the accumulated results and returned by the constraint simplification algorithm.
\text{CHECK-WITNESSES}_{\Delta}(\text{constraints}) =
\text{for } i \leftarrow 1 \text{ to } \text{SIZE}(\text{constraints})
\quad \text{if SEARCH}_{\Delta}(\text{constraints}[i]) \text{ returns error}
\quad \quad \text{return false}
\text{return true}

Figure 9.12: Pseudocode: Witness Checking

\textbf{Algorithm: Constraint Solving Pseudocode}

Witnesses for free type variables are constructed by passing simplified constraints to the witness construction algorithm shown in Figure 9.11. In this algorithm, we denote the type that is substituted for \( X \) by substitution \( \text{sub} \) as \( \text{sub}[X] \). If a substitution \( \text{sub} \) does not substitute for a type variable \( X \) then:

\[ \text{sub}[X] = \text{undefined}. \]

The empty substitution \( \text{NIL} \) is undefined for all type variables. That is, for every type variable \( X \):

\[ \text{NIL}[X] = \text{undefined}. \]

Also, the notation:

\[ \text{sub}[X] \leftarrow A \]

is used to assign \( A \) to \( \text{sub}[X] \). After this assignment, the substitution \( \text{sub} \) is defined to substitute type \( A \) for type variable \( X \).

Given a list of simplified constraints, this algorithm returns a substitution, which replaces each free type variable with the union of its lower bounds. The algorithm is composed of three loops. The first loop constructs the union of the lower bounds of each free type variable. The second loop substitutes these unions for occurrences of the variables. Notice that the second loop returns an error if a recursive constraint is found. The third loop checks for occurrences of free type variables in the constructed witnesses. If a free type variable is found then \text{error} is returned. Otherwise, the constructed witnesses are returned. This check ensures that a witness is constructed for every free type variable.
\begin{verbatim}
SOLVE\(_{p,\Delta}(\text{constraints}) = \)
\((\text{paths}, \text{simplified}, \text{sub}) \leftarrow \text{SIMPLIFY}_{p,\Delta}(\text{constraints})\)
\(\text{sub}' \leftarrow \text{CONSTRUCT-WITNESSES}_{\Delta}(\text{simplified})\)
\(\text{simplified} \leftarrow \text{APPLY-SUB}(\text{sub}', \text{simplified})\)
\(\text{paths} \leftarrow \text{APPLY-SUB}(\text{sub}', \text{paths})\)
\(\text{sub} \leftarrow \text{COMPOSE}(\text{sub}, \text{sub}')\)
\(\text{if } \text{CHECK-WITNESSES}_{p,\Delta}(\text{simplified})\)
\(\quad \text{return } (\text{paths}, \text{sub})\)
\(\text{else}\)
\(\quad \text{return } \text{error}\)
\end{verbatim}

Figure 9.13: Pseudocode: Constraint Solving

The auxiliary algorithm that verifies whether witnesses are, in fact, valid is shown in Figure 9.12. The construction of a witness ensures that the witness satisfies the lower bounds of the corresponding type variable. However, the upper bounds may not be satisfied. The search algorithm is used to guarantee that a witness satisfies the upper bounds. Because none of the input constraints contain free type variables, the search algorithm acts as a subtype checker (see Lemma 17 in Appendix D for more details). Each input constraint is passed to the search algorithm. If the algorithm does not return an error then the upper bounds are satisfied.

The driver algorithm, SOLVE, is shown in Figure 9.13. This algorithm simplifies the input constraints by calling SIMPLIFY. Once all constraints have been simplified, CONSTRUCT-WITNESSES is called to construct witnesses for the free type variables. After constructing witnesses, SOLVE checks whether the witnesses are valid, and, if so, substitutes the witnesses into the paths returned by SIMPLIFY.

The constraint solving algorithm is sound. The following theorem is proved in Appendix D.

**Theorem 5** (Soundness of Constraint Solving). *If:*

\(p\) is a well-typed program, and
\(\Delta\) is a type variable environment, and
\(\text{constraints}\) is a list of constraints, and
for each \(i\) such that \(1 \leq i \leq \text{SIZE}((\text{constraints}))\),
\(\text{constraints}[i]\) is of the form \(A_i <: B_i\), and
\(\text{SOLVE}_{p,\Delta}(\text{constraints})\) returns \((\text{paths}, \text{sub})\)
then:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \),

\( p; \Delta \vdash \text{APPLY-SUB}(\text{sub}, A_i) \prec \text{APPLY-SUB}(\text{sub}, B_i) \).

**Restrictions for Termination**

Various restrictions can be applied to ensure termination of the constraint solving algorithm. The most straightforward way to ensure termination is to add an iteration limit to the algorithm. However, this leads to cryptic error messages; an in-depth knowledge of the constraint solving algorithm (and the constraints to be solved) is required for the programmer to understand why type checking fails. In addition, remember that constraint solving occurs at run time in HTV1. A program may exceed the constraint solver’s iteration limit at run time, even though it did not exceed the limit at compile time.

A better restriction is one that is related to the language itself, rather than to the algorithm. Not only does such a restriction help a programmer to reason about constraint solving, but it allows for a more portable language. Since the restriction is based on the language, termination will not be dependent on a particular implementation of the constraint solver. That is, if a program is well typed according to one implementation of the language then it is guaranteed to be well typed according to all other implementations.

One such restriction is a limit on the nesting depth of generic types. We define the nesting depth of a type in HTV1 as follows:

\[
\text{depth}(A) = \begin{cases} 
1 & \text{if } A = X \\
1 & \text{if } A = \text{Object} \\
1 & \text{if } A = \text{Bottom} \\
1 + \max(\text{depth}(A_1), \ldots, \text{depth}(A_n)) & \text{if } A = S[A_1 \ldots A_n] \\
\max(\text{depth}(A), \text{depth}(B)) & \text{if } A = A \cup B 
\end{cases}
\]

For any given nesting depth, there is a finite number of distinct types (modulo renaming of free type variables) for a given program and type variable environment (see Lemma 18 in Appendix E for a proof). Because constraints are simply pairs of types, there is also a finite number of con-
straints (modulo renaming of free type variables) for a given program and type variable environment. Therefore, the constraint solving algorithm can be made to terminate on all inputs with two slight modifications:

1. the type hierarchy search algorithm must record its input constraint and fail if a duplicate constraint is found, and
2. the type hierarchy search algorithm must prevent duplicate types in the path of a queue element.

Both modifications detect cycles in the computation of the constraint solving algorithm. The first modification detects a cycle in the sequence of constraints that are solved. The second modification detects a cycle in the sequence of types that are traversed.

Appendix E shows the modifications to the pseudocode of the constraint solving algorithm. We refer to the terminating version of SOLVE as T-SOLVE. Appendix E also provides a proof of the following theorem.

**Theorem 6 (Termination of Constraint Solving).** If:

- there is a fixed nesting depth for types, and
- \( p \) is a well-typed program, and
- \( \Delta \) is a type variable environment, and
- constraints is a list of constraints

then:

\[
\text{T-SOLVE}_{p, \Delta}(\text{constraints}) \text{ terminates.}
\]

In practice, we do not anticipate a nesting depth limit to be a serious impediment to expressiveness, especially if the maximum depth is programmer-configurable. Many existing compilers, including the C# compiler [28], include a fixed nesting depth. Using PLT Redex, we have shown that many programs terminate without any such restriction. We have developed a test suite of approximately 180 tests that exercise the constraint solving algorithm, none of which loop infinitely. This evidence supports the conjecture that a tunable metric, like a nesting depth limit, will not be a serious impediment to expressiveness in practice.
9.3.2 Inferring Paths

The constraint solving algorithm returns a path between two given types. However, when inferring path annotations for method calls or checking valid overriding, only the initial type in the path is known. In these cases, a path must be inferred by searching the hierarchy of trait and object definitions.

When inferring path annotations for method calls, the search must start from the definition corresponding to the static type of the receiver. It must end at the definition enclosing the inherited method. When checking valid overriding, the search must start from the definition enclosing the overriding method and end at the definition enclosing the overridden method.

Recall that traits may extend themselves as long as the restrictions from Section 8.6 are satisfied. This means the trait and object definition hierarchy may contain cycles. To ensure the search of this hierarchy will terminate, the trait and object definitions that have been visited are recorded. By recording these definitions, repeatedly visiting a definition (i.e., traversing a cycle) can be avoided.

Also, recall that at most one method definition of a given name can be inherited by a trait or object. This restriction, combined with the fact that traits and objects are recorded, ensures that the order of traversal is irrelevant. Since there is only one path to the method, and all paths terminate, the method must be found eventually.

Once the method has been found, a path in the type hierarchy must be computed. During the search, a skeleton of this path is constructed. That is, a path without witnesses for hidden type variables is constructed. Ordinary type variables are substituted with the corresponding type arguments, but hidden type variables are not substituted. Instead, the constraints on hidden type variables are collected. After the search is complete, the constraint solving algorithm is used to find witnesses for the hidden type variables that occur in the path.

9.3.3 The Rest of the Type Checker

The rest of the type checker for HTV1 is, for the most part, a straightforward encoding of the static semantics. Program type checking simply delegates the checking to the definition type checker and the expression type checker. Definition typing checking requires single inheritance checking,
type well-formedness checking, and method type checking. Single inheritance checking traverses
the hierarchy of the trait and object definitions in much the same way that path inference does.
Type well-formedness checking requires finding witnesses for hidden type variables. By passing
the bounds on the hidden type variables to the constraint solving algorithm, these witnesses are
inferred. Method type checking requires well-formedness checking, subtype checking, override
checking, and expression type checking. As discussed, subtype checking is done by the constraint
solving algorithm. Override checking requires path inference and subtype checking. Expression
typing requires well-formedness checking and subtype checking.
Chapter 10

Semantics of Conditional Extension

Conditional extension can be added to a language with hidden type variables. This chapter presents the semantics of CE by adjusting the formal semantics of the intermediate language for HTV1. In fact, very few changes need to be made to the semantics of HTV1.

Recall that HTV0 and HTV1 require an intermediate language for two reasons:

1. the syntax of method calls must store witnesses for hidden type variables, and
2. union types and Bottom are required for unique most-specific witnesses.

Also, recall that conditional constraints in CE do not introduce hidden type variables. Instead, conditional constraints are used to constrain existing type variables. Therefore, conditional extension in CE does not require an intermediate language that differs from the source language. However, by extending the intermediate language of HTV1, we show that conditional extension can be smoothly integrated into a language with hidden type variables.

10.1 Syntax of CE

The differences between the syntax of the intermediate language for CE and that of HTV1 are, not surprisingly, the definitions of traits and objects. Figure 10.1 shows these differences. Recall that Figures 8.3 and 8.22 show the syntax of the intermediate language for HTV1. The extended types in both trait and object definitions contain where clauses, which specify the conditional constraints.
trait definition

\[ \text{td} ::= \text{trait } T \llbracket \overrightarrow{\text{bnd}} \rrbracket \text{ extends } \{ M \text{ where } \{ \overrightarrow{\text{bnd}} \} \text{ where } \{ \overrightarrow{\text{bnd}} \} \text{ md } \text{ end} } \]

object definition

\[ \text{od} ::= \text{object } O \llbracket \overrightarrow{\text{bnd}} \rrbracket (x:A) \text{ extends } \{ \overrightarrow{\text{bnd}} \text{} \text{where } \{ \overrightarrow{\text{bnd}} \} \text{ where } \{ \overrightarrow{\text{bnd}} \} \text{ md } \text{end} } \]

Figure 10.1: Difference between the Syntax of Intermediate Languages for CE and HTV1

Paths: \[ p; \Delta \vdash A <:: A \]

\[ S[\overrightarrow{\text{bnd}}_1] \text{ extends } \{ M \text{ where } \{ \overrightarrow{\text{bnd}}_2 \} \text{ where } \{ \overrightarrow{\text{bnd}}_3 \} \text{ end } \in p \]

\[ \overrightarrow{\text{bnd}}_1 = \overrightarrow{X} \quad \text{convert}(\overrightarrow{\text{bnd}}_1) = \overrightarrow{J}_1 <:: \overrightarrow{J}_2 \]

\[ \text{convert}(\overrightarrow{\text{bnd}}_2) = \overrightarrow{K}_1 <:: \overrightarrow{K}_2 \]

\[ \overrightarrow{\text{bnd}}_3 = \overrightarrow{Y} \quad \text{convert}(\overrightarrow{\text{bnd}}_3) = \overrightarrow{L}_1 <:: \overrightarrow{L}_2 \]

\[ p; \Delta \vdash A \text{ ok} \quad p; \Delta \vdash [A/X][B/Y] \overrightarrow{J}_1 <:: [A/X][B/Y] \overrightarrow{J}_2 \]

\[ p; \Delta \vdash B \text{ ok} \quad p; \Delta \vdash [A/X][B/Y] \overrightarrow{K}_1 <:: [A/X][B/Y] \overrightarrow{K}_2 \]

\[ p; \Delta \vdash S[\overrightarrow{A}] \text{ ok} \quad p; \Delta \vdash S[\overrightarrow{A}] <:: [A/X][B/Y]M_i \]

Figure 10.2: Difference between the Path Rules for CE and HTV1

10.2 Static Semantics of CE

The static semantics of HTV1 requires a few small adjustments to allow for conditional extension.

The main change occurs in rule \[\text{[P-Ext]}\]. The HTV1 version of this rule is shown in Figure 8.24. Figure 10.2 shows this rule adjusted for CE.

Figure 10.2 makes use of the following notational convention: \(\overrightarrow{\text{bnd}}\) is short-hand notation for \(\overrightarrow{\text{bnd}}_1 \ldots \overrightarrow{\text{bnd}}_n\). Therefore, rule \[\text{[P-Ext]}\] denotes the \(i^{th}\) element of \(\overrightarrow{\text{bnd}}_2\) by \(\overrightarrow{\text{bnd}}_{2i}\).

The CE version of rule \[\text{[P-Ext]}\] requires an additional subtyping check. For a type \(S[\overrightarrow{A}]\) to extend another type \(M_i\) that is listed in the extends clause of trait \(S\), the conditional constraints \(\overrightarrow{\text{bnd}}_{2i}\) on \(M_i\) must be satisfied. This is enforced by the second to last antecedent of the rule. If this subtyping relation holds then the types \(S[\overrightarrow{A}]\) and \([A/X][B/Y]M_i\) form a valid path.
Definition typing: \[ p \vdash d \text{ ok} \]

\[
\text{[T-TraitDef]} \quad \begin{align*}
\text{bnd}_1 &= X \to J \\
\text{bnd}_2 &= Y \to K \\
\text{bnd}_3 &= Z \to L \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \ \text{convert}(\text{bnd}_3) \\
\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) &= \emptyset \quad i \neq j \quad i, j \in \{1, \ldots, |M|\} \\
p;\Delta' \vdash \bar{J} \text{ ok} & \quad p;\Delta' \vdash \bar{M} \text{ ok} & \quad p;\Delta' \vdash \bar{K} \text{ ok} & \quad p;\Delta' \vdash \bar{L} \text{ ok} \\
p;\Delta;\Delta';\text{self} : T[\llbracket X \rrbracket];T[\llbracket X \rrbracket] \vdash \bar{md} \text{ ok} \\
p \vdash \text{trait } T[\llbracket \text{bnd}_1 \rrbracket] \text{ extends } \{ M \ \text{where } \{ \text{bnd}_2 \}\} \text{ where } \{ \text{bnd}_3 \} \to \bar{md} \text{ end ok}
\end{align*}
\]

\[
\text{[T-ObjectDef]} \quad \begin{align*}
\text{bnd}_1 &= X \to J \\
\text{bnd}_2 &= Y \to K \\
\text{bnd}_3 &= Z \to L \\
\Delta &= \text{convert}(\text{bnd}_1) \\
\Delta' &= \text{convert}(\text{bnd}_1) \ \text{convert}(\text{bnd}_3) \\
\text{inherited}_p(M_i) \cap \text{inherited}_p(M_j) &= \emptyset \quad i \neq j \quad i, j \in \{1, \ldots, |M|\} \\
p;\Delta' \vdash \bar{J} \text{ ok} & \quad p;\Delta \vdash \bar{A} \text{ ok} & \quad p;\Delta' \vdash \bar{M} \text{ ok} & \quad p;\Delta' \vdash \bar{K} \text{ ok} & \quad p;\Delta' \vdash \bar{L} \text{ ok} \\
p;\Delta;\Delta';\text{self} : O[\llbracket X \rrbracket] ; x : A; O[\llbracket X \rrbracket] \vdash \bar{md} \text{ ok} \\
p \vdash \text{object } O[\llbracket \text{bnd}_1 \rrbracket](x:A) \text{ extends } \{ M \ \text{where } \{ \text{bnd}_2 \}\} \text{ where } \{ \text{bnd}_3 \} \to \bar{md} \text{ end ok}
\end{align*}
\]

Figure 10.3: Trait and Object Definition Typing Rules for CE

The rules for trait and object definition typing in HTV1 must also be adjusted. Figure 8.28 shows the trait and object definition typing rules for HTV1. Figure 10.3 shows the updated rules. These rules must check the well-formedness of the bounds in the conditional constraints. Notice that type well-formedness is checked using a type variable environment that includes ordinary type variables and hidden type variables, but conditional constraints are not included. This ensures that new type variables are not introduced by the conditional constraints.

The definition of the auxiliary function for determining inherited methods requires a minor cosmetic change. Figure 8.11 shows the function for determining inherited methods in HTV1. This function is adjusted for CE in Figure 10.4. Because the definition of this function includes a trait definition, the extends clause of the trait definition must be updated to include conditional constraints. The conditional constraints play no role in the function and therefore are ignored by
Inherited methods: \( \text{inherited}_p(M) = \{ \overrightarrow{m} \} \)

\[
inherited_p(M) =
\begin{cases}
\emptyset & \text{if } M = \text{Object} \\
\{ \overrightarrow{m} \} \cup \inherited_p(M_1) \cup \ldots \cup \inherited_p(M_n) & \text{if } M = T[\overrightarrow{A}] \\
& \text{and trait } T \text{ extends } \{ \overrightarrow{M} \} \text{ end } \in p \\
& \text{and } \overrightarrow{m} = \text{name}(\overrightarrow{md}) \\
& \text{and } |\overrightarrow{M}| = n
\end{cases}
\]

Figure 10.4: Inherited Methods for CE

using \( \_ \). In other words, when determining methods that are inherited by a trait or object, we assume all the conditional constraints are satisfied. Conceptually, this function is no different than that of HTV1. However, we include its definition for completeness.

10.3 Dynamic Semantics of CE

The dynamic semantics of the intermediate language for CE is mostly identical to that of HTV1. The one minor difference between the two is the definition of the path comparison relation. Figure 8.15 defines the path comparison relation for HTV1. Because this definition uses a trait or object definition, the extends clause of the trait or object definition must be updated to include conditional constraints. The conditional constraints play no role in the function and therefore are ignored by using \( \_ \) in Figure 10.5. Again, this definition is included, despite no conceptual differences, for completeness.

10.4 Type Soundness

The semantics of CE is type sound. The proof is almost identical to the soundness proof of HTV1 in Appendix A. Only the case for rule \([P-\text{EXT}]\) in Lemma 2 must be adjusted. This case is proved
Path comparison: $\vec{A} \prec \vec{A}$

$$\vec{A} \prec \vec{B}$$

if $p; \emptyset \vdash \vec{A}$ path-ok
and $p; \emptyset \vdash \vec{B}$ path-ok
and either $|\vec{A}| < |\vec{B}|$
and $|\vec{A}| = |\vec{B}|$
and $p; \emptyset \vdash \vec{E} <: \vec{F}$
where $A_i = S[\vec{C}]$ and $B_i = S[\vec{D}]$
and $1 \leq i \leq |\vec{A}| - 1$
and $S[bnd_1]$ extends $\{M\}$ where $\{bnd_2\}$ end $\in p$
and $bnd_1 = \vec{X}$ and $bnd_2 = \vec{Y}$
and $[\vec{C}/\vec{X}][\vec{E}/\vec{Y}]M_j = A_{i+1}$ and $[\vec{D}/\vec{X}][\vec{F}/\vec{Y}]M_j = B_{i+1}$
and $1 \leq j \leq |\vec{M}|$

Figure 10.5: Path Comparison Relation for CE

using the induction hypothesis and rule [S-PATH].
Chapter 11

Implementing Conditional Extension

The semantics of CE has been implemented using PLT Redex. The implementation of HTV1, discussed in Chapter 9, can be easily adjusted for conditional extension. The adjustments are described in this chapter.

11.1 Syntax of CE in PLT Redex

The implementation of the syntax for the intermediate language of HTV1, which is shown in Figure 9.1, must be adjusted in order to implement the intermediate language of CE. In particular, the implementation of trait and object definitions must be augmented to include a list of bounds with each extended type. The new implementation, which is a straightforward encoding of the specification given in Figure 10.1, is shown in Figure 11.1. Except for the implementation of trait and object definitions, the implementation of the intermediate language for HTV1 is unchanged.

