Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Type Constraints and Deny Capabilities

Author:
Michael Gillan

Supervisor:
Prof. Sophia Drossopoulou

Second Marker:
Prof. Nobuko Yoshida

May 17, 2021

Abstract

Generics are a vital and powerful part of nearly all modern programming languages, which allow code reuse across various types. However, with great power comes great responsibility; generics, and how they interact with the rest of a language, are highly complex, and in some cases require immense programmer effort to understand and apply.

Generic variables are usually bounded using subtyping; the range of types a variable is permitted to be is defined by the subtype of its bound. When types are referred to in their bounds, known as f-bounded polymorphism, the situation gets more complicated still.

An alternative to subtype bounding has been proposed in Genus, a type system developed on top of Java that uses type predicates known as constraints to bound type variables. Genus is highly expressive and powerful, but lacks a proof of soundness.

We present Genus-, a distilled, imperative formal model of generics that uses the same generic mechanisms as Genus. We discuss the well-formedness of the system, and show that it is sound.

We also describe DeGen, a type system we designed to incorporate the deny capabilities and actor-based execution from Pony into Genus-. We define its well-formedness, and discuss proving preservation of soundness and data-race freedom.

Acknowledgements

I would like to thank my supervisor, Sophia Drossopoulou, for her incredible patience and guidance over the course of this project. Her feedback and support has been vital for getting this project off the ground.

I'd also like to thank my family, for not getting mad when I didn't return their calls, and my flatmates for putting up with the replica North Atlantic garbage patch slowly migrating out of my room.

And finally, a shout-out to Michel Jouvet, without whose works on analeptics, this project would not and could not possibly exist. RIP to a real one.

Contents

1. Introduction

1.1 Motivations

There is inherent complexity in writing concurrent applications; care must be taken to prevent concurrent operations from interfering with each others' data as it could potentially create data races. The widely used solution this problem is explicit concurrency management through concurrency primitives. These require programmers to be able to reason about how threads will interact with each other, and to be constantly mitigating against concurrency issues. Despite this additional burden on programmers, they are exposed to risks such as deadlocking.

A solution to relieve the burden of concurrency on programmers is actor-based languages with capability systems; augmented type systems that can statically check that programs are data-race free. One such language is Pony, which uses a system called *deny capabilities* to determines what can and cannot be done with every reference in a program.

Like many other object-oriented programming languages, Pony uses a system of generics based on bounding type variables by subtyping. This is powerful and expressive, but can lead to unsoundness and programmer burden, especially when types appear in their own type bounds. This is referred to as 'f-bounded polymorphism'. Recently, alternative systems of generics have been developed, which constrain generic type variables using mechanics other than subtyping.

One alternative to f-bounded polymorphism is to use type predicates to ensure the interfaces that types present to generic code have an expected structure, without creating a subtype relation between the type and the interface definition. This is the approach taken by Genus [1], a highly expressive system of generics built on top of Java that uses type predicates (called *constraints*) to bound type variables.

We give a minimal formal specification of the ideas presented in Genus, and prove that it is sound. On top of this model, we introduce deny capabilities, resulting in a minimal, sound proof of concept that marries the major ideas from both systems.

1.2 Contributions

This project presents Genus-, a formal specification for the mechanism of genericity presented in Genus. We describe an augmented type system incorporating deny capabilities, DeGen, so as to provide an alternative to generics as they currently exist in Pony.

- We developed Genus-, a distilled, minimal formal model for the mechanism of genericity presented in Genus. This is an imperative model, which is more faithful to real programming languages and allows us to uncover more possibilities for unsoundness. It is described in Chapter 3
- We give a proof of soundness for Genus- in Section 3.8. Genus does not have a proof of soundness, so this is a novel contribution that establishes Genus- as a valid basis for designing a more complex type system on top of.
- We designed DeGen, an augmented version of Genus- that includes deny capabilities. We describe the design process in 4
- We also give a formal description of DeGen in Chapter 4, and discuss some thoughts about how we would show soundness and preservation of well-formed visibility. Our formal description includes a parametric formulation of viewpoints, based on previously established requirements.

2. Background

2.1 Types and Type Systems

Type systems are lightweight formal methods that allow a programmer to reason that a program will behave as intended with respect to a specification of the language it is created in. In his book *Types and Programming Languages* [2], Pierce defines a type system:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

From this definition, we can infer that types are the "kinds of values" that can result from computations within a program.

Traditionally, the concept of a *record type* is used to define composite types made from more than one primitive type; a record, which can be thought of as an *object* with its components representing the object's methods and fields, is an unordered set of labeled values, and a record type is specified by associating a type with each of its labels [3]. In addition to composite and primitive types, we also have function types, which express the types of the parameters and the return type of a function.

Formally, a record consists of labels $x_1, ..., x_j$ with associated types $\sigma_1, ...\sigma_j$, and is represented as the set $\{x_1 : \sigma_1, ..., x_j : \sigma_j\}$ [4].

2.1.1 Structural Type Systems

A structural type system is one in which type equivalence and compatibility is determined entirely by the type's structure; two types are compatible if for each feature in the first type, a corresponding and equivalent feature exists in the second type.

A language that employees this sort of type system is Go, when checking against interfaces. In the example below, `fmt.Println()` expects the parameter to implement the `fmt.Stringer` interface, which requires a method `String() string`. Since `User` implements this, it is accepted by `fmt.Println()`

```
type User struct {
  name string
}

func (user User) String() string {
  return fmt.Sprintf("User: name = %s", user.name)
}

func main() {
  user := User{name: "foo"}
  fmt.Println(user)         // User: name = foo
}
```

2.1.2 Nominal Type Systems

A nominal type system is one in which type equivalence and compatibility is determined entirely by type declarations and names; two structurally identical types are not considered the same if they are declared with different names.

The Java example given below shows that, although `Customer` and `Employee` are structurally identical, they cannot be assigned to each other.

```
class Customer {
  String name;
}

class Employee {
  String name;
}

class Main {
  public static void main(String[] args) {
    Customer c = new Customer();
    Employee e = new Employee();
    c = e; // incompatible types
  }
}
```

2.1.3 Union and Intersection Types

Union and intersection types are forms of type expressions that allow the combination of types into more complex types, without the need for a definition.

Union Types

A union type describes values that belong to either of two types. A good example of this is `char` in C. A signed char generally has a range of -127 to 128, and an unsigned char has a range of 0 to 255. The union of a signed char and unsigned char could accept values in the range -127 to 255. Generally, valid operations on union types have to be valid of both of its component types.

Intersection Types

An intersection type describes values that belong to both of two types. Using the char example above, the intersection type of a signed and unsigned char would accept values in the range 0 to 128. This intersection type could then be used in any operation expecting either of its component types, because it is compatible with both.

2.2 Subtyping

A type S is a structural subtype of T if S's interface (its public fields and methods) is a superset of T's. Intuitively, any operation that we perform on T through its interface can also be performed on S. Formally, we can use the concept of a record type, which describes the interface of an object as a mapping of labels to types. The record subtyping axiom is a formalization of the intuition above:

$$\frac{\sigma_1 \subseteq \rho_1, ..., \sigma_k \subseteq \rho_k}{\{x_1 : \sigma_1, ..., x_k : \sigma_k, ..., x_l : \sigma_l\} \subseteq \{x_1 : \rho_1, ..., x_k : \rho_k\}}$$

Nominal subtyping is an explicitly defined subtype relation between two types. The below example shows nominal subtyping in Java; the relationship between 'Bob' and 'Named' is declared, and says that 'Bob' implements 'Named'.

```
interface Named {
    public String name();
}

class Bob implements Named {
    public String name() {
        return "Bob";
    }
}
```

With nominal subtyping, it is the compiler's responsibility to check that the subtype implements everything required by the subtyping relation.

2.3 Covariance and Contravariance

The concept of variance refers to how the subtyping between complex type expressions relates to the subtyping of the components of said expressions; the type constructor of complex types may preserve, reverse, or ignore the subtype relation between its component types.

A covariant type constructor is one that preserves the relationship between subtypes; if `foo` is a subtype of `bar`, and the type constructor of a `List` is covariant, then a `List<foo>` is a subtype of `List<bar>`.

A contravariant type constructor is one that reverses the relationship between subtypes; given the same subtype relationship between `foo` and `bar`, and a contravariant type constructor for a `Printer`, we have that `Printer<bar>` is a subtype of `Printer<foo>`.

Additionally, there is the concept of invariance, where the relationship between simple types is ignored in complex type constructors, and two different instances of a type constructor are neither a subtype or a supertype of one another.

When typing a function, the return type may be covariant when the function is overridden, and parameter types may be contravariant. Generally, wherever the base method's function may be called, the overridden function must be permitted. Consider the psuedocode example below.

```
class Foo extends Bar {}
class Foo1 extends Bar1 {}

class Thing {
  Bar doThing(Foo1 x) {
    // implementation
  }
}

class OtherThing extends Thing {
  Foo doThing(Bar1 x) {
    // implementation
  }
}
```

First, it is clear that `OtherThing` can make `doThing` return a `Foo` as `Foo` is a subtype of `Bar`, and intuitively anything expecting a `Bar` can use a `Foo` instead. Secondly, `otherThing` can make `doThing` take a `Bar1` as a parameter since anything calling `Thing.doThing` must pass a `Foo1`, which is guaranteed to be a subtype of `Bar1`, and can be treated that as a `Bar1`.