11.2 Evaluation Rules of CE in PLT Redex

The implementation of the evaluation rules of CE is identical to that of HTV1. However, notice that the path update function (Figures 8.17 and 8.31) calls the constraint solver indirectly via the path extension function (Figure 8.16). The next section will discuss adjustments to the constraint solving algorithm for CE. These adjustments will indirectly affect the evaluation rules of CE.
; trait definitions
(td (trait T (bnd ...)
    ((M (bnd ...)) ...)
    (bnd ...)
    (md ...)))

; object definitions
(od (object O (bnd ...)
    ((x A) ...)
    ((M (bnd ...)) ...)
    (bnd ...)
    (md ...)))

...
else if $C$ is of the form $S[D]$

if $p$ contains a declaration of the form

$S'[X,K]$ extends $\{M \text{ where } \{bnd\}\}$ where $\{Y,L\}$ end

where $S' = S$

$Z \leftarrow \text{FRESH-VARS}(Y)$

$\text{sub} \leftarrow \text{COMPOSE}([D/X], [Z/Y])$

$\text{constraints}' \leftarrow \text{APPEND} (\text{convert}(X,K), \text{convert}(Y,L))$

$\text{constraints}' \leftarrow \text{APPEND} (\text{constraints}, \text{constraints}')$

$\text{constraints}' \leftarrow \text{APPLY-SUB}(\text{sub}, \text{constraints}')$

for $i \leftarrow 1$ to $\text{SIZE}(M)$

$\text{path}' \leftarrow \text{APPEND}(\text{path}, \text{CONS}(\text{APPLY-SUB}(\text{sub}, M_i), \text{NIL}))$

$\text{constraints}'' \leftarrow \text{APPEND}(\text{constraints}', \text{APPLY-SUB}(\text{sub}, \text{convert}(\text{bnd}_i)))$

$\text{queue} \leftarrow \text{ENQUEUE}((\text{path}', \text{constraints}''), \text{queue})$

$\text{uppers} \leftarrow \text{UPPER-BOUNDS}_\Delta(C)$

for $i \leftarrow 1$ to $\text{SIZE}(\text{uppers})$

$\text{path}' \leftarrow \text{APPEND}(\text{path}, \text{CONS}(\text{uppers}[i], \text{NIL}))$

$\text{queue} \leftarrow \text{ENQUEUE}((\text{path}', \text{constraints}''), \text{queue})$

else

return error

This pseudocode implements the sixth branch of the conditional in the body of the while loop in Figure 9.8. Of this pseudocode, only the third line:

$S'[X,K]$ extends $\{M \text{ where } \{bnd\}\}$ where $\{Y,L\}$ end

and the twelfth and thirteenth lines:

$\text{constraints}'' \leftarrow \text{APPEND}(\text{constraints}', \text{APPLY-SUB}(\text{sub}, \text{convert}(\text{bnd}_i)))$

$\text{queue} \leftarrow \text{ENQUEUE}((\text{path}', \text{constraints}''), \text{queue})$

differ from the sixth branch given in Section 9.3.1. The third line is adjusted for the new syntax of trait and object definitions. The twelfth and thirteenth lines are added to collect the conditional constraints.

If this branch is executed then the path has reached a trait or object type $S[D]$. As before, a new element of the queue is created for each supertype of $S[D]$. The supertypes include the types in the extends clauses of the $S$ and the upper bounds of $S[D]$ in the type variable environment. As before, the path for the new queue element is the result of adding the supertype to the current path.
For extended types, the constraints for the new queue element are the combination of the current constraints, the constraints on the type variables of the trait or object, and the conditional constraints of the extended type. For upper bounds, the constraints for the new queue element are the current constraints.

The soundness and termination results from Section 9.3.1 continue to hold for the updated constraint solving algorithm. In fact, to prove the soundness of the updated algorithm, only Lemma 14 in Appendix D must be adjusted. This lemma must show that the updated sixth branch of the search algorithm (shown above) implements rule [P-EXT] in Figure 10.2. This follows from the induction hypothesis and Lemma 15.

To prove that the updated algorithm terminates, only Lemma 19 in Appendix E must be adjusted. In particular, this lemma must show that the updated sixth branch of the search algorithm (shown above) terminates. This follows from the fact that all three lines that were added to the sixth branch must terminate.

11.3.2 Inferring Paths

Recall from Section 9.3.2 that paths must be inferred for method calls and override checking. This inference requires a search of the trait and object definition hierarchy. Also recall that constraints on hidden type variables are collected during this search. After the search is complete, the constraints are solved to provide witnesses for the hidden type variables.

The constraint solver for CE is permitted to search from a trait or object definition to an extended trait definition only if the conditional constraints on the extended trait are satisfied. Therefore, during the search, the constraint solver must collect the conditional constraints as well as the constraints on hidden type variables. After the search, all the collected constraints are passed to the constraint solver. If the constraint solver succeeds then the path constructed by the search contains valid extensions.
11.3.3 The Rest of the Type Checker

The remaining changes to the implementation of HTV1 occur in the trait and object definition type checker. The check for single inheritance must be adjusted to account for the new form of extends clauses. As discussed in Section 10.2, this check is conceptually no different than the corresponding check in HTV1.

The trait and object definition type checker itself must account for the new form of extends clauses. The type variable environments must be adjusted to include bounds on ordinary type variables and hidden type variables, but not the conditional constraints. Also, the types in the conditional constraints must be checked for well formedness.
Chapter 12

Related Work

There have been numerous proposals for adding generics (i.e., parameterized types) to object-oriented languages. A chief difficulty is integrating parametric and subtype polymorphism [41]. In this chapter, we discuss the most prominent of these proposals and how they compare to the languages developed in this dissertation.

12.1 System $F_\ll$

One of the most widely accepted foundational studies of programming languages that combine parametric and subtype polymorphism is System $F_\ll$ [40]. $F_\ll$ extends System F (the polymorphic lambda calculus) with subtyping. Although $F_\ll$ provides a formal calculus to study the interactions between these two forms of polymorphism, the design space is very large and $F_\ll$ does not cover all of it. In particular, because $F_\ll$ is structurally subtyped, there is no facility for defining a new type as there is in a nominal type system. In addition, $F_\ll$ does not allow a subtype relationship between types with a different number of type parameters. A language with hidden type variables allows both of these relationships. Finally, there is nothing comparable to conditional extension in $F_\ll$. 
12.2 Pointwise Subtyping

In most object-oriented languages, the only subtype relation that can exist between parameterized types is pointwise subtyping, where a parametric type extends another parametric type with the same type parameters. For example, \texttt{List[String]} is a subtype of \texttt{Collection[String]} if the parametric definition of \texttt{List[X]} is defined to extend the parametric definition of \texttt{Collection[X]}. However, pointwise subtyping cannot express subtype relationships between two different instantiations of one type name. For example, one cannot express that \texttt{List[String]} is a subtype of \texttt{List[Object]}.

Pointwise subtyping is therefore strictly less expressive than a language with hidden type variables.

Objective Caml [31] supports parametric polymorphism over a class system with structural subtyping, resulting in significantly different properties than a language with nominal subtyping. For example, bounds on type parameters are not supported in Objective Caml. Eiffel [36] supports covariant method parameter types, causing the type system to be unsound [16]. CLU [32] provides constraints on type parameters declared in “where clauses”, which were adapted for Theta [19] and a Java extension called PolyJ [8]. Each constraint in a where clause identifies the names and types of required methods for the type parameters. These where clauses cannot declare new, hidden, type variables; they are used only for constraining pre-existing parameters.

Pizza [39] and GJ [10] are Java extensions with generic types that do not require modification of the Java virtual machine. Both Pizza and GJ use a type-erasure semantics, where type arguments are removed before running the program. A type-erasure semantics (which was ultimately adopted for the Java 2 Standard Edition Development Kit version 5.0 (J2SDK5.0)) was chosen for better compatibility with legacy code and run-time performance benefits; however, this choice severely restricts the expressiveness of the language, preventing safe type-dependent operations with generic types. In addition, only invariant subtyping for parameterized types is supported (except for arrays, which have dynamically checked covariant subtyping, as in the Java Programming Language). GJ has been formalized in a core calculus (with severe restrictions on statically valid casts) and proved type sound [24].
12.3 Variance Types

Variance types are parameterized types with subtyping relationships between their different instantiations. In some languages, notation (+ for covariance and - for contravariance, for example) for expressing variance types is built into the language as primitives. The use of these primitives is restricted: method parameter types cannot be annotated as covariant and method return types cannot be annotated as contravariant.

When defining variance types with language primitives, there are two competing schools of thought. Languages with declaration-site variance, such as Eiffel [36], POOL [7], Strongtalk [9], NextGen [13], and Scala [37], attach variance annotations to type parameters when they are declared. Languages with use-site variance, such as structural virtual types [43] and variant parametric types (VPTs) [25], attach variance annotations to instantiations of type parameters.

With declaration-site variance, the properties of a type are declared up-front and in one place. However, the restrictions on the placement of variance annotations often require multiple definitions of a type. For example, a covariant list must be read-only. In order to add an element to a list, a separate, invariant, definition must be provided.

Use-site variance avoids the difficulties of declaration-site variance by delaying the variance decision until the type is instantiated. The instantiation of a type parameter determines its variance. This approach allows a type to be designed without variance in mind. However, for type soundness to hold, the methods that are available to an instantiated type must be restricted. In particular, a method that is defined in a type, but, as a result of instantiation, contains a covariant parameter type or a contravariant return type is not available for use. In this way, use-site variance allows a single type definition to be used in all contexts. However, reasoning about types instantiated with variance annotations can be difficult.

Hidden type variables can encode declaration-site variance as a special case. This is shown by the covariant list example in Section 4.2. Indeed, that example is more general than can be encoded with variance annotations. In particular, the cons method can “expand” the type of the list. That is, the cons method can return a list whose type is more general than the receiver type. It is not
possible to define such a method using variance annotations alone. The encoding of such a method (in Java syntax) using declaration-site variance would be:

```java
class List<+X> extends Object {
    List<X> cons(X x) { return new Cons<X>(x, self); }
    // static error
}
class Empty<X> extends List<X> {}
class Cons<X> extends List<X> {
    X head;
    List<X> rest;
    Cons(X hd, List<X> rs) {
        self.head = hd;
        self.rest = rs;
    }
}
```

This definition would be rejected because the parameter type of method `cons` is covariant. Using use-site variance, the definition of such a method would be the same as above, but without the variance annotation. The use of this method would be encoded (again in Java syntax) as:

```java
List<+Number> l = new Empty<Integer>().cons(new Integer(1));
l.cons(new Float(1.2)); // static error
```

The second method call is rejected because the `cons` method is unavailable to a receiver of type `List<+Number>`. In Section 12.5, we show that the combination of generic methods and lower bounds on type parameters can be used to express such a method.

Using variance annotations, there is no way to express a non-parametric base case of a parametric inductive type, as shown in Section 4.1. In addition, there is no way to define variance relations between different types, as shown in Section 4.3. Variance annotations also do not allow a general form of self extension. It is not possible to conditionally extend a type using variance annotations.

---

1We avoid reliance on the autoboxing functionality of J2SDK5.0 so as not to obfuscate the cause of the problem.
12.4 Wildcards

Wildcards [45] are an alternative to VPTs that were incorporated into J2SDK5.0. Like VPTs, wildcards provide a mechanism for covariant and contravariant instantiations of type parameters at use sites. But, instead of annotating type instantiations, a special type argument $\forall$ ranging over all possible type arguments is introduced. The type $C<\forall \text{ extends } T>$ corresponds to the covariant type $C<+T>$ in VPTs, while $C<\forall \text{ super } T>$ corresponds to the contravariant type $C<-T>$, and $C<\forall>$ to the bivariant type $C<\star T>$ (i.e., a type which is both covariant and contravariant).

By simulating variance types, wildcards can be used to infer more precise types than would be possible in a language with pointwise subtyping. Consider the following example from Torgersen, et. al. [45]:

\[<T> \ T \text{ choose}(T \ a, \ T \ b) \ \{ \ \cdots \ \} \]
\[\text{Set<Integer>} \ \text{intSet} = \cdots \]
\[\text{List<String>} \ \text{stringList} = \cdots \]
\[\text{choose} \ (\text{intSet}, \ \text{stringList}) \]

In the method call choose(intSet, stringList) above, a type for $T$ must be found, and it must be a supertype of both Set<Integer> and List<String>. In pointwise subtyping systems, such as GJ, Collection<Object> is not a supertype of both Set<Integer> and List<String> because Set and List are not covariant. Instead, the type Object is determined to be their most precise supertype. However, with wildcards, the more precise type Collection<$\forall$> may be inferred. A language with hidden type variables can provide the same precision. The above example can be written as:

\[
\text{trait Set[$X$ extends $Y$] extends } \{ \text{Set[$Y$], Collection[$X$]} \}
\text{ where } \{ Y \text{ extends Object} \}
\text{end}
\text{trait List[$X$ extends $Y$] extends } \{ \text{List[$Y$], Collection[$X$]} \}
\text{ where } \{ Y \text{ extends Object} \}
\text{end}
\text{object O extends Object}
\text{choose[$Z$]($x:Z,y:Z$):Z = \ldots}
\text{end}
\]
object IntSet extends Set[\mathbb{Z}]
    ...
end

object StrList extends List[String]
    ...
end

O.choose[Collection[Object]][(IntSet, StrList)]

Wildcards can be interpreted as a restricted form of existential types. The type \( C<?> \) corresponds to \( \exists X. C<X> \). Hidden type variables on the other hand are universally quantified. For this reason, it is not possible to encode the covariant list example from Section 4.2 in a language with wildcards. As an illustration of this, notice that only \texttt{null} can be written to the following list:

```java
List<?> extends Number> l =
    new Empty<Integer>().cons(new Integer(1));
l.cons(null); // OK
l.cons(new Float(1.2)); // static error
```

The third line causes an error because \( l \) is a list of some subtype of \texttt{Number}; the type system does not know the actual type of the list’s elements. In other words, precision is lost in order to define a covariant list type. In a language with hidden type variables, precision need not be lost.

It is also not possible to extend infinitely many instantiations of another type. For example, the encoding of the ML option type (as in Section 4.1) results in a static error:

```java
class None extends Option<?> {
    ...
} // static error
```

Although wildcards have been incorporated into J2SDK5.0, their use has not been proven type sound. The Wild FJ type system [44] has been proposed to investigate the type soundness of wildcards, but the subtyping rules in Wild FJ differ from those specified in \textit{The Java Language Specification} (JLS) [22]. In addition, Wild FJ has not been proved type sound.
12.5 Scala

Scala [37] is an object-oriented language that shares a number of features with Fortress. It can define both objects and traits, as well as parameterized types. Unlike Fortress, Scala provides variance annotations on type parameters for covariance and contravariance. Scala also allows upper and lower bounds on type parameters.

With these features, Scala is able to encode a number of the examples discussed in this dissertation. For example, the covariant list trait from Section 4.2 can be encoded in Scala as follows:

```scala
trait List[+X] {
  def cons[Y :> X](y:Y):List[Y] = new Cons[Y](y,this);
}
```

In addition, a non-parametric empty list, as discussed in Section 4.1, can be defined as follows:

```scala
trait List[+X] {}
object Empty extends List[Nothing] {}
```

This definition makes use of the Scala type `Nothing`, which is a subtype of all other types.

However, the Scala type system cannot encode all possible uses of hidden type variables. For example, the above technique used to define a non-parametric empty list requires the list type to be covariant, and would not apply to an invariant list. As another example, there is no way to eliminate extraneous type parameters, as shown in Section 4.6. Scala’s type system also does not allow a type to be conditionally extended.

In Scala, as in the Java Programming Language, a type-erasure semantics is used. Thus, type-dependent operations on parameterized types are unsafe. For example, consider the following Scala code:

```scala
var l : List[int] = Nil.::("foo").asInstanceOf[List[int]]
var i : int = l.head + 4
```
The first assignment will succeed without complaint, but the second will produce a run-time error, whereas the corresponding code in a language with hidden type variables will throw an exception when the run-time cast is invoked:

```scala
trait List[X extends Y] extends List[Y] where {Y extends Object}
  first(): X = self.first()
  cons(y: Y): List[Y] = Cons[Y](y, self)
  cast[Z extends Object](): Z =
    typecase x = self of
    Z ⇒ x
    else ⇒ throw IllegalCast
  end
end
object Cons[X] extends List[X] end
object Empty extends List[X] where {X extends Object} end
Empty.cons("foo").cast[List[Z]]().first() + 4
```

Various subsets of Scala have been formalized [38, 6, 18]; however, these calculi have not considered parametric types.

### 12.6 Cecil

Cecil [14] and its successor, Diesel, have static type systems capable of expressing many of the type relationships expressible in a language with hidden type variables and conditional extension. The programmer can specify subtyping constraints (similar to those in the where clause of this dissertation), and signature constraints (similar to those in the where clauses of Theta) on type variables.

However, Cecil’s type system is merely descriptive: static types are used for compile-time checking, but they have no run-time consequences. Rather, the run-time semantics of a Cecil program is determined by its “dynamically-typed core”, where a “dynamic type” in Cecil consists of an object and its descendents in the inheritance hierarchy. Furthermore, even when the static subtyping hierarchy coincides with the inheritance hierarchy (Cecil, but not Diesel, allows these hierarchies to be defined independently), there is not a one-one correspondence between static types and dynamic
types. One object in the inheritance hierarchy (and thus, one dynamic type) corresponds to every instantiation of a parameterized static type. Therefore, the semantics of Cecil is similar to those of languages with type erasure. In contrast, the languages developed in this dissertation maintain instantiated types at run time, and can therefore perform type-dependent operations that are not possible in Cecil. For example,

```
object O
    m[X extends Object](v: Vector[X]): Z =
    typecase x = v of
        Vector[Z] ⇒ 1
        Vector[String] ⇒ 2
        else ⇒ 3
    end
end
```

The equivalent program in Cecil is statically rejected:

```
method m(a@:i_vector[int]):int { 1 }
method m(a@:i_vector[i_string]):int { 2 }
```

A simplified variant of the Cecil type system has been formalized by Litvinov [33, 34]. This formalization also considers conditional extension, which is called “partial subtyping”.

To the best of our knowledge, there does not exist a terminating type checking algorithm for the Cecil type system presented in [33] and [34]. Litvinov developed a set of restrictions on programs to allow for a terminating type checking algorithm. However, these restrictions severely limit expressivity. One restriction disallows self extension, making covariant and contravariant relationships impossible to declare.

Another restriction requires extensions to be of the form:

```
extend <\bar{X}\backslash\bar{C}> N1<\bar{X}> isa N2<\bar{X}'>
```

where \(\{\bar{X}'\} \subseteq \{\bar{X}\} \). In Cecil syntax, \(\bar{X}\) and \(\bar{X}'\) are lists of type variables, \(\bar{C}\) is a list of constraints, and \(N1\) and \(N2\) are type constructors. The notation \(<\bar{X}\backslash\bar{C}>\) is used to associate constraints \(\bar{C}\) with
type variables $\bar{x}$. This restriction requires all type variables to be parameters of the extending type. In other words, hidden type variables are disallowed. Because hidden type variables are disallowed, the applications of hidden type variables shown in Chapter 4 are also disallowed.\(^2\)

Two other restrictions limit the form of a bound on a type variable. A bound must be either a type variable or a class type that is instantiated with type variables. In addition, a type variable cannot occur more than once in a bound. These restrictions limit the form of bounds in general, and conditional constraints in particular. No such restrictions are imposed on the languages developed in this dissertation.

### 12.7 Generalized Algebraic Data Types

Generalized algebraic data types (GADTs) [26], which subsume first-class phantom types [15] and are a restriction of guarded recursive data types [47], extend the functionality of parameterized algebraic data types of ML and Haskell by allowing type constructors to return specific instantiations of the data type being defined. GADTs have several applications including typed evaluators, generic pretty-printing, and typed LR parsing. GADTs have also been explored in the context of object-oriented programming [30]. Kennedy and Russo showed that subclassing, generics, and virtual dispatch are sufficient to encode GADTs in an object-oriented language such as the Java Programming Language or C#. In addition, they introduced a where clause into C# to express equality type relationships and thus remove the need for unsafe run-time casts in programs manipulating GADTs. They formalize the language and prove it type sound. However, the where clause defined by Kennedy and Russo cannot introduce new type variables and therefore cannot express relationships that are possible in a language with hidden type variables. For example, a non-parameterized empty list extending all instantiations of a parameterized list is not expressible.

\(^2\)Notice that this restriction not only disallows hidden type variables, but also requires all extended types to be instantiated with type variables. This is a stringent requirement in itself.
Chapter 13

Conclusions and Future Work

13.1 Conclusions

This dissertation explores the generalization of pointwise subtyping in generic object-oriented languages. This exploration has yielded the following three novel contributions:

1. We have identified two features that allow the expression of many advanced type relationships, which are impossible in a pointwise-subtyping language.
2. We have generalized pointwise subtyping in a programming language without type erasure.
3. We have developed and exposed the implementation of a generalized pointwise-subtyping language.