The above leads to a standard rule for function subtyping (using the arrow for function typing):

$$\frac{\sigma' \subseteq \sigma \qquad \tau \subseteq \tau'}{\sigma \to \tau \subseteq \sigma' \to \tau'}$$

Some languages, such as Java, limit parameter types to be invariant, while others including C# allow parameter types to be contravariant.

2.4 Polymorphism

Polymorphism is defined as providing a single interface to entities of different types [5].

2.4.1 Ad hoc Polymorphism

Ad hoc polymorphism, commonly referred to as *function overloading*, is the mechanism by which a function appears to work on multiple different types, even if these types do not share a similar structure [3].

The common example for this is the `+` operator; when applied to integers, we have operations of the form `1 + 2 == 3`, where `+` is a summation function; when applied to strings, we can use it to perform concatenation, such as `"foo" + "bar" == "foobar"`. It is therefore clear that although this operator presents the same interface for many different types, it actually represents a number of distinct, heterogeneous functions.

2.4.2 Subtype Polymorphism

Since we know that subtypes present the same interface as their base type, polymorphism through subtyping allows a subtype to be used anywhere a supertype is expected. This is clear in the below example:

```
class A {}
class B {}

class Pair {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
      this.fst = fst;
      this.snd = snd;
  }

  Object getFst() {
      return this.fst;
  }
}

class Main {
  public static void main(String[ ] args) {
    Pair pairA = new Pair(new A(), new A());
    Pair pairB = new Pair(new B(), new B());
  }
}
```

Both `A` and `B` can be stored using the same `Pair` class, as they are both subtypes of `Object`. However, all type information is lost once the reference is stored. The following execution produces an error, as `getFst()` will only return an `Object`.

```
class Main {
  public static void main(String[] args) {
    Pair pairA = new Pair(new A(), new A());

    A a = pairA.getFst(); // incompatible types
  }
}
```

2.4.3 Parametric Polymorphism

Parametric polymorphism allows a single implementation of a function to work uniformly on many different types, which usually have some common structure. They are said to be *generic* and accept type parameters as placeholders for types:

```
class Pair<T> {
  T fst;
  T snd;

  Pair(T fst, T snd) {
      this.fst = fst;
      this.snd = snd;
  }

  T getFst() {
      return this.fst;
  }
}

class Main {
  public static void main(String[] args) {
    Pair<A> pairA = new Pair(new A(), new A());

    A a = pairA.getFst(); // successful
  }
}
```

The above leads to a successful execution, as the type information about `fst` is preserved through the type parameter. The parameter `T` is unbounded, and ranges over all possible types.

Bounded quantification allows us to specify type parameters that range over a subset of all types. Generally, this set is given by all subtypes of a given type.

```
interface Stringable {
  public String string();
}

class A implements Stringable {
  public String string() {
    return "A";
  }
}

class Pair<T extends Stringable> implements Stringable {
  T fst;
  T snd;

  Pair(T fst, T snd) {
    this.fst = fst;
    this.snd = snd;
  }

  public String string() {
    return this.fst.string() + ", " + this.snd.string();
  }
}

class Main {
  public static void main(String[] args) {
    Pair<A> pairA = new Pair(new A(), new A());

    System.out.println(pairA.string()); // "A, A"
  }
}
```

Here `T` can be any subtype of `Stringable`.

2.4.4 F-bounded Polymorphism

Recursively bounded quantification, also known as f-bounded quantification, allows type parameters to appear in their own bounds.

```
interface Equatable<T extends Equatable<T>> {
  public boolean eq(T that);
}
```

In the above, the type parameter `T` is required to be a subtype of `Equatable[T]`. This means, if `String` were to implement `Equatable[String]`, it would require a method `boolean eq(String that)`. This means that `Strings` are comparable to other `Strings` but not to, say, `Integers`.

2.4.5 Issues with F-Bounded Polymorphism

F-bounded polymorphism introduces type-checking complexities. It requires that inheritance to be recursive; `String` implements `Comparable<String>`, meaning it is defined in terms of its inherited type. When recursive inheritance is mixed with variance in type parameters, polymorphic subtyping becomes undecidable [6]. However, this does not apply in Pony, due to type invariance relative to type parameters.

F-bounded polymorphism can lead to significant code complexity. Below is an example of two classes taken from the *FindBugs* [7] open-source project.

```
class AbstractVertex<
    EdgeType extends AbstractEdge<EdgeType, ActualVertexType>,
    ActualVertexType extends AbstractVertex<EdgeType, ActualVertexType>> {...}
class AbstractEdge <
    ActualEdgeType extends AbstractEdge<ActualEdgeType, VertexType>,
    VertexType extends AbstractVertex<ActualEdgeType, VertexType>> {...}
```

The mutual dependency between `AbstractVertex` and `AbstractEdge` leads both classes to have very complex signatures.

2.5 Genus, Models and Constraints

An alternative generics mechanism to subtyping is presented in Genus [1], a system where named constructs, called models, explicitly witness constraints.

2.5.1 Constraints

In Genus, types are constrained using explicit *type constraints* that exist as a predicate on types. To require that a type has, for example, an `equals(...)` method, the below constraint can be used:

```
  constraint Eq[T] {
    boolean equals(T that);
}
```

This constraint can then be applied to type parameters to establish the ability to test equality on them:

```
  interface Set[T where Eq[T]] { ... }
```

Thus, an instantiation of `Set` requires a model, supplied either implicitly by the compiler or chosen explicitly by the programmer, to witness that `Eq[T]` is satisfied by whatever the type argument `T` is.

Prerequisite Constraints

Constraints may have other constraints as their prerequisites, and the satisfaction of a constraint entails the satisfaction of its prerequisite constraints. For example, `Eq[T]` is prerequisite of `Comparable[T]`, and the relationship can be expressed as:

```
  constraint Comparable[T] extends Eq[T] {
    int compareTo(T other);
  }
```

2.5.2 Models

Models are witnesses of constraints that are provided with the type argument.

Natural Models

If a type structurally conforms to a constraint, i.e. already has the required operations, then it forms a natural model for the constraint. For example, `String` has a built-in `equals(...)` method, and so `Set[String]` can use this method for establishing equality.

Explicit Models

The programmer can create models to satisfy a constraint if a natural model does not exist or does not have the desired behaviour. For example, case-insensitive equality can be defined as a model for `Eq[String]`:

```
model CIEq for Eq[String] {
  bool equals(String str) {
    return equalsIgnoreCase(str);
  }
}
```

Models can be extended and reused through model inheritance; a model for `Comparable[String]` can reuse the `CIEq` model for `Eq[String]`:

```
model CICmp for Comparable[String] extends CIEq {
  int compareTo(String str) {
    return compareToIgnoreCase(str);
  }
}
```

Alternatively, `CICmp` could define its own `equals(...)` method.

The model chosen to witness a constraint on a type is contained as part of the type; this is illustrated below:

```
Set[String] s0;
Set[String with CIEq] s1;
s1 = s0; // illegal assignment: different types.
```

The model of equality chosen for each `Set` is a static part of the type, and so `s0` and `s1` represent different types.

2.6 Dependent Types

A dependent type is one defined based upon a value. For example, a "pair of integers" is a type, and a "pair of integers where the first value is greater than the second" is a dependent type. Pony does not have dependent types, but an example, adapted from [8], in Agda, a language that does, is below:

```
Vec : Set -> Nat -> Set
Vec A zero = Unit
Vec A (succ n) = A X Vec A n
```

The signature of a `zip` function, which requires both `Vec` to be of the same length, can be defined as:

```
zip : {A B : Set} -> (n : Nat) ->
    Vec A n -> Vec B n -> Vec (A X B) n
```

We can draw a parallel with polymorphic types: polymorphic types are parameterised by types, and dependent type are parameterised by values.

2.6.1 Path Dependent Types

Path dependent types are a form of dependent type which depends on the value of an abstract type parameter. In Scala, a nested type depends on a specific instance of the outer type, rather than the outer type itself.

```
trait Animal { a =>
  type Food
  def eats(food: a.Food): Unit = {}
}

trait Grass
trait Meat

trait Cow extends Animal with Meat {
  type Food = Grass
}

trait Lion extends Animal {
  type Food = Meat
}
```

In the definitions above, adapted from [9], `Animal` has an *abstract type member* `Food`. We can see that `eats` requires its parameter to be of type `a.Food`, which is a *path-dependent type*. We therefore define `Cow` to be an `Animal` that eats `Grass`, and `Lion` to be an `Animal` that eats `Meat`.

```
val leo = new Lion {}
val milka = new Cow {}
val daisy = new Cow {}

leo.eats(milka)     //ok
milka.eats(daisy)   //error
```

Whilst `leo` can eat `milka`, since `leo.eats()` expects type `Meat` which `milka` is, `milka` cannot eat `daisy` as she isn't `Grass`.

2.7 Soundness

A sound type system is one in which well-typed programs cannot lead to type errors [10]. We can prove the type soundness of a system using *subject reduction* and *progress* [11].

Subject Reduction

A type system with subject reduction says that the evaluation of expressions does not cause their type to change. This can be expressed formally with the type rule:

$$\frac{\Gamma \vdash e_1 : \tau \qquad e_1 \rightarrow e_2}{\Gamma \vdash e_2 : \tau}$$

Progress

Progress indicates that the evaluation of a program cannot become *stuck*. This means that each program can either be evaluated infinitely, or will reach a state in which it cannot be evaluated further, that is, a value or a type error; the program never gets into an undefined state where no further transitions are possible.