Each of the above contributions has helped to extend the expressiveness of generic object-oriented programming languages. In the rest of this section, we elaborate on these contributions.

This dissertation introduces hidden type variables and conditional extension as a means to generalize pointwise subtyping. Hidden type variables, which can be used to quantify types beyond their ordinary type parameters, allow one to express many type relationships that cannot be expressed in conventional generic object-oriented languages. Using hidden type variables, one can express covariance and contravariance without additional variance annotations. A single type can extend, and inherit methods from, infinitely many instantiations of another type. Hidden type variables also enable a programmer to omit redundant type arguments.
Conditional extension allows extension to be dependent on the satisfiability of one or more constraints. Using conditional extension, the properties of a collection type can depend on the properties of its element type. For example, a list type can be printable if and only if its element type is printable. Conditional extension has applications beyond collection types, including the implementation of algebraic properties in the Fortress standard library.

This dissertation investigates the semantics of hidden type variables in a generic, object-oriented type system. The semantics requires most-specific witnesses for hidden type variables. Union types and the type Bottom, which is a subtype of every type, ensure that most-specific witnesses exist.

A chief design requirement for the semantics of hidden type variables is the preservation of types at run time. Thus the language developed in this dissertation avoids the idiosyncrasies associated a type erasure semantics. In particular, type variables (including hidden type variables) can be used in type-dependent operations. However, this design requires witnesses for hidden type variables to be found at run time. To ensure that witnesses found at run time do not violate type soundness, the notion of a path in the type hierarchy is developed and path annotations are added to method calls.

The semantics of hidden type variables can be adjusted to allow conditional extension. Relatively few changes are required. Trait and object definition checking must ensure that the types in the conditional constraints are well formed, and judging a valid extension requires an additional subtype check to ensure that the conditional constraints are satisfied.

The problem of determining whether two types are in the subtype relation of a language with hidden type variables is shown to be undecidable. Therefore, a constraint solving algorithm, which approximates this relation, is developed. This algorithm infers witnesses for hidden type variables. If the algorithm successfully terminates then it returns a path, which shows that the two input types are in the subtype relation. Although initial experimentation suggests that the algorithm terminates in practice, additional restrictions, such as a limit on the nesting depth of generic types, must be imposed to guarantee termination.

To help ensure that the semantics of hidden type variables and conditional extension are sound and has other desirable properties, we have mechanized the semantics using PLT Redex. This mechanization serves as a proof of concept of the formal semantics and was useful in drawing out
subtle bugs in the semantics. Testing the semantics against a suite of examples has provided further confidence in its soundness.

In addition to testing the soundness of the semantics, we have used the implementation to test the expressiveness of the language. This has provided confidence that the applications of hidden type variables and conditional extension are in fact valid programs. The implementation is available online, allowing readers to further experiment with the semantics of hidden type variables and conditional extension.

The implementation was also used to test the expressiveness of the constraint solving algorithm. We have shown that the constraint solver terminates on many constraints without a nesting depth limit. By using our implementation to generate constraints from practical programs, we can be assured that the constraints are themselves practical. This has enabled us to conjecture that a nesting depth limit will not be a serious impediment to expressiveness in practice.

We compare hidden type variables and conditional extension to other mechanisms for combining parametric and nominal subtype polymorphism. Most generic object-oriented languages employ a pointwise subtyping scheme. Such languages disallow hidden type variables and conditional extension.

Many generic object-oriented languages allow the definition of variance types by additional language primitives. A language with hidden type variables is no different (i.e., where clauses must added to the language to support variance types). However, hidden type variables are a general mechanism that allow the expression of numerous advanced type relationships beyond variance types. We describe several of these relationships in the Chapter 4. In addition, most of these languages impose a type-erasure semantics and provide no facility for conditional extension.

To the best of our knowledge, the type system of the Cecil programming language is the only previously developed type system that allows both hidden type variables and conditional extension. However, Cecil employs a type-erasure semantics. In doing so, Cecil prevents type-dependent operations on type variables and avoids one of the major challenges that have been confronted in this dissertation: finding and maintaining witnesses for hidden type variables at run time. In addition, the only constraint solving algorithms devised for the Cecil type system impose severe restrictions
on programs. These restrictions disallow hidden type variables and limit the expressiveness of conditional extension.

Although designed for the Fortress programming language, both hidden type variables and conditional extension can be incorporated into other generic object-oriented languages. Many of the same problems would arise, and solutions analogous to those we present would apply.

13.2 Future Work

The Fortress programming language allows method definitions to be overloaded and employs a multiple dispatch semantics to resolve method calls. However, the semantics developed in this dissertation is a single dispatch semantics. It would be interesting to extend the semantics developed in this dissertation to a multiple dispatch semantics. Most likely, this will require associating path annotations with each argument of a method call.

Recall that the overriding rules for HTV0 require the overriding trait to have the same number of the hidden type variables as the overridden trait. This restriction ensures that the additional witness annotations provide sufficient witnesses when a method call is dynamically dispatched. This restriction can be removed if the additional witness annotations are inferred at run time. This is left for future work.

Adding where clauses to method definitions, as discussed in Section 6.1, is another area of future work. An extra check will be needed when dispatching to methods to determine whether the constraints listed in the method’s where clause are satisfied. Beyond that, most of the infrastructure developed for hidden type variables can be utilized.

The languages developed in this dissertation are purely functional. That is, there are no side effects, and in particular, there is no state mutation. Yet, “real world” languages make heavy use of mutable state. For example, collection types often update their elements by state mutation. Another interesting area of future work is to add mutable state to the semantics of hidden type variables and conditional extension.

The research presented in this dissertation would benefit from further experimental and theoret-
ical analyses. Experimental analysis would help determine whether the restrictions for termination of the constraint solver are prohibitive in practice. Experimental analysis would also help determine the cost of inferring witnesses at run time. Theoretical analysis would help to understand the asymptotic behavior of the type checker.

Other future work includes investigating alternative restrictions for termination of the constraint solver. Kennedy and Pierce investigate a promising approach in the context of a language with nominal inheritance and variance for generic types [29]. They introduce a non-expansiveness criterion on extension, which ensures that a finite set of types is explored by the subtype checker. This restriction is specified (but not yet implemented) by the .NET CLR and implemented by the Scala type checker [28]. Although there may be other sources of undecidability in the subtype systems of HTV0, HTV1 and CE, a non-expansiveness check is a promising alternative to a limit on the nesting depth of generic types.
Appendix A

Type Soundness Proof for HTV1

Theorem 1 (Progress). If program \( p \) is well typed and \( p; \emptyset; \emptyset \vdash e : A \) for some expression \( e \) and type \( A \) then either \( e \) is a value or \( p \vdash e \rightarrow f \) for some expression \( f \).

Proof. The proof is by case analysis on the current redex in \( e \) (in the case that \( e \) is not a value). In each case, we show that a redex evaluation rule applies. This, in combination with rules [R-EVAL] and [R-CONTEXT], yields the result.

Case \( O[\overrightarrow{B}](\overrightarrow{v}).x_i \):
By the well-typedness of \( e \):
\[
p; \emptyset; \emptyset \vdash O[\overrightarrow{B}](\overrightarrow{v}).x_i : [\overrightarrow{B}/\overrightarrow{X}]C_i
\]
where:
\[
\text{object } O[\overrightarrow{X}]_{x:C}\text{ end } \in p
\]
and:
\[
1 \leq i \leq |\overrightarrow{x}|.
\]
Therefore, [R-FIELD] can be applied.

Case \( O[\overrightarrow{B}](\overrightarrow{v}) \) path \( \overrightarrow{C}.m[\overrightarrow{D}](\overrightarrow{w}) \):
By the well-typedness of \( e \):
\[
p; \emptyset; \emptyset \vdash O[\overrightarrow{B}](\overrightarrow{v}) \text{ path } \overrightarrow{C}.m[\overrightarrow{D}](\overrightarrow{w}) : [\overrightarrow{D}/\overrightarrow{X}]F
\]
where:
\[
\text{method}_p(m, \overrightarrow{C}) = \{m[\overrightarrow{X}]_{x:E} : F = e\}.
\]
Notice that \( \overrightarrow{C} \) must begin with \( O[\overrightarrow{B}] \). Therefore, by the definition of method lookup:
\[
\text{object } O_{y:C}\text{ end } \in p.
\]
By the well-typedness of the method call:
\[
|\overrightarrow{y}| = |\overrightarrow{v}|, \quad |\overrightarrow{x}| = |\overrightarrow{w}|, \text{ and } |\overrightarrow{X}| = |\overrightarrow{D}|.
\]
Therefore, [R-METHOD] can be applied.

Case typecase \(x = v\) of \(B \Rightarrow g\) else \(h\) end:
One of [R-TYPECASE] or [R-TYPECASE-ELSE] must apply.

Case \(g\) as \(B\):
Then rule [R-ASCRITION] applies.

\begin{lemma}[Weakening]. Suppose\end{lemma}

\[ p; \Delta \vdash B <: C \text{ then } p; \Delta \vdash L <: B <: C. \]
\[ p; \Delta \vdash B \text{ ok then } p; \Delta \vdash L \text{ ok.} \]
\[ p; \Delta \vdash B \text{ path-ok then } p; \Delta \vdash L \text{ path-ok.} \]
\[ p; \Delta; \Gamma \vdash e : B \text{ then } p; \Delta \vdash L ; \Gamma \vdash e : B \text{ and } p; \Delta; \Gamma x \vdash e : B. \]

\begin{proof}
Each part is proved by straightforward induction on the derivation of \(p; \Delta \vdash B <: C\), \(p; \Delta \vdash B \text{ ok}\), \(p; \Delta \vdash B \text{ path-ok}\), and \(p; \Delta; \Gamma \vdash e : B\), respectively.
\end{proof}

\begin{lemma}[Type Substitution Preserves Well-Formedness, Paths, and Subtyping].

1. If:
\[ p; \Delta_1 \vdash X <: K \Delta_2 \vdash A \text{ ok,} \]
\[ p; \Delta_1 \vdash B <: [B/X]K, \]
\[ p; \Delta_1 \vdash B \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv (K) <: L \Delta_2 = \emptyset \]

then \(p; \Delta_1 [B/X]\Delta_2 \vdash [B/X]A \text{ ok.}\)

2. If:
\[ p; \Delta_1 \vdash X <: K \Delta_2 \vdash A \text{ ok,} \]
\[ p; \Delta_1 \vdash [B/X]K <: B, \]
\[ p; \Delta_1 \vdash B \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv (K) <: X \Delta_2 = \emptyset \]

then \(p; \Delta_1 [B/X]\Delta_2 \vdash [B/X]A \text{ ok.}\)
3. If:
\[ p; \Delta_1 \overrightarrow{X} <: \overrightarrow{K} \Delta_2 \vdash A :: B, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} <: [\overrightarrow{C}/\overrightarrow{X}]\overrightarrow{K}, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv(\overrightarrow{X} <: \overrightarrow{K} \Delta_2) = \emptyset \]

then \( p; \Delta_1 [\overrightarrow{C}/\overrightarrow{X}]\Delta_2 \vdash [\overrightarrow{C}/\overrightarrow{X}]A <: [\overrightarrow{C}/\overrightarrow{X}]B. \)

4. If:
\[ p; \Delta_1 \overrightarrow{K} <: \overrightarrow{X} \Delta_2 \vdash A :: B, \]
\[ p; \Delta_1 \vdash [\overrightarrow{C}/\overrightarrow{X}]\overrightarrow{K} <: \overrightarrow{C}, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv(\overrightarrow{K} <: \overrightarrow{X} \Delta_2) = \emptyset \]

then \( p; \Delta_1 [\overrightarrow{C}/\overrightarrow{X}]\Delta_2 \vdash [\overrightarrow{C}/\overrightarrow{X}]A <: [\overrightarrow{C}/\overrightarrow{X}]B. \)

5. If:
\[ p; \Delta_1 \overrightarrow{X} <: \overrightarrow{K} \Delta_2 \vdash A :: B, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} <: [\overrightarrow{C}/\overrightarrow{X}]\overrightarrow{K}, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv(\overrightarrow{X} <: \overrightarrow{K} \Delta_2) = \emptyset \]

then \( p; \Delta_1 [\overrightarrow{C}/\overrightarrow{X}]\Delta_2 \vdash [\overrightarrow{C}/\overrightarrow{X}]A <: [\overrightarrow{C}/\overrightarrow{X}]B. \)

6. If:
\[ p; \Delta_1 \overrightarrow{K} <: \overrightarrow{X} \Delta_2 \vdash A :: B, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} <: [\overrightarrow{C}/\overrightarrow{X}]\overrightarrow{K}, \]
\[ p; \Delta_1 \vdash \overrightarrow{C} \text{ ok, and} \]
\[ tv(\Delta_1) \cap tv(\overrightarrow{K} <: \overrightarrow{X} \Delta_2) = \emptyset \]

then \( p; \Delta_1 [\overrightarrow{C}/\overrightarrow{X}]\Delta_2 \vdash [\overrightarrow{C}/\overrightarrow{X}]A <: [\overrightarrow{C}/\overrightarrow{X}]B. \)

**Proof.** The proof is by simultaneous induction on the derivations of:

1. \( p; \Delta_1 \overrightarrow{X} <: \overrightarrow{K} \Delta_2 \vdash A \text{ ok}, \)
2. \( p; \Delta_1 \overrightarrow{K} <: \overrightarrow{X} \Delta_2 \vdash A \text{ ok}, \)
3. \( p; \Delta_1 \overrightarrow{X} <: \overrightarrow{K} \Delta_2 \vdash A :: B, \)
4. \( p; \Delta_1 \overrightarrow{K} <: \overrightarrow{X} \Delta_2 \vdash A :: B, \)
5. \( p; \Delta_1 \overrightarrow{X} <: \overrightarrow{K} \Delta_2 \vdash A :: B, \) and
6. \( p; \Delta_1 \overrightarrow{K} <: \overrightarrow{X} \Delta_2 \vdash A :: B. \)

We show the proofs for 1, 3, and 5. The proofs for 2, 4, and 6 are similar. We proceed by case analysis on the last rule in each derivation.
Case [W-OBJ], [W-BOT]:
Straightforward.

Case [W-VAR]:
Then:
\[ A \in \text{tv}(\Delta_1 \bar{X} <: \bar{K} \Delta_2). \]

If:
\[ A = X_i \]
then:
\[ \begin{array}{l}
\overline{[B/X]}A = B_i.
\end{array} \]

By assumption:
\[ p; \Delta_1 \vdash B_i \text{ ok}. \]

By Lemma 1:
\[ p; \Delta_1 [\overline{B/X}]\Delta_2 \vdash B_i \text{ ok}. \]

Otherwise:
\[ A \in \text{tv}(\Delta_1 [\overline{B/X}]\Delta_2) \text{ and } \overline{[B/X]}A = A. \]

By rule [W-VAR]:
\[ p; \Delta_1 [\overline{B/X}]\Delta_2 \vdash [\overline{B/X}]A \text{ ok}. \]

Case [W-UNION]:
Follows from the induction hypothesis.

Case [W-TAPP]:
Follows from the induction hypothesis.

Case [P-OBJ], [P-BOT]:
An application of rule [P-OBJ] or [P-BOT] followed by rule [S-PATH] proves the desired result.

Case [P-VAR]:
Then:
\[ (A <: B) \in (\Delta_1 \bar{X} <: \bar{K} \Delta_2). \]

If:
\[ (A <: B) \in \Delta_1 \text{ or } (A <: B) \in \Delta_2 \]
then the conclusion is immediate. Otherwise:
\[ A = X_i \text{ and } B = K_i. \]

By assumption:
\[ p; \Delta_1 \vdash C_i <: [\overline{C/X}]K_i. \]

Lastly, Lemma 1 gives us the desired result.

Case [P-EXT]:
Follows from the induction hypothesis and rule \[\text{S-PATH}\].

Case \[\text{P-UNIONSUB}\]:
Follows from the induction hypothesis.

Case \[\text{P-UNIONSUPER}\]:
Then:
\[
B = A \cup D \text{ or } B = D \cup A
\]
for some \(D\). Therefore:
\[
\frac{\vec{C}/\vec{X}}{\vec{X}} B = \frac{\vec{C}/\vec{X}}{(A \cup D)}
\]
or:
\[
\frac{\vec{C}/\vec{X}}{\vec{X}} B = \frac{\vec{C}/\vec{X}}{(D \cup A)}
\]
and rule \[\text{P-UNIONSUPER}\] applies. Applying rule \[\text{S-PATH}\] proves the desired result.

Case \[\text{S-REFL}\]:
Trivial.

Case \[\text{S-TRANS}\]:
Follows from the induction hypothesis.

Case \[\text{S-PATH}\]:
Follows from the induction hypothesis.

\[\text{Lemma 3 (Valid Overriding).}\] If:
\[
p;\vec{X} <: \vec{K} \vec{L} <: \vec{Y} \vdash \vec{A} \vec{B} \text{ path-ok},
\]
\[
\text{method}_p(m, \vec{A} \vec{B}) = \{m[\text{bind}](\_::\vec{C});D = \_\},
\]
\[
\text{bind} = \vec{Z},
\]
\[
p;\emptyset \vdash \vec{E} \text{ ok,}
\]
\[
p;\emptyset \vdash \vec{F} \text{ ok,}
\]
\[
p;\emptyset \vdash \vec{E} <: \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} \vec{K},
\]
\[
p;\emptyset \vdash \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} \vec{L} <: \frac{\vec{F}}{\vec{F}},
\]
\[
p;\emptyset \vdash \vec{G} <: \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} \vec{A}, \text{ and}
\]
\[
\text{min-path}_p(G[\vec{E}/\vec{X}][\vec{F}/\vec{Y}] (\vec{A} \vec{B})) = \vec{H}
\]
then:
\[
\text{method}_p(m, \vec{H}) = \{m[\text{bind}'](\_::\vec{C}');D' = \_\},
\]
\[
\text{bind}' = \vec{Z}',
\]
\[
\text{convert}(\text{bind}') = \vec{K}_1, <: \vec{K}_2,
\]
\[
\Delta = \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} \text{convert}(\text{bind}),
\]
\[
p;\Delta \vdash \frac{\vec{Z}/\vec{Z}'}{\vec{Z}/\vec{Z}'} \vec{K}_1 <: \frac{\vec{Z}/\vec{Z}'}{\vec{Z}/\vec{Z}'} \vec{K}_2,
\]
\[
p;\Delta \vdash \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} \vec{C} <: \frac{\vec{Z}/\vec{Z}'}{\vec{Z}/\vec{Z}'} \vec{C}', \text{ and}
\]
\[
p;\Delta \vdash \frac{\vec{Z}/\vec{Z}'}{\vec{Z}/\vec{Z}'} D' <: \frac{\vec{E}/\vec{X}}{\vec{X}} \frac{\vec{F}/\vec{Y}}{\vec{Y}} D.
\]

\[\text{Proof.}\] The proof proceeds by induction on the size of path \(\vec{H}\).
Base case: $|H| = 1$

Then:

$$H = \text{min-path}_{p}(G[\overline{E/\overline{X}}][\overline{F/\overline{Y}}](A\overline{B}))$$

$$= \text{min-path}_{g}(\overline{E/\overline{X}})[\overline{F/\overline{Y}}]A$$

Therefore:

$$\text{method}_{p}(m, [\overline{E/\overline{X}}][\overline{F/\overline{Y}}](A\overline{B})) = \text{method}_{p}(m, H)$$

or:

$$[\overline{E/\overline{X}}][\overline{F/\overline{Y}}](m\llbracket \text{bnd}\rrbracket (\overline{Z:C}); D = \_ ) = m\llbracket \text{bnd}''\rrbracket (\overline{Z:C}'); D' = \_$$

and the result holds trivially.

Inductive case: $H = H'\overline{H''}$ where $|\overline{H''}| > 0$

If $H'$ is not a trait or object type then the result follows from the induction hypothesis. Otherwise:

$$H' = S\llbracket \overline{T} \rrbracket$$

for some $S$ and $\overline{T}$. By the induction hypothesis:

$$\text{method}_{p}(m, \overline{H''}) = \{ m\llbracket \text{bnd}''\rrbracket (\overline{Z:C}'); D' = \_ \},$$

$$\text{convert}(\text{bnd}') = \overline{K}'_1 <: \overline{K}'_2,$$

$$\Delta = [\overline{E/\overline{X}}][\overline{F/\overline{Y}}]\text{convert}(\text{bnd}),$$

$$p; \Delta \vdash [\overline{Z/Z}]{\overline{K}'_1} <: [\overline{Z/Z}]{\overline{K}'_2},$$

$$p; \Delta \vdash [\overline{E/\overline{X}}][\overline{F/\overline{Y}}]{\overline{C}} <: [\overline{Z/Z}]{\overline{C}'},$$

and

$$p; \Delta \vdash [\overline{Z/Z}]{\overline{D}'} <: [\overline{E/\overline{X}}][\overline{F/\overline{Y}}]D.$$
Notice that:

\[ p; \Delta' \vdash S[\overline{X'}]\overline{H''} \text{ path-ok} \]

where:

\[ \Delta' = \text{convert}(\overline{bnd_1}) \text{ convert}(\overline{bnd_2}). \]

Then by rule [OVERRIDE]:

\[
\begin{align*}
\overline{bnd''} &= \overline{Z'_{\overline{\Delta}}}, \\
\overline{bnd''} &= \overline{Z''_{\overline{\Delta}}}, \\
\text{convert}(\overline{bnd''}) &= \overline{K''_1} <: \overline{K''_2}, \\
\Delta'' &= \Delta' \text{ convert}(\overline{bnd''}),
\end{align*}
\]

p; \Delta'' \vdash [\overline{Z'}/\overline{Z''}]\overline{K''_1} <: [\overline{Z'}/\overline{Z''}]\overline{K''_2},

p; \Delta'' \vdash \overline{C''} <: [\overline{Z'}/\overline{Z''}]\overline{C''}, \text{ and}

p; \Delta'' \vdash [\overline{Z'}/\overline{Z''}]\overline{D''} <: \overline{D''}.

By Lemma 2:

\[
\begin{align*}
\Delta'' &= [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\text{convert}(\overline{bnd''}), \\
p; \Delta'' &\vdash [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}' \overline{K''}_1 <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{K''}_2, \\
p; \Delta'' &\vdash [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{C''} <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{C''}'' \overline{D''}, \text{ and}
\end{align*}
\]

Notice that:

\[
\begin{align*}
\Delta''' &= [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\text{convert}(\overline{bnd''}) \\
&= \text{convert}(\overline{bnd'}) \\
&= \overline{K''}_1 <: \overline{K''}_2, \\
[\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{C''} &= \overline{C''}, \text{ and}
\end{align*}
\]

Therefore:

\[
\begin{align*}
p; \Delta'' &\vdash \overline{C''} <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{C''}'' \overline{D''} \text{ and}
\end{align*}
\]

By Lemma 2:

\[
\begin{align*}
p; \Delta'' &\vdash [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{K''}_1 <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{K''}_2.
\end{align*}
\]

By Lemma 1:

\[
\begin{align*}
p; \Delta''' &\vdash [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{K''}_1 <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{K''}_2, \\
p; \Delta''' &\vdash \overline{C''} <: [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{C''}'' \text{, and}
\end{align*}
\]

By Lemma 1:

\[
\begin{align*}
p; \Delta''' &\vdash [\overline{T}/\overline{X'}][\overline{T}/\overline{X''}]\overline{Z'}/\overline{Z''}'' \overline{D''} <: \overline{D''}.
\end{align*}
\]
From:

\[ p; \Delta \vdash [E / X]K_1 \vdash [E / X]K_2, \]

and Lemma 2:

\[ p; \Delta \vdash [T / X'][[I / X']\ldots Z / Z'\ldots K_1 \vdash [T / X'][[I / X']\ldots Z / Z'\ldots K_2, \]

\[ p; \Delta \vdash [Z / Z']C' \vdash [I / X'][[I / X']\ldots Z / Z'\ldots C'', \]

\[ p; \Delta \vdash [T / X'][[I / X']\ldots [Z / Z']D'' \vdash [T / Z']D'. \]

By rule [S-TRANS]:

\[ p; \Delta \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]C' \vdash [\bar{I} / \bar{X}][\bar{I} / \bar{X}']\ldots [Z / Z']C'' \]

\[ p; \Delta \vdash [\bar{I} / \bar{X}][\bar{I} / \bar{X}']\ldots [Z / Z']D'' \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]D, \]

which provides the result of the lemma.

\[ \square \]

**Lemma 4** (Term and Type Substitution Preserves Typing). If \( p \) is well typed and:

\[ p; \bar{X} \vdash \bar{K} \bar{L} \vdash \bar{Y}; x : A \ y : B \vdash e : C, \]

\[ p; \emptyset; \emptyset \vdash \bar{f} : \bar{D}, \]

\[ p; \emptyset \vdash \bar{E} \text{ ok}, \]

\[ p; \emptyset \vdash \bar{f} \text{ ok}, \]

\[ p; \emptyset \vdash \bar{D} \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]\bar{A}, \]

\[ p; \emptyset \vdash \bar{E} \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]\bar{K}, \]

\[ p; \emptyset \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]\bar{L} \vdash \bar{F} \]

then \( p; \emptyset; y : [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]B \vdash \text{path}_p([\bar{E} / \bar{X}][\bar{F} / \bar{Y}][\bar{f} / \bar{x}]e) : G \)

\[ p; \emptyset \vdash G \vdash [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]C. \]

**Proof.** By induction on the derivation of:

\[ p; \bar{X} \vdash \bar{K} \bar{L} \vdash \bar{Y}; x : A \ y : B \vdash e : C. \]

The proof proceeds by case analysis on the last rule applied in this derivation. We show several cases in detail and sketch the remaining cases.

**Case [T-VAR]** \( p; \bar{X} \vdash \bar{K} \bar{L} \vdash \bar{Y}; x : A \ y : B \vdash x' : C; \)

If:

\[ x' = x_i \text{ and} \]

\[ C = A_i \]

then:

\[ \text{path}_p([\bar{E} / \bar{X}][\bar{F} / \bar{Y}][\bar{f} / \bar{x}]x') = f_i. \]

By Lemma 1:

\[ p; \emptyset; y : [\bar{E} / \bar{X}][\bar{F} / \bar{Y}]B \vdash f_i : D_i. \]
By the assumption: $$p; \emptyset \vdash D_i <: [E/X][F/Y]A_i.$$ 

Otherwise: $$x' = y_i$$ and 
$$C = B_i.$$ 

Then: 
$$path_p([E/X][F/Y][f/x]x') = y_i.$$ 

By rule [T-VAR]: 
$$p; \emptyset; y : [E/X][F/Y]B \vdash y_i : [E/X][F/Y]B_i.$$ 

By rule [S-REFL]: 
$$p; \emptyset \vdash [E/X][F/Y]B_i <: [E/X][F/Y]B_i.$$ 

Case [T-Self]: 
Similar to the previous case.

By Lemma 2: 
$$p; \emptyset \vdash [E/X][F/Y]O[H] ok.$$ 

By the induction hypothesis:

$$p; \emptyset; y : [E/X][F/Y]B \vdash path_p([E/X][F/Y][f/x]y) : I''$$ and 
$$p; \emptyset \vdash I'' <: [E/X][F/Y]I'.$$ 

By Lemma 2: 
$$p; \emptyset \vdash [E/X][F/Y]I' <: [E/X][F/Y][H/Z]I'.$$ 

By rule [S-TRANS]: 
$$p; \emptyset \vdash I'' <: [E/X][F/Y][H/Z]I'.$$ 

Notice that:

$$path_p([E/X][F/Y][f/x]O[H](\bar{g})) = [E/X][F/Y]O[H](path_p([E/X][F/Y][f/x]\bar{g})).$$ 

By rule [T-OBJECT]: 
$$p; \emptyset; y : [E/X][F/Y]B \vdash path_p([E/X][F/Y][f/x]O[H](\bar{g})) : [E/X][F/Y]O[H],$$
By rule [S-REFL]:

\[ p; \emptyset \vdash [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]O[\overline{H}] <: [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]O[\overline{H}] \]

**Case [T-FIELD]:**

Follows from the induction hypothesis.

\[ p; \overline{X} <: \overline{K} \overline{L} <: \overline{Y}; x : A \ y : B \vdash g : H \]
\[ p; \overline{X} <: \overline{K} \overline{L} <: \overline{Y} \vdash HH^\prime \text{ path-ok} \]

\[ \text{method}_p(m, HH^\prime) = \{ m[\text{bnd}](\overline{\cdot}:A) : B^\prime \cdot \} \]

\[ \text{bnd} = Z \overline{\cdot} \]
\[ \text{convert}(\text{bnd}) = K_1 <: K_2 \]

\[ p; \overline{X} <: \overline{K} \overline{L} <: \overline{Y}; x : A \ y : B \vdash h : A^\prime \]
\[ p; \overline{X} <: \overline{K} \overline{L} <: \overline{Y} \vdash \overline{T} \text{ ok} \]

**Case [T-METHOD]**

\[ p; \overline{X} <: \overline{K} \overline{L} <: \overline{Y}; x : A \ y : B \vdash g \text{ path } HH^\prime \cdot m[\overline{T}](\overline{h}) : [\overline{T}/\overline{Z}]B^\prime \]

By the induction hypothesis:

\[ p; \emptyset; y : [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]B \vdash \text{path}_p([\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{f}/\overline{x}]g) : H'' \]
\[ p; \emptyset \vdash H'' <: [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]H \]
\[ p; \emptyset; y : [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]B \vdash \text{path}_p([\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{f}/\overline{x}]\overline{h}) : \overline{A}'' \]
\[ p; \emptyset \vdash \overline{A}'' <: [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]\overline{A}'' \]

By [S-TRANS] and Lemma 2:

\[ p; \emptyset \vdash \overline{A}'' <: [\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{T}/\overline{Z}]\overline{A}'. \]

By Lemma 2:

\[ p; \emptyset \vdash [\overline{E}/\overline{X}][\overline{F}/\overline{Y}]\overline{T} \text{ ok.} \]

Notice that:

\[ \text{path}_p([\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{f}/\overline{x}](g \text{ path } HH^\prime \cdot m[\overline{T}](\overline{h}))) = \text{path}_p([\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{f}/\overline{x}]g) \text{ path } HH^\prime \cdot m[\overline{T}]([\text{path}_p([\overline{E}/\overline{X}][\overline{F}/\overline{Y}][\overline{f}/\overline{x}])]) \]

where:

\[ \text{min-path}_p(H''[\overline{E}/\overline{X}][\overline{F}/\overline{Y}](HH^\prime)) = \overline{H}'' \]

and:

\[ p; \emptyset \vdash \overline{H}'' \text{ path-ok.} \]
By Lemma 3:
\[
\text{method}_p(m, H') = \{ m \left[ \text{bnd}' \right] (x : \text{type}) : B'' = \_ \},
\]
\[
\text{convert} (\text{bnd}') = \overrightarrow{K}'_1 <: \overrightarrow{K}'_2,
\]
\[
\Delta = \mathbb{E}/x = \mathbb{F}/y]
\]
\[
p; \Delta \vdash \overrightarrow{Z}/x = \overrightarrow{Z}'_1 <: \overrightarrow{Z}'_2,
\]
\[
p; \Delta \vdash \mathbb{E}/x(\mathbb{F}/y)[\overrightarrow{I} / \overrightarrow{Z}] A' <: \overrightarrow{Z}/x = \overrightarrow{Z}'_2 A''' ,
\]
\[
p; \Delta \vdash \overrightarrow{Z}/x = \overrightarrow{Z}'_2 B'' <: \mathbb{E}/x(\mathbb{F}/y) [\overrightarrow{I} / \overrightarrow{Z}] B'.
\]

By Lemma 2:
\[
p; \emptyset \vdash \overrightarrow{I}/x = \overrightarrow{I}'_1 <: \overrightarrow{I}'_2 ,
\]
\[
p; \emptyset \vdash \mathbb{E}/x(\mathbb{F}/y)[\overrightarrow{I} / \overrightarrow{Z}] A' <: \overrightarrow{I}/x = \overrightarrow{I}'_2 A''' ,
\]
\[
p; \emptyset \vdash \overrightarrow{I}/x = \overrightarrow{I}'_2 B'' <: \mathbb{E}/x[\overrightarrow{I} / \overrightarrow{Z}] B'.
\]

By rule [S-TRANS]:
\[
p; \emptyset \vdash A''' <: \overrightarrow{I} / \overrightarrow{Z} A'''.
\]

By rule [T-METHOD]:
\[
p; \emptyset; y: \mathbb{E}/x(\mathbb{F}/y) B \vdash \text{path}_p(\mathbb{E}/x(\mathbb{F}/y)[\overrightarrow{I} / \overrightarrow{Z}] (g \text{ path } HH'.m[\overrightarrow{I}](h))) : [\overrightarrow{I} / \overrightarrow{Z}] B'',
\]
which finishes the case.

Case [T-TYPECASE]:
Follows from the induction hypothesis and rule [S-TRANS].

Case [T-ASCRPTION]:
Follows from the induction hypothesis and rule [S-TRANS].

Lemma 5. If \(p; \Delta; \Gamma \vdash e : A\) then \(\text{path}_p(e) = e\).

Proof. By induction on the structure of \(e\). If \(e\) is well typed then the path annotation on each method call must start from the type of the receiver and end at a type defining the method (this follows from rule [T-METHOD]). Therefore, the path annotation on each method call does not need to be extended and the path update function returns the expression unaltered.

Lemma 6. \(\text{path}_p(\mathbb{E}[e]) = \text{path}_p(\mathbb{E}[\text{path}_p(e)])\).

Proof. By induction on the structure of \(e\). Follows from the fact that the path update function is recursively applied to subexpressions.

Lemma 7 (Replacement). If \(p; \emptyset \vdash \mathbb{E}[e] : A\) with the subderivation \(p; \emptyset \vdash e : B\) and \(p; \emptyset \vdash f : C\) where \(p; \emptyset \vdash C <: B\) then \(p; \emptyset \vdash \text{path}_p(\mathbb{E}[f]) : D\) where \(p; \emptyset \vdash D <: A\).

Proof. This proof is by induction on the structure of \(\mathbb{E}\).
Case \( \square \):
Then:
\[
A = B \quad \text{and} \quad D = C.
\]

Lemma 5 gives the final result.

Case \( O[\overrightarrow{E}(\overrightarrow{v}EC'\overrightarrow{g})] \):
By \([T\text{-OBJECT}]\):
\[
p; \emptyset; \emptyset \vdash EC[e] : F,
p; \emptyset \vdash F : [E/X]G_i, \quad \text{and} \quad \text{object } O[X]\overrightarrow{g}(\overrightarrow{w}:G) \in p.
\]

By the induction hypothesis:
\[
p; \emptyset; \emptyset \vdash \text{path}_p(EC[f]) : H
\]
where:
\[
p; \emptyset \vdash H' : F.
\]
By \([S\text{-TRANS}]\):
\[
p; \emptyset \vdash \text{path}_p(EC[f]) : \overrightarrow{H}
\]
By Lemma 5:
\[
\text{path}_p(\overrightarrow{v}) = \overrightarrow{v} \quad \text{and} \quad \text{path}_p(\overrightarrow{g}) = \overrightarrow{g}.
\]
Therefore:
\[
\text{path}_p(O[\overrightarrow{E}](\overrightarrow{v}EC'[f]\overrightarrow{g})) = O[\overrightarrow{E}](\overrightarrow{v} \text{path}_p(EC[f])\overrightarrow{g}).
\]
By \([T\text{-OBJECT}]\):
\[
p; \emptyset; \emptyset \vdash \text{path}_p(EC[f]) : A.
\]
Then the lemma holds with \( A = D \).

Case \( EC'.x \):
Similar to the previous case.

Case \( EC' \text{ path } E\overrightarrow{F}.m[\overrightarrow{G}](\overrightarrow{g}) \):
By \([T\text{-METHOD}]\):
\[
p; \emptyset; \emptyset \vdash EC'[e] : E,
p; \emptyset \vdash E\overrightarrow{F} \text{ path-ok},
\text{method}(m, E\overrightarrow{F}) = \{ m[\text{bind}]((x:H);I) \},
\text{bind} = X \overrightarrow{m},
\text{convert}(\text{bind}) = \overrightarrow{L_1} <: \overrightarrow{L_2},
p; \emptyset; \emptyset \vdash \overrightarrow{g} : H',
p; \emptyset \vdash \overrightarrow{H'} : [\overrightarrow{G}/X]H',
p; \emptyset \vdash \overrightarrow{G} \text{ ok},
p; \emptyset \vdash [\overrightarrow{G}/X]L_1 : [\overrightarrow{G}/X]L_2, \quad \text{and} \quad p; \emptyset; \emptyset \vdash EC[e] : [\overrightarrow{G}/X]I.
By the induction hypothesis:

\[ p; \emptyset \vdash \text{path}_p(EC'[f]) : E' \]

where:

\[ p; \emptyset \vdash E' <: E. \]

By Lemma 5:

\[ \text{path}_p(\overline{g}) = \overline{g}. \]

Therefore:

\[ \text{path}_p(EC'[f]) \ \text{path} \ E\overline{F}.m[\overline{G}] f \]

\[ = \text{path}_p(EC'[f]) \ \text{path} \ \overline{F}.m[\overline{G}] f \]

where:

\[ \text{min-path}_p(E'E\overline{F}) = \overline{F} \]

and:

\[ p; \emptyset \vdash \overline{F} \ \text{path-ok}. \]

By Lemma 3:

\[ \text{method}_p(m, \overline{F}') = \{ m[\ bnd'] (\overline{G}:I'') : I'' = \} \]

\[ \ bnd' = X' \]

\[ \text{convert}\ (bnd') = \overline{K}_1 <: \overline{K}_2, \]

\[ \Delta = \text{convert}(\ bnd), \]

\[ p; \Delta \vdash [X/X']K_1 <: [X/X']K_2, \]

\[ p; \Delta \vdash \overline{H} <: [X/X']\overline{K}_2, \]

\[ p; \Delta \vdash \overline{H}' <: [X/X']\overline{K}_2', \text{ and} \]

\[ p; \Delta \vdash [X/X']I'' <: I. \]

By Lemma 2:

\[ p; \emptyset \vdash [G/X']K_1 <: [G/X']K_2, \]

\[ p; \emptyset \vdash [G/X']\overline{H} <: [G/X']\overline{K}_2, \]

\[ p; \emptyset \vdash [G/X']I'' <: [G/X']I. \]

By rule [S-TRANS]:

\[ p; \emptyset \vdash \overline{H}' <: [G/X']I''. \]

By rule [T-METHOD]:

\[ p; \emptyset \vdash \text{path}_p(EC[f]) : [G/X']I'\]

which finishes the case.

Case \( \text{v \ path} \ \overrightarrow{E}.m[\overrightarrow{F}] \overrightarrow{w} \ EC' \ \overrightarrow{f} : \)

Similar to the previous cases.

Case \( \text{typecase } x = EC' \text{ of } A \Rightarrow e \text{ else } e \text{ end} : \)

Similar to the previous cases.

\[ \textbf{Theorem 2 (Subject Reduction).} \] If program \( p \) is well typed and \( p; \emptyset \vdash e : A \) for some expression \( e \) and type \( A \) and \( p \vdash e \rightarrow f \) for some expression \( f \) then \( p; \emptyset \vdash f : B \) for some type \( B \) where \( p; \emptyset \vdash B <: A. \)
Proof. The proof is by case analysis on the redex evaluation rule applied.

Case [R-FIELD]:

Then:
\[
e = EC[\overline{O}] (\overline{v}) . x_i \quad \text{and} \quad f = \text{path}_p(EC[v_i])
\]
where:
\[
\text{object} \ O[\overline{X}] (x:D) \ \text{end} \in p.
\]

By the well-typedness of \(e\):
\[
p; \emptyset; \emptyset \vdash O[\overline{C}] (\overline{v}) . x_i : [\overline{C}/\overline{X}] D_i.
\]

By [T-OBJECT]:
\[
p; \emptyset; \emptyset \vdash v_i : D'_i
\]
where:
\[
p; \emptyset \vdash D'_i <: [\overline{C}/\overline{X}] D_i.
\]

By Lemma 7:
\[
p; \emptyset; \emptyset \vdash \text{path}_p(EC[v_i]) : B
\]
where:
\[
p; \emptyset \vdash B <: A.
\]

Case [R-METHOD]:

Then:
\[
e = EC[\overline{O}] (\overline{v}) \ \text{path} \ O[\overline{C}] D. m[\overline{E}] (\overline{w}) \quad \text{and} \quad f = \text{path}_p(EC[\overline{v}/\overline{x}] [O[\overline{C}] (\overline{v}) / \text{self}][\overline{w}/\overline{y}][\overline{E}/\overline{Y}] g))
\]
where:
\[
\text{object} \ O[\overline{\text{bnd}}] (x:F) \in p,
\]
\[
\text{method}_p(m, O[\overline{C}] D) = \{ m[\overline{\text{bnd}'}] (y:G): H = g \},
\]
\[
\overline{\text{bnd}'} = \overline{X}, \text{ and} \ 
\]
\[
\overline{\text{bnd}'} = \overline{Y}.
\]

By the well-typedness of \(e\):
\[
p; \emptyset; \emptyset \vdash O[\overline{C}] (\overline{v}) \ \text{path} \ O[\overline{C}] D. m[\overline{E}] (\overline{w}) : [\overline{E}/\overline{Y}] H.
\]

There are two subcases to consider.