Type Soundness

By combining the above mechanisms, if we can show that a well-typed program [1] reduces to only other well-typed programs, and will never become stuck, we have shown that the system is type sound.

2.7.1 Proving Type Soundness in the Presence of Generics

To prove that a generic type system is sound given a proof of soundness for a non-generic version for the same system, it is sufficient to show that any valid type instantiation on a generic declaration yields a valid concrete declaration [12] .

2.8 Actor Based Models

Actor based languages [13] provide a simpler model of concurrency than traditional object-oriented languages; instead of relying on mutual exclusion primitives to control threads' access to shared memory, *actors* have their own private state which only they can modify, and rely on message passing to interact with other actors. Actors process messages sequentially, but can execute concurrently. This message passing system eliminates the need for concurrency primitives such as locks and semaphores, and grants data race freedom.

2.8.1 Capabilities

In languages where the programmer can use pointers to reference objects in the heap, a number of potential problems can occur; the aliasing and sharing of mutable state can lead to unintended behaviour.

A solution to this problem is *capabilities*. In the system proposed by [14], capabilities consist of a pointer to an object, and a set of access rights that detail how the pointer can be used. [14] introduces the concepts of *base rights* and *exclusive rights*; base rights say a reference can perform an operation, such as writing, on an object, and exclusive rights say that no other reference can perform that operation on the object[2]. This leads to the concept of *compatibility* – two capabilities on the same object are compatible if their access rights permit the other to exist. Corresponding base and exclusive rights are incompatible as asserting one denies other references of the other.

Capabilities in Actor Based Models

We can use capabilities in actor based models to ensure data-race free programs. By ensuring that all references to an object have compatible capabilities, which can be ensured by the type system statically, we know that no two actors can access the same memory in a way that causes a data-race.

Capabilities in actor models give actors the ability to read, mutate, and send a reference safely. Only one actor may possess a mutable reference to an object at a time, and so to send a mutable reference, the sending actor must give up its reference to the object [15].

Isolation

An isolated reference is one where all paths to non-immutable objects reachable from the reference pass through it [12]. These isolated references form static regions where mutable objects reachable from the reference can only be reached through the reference, and immutable objects must be either only reachable through the reference of globally immutable [16].

We can *recover* mutable types back to an isolated state if we can ensure it is an isolated reference. The resulting alias makes stronger local guarantees than the previous mutable alias.

[1]We consider the entire program to be an expression for the definitions above.

[2]This does not necessarily mean the reference can perform the operation, it would still require the base right to do so.

2.8.2 Deny Capabilities

While earlier capability systems [12,14,15] detail what operations are allowed on an object, an other approach is to use capabilities to detail what aliases are *denied* by the existence of a reference [16]. This system of deny capabilities is what we will use to frame the rest of our exploration.

Aliases

Local aliases are other references in the same actor to an object that the actor already has a reference to. **Global aliases** are references to the object that are held by other actors. Deny capabilities describe what a local and global alias is forbidden to do.

A reference is mutable if it denies global read and write aliases, as no other actor can concurrently read or write to the object. Immutability is given by denying global write aliases, so a data race where an actor is reading an object as another is writing to it cannot occur. Denying no glabal aliases makes a reference *opaque*, meaning it can be neither read nor written to.

A reference can be safely sent to another actor if its global and local deny properties are the same, i.e. it does not matter which actor is holding the reference.

A full description of the deny capabilities is given in Table 2.1.

	Deny global read/ write aliases	Deny global write aliases	Allow all global aliases
Deny local read/ write aliases	*Isolated (iso)*		
Deny local write aliases	Transition (`trn`)	*Value (val)*	
Allow all local aliases	Reference (`ref`)	Box (`box`)	*Tag (tag)*
	(Mutable)	(Immutable)	(Opaque)

Table 2.1: Capability matrix, reproduced from [16]

2.8.3 View Point Adaptation

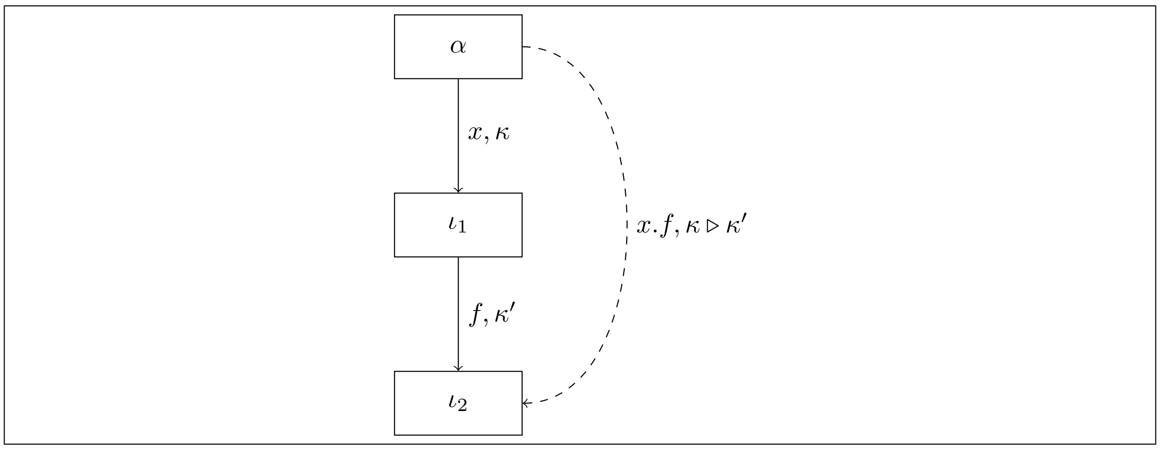
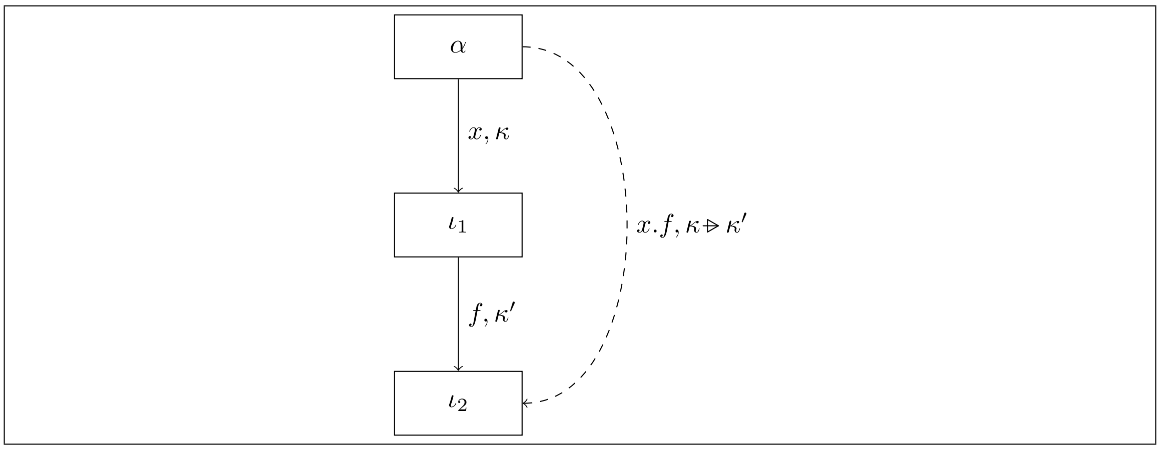


Figure 2.1: Viewpoint Adaptation

When reading a field from an object, as seen in Figure 2.1, we create a temporary alias whose capability depends on the capabilities of the object and its field. We call this *viewpoint adaptation*, denoted by $\triangleright$ and it can intuitively be thought of as "if a_1 sees a_2 as κ and a_2 sees a_3 as κ', then a_1 sees a_3 as $\kappa \triangleright \kappa'$ [17]. This alias is *temporary*, and does not outlive the execution of the expression which created it, such as field read [16].


α
x, κ
ι_1
f, κ'
ι_2
$x.f, \kappa \triangleright\!\!\!\!\!\!\triangleright \kappa'$
Figure 2.2: Extracting Adaptation

2.8.4 Extracting Adaptation

The extracting adaptation introduced by Steed [18] considers the case in Figure 2.2, where an actor α has an object x, which in turn has a field f. If f is overwritten with a new value, the return value of the operation is a temporary reference to the old value of the field, whose reference capability is given by $\kappa \triangleright\!\!\!\!\!\!\triangleright \kappa'$.

2.8.5 Aliasing and Unaliasing

Temporary References

Here, we distinguish between stable references, such as object fields and local variables, and temporary references, such as those created by the adaptations described above.

Considering the capabilities of temporary aliases, we are presented with two additional scenarios which affect unique aliases:

- zero stable references exist to an object anywhere in the program. This is similar to the isolation described in Section 2.8.1, but its sole reference is temporary (it has one reference removed). This is represented by an additional deny capability `iso`$^-$.
- zero stable *mutable* references exist to an object anywhere in the program. This is similar to the transition deny capability `trn`, but with an alias removed, and we denote it using `trn`$^-$.

We refer to `iso`$^-$ and `trn`$^-$ as ephemeral capabilities.

Aliasing

Aliasing occurs when a new stable reference is created from an existing one. The capability of the new alias depends on the capability of the reference it was created from, and cannot deny more than the original capability.