Subcase: \(m\) is defined in \(O\)

By [T-METHODDEF]:
\[
p; \Delta; \text{self} : O[\overline{X}] x : F \ y : G' \vdash g' : H'
\]
where:
\[ \Delta = \text{convert} (\text{bnd}) \text{ convert} (\text{bnd}') \]
\[ \overline{G} = [\overline{C} / \overline{X}] \overline{G}' \]
\[ g = [\overline{C} / \overline{X}] g' \]
\[ H = [\overline{C} / \overline{X}] H'' \text{ for some } H'' \text{, and } \]
\[ p; \Delta \vdash H' <: H''. \]

By [T-METHOD] and Lemma 2:
\[ p; \emptyset \vdash \overline{E} \text{ ok}, \]
\[ \text{convert} (\text{bnd}') = \overline{K}'_1 <: \overline{K}'_2 , \]
\[ p; \emptyset \vdash [\overline{E} / \overline{Y}] K'_1 <: [\overline{E} / \overline{Y}] K'_2 , \]
\[ p; \emptyset \vdash \overline{w} : \overline{G}'' , \text{ and } \]
\[ p; \emptyset \vdash \overline{G}'' <: [\overline{E} / \overline{Y}] \overline{G} . \]

By [T-OBJECT] and Lemma 2:
\[ p; \emptyset \vdash \overline{C} \text{ ok}, \]
\[ \text{convert} (\text{bnd}) = \overline{K}_1 <: \overline{K}_2 , \]
\[ p; \emptyset \vdash [\overline{C} / \overline{X}] \overline{K}_1 <: [\overline{C} / \overline{X}] \overline{K}_2 , \]
\[ p; \emptyset \vdash \overline{v} : \overline{F}'; \text{ and } \]
\[ p; \emptyset \vdash \overline{F}' <: [\overline{C} / \overline{X}] \overline{F} . \]

By Lemma 4:
\[ p; \emptyset \vdash \overline{v} / \overline{x} / [\overline{C} / \overline{y}] / [\text{self}] / [\overline{w} / \overline{y}] / [\overline{E} / \overline{Y}] g ) \vdash H'' \text{ and } \]
\[ p; \emptyset \vdash H''' <: [\overline{E} / \overline{Y}] [\overline{C} / \overline{X}] H'. \]

By Lemma 2:
\[ p; \emptyset \vdash [\overline{E} / \overline{Y}] [\overline{C} / \overline{X}] H' <: [\overline{E} / \overline{Y}] H. \]

By [S-TRANS]:
\[ p; \emptyset \vdash H''' <: [\overline{E} / \overline{Y}] H. \]

By Lemma 7:
\[ p; \emptyset \vdash \text{path}_p ( \text{EC}[\text{path}_p ([\overline{v} / \overline{x}] [\overline{C} / \overline{y}] / [\text{self}] / [\overline{w} / \overline{y}] / [\overline{E} / \overline{Y}] g )) : B \]

where:
\[ p; \emptyset \vdash B <: A. \]

By Lemma 6:
\[ \text{path}_p ( \text{EC}[\text{path}_p ([\overline{v} / \overline{x}] [\overline{C} / \overline{y}] / [\text{self}] / [\overline{w} / \overline{y}] / [\overline{E} / \overline{Y}] g )) = \text{path}_p ( \text{EC}[\overline{v} / \overline{x}] [\overline{C} / \overline{y}] / [\text{self}] / [\overline{w} / \overline{y}] / [\overline{E} / \overline{Y}] g )) \]
finishes the case.

Subcase: \( m \) is not defined in \( O \)

Almost identical to the above case. The only difference is that substituting the object’s value parameters is superfluous.

Case \([R\text{-}TYPECASE]\):

Then:

\[
e = EC[typecase \ x = O[\overrightarrow{C}](\overrightarrow{v}) \text{ of } D \Rightarrow g \ D' \Rightarrow g' \ D'' \Rightarrow g'' \text{ else } \Rightarrow h \text{ end}] \text{ and }
f = path_p(EC[[O[\overrightarrow{C}](\overrightarrow{v})/x]g'])
\]

where:

\[
p; \emptyset \vdash O[\overrightarrow{C}] <: D'.
\]

By the well-typedness of \( e \):

\[
p; \emptyset; \emptyset \vdash typecase \ x = O[\overrightarrow{C}](\overrightarrow{v}) \text{ of } D \Rightarrow g \ D' \Rightarrow g' \ D'' \Rightarrow g'' \text{ else } \Rightarrow h \text{ end} : E
\]

for some \( E \). By rule \([T\text{-}TYPECASE]\):

\[
p; \emptyset; x : D' \vdash g' : E' \text{ for some } E', \text{ and }
p; \emptyset \vdash E' <: E.
\]

By Lemma 4:

\[
p; \emptyset; \emptyset \vdash path_p([O[\overrightarrow{C}](\overrightarrow{v})/x]g') : E''
\]

for some \( E'' \) such that:

\[
p; \emptyset \vdash E'' <: E'.
\]

By rule \([S\text{-}TRANS]\):

\[
p; \emptyset \vdash E'' <: E.
\]

By Lemma 7:

\[
p; \emptyset; \emptyset \vdash path_p(EC[path_p([O[\overrightarrow{C}](\overrightarrow{v})/x]g')]) : B
\]

where:

\[
p; \emptyset \vdash B <: A.
\]

By Lemma 6:

\[
\begin{align*}
\text{path}_p(EC[path_p([O[\overrightarrow{C}](\overrightarrow{v})/x]g')]) \\
= \text{path}_p(EC[[O[\overrightarrow{C}](\overrightarrow{v})/x]g'])
\end{align*}
\]

finishes the case.

Case \([R\text{-}TYPECASE}\_ELSE]\):

Similar to the previous case.

Case \([R\text{-}ASCRIPITION]\):
Then:  
\[ e = EC[g \text{ as } C] \text{ and } \]
\[ f = \text{path}_p(EC[g]). \]

By the well-typedness of \( e \):
\[ p; \emptyset; \emptyset \vdash g \text{ as } C : C. \]

By rule [T-ASCRIPITION]:
\[ p; \emptyset; \emptyset \vdash g : D \text{ for some } D, \text{ and } \]
\[ p; \emptyset \vdash D <: C. \]

By Lemma 7:
\[ p; \emptyset; \emptyset \vdash \text{path}_p(EC[g]) : B \]

where:
\[ p; \emptyset \vdash B <: A. \]
Appendix B

Acyclicity Proof for HTV1 Programs

This appendix gives restrictions on HTV1 programs to prevent cycles in the type hierarchy. The restrictions are permissive enough to allow the definition of the covariant and contravariant types. Note that we use these restrictions only to guarantee the acyclicity of the type hierarchy; they can be replaced by any other set of restrictions that makes this guarantee. After introducing the restrictions, we prove that they are sufficient.

Suppose we have the following trait definition in a program $p$:

\[
\begin{align*}
\text{trait } T & \text{ extends } \{ M \} \text{ where } \{ bnd_i \} \text{ end} \\
bnd_1 &= X - K \\
bnd_2 &= Y - L
\end{align*}
\]

We say that $T$ directly depends on $S \neq T$ in $p$ if $M_i = S[\overrightarrow{A}]$ for some $i$ and $\overrightarrow{A}$, and that $T$ depends on $S$ in $p$ if $T$ directly depends on $S$ or $T$ depends on some $R$ that directly depends on $S$; the depends relation is the transitive closure of the directly-depends relation. Note that $T$ never directly depends on itself. We say that $T$ is self-extending in $p$ if $M_i = T[\overrightarrow{A}]$ for some $i$ and $\overrightarrow{A}$; we say that $M_i$ is a self-supertype of $T$. We often omit mentioning the program $p$ when it is obvious from context. Note that objects cannot be self-extending.

We impose the following three restrictions on trait definitions:

(R1) $T$ must not depend on itself (i.e., there are no cycles except through self-extension).

(R2) $T$ has at most one self-supertype.

(R3) If $T$ is self-extending with self-supertype $T[\overrightarrow{A}]$, then for each type parameter $X_i$, one of the following holds:

(a) $A_i = X_i$;

(b) $A_i = K_i$;

(c) $A_i = Y_j$ and $X_i = L_j$ for some $j$; or

(d) $A_i = X_j$ and $X_i = K_j$ for some $j \neq i$.

We now show that these restrictions guarantee the acyclicity of the type hierarchy.
Lemma 8. If $p; \emptyset \vdash T[\overrightarrow{A}] \llbracket \rightarrow B \rrbracket$ then either $T = S$ or $T$ depends on $S$.

Proof. By induction on the derivation (call it $Q$) of:

$$p; \emptyset \vdash T[\overrightarrow{A}] \llbracket \rightarrow B \rrbracket.$$

We proceed by case analysis on the last rule applied in the derivation (see Figure 8.5 for the possible rules).

Case [S-REFL]:
Then $T = S$.

Case [S-TRANS]:
Suppose:

$$p; \emptyset \vdash T[\overrightarrow{A}] \llbracket \rightarrow C \rrbracket \quad \text{and} \quad p; \emptyset \vdash C \llbracket \rightarrow S \rrbracket$$

for some $C$. Then $C$ has the form $R[\overrightarrow{D}]$ for some $R$ and $\overrightarrow{D}$. By the induction hypothesis either $T = R$ or $T$ depends on $R$.

Subcase $T = R$:
By the induction hypothesis either $R = S$ or $R$ depends on $S$. If $R = S$ then $T = S$. Otherwise, $T$ depends on $S$.

Subcase $T$ depends on $R$:
By the induction hypothesis either $R = S$ or $R$ depends on $S$. If $R = S$ then $T$ depends on $S$. Otherwise, $T$ depends on $S$ because the depends relation is transitive.

Subcase [S-PATH]:
Consider the previous rule applied in $Q$ (see Figures 8.4 and 8.24 for the possible rules). It cannot be [P-OBJ], [P-BOT], [P-VAR], [P-UNIONSUB], or [P-UNIONSUPER] because either $T[\overrightarrow{A}]$ or $S[\overrightarrow{B}]$ does not have the correct form. Therefore, it must be [P-EXT]. In this case, either $T = S$ or $T$ depends on $S$ by the definition of the depends relation. \qed

Lemma 9. Consider a well-typed program $p$ with the following trait definition that meets the restrictions above:

$$\text{trait } T[\overrightarrow{bnd_1}] \text{ extends } \{\overrightarrow{M}\} \text{ where } \{\overrightarrow{bnd_2}\} \_ \text{ end},$$

$$\overrightarrow{bnd_1} = \overrightarrow{X} \_ K,$$

$$\overrightarrow{bnd_2} = \overrightarrow{Y} \_ L,$$

$$\text{convert}(\overrightarrow{bnd_1}) = \overrightarrow{K}^i \llbracket \rightarrow K^0 \rrbracket, \quad \text{and} \quad \text{convert}(\overrightarrow{bnd_2}) = \overrightarrow{L}^i \llbracket \rightarrow L^0 \rrbracket$$

where $T[\overrightarrow{A}] \in \{\overrightarrow{M}\}$ for some $\overrightarrow{A}$. For all $\overrightarrow{B}$ and $\overrightarrow{C}$ such that

$$p; \emptyset \vdash T[\overrightarrow{B}] \llbracket \rightarrow C \rrbracket,$$

and for any $i$ such that $1 \leq i \leq |\overrightarrow{X}|$,

1. if $A_i = X_i$ then $C_i = B_i$;
2. if $A_i = K_i$ then either $p; \emptyset \vdash B_i <: C_i$ or $p; \emptyset \vdash C_i <: B_i$;
3. if $A_i = Y_j$ and $X_i = L_j$ for some $j$
   then either $p; \emptyset \vdash B_i <: C_i$ or $p; \emptyset \vdash C_i <: B_i$; and
4. if $A_i = X_j$ and $X_i = K_j$ for some $j \neq i$
   then either $p; \emptyset \vdash B_i <: C_i$ or $p; \emptyset \vdash C_i <: B_i$.

Proof. We use proof by contradiction. Suppose there are $p, T, \vec{B}, \vec{C}$, and $i$ such that $p$ contains $T$
$T$ is of the form indicated in the statement of the theorem, and

$$p; \emptyset \vdash T[\vec{B}] <: T[\vec{C}],$$

but one of the four numbered implications fails. Among all choices of $p, T, \vec{B}, \vec{C}$, and $i$,
consider a choice such that the derivation (call it $Q$) of

$$p; \emptyset \vdash T[\vec{B}] <: T[\vec{C}],$$

is of minimal length. Consider the last rule applied in this derivation (see Figure 8.5 for the possible
rules). It cannot be $[S\text{-TRANS}]$ by the minimality assumption. It cannot be $[S\text{-REFL}]$ because in
that case $\vec{B} = \vec{C}$, so each of the four numbered implications would hold. Therefore, it must be
$[S\text{-PATH}]$.

Consider the previous rule in $Q$ (see Figures 8.4 and 8.24 for the possible rules). It cannot be $[P\text{-OBJ}]$, $[P\text{-BOT}]$, $[P\text{-VAR}]$, $[P\text{-UNIONSUB}]$, or $[P\text{-UNIONSUPER}]$ because none of these
produces a judgment of the correct form. Therefore, it must be $[P\text{-EXT}]$, with:

$$T[\vec{C}] = [\vec{B}/X][\vec{D}/Y]M_l$$

for some $l$ and $\vec{D}$ such that:

$$1 \leq l \leq |\vec{M}|,$$

$$p; \emptyset \vdash [\vec{B}/X][\vec{D}/Y]\vec{K}^l <: [\vec{B}/X][\vec{D}/Y]\vec{K}^h,$$

and

$$p; \emptyset \vdash [\vec{B}/X][\vec{D}/Y]\vec{L}^l <: [\vec{B}/X][\vec{D}/Y]\vec{L}^h.$$

Because $T$ has at most one self-supertype, we have:

$$M_l = T[\vec{A}]$$

and

$$\vec{C} = [\vec{B}/X][\vec{D}/Y]\vec{A}.$$ 

One of the following cases must apply. Since that case must hold, all of the numbered implications
must hold (contrary to the assumption that it doesn’t).

case: $A_i = X_i$

Then:

$$C_i = [\vec{B}/X][\vec{D}/Y]X_i = [\vec{D}/Y]B_i.$$ 

Because $B_i$ has no free type variables, we have $C_i = B_i$. 

case: \( A_i = K_i \)

Then:

\[ C_i = \overline{B/X} \overline{D/Y} K_i. \]

Because either:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} K'_i &= \overline{B/X} \overline{D/Y} K_i = B_i \\
\overline{B/X} \overline{D/Y} K''_i &= \overline{B/X} \overline{D/Y} K_i
\end{align*}
\]

or:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} K'_i &= \overline{B/X} \overline{D/Y} K_i = B_i \\
\overline{B/X} \overline{D/Y} K''_i &= \overline{B/X} \overline{D/Y} K_i
\end{align*}
\]

we have either:

\[ p; \emptyset \vdash B_i <: C_i \]

or:

\[ p; \emptyset \vdash C_i <: B_i. \]

case: \( A_i = Y_j \) and \( X_i = L_j \) for some \( j \)

Then:

\[ C_i = \overline{B/X} \overline{D/Y} Y_j = D_j. \]

Because either:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} L'_j &= \overline{B/X} \overline{D/Y} Y_j = D_j \\
\overline{B/X} \overline{D/Y} L''_j &= \overline{B/X} \overline{D/Y} Y_j
\end{align*}
\]

or:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} L'_j &= \overline{B/X} \overline{D/Y} Y_j = D_j \\
\overline{B/X} \overline{D/Y} L''_j &= \overline{B/X} \overline{D/Y} Y_j
\end{align*}
\]

we have either:

\[ p; \emptyset \vdash C_i = D_j <: B_i \]

or:

\[ p; \emptyset \vdash B_i <: C_i = D_j. \]

case: \( A_i = X_j \) and \( X_i = K_j \) for some \( j \neq i \)

Then:

\[ C_i = \overline{B/X} \overline{D/Y} X_j = \overline{D/Y} B_j = B_j. \]

Because either:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} K'_j &= \overline{B/X} \overline{D/Y} X_j = B_j \\
\overline{B/X} \overline{D/Y} K''_j &= \overline{B/X} \overline{D/Y} K_j
\end{align*}
\]

or:

\[
\begin{align*}
\overline{B/X} \overline{D/Y} K'_j &= \overline{B/X} \overline{D/Y} X_j = B_j \\
\overline{B/X} \overline{D/Y} K''_j &= \overline{B/X} \overline{D/Y} X_j
\end{align*}
\]
we have either:

\[ p; \emptyset \vdash C_i = B_j <: B_i \]

or:

\[ p; \emptyset \vdash B_i <: C_i = B_j. \]

\[ \square \]

**Theorem 3.** If every trait definition in a well-typed program \( p \) satisfies the restrictions above, and \( p; \emptyset \vdash A \ ok \), and \( p; \emptyset \vdash B \ ok \), then \( p; \emptyset \vdash A <: B \) and \( p; \emptyset \vdash B <: A \) imply that \( A = B \).

**Proof.** The proof is by induction on the structure of \( A \). We proceed by case analysis (see Figure 8.3 for the definition of types).

Case \( A = \text{Object} \):

Then \( B = \text{Object} \) because:

\[ p; \emptyset \vdash \text{Object} <: B. \]

Case \( A = \text{Bottom} \):

Then \( B = \text{Bottom} \) because:

\[ p; \emptyset \vdash B <: \text{Bottom}. \]

Case \( A \) has the form \( O[\overrightarrow{C}] \) for some \( O \) and \( \overrightarrow{C} \):

Then \( B = O[\overrightarrow{C}] \) because:

\[ p; \emptyset \vdash B <: O[\overrightarrow{C}]. \]

Case \( A \) is a type variable:

Notice that \( A \) cannot be a type variable because \( p; \emptyset \vdash A \ ok \) and the type variable environment is empty (see Figures 8.7 and 8.25 for the definition of well-formed types).

Case \( A \) is a union type:

Then \( A = B \) by the definition of type equivalence for union types (see the end of Section 8.2.1 for the definition of type equivalence).

Case \( A \) has the form \( T[\overrightarrow{C}] \) for some \( T \) and \( \overrightarrow{C} \):

We proceed by case analysis on the structure of \( B \) (see Figure 8.3 for the definition of types).

Subcase \( B = \text{Object} \):

Notice that \( B \) cannot be \( \text{Object} \) because:

\[ p; \emptyset \vdash \text{Object} <: A \]

only if \( A = \text{Object} \).

Subcase \( B = \text{Bottom} \):

Notice that \( B \) cannot be \( \text{Bottom} \) because:

\[ p; \emptyset \vdash A <: \text{Bottom} \]

only if \( A = \text{Bottom} \).
Subcase $B$ has the form $O[D]$ for some $O$ and $D$:
Notice $B$ cannot have the form $O[D]$ because:

$$p; \emptyset \vdash A : O[D]$$

only if $A = O[D]$.

Subcase $B$ is a type variable:
Notice that $B$ cannot be a type variable because $p; \emptyset \vdash B$ ok and the type variable environment is empty (see Figures 8.7 and 8.25 for the definition of well-formed types).

Subcase $B$ is a union type:
Then $A = B$ by the definition of type equivalence for union types (see the end of Section 8.2.1 for the definition of type equivalence).

Subcase $B$ has the form $S[D]$ for some $S$ and $D$:
By Lemma 8, either:

- $T = S$
- $T$ depends on $S$ and $S$ depends on $T$.

Notice that restriction (R1) prevents the latter case from holding. Thus, because $A = T[C]$ and $B = T[D]$, we have:

$$p; \emptyset \vdash T[C] : T[D]$$
$$p; \emptyset \vdash T[D] : T[C].$$

By rule [W-TAPP], for each $i$ such that $1 \leq i \leq |C|:

$$p; \emptyset \vdash C_i$$ ok and
$$p; \emptyset \vdash D_i.$$

By Lemma 9 and the restriction (R3), for each $i$ such that $1 \leq i \leq |C|$, either:

- $C_i = D_i$
- $p; \emptyset \vdash C_i : D_i$ and $p; \emptyset \vdash D_i : C_i$.

In the latter case, we also get $C_i = D_i$ by the induction hypothesis. Thus:

$$\overline{C} = \overline{D}$$

and:

$$A = T[\overline{C}] = T[\overline{D}] = B,$$

contradicting the assumption that $A \neq B$. 

\qed
Appendix C

Undecidability Proof for Subtyping in HTV1

This appendix proves the undecidability of the subtype checking problem for the language of HTV1. This is proved by reducing the subtype checking problem for a language developed by Kennedy and Pierce [29] to the subtype checking problem for HTV1. Section 9.3.1 defines this reduction. This section also defines the syntax of the language of Kennedy and Pierce. For the definition of subtyping and well-formed types in the language of Kennedy and Pierce, the reader should see the report written by Kennedy and Pierce.