Any capability that is locally compatible with itself aliases as itself. An isolated reference doesn't allow mutable nor immutable references, and so aliases as an opaque reference. The deny capability `trn` only allows mutable local aliases, and must alias as a capability that allows a local mutable alias (the initial `trn`). Thus `trn` aliases as `box`.

The ephemeral capabilities described above are similar to their stable counterparts, but with one reference removed. Aliasing them simply adds that reference, and thus they alias as their stable counterparts.

Destructive Reads and Unaliasing

An operation which discards an alias, such as the extracting write described in Section 2.8.4, removes a stable reference to that object and returns an ephemeral reference to the object.

2.9 Pony

Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language that employees the strategy of deny capabilities described in the above sections to ensure static data-race freedom [19].

2.9.1 Pony Types

Types in Pony differ from those in other languages as, in addition to a set of values, they contain a reference capability.

2.9.2 Polymorphism in Pony

Pony supports all the forms of polymorphism detailed in Section 2.4, but with some restrictions introduced by reference capabilities.

Capability Subtyping

When thinking about substitutability of types, we must think about how capabilities can be related to one another.

By considering some of the deny capabilities, we can see how they can be substituted for one another; `iso` is read and write unique, and `trn` only guarantees write uniqueness, so it is clear that we can substituted an `iso` for a `trn`.

The deny capability type system [16] allows us to build the whole subcapability relation, shown in Figure 2.3.

$$\begin{tikzcd}
tag & & & \\
& box \arrow[ul] & & \\
& trn \arrow[u] & ref \arrow[ul] & val \arrow[ull] \\
iso \arrow[uuu] & trn^{-} \arrow[u] \arrow[ur] \arrow[urr] & & \\
iso^{-} \arrow[u] \arrow[ur] & & &
\end{tikzcd}$$

Figure 2.3: Subcapability relation

Capability Constraints

When instantiating a type variable, Pony requires that the capability of the parameter matches the capability of the bound exactly, and is not a subcapability. If a capability is not supplied with the type bound, the method or class must work for all possible capabilities.

Instead of a specific capability, Pony allows you to specify *capability constraints* in type bounds, which allows any of a set of concrete capabilities. The full list of capability constraints is detailed in Table 2.2.

Constraint	Allowed Capabilities	Description
`#any`	`iso, trn, ref, val, box, tag`	Default constraint
`#read`	`ref, val, box`	Anything that can be read from
`#share`	`val, tag`	Anything that can be sent to more than one actor
`#send`	`iso, val, tag`	Anything that can be sent to an actor
`#alias`	`ref, val, box, tag`	Set of capabilities that alias as themselves (used by compiler)

Table 2.2: Capability Constraints, reproduced from [20]

Variance in Pony Generic Parameters

In Pony, type parameters are invariant: a `List[Any box]` is neither a subtype nor a supertype of a `List[String box]`. This is due to the fact that a type parameter may be used both covariantly (as a return type) and contravariantly (as a parameter type) in the same method [11].

2.9.3 Pony Formal Models

There have been several formal models for Pony proposed since its creation. The first, introduced in [16] showed that the Pony type system guarantees freedom from data-races but lacked a number of important language features. $Pony^{GS}$ [18] then formalised a larger subset of Pony, simplifying and enhancing it. However, it still missed a number of features required for a complete formal model, such as recovery and generics. Most recently, $Pony^{PL}$ [21] introduced a formal model incorporating generics, involving rules for translation to a non-generic model $Pony^{0}$.

3. Genus-

Here we present Genus-, a minimal formal model that represents the ideas presented by Genus [1], as discussed in Section 2.5. Our major restriction is that we drop the concepts of models and interfaces entirely. While having models as named constructs adds a large amount of expressiveness to the system, we were able to simplify the system greatly by removing them. This change is explained and justified in Section 3.4. We discuss the core ideas of Genus- in Section 3.1, and discuss its differences with Genus. The full formal model of Genus- is provided in Appendix A

3.1 Introduction

3.1.1 Constraints

Genus- constraints exist as predicates on classes; they ensure that the type arguments to parametric classes and methods fulfill requirements imposed upon the type variables. A constraint provides a list of method signatures. Any class intended to witness that constraint must implement compatible methods.