Lemma 10. Let $CT$ and $T$ be a class table and type in the language of Kennedy and Pierce such that:

$$\emptyset \vdash T \text{ ok}.$$

Let program $p$ (in the language of HTV1) be the result of encoding $CT$ with $E$. Then:

$$p; \emptyset \vdash E(T) \text{ ok}.$$

Proof. By induction on the structure of $T$. Notice that $T$ must be a constructed type because the type variable environment is empty. Let:

$$T = C\langle \overrightarrow{U} \rangle$$

where:

$$C\langle v_1X_1 \ldots v_nX_n \rangle$$

is in $CT$. Let:

trait $S[\ bnd_1 \ldots bnd_n\ ]$
extends \ldots
where \{bnd'\_1 \ldots bnd'\_n\}
end

be in $p$, where $S = E(C)$. By the induction hypothesis:

$$p; \emptyset \vdash E(\overrightarrow{U}) \text{ ok.}$$
Next we show that \( \mathcal{E}(\overline{U}) \) satisfies the bounds on the corresponding type variables and therefore:

\[
p;\emptyset \vdash \mathcal{E}(C(\overline{U})) \text{ ok.}
\]

Let \( 1 \leq i \leq n \). We proceed by case analysis on the variance of \( X_i \).

Case \( X_i \) is covariant:

Then:

\[
\begin{align*}
bnd_i &= Y_i \text{ extends } Z_i \text{ and} \\
bnd'_i &= Z_i \text{ extends } \text{Object}.
\end{align*}
\]

By rule [W-OBJ]:

\[
p;\emptyset \vdash \text{Object} \text{ ok.}
\]

By rules [P-OBJ] and [S-PATH]:

\[
\begin{align*}
p;\emptyset \vdash \mathcal{E}(U_i) &<: \text{Object} \text{ and} \\
p;\emptyset \vdash \text{Object} &<: \text{Object}.
\end{align*}
\]

Therefore \( \text{Object} \) is a valid witness for \( Z_i \).

Case \( X_i \) is contravariant:

Then:

\[
\begin{align*}
bnd_i &= Y_i \text{ bounds } Z_i \text{ and} \\
bnd'_i &= Z_i \text{ extends } \text{Object}.
\end{align*}
\]

By rule [W-BOT]:

\[
p;\emptyset \vdash \text{Bottom} \text{ ok.}
\]

By rules [P-BOT] and [S-PATH]:

\[
\begin{align*}
p;\emptyset \vdash \text{Bottom} &<: \mathcal{E}(U_i) \text{ and} \\
p;\emptyset \vdash \text{Bottom} &<: \text{Object}.
\end{align*}
\]

Therefore \( \text{Bottom} \) is a valid witness for \( Z_i \).

Case \( X_i \) is invariant:

Then:

\[
\begin{align*}
bnd_i &= Y_i \text{ extends } Z_i \text{ and} \\
bnd'_i &= Z_i \text{ extends } Y_i.
\end{align*}
\]

By the induction hypothesis:

\[
p;\emptyset \vdash \mathcal{E}(U_i) \text{ ok.}
\]

By rule [S-REFL]:

\[
p;\emptyset \vdash \mathcal{E}(U_i) <: \mathcal{E}(U_i).
\]

Therefore \( \mathcal{E}(U_i) \) is a valid witness for \( Z_i \).

\[\square\]

**Lemma 11.** Let \( CT \) be a class table in the language of Kennedy and Pierce. Let program \( p \) (in the language of HTV1) be the result of encoding \( CT \) with \( \mathcal{E} \). Let \( T_1 \) and \( T_2 \) be two types in the language
of Kennedy and Pierce such that:
\[ \emptyset \vdash T_1 \text{ ok and } \emptyset \vdash T_2 \text{ ok.} \]

If:
\[ T_1 <: T_2 \]

then:
\[ p; \emptyset \vdash \mathcal{E}(T_1) <: \mathcal{E}(T_2). \]

**Proof.** The proof is by induction on the derivation of:
\[ T_1 <: T_2. \]

We proceed by case analysis on the last rule in the derivation (see the report by Kennedy and Pierce for the definition of the two subtyping rules (VAR) and (SUPER)).

**Case (VAR):**
Then:
\[ T_1 = C(\vec{U}) \text{ and } T_2 = C(\vec{V}). \]

Let:
\[ C(v_1X_1 \ldots v_nX_n) <: \ldots \]

be in $CT$. Then:
\[
\begin{align*}
\text{trait } S[\text{bnd}_1 \ldots \text{bnd}_n] \\
\text{extends } \{S[Z_1 \ldots Z_n]\} \\
\text{where } \{\text{bnd}'_1 \ldots \text{bnd}'_n\}
\end{align*}
\]

is in $p$, where:
- $S = \mathcal{E}(C),$
- $Y_1 = \mathcal{E}(X_1) \ldots Y_n = \mathcal{E}(X_n),$
- $\text{bnd}_i = \begin{cases} Y_i \text{ extends } Z_i & \text{if } X_i \text{ is covariant} \\ Y_i \text{ bounds } Z_i & \text{if } X_i \text{ is contravariant} \\ Y_i \text{ extends } Z_i & \text{if } X_i \text{ is invariant} \end{cases}$
  where $1 \leq i \leq n,$ and
- $\text{bnd}'_i = \begin{cases} Z_i \text{ extends Object} & \text{if } X_i \text{ is covariant} \\ Z_i \text{ extends } Y_i & \text{if } X_i \text{ is contravariant} \\ Z_i \text{ extends } Y_i & \text{if } X_i \text{ is invariant} \end{cases}$
  where $1 \leq i \leq n.$

By rule (VAR), either:
\[ U_i <: V_i, \]
\[ V_i <: U_i, \text{ or } \]
\[ U_i = V_i \]
for $1 \leq i \leq n$. Therefore, by the induction hypothesis, either:

$$p; \emptyset \vdash \mathcal{E}(U_i) <: \mathcal{E}(V_i),$$
$$p; \emptyset \vdash \mathcal{E}(V_i) <: \mathcal{E}(U_i),$$
$$\mathcal{E}(U_i) = \mathcal{E}(V_i)$$

for $1 \leq i \leq n$. By Lemma 10:

$$p; \emptyset \vdash \mathcal{E}(U) \text{ ok and}$$
$$p; \emptyset \vdash \mathcal{E}(V) \text{ ok.}$$

By rules [P-EXT] and [S-PATH]:

$$p; \emptyset \vdash S[[\mathcal{E}(U)]] <: [\mathcal{E}(U)/Y][\mathcal{E}(V)/Z]S[Z].$$

Noticing that:

$$\mathcal{E}(T_1) = \mathcal{E}(C(\overrightarrow{U}))$$
$$= S[[\mathcal{E}(U)]]$$

and:

$$\mathcal{E}(T_2) = \mathcal{E}(C(\overrightarrow{V}))$$
$$= S[[\mathcal{E}(V)]]$$
$$= [\mathcal{E}(U)/Y][\mathcal{E}(V)/Z]S[Z]$$

finishes the case.

Case (SUPER):

Then:

$$T_1 = C(\overrightarrow{U}) \text{ and}$$
$$T_2 = D(\overrightarrow{U}')$$

where:

$$C \neq D,$$
$$C(v_1X_1\ldots v_nX_n) <: V, \text{ and}$$
$$[\overrightarrow{U}/X]V <: D(\overrightarrow{U}).$$

By the induction hypothesis:

$$p; \emptyset \vdash \mathcal{E}([\overrightarrow{U}/X]V) <: \mathcal{E}(D(\overrightarrow{U})).$$

Also:

$$\text{trait } S[bnd_1\ldots bnd_n]$$
$$\text{ extends } \{S[Z_1\ldots Z_n]/M\}$$
$$\text{ where } \{bnd_1'\ldots bnd_n'\}$$
$$\text{ end}$$

is in $p$, where:

- $S = \mathcal{E}(C)$,
- $Y_1 = \mathcal{E}(X_1) \ldots Y_n = \mathcal{E}(X_n)$,
\[ bnd_i = \begin{cases} Y_i \text{ extends } Z_i & \text{if } X_i \text{ is covariant} \\ Y_i \text{ bounds } Z_i & \text{if } X_i \text{ is contravariant} \\ Y_i \text{ extends } Z_i & \text{if } X_i \text{ is invariant} \end{cases} \]

where \( 1 \leq i \leq n \), and

\[ bnd'_i = \begin{cases} Z_i \text{ extends Object} & \text{if } X_i \text{ is covariant} \\ Z_i \text{ extends Object} & \text{if } X_i \text{ is contravariant} \\ Z_i \text{ extends } Y_i & \text{if } X_i \text{ is invariant} \end{cases} \]

where \( 1 \leq i \leq n \), and

\[ M = \mathcal{E}(V). \]

Notice that \( V \) does not contain any type variables beyond \( X_i \ldots X_n \). Therefore \( M \) does not contain any hidden type variables. By Lemma 10:

\[ p; \emptyset \vdash \mathcal{E}(\overline{U}) \text{ ok.} \]

Next we show that \( \mathcal{E}(\overline{U}) \) satisfies the bounds on the corresponding type variables. Let \( 1 \leq i \leq n \). We proceed by case analysis on the variance of \( X_i \).

Subcase \( X_i \) is covariant:

Then:

\[ bnd_i = Y_i \text{ extends } Z_i \text{ and} \]

\[ bnd'_i = Z_i \text{ extends Object}. \]

By rules \([P-OBJ]\) and \([S-PATH]\):

\[ p; \emptyset \vdash \mathcal{E}(U_i) <: \text{Object} \]

\[ p; \emptyset \vdash \text{Object} <: \text{Object}. \]

Therefore, \( \mathcal{E}(U_i) \) satisfies the bounds of \( Y_i \) with \( \text{Object} \) as a valid witness for \( Z_i \).

Subcase \( X_i \) is contravariant:

Then:

\[ bnd_i = Y_i \text{ bounds } Z_i \text{ and} \]

\[ bnd'_i = Z_i \text{ extends Object}. \]

By rules \([P-BOT]\) and \([S-PATH]\):

\[ p; \emptyset \vdash \text{Bottom} <: \mathcal{E}(U_i) \]

\[ p; \emptyset \vdash \text{Bottom} <: \text{Object}. \]

Therefore, \( \mathcal{E}(U_i) \) satisfies the bounds of \( Y_i \) with \( \text{Bottom} \) as a valid witness for \( Z_i \).

Subcase \( X_i \) is invariant:

Then:

\[ bnd_i = Y_i \text{ extends } Z_i \text{ and} \]

\[ bnd'_i = Z_i \text{ extends } Y_i. \]

By rule \([S-REFL]\):

\[ p; \emptyset \vdash \mathcal{E}(U_i) <: \mathcal{E}(U_i). \]
Therefore, $\mathcal{E}(U_i)$ satisfies the bounds of $Y_i$ with $\mathcal{E}(U_i)$ as a valid witness for $Z_i$.

Now we have shown that $\mathcal{E}(\overrightarrow{U})$ satisfies the bounds on the corresponding type variables. By rules [P-EXT] and [S-PATH]:

$$p; \emptyset \vdash S[\mathcal{E}(\overrightarrow{U})] <: [\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}]M.$$ 

Notice that:

$$[\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}]M = \mathcal{E}([\overrightarrow{U}/\overrightarrow{X}]V).$$ 

Therefore:

$$p; \emptyset \vdash [\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}]M <: \mathcal{E}(D(\overrightarrow{U})).$$ 

By rule [S-TRANS]:

$$p; \emptyset \vdash S[\mathcal{E}(\overrightarrow{U})] <: \mathcal{E}(D(\overrightarrow{U})).$$

Noticing that:

$$\mathcal{E}(T_1) = \mathcal{E}(C(\overrightarrow{U})) = S[\mathcal{E}(\overrightarrow{U})]$$

and:

$$\mathcal{E}(T_2) = \mathcal{E}(D(\overrightarrow{U}'))$$

finishes the case. □

**Lemma 12.** Let $CT$ be a class table in the language of Kennedy and Pierce. Let program $p$ (in the language of HTV1) be the result of encoding $CT$ with $\mathcal{E}$. Let $T_1$ and $T_2$ be two types in the language of Kennedy and Pierce such that:

$$\emptyset \vdash T_1 \text{ ok and}$$
$$\emptyset \vdash T_2 \text{ ok.}$$

If:

$$p; \emptyset \vdash \mathcal{E}(T_1) <: \mathcal{E}(T_2)$$

then:

$$T_1 <: T_2.$$ 

**Proof.** The proof is by induction on the size of the derivation of:

$$p; \emptyset \vdash \mathcal{E}(T_1) <: \mathcal{E}(T_2).$$

Case [S-REFL]:

Then $\mathcal{E}(T_1) = \mathcal{E}(T_2)$. Therefore $T_1 = T_2$ and rule (VAR), which is defined in the report by Kennedy and Pierce, finishes the case.

Case [S-TRANS]:

Then there exists $\mathcal{E}(T_3)$ such that:

$$p; \emptyset \vdash \mathcal{E}(T_1) <: \mathcal{E}(T_3) \text{ and}$$
$$p; \emptyset \vdash \mathcal{E}(T_3) <: \mathcal{E}(T_2).$$
By the induction hypothesis:

\[ T_1 <: T_3 \text{ and } T_3 <: T_2. \]

Kennedy and Pierce prove that the subtype relation is transitive. Therefore:

\[ T_1 <: T_2. \]

Case \([S-PATH]\):
Then the previous rule in the derivation must be \([P-EXT]\). Let:

\[ T_1 = C(\overrightarrow{U}) \]

and:

\[ C(v_1X_1\ldots v_nX_n) <: V \]

Then:

\[
\begin{align*}
\text{trait } S[\text{bnd}_1\ldots \text{bnd}_n] \\
\text{extends } \{S[Z_1\ldots Z_n] M\} \\
\text{where } \{\text{bnd}'_1\ldots \text{bnd}'_n\}
\end{align*}
\]

is in \(p\), where:

- \(S = \mathcal{E}(C)\),
- \(Y_1 = \mathcal{E}(X_1)\ldots Y_n = \mathcal{E}(X_n)\),
- \(\text{bnd}_i = \begin{cases} Y_i \text{ extends } Z_i & \text{if } X_i \text{ is covariant} \\ Y_i \text{ bounds } Z_i & \text{if } X_i \text{ is contravariant} \\ Y_i \text{ extends } Z_i & \text{if } X_i \text{ is invariant} \end{cases}\)
  where \(1 \leq i \leq n\),
- \(\text{bnd}'_i = \begin{cases} Z_i \text{ extends Object} & \text{if } X_i \text{ is covariant} \\ Z_i \text{ extends Object} & \text{if } X_i \text{ is contravariant} \\ Z_i \text{ extends } Y_i & \text{if } X_i \text{ is invariant} \end{cases}\)
  where \(1 \leq i \leq n\), and
- \(M = \mathcal{E}(V)\).

There are two subcases:

Subcase \(T_2 = C(\overrightarrow{U}')\):

By \([P-EXT]\), either:

\[
p; \emptyset \vdash \mathcal{E}(U_i) <: \mathcal{E}(U'_i),
\]

\[
p; \emptyset \vdash \mathcal{E}(U'_i) <: \mathcal{E}(U_i), \text{ or }
\]

\[
\mathcal{E}(U_i) = \mathcal{E}(U'_i)
\]
for \(1 \leq i \leq n\). By the induction hypothesis:

\[
\begin{align*}
p; \emptyset \vdash U_i \prec U_i', \\
p; \emptyset \vdash U_i' \prec U_i, \text{ or} \\
U_i = U_i'
\end{align*}
\]

for \(1 \leq i \leq n\). Rule (VAR), which is defined in the report by Kennedy and Pierce, finishes the case.

Subcase \(T_2 = D(U_i)\) where \(C \neq D\):

By rule [P-EXT]:

\[
\mathcal{E}(D(U_i)) = [\overrightarrow{A}/\overrightarrow{Z}]\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}M
\]

for some \(\overrightarrow{A}\). Notice that \(V\) does not contain any type variables beyond \(X_1 \ldots X_n\). Therefore \(M\) does not contain any hidden type variables. Therefore:

\[
\mathcal{E}(D(U_i')) = [\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}]M.
\]

Therefore:

\[
\mathcal{E}(D(U_i')) = [\mathcal{E}(\overrightarrow{U})/\overrightarrow{Y}]\mathcal{E}(V).
\]

By the definition of \(\mathcal{E}\):

\[
D(U_i') = [\overrightarrow{U}/\overrightarrow{X}]V.
\]

Rule (SUPER), which is defined in the report by Kennedy and Pierce, finishes the case. \(\square\)

**Lemma 13.** Let \(CT\) be a class table in the language of Kennedy and Pierce. Let program \(p\) (in the language of HTV1) be the result of encoding \(CT\) with \(E\). Let \(T_1\) and \(T_2\) be two types in the language of Kennedy and Pierce such that:

\[
\begin{align*}
\emptyset \vdash T_1 \text{ ok} & \quad \text{and} \\
\emptyset \vdash T_2 \text{ ok}
\end{align*}
\]

Then:

\[
T_1 \prec T_2
\]

if and only if:

\[
p; \emptyset \vdash \mathcal{E}(T_1) \prec \mathcal{E}(T_2).
\]

**Proof.** Follows from Lemmas 11 and 12. \(\square\)

**Theorem 4** (Undecidability of Subtyping in HTV1). Subtyping in HTV1 is undecidable.

**Proof.** Follows from Lemma 13 and the fact that subtyping in the language of Kennedy and Pierce is undecidable. \(\square\)
Appendix D

Soundness Proof for Constraint Solving Algorithm

This appendix proves the soundness of the constraint solving algorithm presented in Section 9.3.1. Notice that Lemmas 14 and 15 refer to the algorithms SEARCH and SIMPLIFY, respectively. These algorithms are mutually recursive. As a result, the proofs of the lemmas refer to one another. To ensure that there are no cycles in the proofs, Lemmas 14 and 15 are proved by simultaneous induction. However, for the purpose of readability, the two lemmas are presented separately.

Lemma 14. If:

- $p$ is a well-typed program, and
- $\Delta$ is a type variable environment, and
- $A$ and $B$ are types, and
- $\text{SEARCH}_{p,\Delta}(A <: B)$ returns $(\text{path}, \text{simplified}, \text{sub})$, and
- paths is a list of types, and
- simplified is a list of constraints, and
- sub is a substitution, and
- for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified})$, simplified[$i$] is of the form $C_i <: D_i$, and
- there exists a substitution sub$_1$, and
- for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified})$,
- $p; \Delta \vdash \text{APPLY-SUB}(\text{sub$_1$, C$_i$}) <: \text{APPLY-SUB}(\text{sub$_1$, D$_i$})$

then:

- $\text{COMPOSE}(\text{sub}, \text{sub$_1$})$ returns sub$_2$, and
- $p; \Delta \vdash \text{APPLY-SUB}(\text{sub$_2$, path})$ path-ok, and
- $\text{APPLY-SUB}(\text{sub$_2$, path}) = \text{APPLY-SUB}(\text{sub$_2$, A}) \underbrace{E}_{\text{$E$ is a list of types}} \text{APPLY-SUB}(\text{sub$_2$, B})$, where

Proof. The proof proceeds by induction on the size of path using Lemma 15. Each type in path is added by one of the cases of $\text{SEARCH}_{p,\Delta}(A <: B)$. Each case corresponds directly to a path rule (the second case corresponds to two path rules). In particular, the second case corresponds to rules [P-OBJ] and [P-B OT], the third and fourth to rule [P-UNIONSUPER], the fifth to rule...
[P-UNIONSUB], the sixth to rule [P-EXT], and the seventh to rule [P-VAR]. The first case doesn’t correspond to a rule. Instead, it corresponds to situation where two types are equal.

Lemma 15. If:

\[ p \text{ is a well-typed program, and} \]
\[ \Delta \text{ is a type variable environment, and} \]
\[ \text{constraints is a list of constraints, and} \]
\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{constraints}),} \]
\[ \text{constraints}[i] \text{ is of the form } A_i <: B_i, \text{ and} \]
\[ \text{SIMPLIFY}_{p, \Delta}(\text{constraints}) \text{ returns } (\text{paths}, \text{simplified}, \text{sub}), \text{ and} \]
\[ \text{paths is a list of paths, and} \]
\[ \text{simplified is a list of constraints, and} \]
\[ \text{sub is a substitution, and} \]
\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}),} \]
\[ \text{simplified}[i] \text{ is of the form } C_i <: D_i, \text{ and} \]
\[ \text{there exists a substitution } sub_1, \text{ and} \]
\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}),} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(sub_1, C_i) <: \text{APPLY-SUB}(sub_1, D_i) \]

then for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \), either:

\[ \text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE}(\text{simplified}), \text{ and} \]
\[ \text{constraints}[i] = \text{simplified}[j] \]

or:

\[ \text{COMPOSE}(sub, sub_1) \text{ returns } sub_2, \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(sub_2, \text{paths}[i]) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(sub_2, \text{paths}[i]) = \text{APPLY-SUB}(sub_2, A_i) \overset{\overline{E}}{\rightarrow} \text{APPLY-SUB}(sub_2, B_i), \text{ where} \]
\[ \overline{E} \text{ is a list of types.} \]

Proof. The proof proceeds by induction on the number of recursive calls made by \( \text{SIMPLIFY}_{p, \Delta}(\text{constraints}) \) and nested induction on the size of \( \text{constraints} \).

Case no recursive calls:
Notice that the result vacuously holds if \( \text{constraints} \) is empty. Otherwise, suppose:

\[ \text{constraints} = \text{APPEND}(\text{constraints}', \text{CONS}(G <: H, \text{NIL})). \]

Let:

\[ \text{SIMPLIFY}_{p, \Delta}(\text{constraints}') \text{ return } (\text{paths}', \text{simplified}', \text{sub}'). \]

There are two cases.
Subcase \( \text{BOUND}_\Delta(G <: H) \) returns true:

Then:

\[ \text{SIMPLIFY}_{p, \Delta}(\text{constraints}) \text{ returns } (\text{paths}', \text{CONS}(G <: H, \text{simplified}'), \text{sub}'). \]
By the induction hypothesis, if:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \),
\( \text{constraints}'[i] \) is of the form \( A'_i \ <: \ B'_i \),
\( \text{SIMPLIFY}_{p,\Delta}(\text{constraints}') \) returns \((\text{paths}', \text{simplified}', \text{sub}')\), and
\( \text{paths}' \) is a list of paths, and
\( \text{simplified}' \) is a list of constraints, and
\( \text{sub}' \) is a substitution, and

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{simplified}') \),
\( \text{simplified}'[i] \) is of the form \( C'_i \ <: \ D'_i \), and
there exists a substitution \( \text{sub}'_1 \), and

then for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \), either:

there exists \( j \) such that \( 1 \leq j \leq \text{SIZE}(\text{simplified}') \), and
\( \text{constraints}'[i] = \text{simplified}'[j] \)

or:

\( \text{COMPOSE}(\text{sub}', \text{sub}'_1) \) returns \( \text{sub}'_2 \), and
\( p; \Delta \vdash \text{APPLY-SUB}(\text{sub}'_2, \text{paths}'[i]) \) path-ok, and
\( \text{APPLY-SUB}(\text{sub}'_2, \text{paths}'[i]) = \text{APPLY-SUB}(\text{sub}'_2, A'_i) \overrightarrow{E} \text{APPLY-SUB}(\text{sub}'_2, B'_i) \), where \( \overrightarrow{E} \) is a list of types.

Notice that if there exists \( \text{sub}_1 \) such that:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{simplified}) \),
\( p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C_i) <: \text{APPLY-SUB}(\text{sub}_1, D_i) \)

then:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{simplified}) \),
\( p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C'_i) <: \text{APPLY-SUB}(\text{sub}_1, D'_i) \).

Therefore the induction hypothesis gives the result.

Subcase \( \text{BOUND}_\Delta(G <: H) \) returns \text{false}:

Let:

\( (\text{path}'', \text{simplified}'', \text{sub}'') \leftarrow \text{SEARCH}_{p,\Delta}(G <: H) \)
\( \text{paths}'' \leftarrow \text{APPEND}(\text{paths}', \text{CONS}(\text{path}'', \text{NIL})) \)
\( \text{simplified}''' \leftarrow \text{APPEND}(\text{simplified}', \text{simplified}'') \)
\( \text{sub}''' \leftarrow \text{COMPOSE}(\text{sub}', \text{sub}'') \)
\( \text{paths}''' \leftarrow \text{APPLY-SUB}(\text{sub}''', \text{paths}'') \)
\( \text{simplified}''' \leftarrow \text{APPLY-SUB}(\text{sub}''', \text{simplified}''') \)

Then:

\( \text{SIMPLIFY}_{p,\Delta}(\text{constraints}) \) returns \((\text{paths}'', \text{simplified}''', \text{sub}'')\).
Notice that if there exists $s1$ such that:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified})$, 

\[ p; \Delta \vdash \text{APPLY-SUB}(s1, C_i) \prec \text{APPLY-SUB}(s1, D_i) \]

then:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified''''})$, 

\[ \text{simplified''''}[i] \] is of the form \[ C''''_i \prec D''''_i \], and 

\[ p; \Delta \vdash \text{APPLY-SUB}(s1, C''''_i) \prec \text{APPLY-SUB}(s1, D''''_i) \]

Therefore:

\[ \text{COMPOSE}(s''''_1, s1) \] returns $s3$, and 

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified'})$, 

\[ \text{simplified'}[i] \] is of the form \[ C'_i \prec D'_i \], and 

\[ p; \Delta \vdash \text{APPLY-SUB}(s3, C'_i) \prec \text{APPLY-SUB}(s3, D'_i) \]

and:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified''''})$, 

\[ \text{simplified''''}[i] \] is of the form \[ C''''_i \prec D''''_i \], and 

\[ p; \Delta \vdash \text{APPLY-SUB}(s3, C''''_i) \prec \text{APPLY-SUB}(s3, D''''_i) \]

By the induction hypothesis:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{constraints'}$), 

\[ \text{constraints'}[i] \] is of the form \[ A'_i \prec B'_i \], and 

either:

there exists $j$ such that $1 \leq j \leq \text{SIZE}(\text{simplified'})$, and 

\[ \text{constraints'}[i] = \text{simplified'}[j] \]

or:

\[ \text{COMPOSE}(s'_1, s3) \] returns $s4$, and 

\[ p; \Delta \vdash \text{APPLY-SUB}(s3, \text{paths'}[i]) \text{ path-ok, and} \]

\[ \text{APPLY-SUB}(s4, \text{paths'}[i]) = \text{APPLY-SUB}(s4, A'_i) \overrightarrow{E} \text{APPLY-SUB}(s4, B'_i) \]

where $\overrightarrow{E}$ is a list of types.

Notice that:

\[ s4 = s3. \]

Therefore, for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{constraints'})$ either:

there exists $j$ such that $1 \leq j \leq \text{SIZE}(\text{simplified'})$, and 

\[ \text{constraints'}[i] = \text{simplified'}[j] \]

or:

\[ p; \Delta \vdash \text{APPLY-SUB}(s3, \text{paths'}[i]) \text{ path-ok, and} \]

\[ \text{APPLY-SUB}(s3, \text{paths'}[i]) = \text{APPLY-SUB}(s3, A'_i) \overrightarrow{E} \text{APPLY-SUB}(s3, B'_i) \]

where $\overrightarrow{E}$ is a list of types.
By Lemma 14:

\[
\text{COMPOSE}(\text{sub}'', \text{sub}_3) \text{ returns } \text{sub}_5, \text{ and } \\
p; \Delta \vdash \text{APPLY-SUB(} \text{sub}_5, \text{path}'') \text{ path-ok, and } \\
\text{APPLY-SUB(} \text{sub}_5, \text{path}'') = \text{APPLY-SUB(} \text{sub}_5, G) \overline{\text{E}} \text{ APPLY-SUB(} \text{sub}_5, H), \text{ where } \\
\overline{\text{E}} \text{ is a list of types.}
\]

Notice that:

\[
\text{sub}_5 = \text{sub}_3.
\]

Therefore:

\[
p; \Delta \vdash \text{APPLY-SUB(} \text{sub}_3, \text{path}'') \text{ path-ok, and } \\
\text{APPLY-SUB(} \text{sub}_3, \text{path}'') = \text{APPLY-SUB(} \text{sub}_3, G) \overline{\text{E}} \text{ APPLY-SUB(} \text{sub}_3, H), \text{ where } \\
\overline{\text{E}} \text{ is a list of types.}
\]

Noticing that:

\[
\text{COMPOSE}(\text{sub}'', \text{sub}_1) \text{ returns } \text{sub}_3
\]

gives the result.

Case recursive calls:
Notice that recursive calls cannot be made if \text{constraints} is empty. Therefore, suppose:

\[
\text{constraints} = \text{APPEND}(\text{constraints}', \text{CONS}(G <: H, \text{NIL})).
\]

Let:

\[
\text{SIMPIFY}_{p, \Delta}(\text{constraints}') \text{ return (paths', simplified', sub').}
\]

There are two cases.

Subcase \text{BOUND}_\Delta(G <: H) \text{ returns true:}

Then, let:

\[
\text{constraints}'' = \text{CONS}(G <: H, \text{ simplified'}). 
\]

The recursive call is:

\[
(\text{paths}'', \text{ simplified}'', \text{sub}'') \leftarrow \text{SIMPIFY}_{p, \Delta}(\text{constraints}'')
\]

and:

\[
\text{sub}'' \leftarrow \text{COMPOSE}(\text{sub}', \text{sub}'') \\
\text{paths}''' \leftarrow \text{APPLY-SUB(} \text{sub}'', \text{paths}')
\]

and:

\[
\text{SIMPIFY}_{p, \Delta}(\text{constraints}) \text{ returns (paths''', simplified''', sub''').}
\]

Notice that if there exists \text{sub}_1 \text{ such that:}

- for each \( i \) \text{ such that } 1 \leq i \leq \text{SIZE(simplified)},
- \text{simplified}[i] \text{ is of the form } C_i <: D_i, \text{ and } \\
p; \Delta \vdash \text{APPLY-SUB(} \text{sub}_1, C_i) <: \text{APPLY-SUB(} \text{sub}_1, D_i)
then:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified}')$,
$simplified''[i]$ is of the form $C_i'' <: D_i''$, and
$p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C_i'') <: \text{APPLY-SUB}(\text{sub}_1, D_i'')$.

By the induction hypothesis:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{constraints}'')$,
$\text{constraints}''[i]$ is of the form $A_i'' <: B_i''$, and

either:

there exists $j$ such that $1 \leq j \leq \text{SIZE}(\text{simplified}'')$, and
$\text{constraints}''[i] = \text{simplified}''[j]$

or:

$\text{COMPOSE}(\text{sub}'', \text{sub}_1)$ returns $\text{sub}_2$, and
$p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_2, \text{paths}''[i]) \text{ path-ok}$, and
$\text{APPLY-SUB}(\text{sub}_2, \text{paths}''[i]) = \text{APPLY-SUB}(\text{sub}_2, A_i'') \overrightarrow{E} \text{ APPLY-SUB}(\text{sub}_2, B_i'')$, where
$\overrightarrow{E}$ is a list of types.

Therefore:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{simplified}')$,
$simplified'[i]$ is of the form $C_i' <: D_i'$, and
$p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_2, C_i') <: \text{APPLY-SUB}(\text{sub}_2, D_i')$.

By the induction hypothesis:

for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{constraints}')$,
$\text{constraints}'[i]$ is of the form $A_i' <: B_i'$, and

either:

there exists $j$ such that $1 \leq j \leq \text{SIZE}(\text{simplified}')$, and
$\text{constraints}'[i] = \text{simplified}'[j]$

or:

$\text{COMPOSE}(\text{sub}', \text{sub}_2)$ returns $\text{sub}_3$, and
$p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) \text{ path-ok}$, and
$\text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) = \text{APPLY-SUB}(\text{sub}_3, A_i') \overrightarrow{E} \text{ APPLY-SUB}(\text{sub}_3, B_i')$, where
$\overrightarrow{E}$ is a list of types.

Notice that:

$\text{APPLY-SUB}(\text{sub}_3, \text{paths}') = \text{APPLY-SUB}(\text{sub}_3, \text{paths}'')$.

Therefore, for each $i$ such that $1 \leq i \leq \text{SIZE}(\text{constraints}')$ either:

there exists $j$ such that $1 \leq j \leq \text{SIZE}(\text{simplified}')$, and
$\text{constraints}'[i] = \text{simplified}'[j]$
or:

\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths}''''[i]) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_3, \text{paths}''''[i]) = \text{APPLY-SUB}(\text{sub}_3, A'_i) \text{ \text{E} \text{APPLY-SUB}(sub}_3, B'_i), \]
where \text{E} is a list of types.

Noticing that:

\[ \text{COMPOSE}(\text{sub}'''', \text{sub}_1) \text{ returns } \text{sub}_3 \text{ and} \]
\[ \text{simplified}'''' \text{ contains } G <: H \]
gives the result.

Subcase BOUND\(_\Delta(G <: H)\) returns false:
Let:

\[ \text{Simul}_p, \Delta(G <: H) \]
\[ \text{paths}'''' \leftarrow \text{APPLY-SUB}(\text{sub}''''', \text{paths}'''') \]
\[ \text{simplified}'''''' \leftarrow \text{APPLY-SUB}(\text{sub}''''', \text{simplified}'''') \]
\[ \text{sub}'''''' \leftarrow \text{COMPOSE}(\text{sub}', \text{sub}''''') \]
\[ \text{paths}'''''' \leftarrow \text{APPLY-SUB}(\text{sub}''''', \text{paths}'''') \]
\[ \text{sub}'''''''' \leftarrow \text{COMPOSE}(\text{sub}, \text{sub}''''''') \]
\[ \text{paths}'''''''' \leftarrow \text{APPLY-SUB}(\text{sub}'''''''', \text{paths}''''''') \]

The recursive call is:

\[ (\text{paths}'''''', \text{simplified}'''''', \text{sub}'''''') \leftarrow \text{SIMPLIFY}_p, \Delta(\text{simplified}'''''') \]

and

\[ \text{sub}''''' \leftarrow \text{COMPOSE}(\text{sub}', \text{sub}''''') \]
\[ \text{paths}''''' \leftarrow \text{APPLY-SUB}(\text{sub}'''''', \text{paths}'''') \]

and:

\[ \text{SIMPLIFY}_p, \Delta(\text{constraints}) \text{ returns } (\text{paths}''''', \text{simplified}'''''', \text{sub}'''''') \]

Notice that if there exists \text{sub}_1 \text{ such that:}

\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}), \]
\[ \text{simplified}[i] \text{ is of the form } C_i <: D_i, \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C_i) <: \text{APPLY-SUB}(\text{sub}_1, D_i) \]
then:

\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}'''''), \]
\[ \text{simplified}'''''[i] \text{ is of the form } C'''_{i} <: D'''_{i}, \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C'''_{i}) <: \text{APPLY-SUB}(\text{sub}_1, D'''_{i}) \]

By the induction hypothesis:

\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}''''''), \]
\[ \text{simplified}''''''[i] \text{ is of the form } C'''_{i} <: D'''_{i}, \text{ and} \]

either:

\[ \text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE}(\text{simplified}'''''), \text{ and} \]
\[ \text{simplified}'''''[i] = \text{simplified}''''''[j] \]

\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}''''''), \]
\[ \text{simplified}''''''[i] \text{ is of the form } C'''_{i} <: D'''_{i}, \text{ and} \]

or:

\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths}'''[i]) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_3, \text{paths}'''[i]) = \text{APPLY-SUB}(\text{sub}_3, A'_i) \text{ \text{E} \text{APPLY-SUB}(sub}_3, B'_i), \]
where \text{E} is a list of types.
or:

\[
\text{COMPOSE}(\text{sub}''', \text{sub}_1) \text{ returns } \text{sub}_2, \text{ and }
\]
\[
p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_2, \text{paths}'''; [i]) \text{ path-ok, and }
\]
\[
\text{APPLY-SUB}(\text{sub}_2, \text{paths}'''; [i]) = \text{APPLY-SUB}(\text{sub}_2, \text{C}'''; i) \overrightarrow{E} \text{APPLY-SUB}(\text{sub}_2, \text{D}'''; i), \text{ where } \overrightarrow{E} \text{ is a list of types.}
\]

Therefore:

\[
\text{COMPOSE}(\text{sub}''', \text{sub}_2) \text{ returns } \text{sub}_3, \text{ and }
\]
\[
\text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE} (\text{simplified}'),
\]
\[
\text{simplified}'[i] \text{ is of the form } \text{C}''; i <: \text{D}''; i, \text{ and }
\]
\[
p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{C}''; i) <: \text{APPLY-SUB}(\text{sub}_3, \text{D}''; i)
\]

and:

\[
\text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE} (\text{simplified}''),
\]
\[
\text{simplified}''[i] \text{ is of the form } \text{C}'''; i <: \text{D}'''; i, \text{ and }
\]
\[
p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{C}'''; i) <: \text{APPLY-SUB}(\text{sub}_3, \text{D}'''; i)
\]

By the induction hypothesis:

\[
\text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE} (\text{constraints}''),
\]
\[
\text{constraints}'[i] \text{ is of the form } \text{A}''; i <: \text{B}''; i, \text{ and }
\]

either:

\[
\text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE} (\text{simplified}'), \text{ and }
\]
\[
\text{constraints}'[i] = \text{simplified}'[j]
\]

or:

\[
\text{COMPOSE}(\text{sub}', \text{sub}_3) \text{ returns } \text{sub}_4, \text{ and }
\]
\[
p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) \text{ path-ok, and }
\]
\[
\text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) = \text{APPLY-SUB}(\text{sub}_4, \text{A}'[i]) \overrightarrow{E} \text{APPLY-SUB}(\text{sub}_4, \text{B}'[i]), \text{ where } \overrightarrow{E} \text{ is a list of types.}
\]

Notice that:

\[
\text{sub}_4 = \text{sub}_3.
\]

Therefore, for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}') \) either:

\[
\text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE} (\text{simplified}'), \text{ and }
\]
\[
\text{constraints}'[i] = \text{simplified}'[j]
\]

or:

\[
p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) \text{ path-ok, and }
\]
\[
\text{APPLY-SUB}(\text{sub}_3, \text{paths}'[i]) = \text{APPLY-SUB}(\text{sub}_3, \text{A}'[i]) \overrightarrow{E} \text{APPLY-SUB}(\text{sub}_3, \text{B}'[i]), \text{ where } \overrightarrow{E} \text{ is a list of types.}
\]
By Lemma 14:

\[ \text{COMPOSE}(\text{sub''''}, \text{sub}_3) \text{ returns } \text{sub}_5, \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_5, \text{path''}) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_5, \text{path''}) = \text{APPLY-SUB}(\text{sub}_5, G) \ \overrightarrow{E} \ \text{APPLY-SUB}(\text{sub}_5, H), \text{ where} \]
\[ \overrightarrow{E} \text{ is a list of types.} \]

Notice that:

\[ \text{sub}_5 = \text{sub}_3. \]

Therefore:

\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{path''}) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_3, \text{path''}) = \text{APPLY-SUB}(\text{sub}_3, G) \ \overrightarrow{E} \ \text{APPLY-SUB}(\text{sub}_3, H), \text{ where} \]
\[ \overrightarrow{E} \text{ is a list of types.} \]

Therefore, for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \) either:

\[ \text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE}(\text{simplified''''}), \text{ and} \]
\[ \text{constraints}[i] = \text{simplified''''}[j] \]

or:

\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}[i]) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}[i]) = \text{APPLY-SUB}(\text{sub}_3, A_i) \ \overrightarrow{E} \ \text{APPLY-SUB}(\text{sub}_3, B_i), \text{ where} \]
\[ \overrightarrow{E} \text{ is a list of types.} \]

Notice that:

\[ \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}) = \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}) \text{ and} \]
\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified''''}), \]
\[ \text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE}(\text{simplified''''}), \text{ and} \]
\[ \text{simplified''''}[i] = \text{simplified''''}[j]. \]

Therefore, for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \) either:

\[ \text{there exists } j \text{ such that } 1 \leq j \leq \text{SIZE}(\text{simplified''''}), \text{ and} \]
\[ \text{constraints}[i] = \text{simplified''''}[j] \]

or:

\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}[i]) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_3, \text{paths''''}[i]) = \text{APPLY-SUB}(\text{sub}_3, A_i) \ \overrightarrow{E} \ \text{APPLY-SUB}(\text{sub}_3, B_i), \text{ where} \]
\[ \overrightarrow{E} \text{ is a list of types.} \]

Noticing that:

\[ \text{COMPOSE}(\text{sub''''''}, \text{sub}_1) \text{ returns } \text{sub}_3 \]

gives the result.
Lemma 16. *If:*

\[ p \text{ is a well-typed program, and} \]
\[ \Delta \text{ is a type variable environment, and} \]
\[ A \text{ and } B \text{ are types, and} \]
\[ \text{SEARCH}_{p, \Delta}(A < B) \text{ returns } (\text{path, simplified, } \text{sub}), \text{ and} \]
\[ \text{FREE-VARS}_{\Delta}(A) \text{ returns } \emptyset, \text{ and} \]
\[ \text{FREE-VARS}_{\Delta}(B) \text{ returns } \emptyset \]

\[ \text{then } p; \Delta \vdash A < B. \]

*Proof.* Notice that:

\[ \text{FREE-VARS}_{\Delta}(A) \text{ returns } \emptyset \text{ and} \]
\[ \text{FREE-VARS}_{\Delta}(B) \text{ returns } \emptyset \]

implies that the only free type variables in *simplified* are hidden type variables in program \( p \). The fact that \( p \) is well typed implies that for each type \( A \) in \( p \) there exists a valid witness for each hidden type variable that occurs in \( A \). Therefore, there exists a substitution \( \text{sub}_1 \) such that:

\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{simplified}), \]
\[ \text{simplified}[i] \text{ is of the form } C_i < D_i \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_1, C_i) < \text{APPLY-SUB}(\text{sub}_1, D_i). \]

By Lemma 14:

\[ \text{COMPOSE}(\text{sub}, \text{sub}_1) \text{ returns } \text{sub}_2, \text{ and} \]
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_2, \text{path}) \text{ path-ok, and} \]
\[ \text{APPLY-SUB}(\text{sub}_2, \text{path}) = \text{APPLY-SUB}(\text{sub}_2, A) \overrightarrow{E} \text{APPLY-SUB}(\text{sub}_2, B), \text{ where} \]
\[ \overrightarrow{E} \text{ is a list of types.} \]

From:

\[ \text{FREE-VARS}_{\Delta}(A) \text{ returns } \emptyset \text{ and} \]
\[ \text{FREE-VARS}_{\Delta}(B) \text{ returns } \emptyset \]

we have:

\[ \text{APPLY-SUB}(\text{sub}_2, \text{path}) = A \overrightarrow{E} B. \]

Therefore, the result follows from rules \([\text{MULTIPLE-PATH}], [\text{S-PATH}], \text{ and } [\text{S-TRANS}]\). \(\square\)

Lemma 17. *If:*

\[ p \text{ is a well-typed program, and} \]
\[ \Delta \text{ is a type variable environment, and} \]
\[ \text{constraints is a list of constraints, and} \]
\[ \text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}((\text{constraints})}, \]
\[ \text{constraints}[i] \text{ is of the form } A_i < B_i, \text{ and} \]
\[ \text{FREE-VARS}_{\Delta}(\text{constraints}) \text{ returns } \emptyset, \text{ and} \]
\[ \text{CHECK-WITNESSES}_{p, \Delta}(\text{constraints}) \text{ returns } \text{true} \]
then:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \),
\[ p; \Delta \vdash A_i \prec B_i. \]

Proof. Follows from applying Lemma 16 to each constraint. \( \square \)

**Theorem 5 (Soundness of Constraint Solving).** If:

\[
p \text{ is a well-typed program, and} \\
\Delta \text{ is a type variable environment, and} \\
\text{constraints is a list of constraints, and} \\
\text{for each } i \text{ such that } 1 \leq i \leq \text{SIZE}(\text{constraints}), \\
\text{constraints}[i] \text{ is of the form } A_i \prec B_i, \text{ and} \\
\text{SOLVE}_{p,\Delta}(\text{constraints}) \text{ returns } (\text{paths}, \text{sub})
\]

then:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{constraints}) \),
\[ p; \Delta \vdash \text{APPLY-SUB}(\text{sub}_i, A_i) \prec \text{APPLY-SUB}(\text{sub}_i, B_i). \]

Proof. Let:

\[
\text{SOLVE}_{p,\Delta}(\text{constraints}) = \]
\[
(\text{paths}_1, \text{simplified}_1, \text{sub}_1) \leftarrow \text{SIMP}LIFY_{p,\Delta}(\text{constraints}) \\
\text{sub}_2 \leftarrow \text{CONSTRUCT-WITNESSES}_{\Delta}(\text{simplified}_1) \\
\text{simplified}_2 \leftarrow \text{APPLY-SUB}(\text{sub}_2, \text{simplified}_1) \\
\text{paths}_2 \leftarrow \text{APPLY-SUB}(\text{sub}_2, \text{paths}_1) \\
\text{sub}_3 \leftarrow \text{COMPOSE}(\text{sub}_1, \text{sub}_2) \\
\text{if CHECK-WITNESSES}_{p,\Delta}(\text{simplified}_2) \\
\hspace{1cm} \text{return } (\text{paths}_2, \text{sub}_3) \text{ else} \\
\hspace{1cm} \text{return error}
\]

Notice that \( \text{sub}_2 \) substitutes for all the free type variables in \( \text{simplified}_1 \). Therefore:

\[
\text{FREE-VARS}_{\Delta}(\text{simplified}_2) \text{ returns } \emptyset.
\]

By Lemma 17:

for each \( i \) such that \( 1 \leq i \leq \text{SIZE}(\text{simplified}_2) \),
\[ \text{simplified}_2[i] \text{ is of the form } C_i \prec D_i, \text{ and} \\
\text{p; } \Delta \vdash \text{APPLY-SUB}(\text{sub}_2, C_i) \prec \text{APPLY-SUB}(\text{sub}_2, D_i). \]

The result follows from Lemma 15. \( \square \)
Appendix E

Termination Proof for Constraint Solving Algorithm

Section E.1 shows modifications to the constraint solving pseudocode of Section 9.3.1, which guarantee that the algorithm terminates. Section E.2 proves that the modified algorithm terminates.

E.1 Pseudocode for Terminating Constraint Solving Algorithm

This section shows the parts of the constraint solving pseudocode that are modified for termination. We define three additional auxiliary functions:

\[ \text{CONTAINS}_\Delta (\text{list}, \text{element}) \]
If list contains element (modulo renaming of free type variables) then true. Otherwise false.

\[ \text{NOT-CONTAINS}_\Delta (\text{list}, \text{element}) \]
If list does not contain element (modulo renaming of free type variables) then true. Otherwise false.

We highlight the lines of the pseudocode that have been modified by placing boxes around them.

\[
\begin{align*}
\text{T-SEARCH}_{p,\Delta}(A &<: B, table) = \\
\text{if CONTAINS}_\Delta(table, A <: B) &\quad \text{return error} \\
\text{else} &\quad table \leftarrow \text{CONS}(A <: B, table)
\end{align*}
\]
\[ \begin{align*}
\text{path} & \leftarrow \text{CONS}(A, \text{NIL}) \\
\text{constraints} & \leftarrow \text{NIL} \\
\text{queue} & \leftarrow \text{ENQUEUE}((\text{path}, \text{constraints}), \text{NIL})
\end{align*} \]

\textbf{while} NOT-EMPTY(\text{queue})

\[ \begin{align*}
(\text{path}, \text{constraints}, \text{queue}) & \leftarrow \text{DEQUEUE}(\text{queue}) \\
C & \leftarrow \text{LAST}(\text{path})
\end{align*} \]

\textbf{if} \text{UNIFY}_\Delta(C, B) \text{ returns } \text{sub} \\
\text{and } \text{sub} \not= \text{false} \text{ and } \text{T-SIMPLIFY}_{p, \Delta}(\text{APPLY-SUB}((\text{sub}, \text{constraints}), \text{table})) \text{ does not return error}

\[ \begin{align*}
(\text{simplified}, \text{sub}') & \leftarrow \text{T-SIMPLIFY}_{p, \Delta}(\text{APPLY-SUB}((\text{sub}, \text{constraints}), \text{table})) \\
\text{sub} & \leftarrow \text{COMPOSE}(\text{sub}, \text{sub}') \\
\text{path} & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
\text{return } (\text{path}, \text{simplified}, \text{sub})
\end{align*} \]

\textbf{else if} (\(B = \text{Object or } C = \text{Bottom}\)) \\
\text{and } \text{T-SIMPLIFY}_{p, \Delta}(\text{constraints}, \text{table}) \text{ does not return error}

\[ \begin{align*}
(\text{simplified}, \text{sub}) & \leftarrow \text{T-SIMPLIFY}_{p, \Delta}(\text{constraints}, \text{table}) \\
\text{path} & \leftarrow \text{APPEND}(\text{path}, \text{CONS}(B, \text{NIL})) \\
\text{path} & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
\text{return } (\text{path}, \text{simplified}, \text{sub})
\end{align*} \]

\textbf{else if} \(B\) \text{ is of the form } D \cup E \\
\text{and } \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(C <: D, \text{constraints}), \text{table}) \text{ does not return error}

\[ \begin{align*}
(\text{paths}, \text{simplified}, \text{sub}) & \leftarrow \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(C <: D, \text{constraints}), \text{table}) \\
\text{path} & \leftarrow \text{APPEND}(\text{APPEND}(\text{path}, \text{paths}[1]), \text{CONS}(B, \text{NIL})) \\
\text{path} & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
\text{return } (\text{path}, \text{simplified}, \text{sub})
\end{align*} \]
else if \( B \) is of the form \( D \cup E \)

and \( \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(C \prec E, \text{constraints}), \text{table}) \) does not return error

\[
\text{return error}
\]

\[
\begin{align*}
\text{paths, simplified, sub} & \leftarrow \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(C \prec E, \text{constraints}), \text{table}) \\
path & \leftarrow \text{APPEND}(\text{APPEND}(\text{path}, \text{paths}[1]), \text{CONS}(B, \text{NIL})) \\
path & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
\text{return (path, simplified, sub)} & 
\end{align*}
\]

else if \( C \) is of the form \( D \cup E \)

and \( \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(D \prec B, \text{CONS}(E \prec B, \text{constraints})), \text{table}) \) does not return error

\[
\begin{align*}
\text{paths, simplified, sub} & \leftarrow \text{T-SIMPLIFY}_{p, \Delta}(\text{CONS}(D \prec B, \text{CONS}(E \prec B, \text{constraints})), \text{table}) \\
path & \leftarrow \text{APPEND}(\text{path}, \text{CONS}(B, \text{NIL})) \\
path & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{path}) \\
\text{return (path, simplified, sub)} & 
\end{align*}
\]

else if \( C \) is of the form \( S\{\overrightarrow{D}\} \)

if \( p \) contains a declaration of the form

\[
- S'[\overrightarrow{X - K}] \quad \text{extends} \quad \{\overrightarrow{M} \} \quad \text{where} \quad \{\overrightarrow{Y - L}\} \quad \text{end}
\]

where \( S' = S \)

\[
\begin{align*}
\overrightarrow{Z} & \leftarrow \text{FRESH-VARS}(\overrightarrow{Y}) \\
\text{sub} & \leftarrow \text{COMPOSE}([\overrightarrow{D / X}], [\overrightarrow{Z / Y}]) \\
\text{constraints'} & \leftarrow \text{APPEND}((\text{convert}(\overrightarrow{X - K}), \text{convert}(\overrightarrow{Y - L}))) \\
\text{constraints'} & \leftarrow \text{APPEND}(\text{constraints}, \text{constraints'}) \\
\text{constraints'} & \leftarrow \text{APPLY-SUB}(\text{sub}, \text{constraints'}) \\
\text{for} \quad i & \leftarrow 1 \text{ to } \text{SIZE}(\overrightarrow{M}) \\
\text{if NOT-CONTAINS}_{\Delta}(\text{path}, M_i) & \\
\text{path'} & \leftarrow \text{APPEND}(\text{path}, \text{CONS}(\text{APPLY-SUB}(\text{sub}, M_i), \text{NIL})) \\
\text{queue} & \leftarrow \text{ENQUEUE}((\text{path'}, \text{constraints'}), \text{queue}) \\
\text{uppers} & \leftarrow \text{UPPER-BOUNDS}_{\Delta}(C) \\
\text{for} \quad i & \leftarrow 1 \text{ to } \text{SIZE}(\text{uppers}) \\
\text{if NOT-CONTAINS}_{\Delta}(\text{path}, \text{uppers}[i]) & \\
\text{path'} & \leftarrow \text{APPEND}(\text{path}, \text{CONS}(\text{uppers}[i], \text{NIL})) \\
\text{queue} & \leftarrow \text{ENQUEUE}((\text{path'}, \text{constraints'}), \text{queue}) \\
\text{else} & \\
\text{return error}
\end{align*}
\]
else
  uppers ← UPPERS\(\Delta\)(\(C\))
  for \(i ← 1\) to SIZE(uppers)
  if NOT-CONTAINS\(\Delta\)(\(path, uppers[i]\))
  \(path' ← APPEND(path, CONS(uppers[i], NIL))\)
  \(queue ← ENQUEUE((path', constraints), queue)\)

return error

T-SIMPLIFY\(_{p, \Delta}\)(\(constraints, table\)) =

\(paths ← NIL\)
\(simplified ← NIL\)
\(sub ← NIL\)
for \(i ← 1\) to SIZE(\(constraints\))

constraint ← constraints[\(i\)]
if BOUND\(\Delta\)(\(constraint\))
  \(simplified ← CONS(constraint, simplified)\)
  \(paths ← APPEND(paths, CONS(false, NIL))\)
else
  \((path, simplified', sub') ← T-SEARCH\(_{p, \Delta}\)(constraint, table)\)
  \(paths ← APPEND(paths, CONS(path, NIL))\)
  \(simplified ← APPEND(simplified, simplified')\)
  \(constraints ← APPLY-SUB(sub', constraints)\)
  \(sub ← COMPOSE(sub, sub')\)
  \(paths ← APPLY-SUB(sub, paths)\)
  \(simplified ← APPLY-SUB(sub, simplified)\)
  if for all \(i\) such that \(1 ≤ i ≤ SIZE(simplified)\) we have BOUND\(\Delta\)(\(simplified[i]\))
  return \((paths, simplified, sub)\)
else
  \((..., simplified'', sub'') ← T-SIMPLIFY\(_{p, \Delta}\)(simplified, table)\)
  \(sub ← COMPOSE(sub, sub'')\)
  \(paths ← APPLY-SUB(sub, paths)\)
  return \((paths, simplified', sub)\)

T-CHECK-WITNESSES\(_{p, \Delta}\)(\(constraints\)) =
  for \(i ← 1\) to SIZE(\(constraints\))
  if T-SEARCH\(_{p, \Delta}\)(\(constraints[i]\), NIL) = error
    return false
  return true
\[
\text{T-SOLVE}_{p,\Delta}(\text{constraints}) = \\
(\text{paths}, \text{simplified}, \text{sub}) \leftarrow \text{T-SIMPLIFY}_{p,\Delta}(\text{constraints}, \text{NIL})
\]

\[
\text{sub}' \leftarrow \text{CONSTRUCT-WITNESSES}_\Delta(\text{simplified})
\]

\[
\text{simplified} \leftarrow \text{APPLY-SUB}(\text{sub}', \text{simplified})
\]

\[
\text{paths} \leftarrow \text{APPLY-SUB}(\text{sub}', \text{paths})
\]

\[
\text{sub} \leftarrow \text{COMPOSE}(\text{sub}, \text{sub}')
\]

\[
\text{if T-CHECK-WITNESSES}_{p,\Delta}(\text{simplified})
\]

\[
\text{return (paths, sub)}
\]

\[
\text{else}
\]

\[
\text{return error}
\]

E.2 Proof of Termination

This section of the appendix proves that the constraint solving algorithm presented in Section E.1 terminates. Notice that Lemmas 19 and 20 refer to the algorithms T-SEARCH and T-SIMPLIFY, respectively. These algorithms are mutually recursive. As a result, the proofs of the lemmas refer to one another. To ensure that there are no cycles in the proofs, Lemmas 19 and 20 are proved simultaneously. However, for the purpose of readability, the two lemmas are presented separately.

Lemma 18. If there is a fixed maximum nesting depth for types then there is a finite number of distinct types (modulo renaming of free type variables) for a given program and type variable environment.

Proof. By induction on the nesting depth limit. If the nesting depth limit is one then the only well-formed types are: Object, Bottom, type variables, trait or object types without any type arguments, and union types whose component types have a depth of one. For any given type variable environment, there is a finite number of bound (i.e., non-free) type variables. Also, in any given program, there is a finite number of trait and object definitions. Therefore, there is a finite number of trait or object types without any type arguments. Lastly, notice that union types with the same set of component types are equal (that is, order of the components and duplicate components do not distinguish the types). Since there is a finite number of non-union types of depth one, there is a finite number of sets of non-union types of depth one. Therefore, there is a finite number of distinct union types whose component types have a depth of one.

If the depth is \(n + 1\) then by the induction hypothesis there is a finite number of distinct types (modulo renaming of free type variables) with depth at most \(n\). Notice that, in any given program, there is a finite number of trait and object definitions. Each of these traits and objects can be instantiated with only a finite number of distinct combinations of type arguments where each type argument is a type of depth at most \(n\). Therefore, there is a finite number of types (modulo renaming of free type variables) with depth at most \(n + 1\). \(\square\)
Lemma 19. If:

- there is a fixed maximum nesting depth for types, and
- \( p \) is a well-typed program, and
- \( \Delta \) is a type variable environment, and
- \( A \) and \( B \) are types, and
- table is a list of constraints

then:

\[ \text{T-SEARCH}_{p,\Delta}(A <: B, \text{table}) \text{ terminates.} \]

Proof. Notice that there are two ways this algorithm could loop infinitely. First, the while loop may
not terminate. Second, a call to T-SIMPLIFY may not terminate.

In the first case, notice that the first five cases of the conditional in the body of the while loop
cause the while loop to terminate. The remaining two cases increase the size of an existing path. By
Lemma 18, there is a finite number of types (modulo renaming of free type variables) satisfying the
given nesting depth limit. Therefore, there is a finite number of non-repeating sequences of types
(i.e., potential paths). In addition, there is a finite number of sequences of types in which a type
repeats at most once. Since the last two cases of the while loop disallow paths that repeat a type
more than once, the queue cannot grow infinitely. Therefore, the while loop must terminate.

In the second case, notice that a call to T-SIMPLIFY either terminates, makes a recursive call to
T-SIMPLIFY, or makes a call to T-SEARCH. Lemma 20 guarantees that a recursive call to T-SIMPLIFY
will terminate. Consider the situation in which a call to T-SIMPLIFY results in a call to T-SEARCH.
For the purposes of this proof, we consider this a recursive call to T-SEARCH. By Lemma 18, there is
a finite number of types (modulo renaming of free type variables) satisfying the given nesting depth
limit. Because constraints are simply pairs of types, there is a finite number of constraints (modulo
renaming of free type variables) satisfying the given nesting depth limit. Therefore, by recording
the input constraint and returning an error when cycles are detected, T-SEARCH must terminate. □

Lemma 20. If:

- there is a fixed maximum nesting depth for types, and
- \( p \) is a well-typed program, and
- \( \Delta \) is a type variable environment, and
- constraints is a list of constraints, and
- table is a list of constraints

then:

\[ \text{T-SIMPLIFY}_{p,\Delta}(\text{constraints}, \text{table}) \text{ terminates.} \]

Proof. The only procedure calls that may loop infinitely are T-SEARCH and the recursive call to
T-SIMPLIFY. By Lemma 19, the call to T-SEARCH must terminate.

Suppose, for contradiction, that the recursive call to T-SIMPLIFY does not terminate. By Lemma 18,
there is a finite number of types (modulo renaming of free type variables) satisfying the given nest-
ing depth limit. Because constraints are simply pairs of types, there is a finite number of constraints
(modulo renaming of free type variables) satisfying the given nesting depth limit. Therefore, if
the recursive call to T-SIMPLIFY does not terminate then the call to T-SEARCH must eventually be
passed a duplicate constraint. However, notice that T-SEARCH records each input constraint and returns an error when a duplicate is found. If an error is returned by T-SEARCH then the recursive call to T-SIMPLIFY must terminate, which contradicts our assumption.

Lemma 21. If:

there is a fixed maximum nesting depth for types, and
p is a well-typed program, and
\( \Delta \) is a type variable environment, and
constraints is a list of constraints

then:

T-CHECK-WITNESSES\(_{p,\Delta}(\text{constraints})\) terminates.

Proof. Follows from applying Lemma 19 to each constraint.

Theorem 6 (Termination of Constraint Solving). If:

there is a fixed maximum nesting depth for types, and
p is a well-typed program, and
\( \Delta \) is a type variable environment, and
constraints is a list of constraints

then:

T-SOLVE\(_{p,\Delta}(\text{constraints})\) terminates.

Proof. The only procedure calls that may loop infinitely are T-SIMPLIFY and T-CHECK-WITNESSES. But, by Lemmas 20 and 21, both must terminate.
Bibliography


Vita

Joseph J. Hallett is from Wellesley, Massachusetts. He graduated from Wellesley High School in 1998. Afterward, he attended Colgate University. He graduated from Colgate University with High Honors in Computer Science in 2002. He received his doctoral degree from Boston University in 2008. His dissertation is titled “Hidden Type Variables and Conditional Extension for More Expressive Generic Programs”.

Joe’s interests in computer science, and programming languages in particular, are in the areas of programming language design, formal semantics, type systems, and large-scale program design. As a member of the Boston University Church Project, Joe has developed formal semantics for ambiguous programming language features, type systems to verify advanced program invariants, and techniques for efficient programming language implementation. Joe has also been involved in the development of the Fortress programming language, a new language developed at Sun Microsystems for high-performance computing. As a member of the Fortress programming language team, Joe has worked on generic type systems, overload resolution, and coercion.

Joe has published several peer-reviewed articles and presented his work at various programming-language conferences and workshops. In addition to his theoretical study, Joe has worked on several software development teams in both industry and academia. He has also taught programming courses at both Colgate University and Boston University.