Figure 3.1 shows a simple equality constraint `Eq`, which has a single method `equals(T other)`. Constraining a type variable using the `Eq[T]` constraint ensures that any argument for `T` can be tested for equality. Figure 3.2 shows a parametric method that makes use of this.[1]

Multivariate constraints allow a predicate to be established on the relationship between two classes. The multivariate constraint `Graphlike[E, V]` in Figure 3.3 must be witnessed by two classes, where the method signatures define the required relationship between the arguments. The argument for `E` should have two methods `source()` and `sink()` that return an object of the type provided as an argument for `V`. Which variable should implement which methods is indicated through the method receiver; `E` as the receiver for `source()` indicates it is the argument for `E` that must implement the method. In Figure 3.1, there is only one type variable, so the receiver can be safely omitted.[2] Classes that witness the constraints in Figures 3.1 and 3.3 are detailed in Section 3.1.2.

```
constraint Eq[T] {
    bool equals(T other);
}
```

Figure 3.1: Equality Constraint

```
bool notEquals[T where Eq[T]](T a, T b) {
    return !a.equals(b);
}
```

Figure 3.2: Parametric Method Using `Eq[T]`

Constraints can `extend` other constraints to create constraint entailment, where the satisfaction of one constraint is a prerequisite for the satisfaction of another. The constraint `Comparable[T]` in Figure 3.4 entails `Eq[T]`, meaning any class intended to witness `Comparable[T]` must also provide the `equals(T other)` method required by `Eq[T]`.

[1]This example, and others throughout this section, makes use of syntax that is not formally defined in Genus-. This syntax is intended to take its standard definition in other OOL such as Java.

[2]In the formalisation in Appendix A, we don't allow the receiver to be omitted.

```
constraint Graphlike[V, E] {
    Iterable[E] V.outgoingEdges();
    Iterble[E] V.incomingEdges();
    V E.source();
    V E.sink();
}
```

Figure 3.3: Multivariate Graphlike Constraint

```
constraint Comparable[T] extends Eq[T] {
    int compareTo(T other);
}
```

Figure 3.4: Comparable Constraint Entails Equality

Genus- simplifies Genus by removing interfaces as named constructs. To maintain expressiveness, we have had to augment the behaviour of constraints to act as interfaces. This is a natural extension, as interfaces act as constraints on classes already; a `Set` interface defines the methods that a class requires to be treated as a `Set`, and a class that implements `Set` must implement these methods. However, to accurately provide the behaviour of interfaces, we must allow that the type variables in constraints can themselves be constrained; a constraint that defines the behaviour of sets would be useless if you could not ensure that they type argument witnesses equality! This differs from the original Genus model. Figure 3.5 shows a comparison between a Java `Set` interface, and a Genus- `Set` constraint. Since constraints are predicates on types, a constraint that models a parametric interface must be multivariate, establishing a relationship between a class representing the set, and the type parameter to the set. We do not allow constrained type variables to be used as receivers, restricting them to only act as return or parameter types.

```
interface Set<T
    extends Comparable<T>> {
  ...
  void remove(T item);
  ...
}
```

(a) Basic Java set interface

```
constraint Set[S, T where Eq[T]] {
  ...
  void S.remove(T item);
  ...
}
```

(b) Equivalent constraint to the set interface

Figure 3.5: Set Interface vs Constraint

We show in Section 3.3 that any interface can be translated into a Genus- constraint, ensuring that no expressivity is lost by the removal of interfaces as named constructs.

3.1.2 Classes

Genus- classes serve largely the same purpose as classes in other languages in that they are an extensible template for creating objects. Our interest in classes in Genus- comes from their interaction with constraints. In the original Genus system, 'models' exist as named constructs that define behaviours for classes intended to witness constraints. Genus- removes 'models', and instead only permits classes to witness constraints structurally.

In order to witness a constraint, a class must provide implementations for the constraint's method signatures such that they are correct when the class is the argument for the type variable.

This is demonstrated in Figure 3.6, where `IntPair` structurally witnesses the `Eq[T]` constraint we defined in Figure 3.1. Supplying `IntPair` as the type argument to `Eq[T]` makes the required method signature `bool equals(IntPair other)`, which is correctly implemented in the class. Thus `IntPair` could be correctly used as the argument for a type parameter constrained by `Eq`, such as the `notEquals` method in Figure 3.2.

```
class IntPair {
    int fst;
    int snd;
    bool equals(IntPair other) {
        return this.fst == other.fst && this.snd == other.snd;
    }
}
```

Figure 3.6: `IntPair` Class Witnessing `Eq[T]`

To witness multivariate constraints, multiple classes are required. As described in the previous section, the constraint defines what methods must exist on which class. To demonstrate this, Figure 3.7 shows the `Graphlike[V, E]` constraint with two classes `Vertex` and `Edge` used as type arguments. We can see that the constraint creates a predicate on the relationship between `Vertex` and `Edge`, and show the class implementations that would fulfil this predicate.

```
constraint GraphLike[Vertex, Edge] {
  Iterable[Edge] Vertex.outgoingEdges();
  Iterable[Edge] Vertex.incomingEdges();
  Vertex Edge.source();
  Vertex Edge.sink();
}
```

```
class Vertex {
  List[Edge] _out;
  List[Edge] _in;
  List[Edge] outgoingEdges() {
    return _out;
  }
  List[Edge] incomingEdges() {
    return _in;
  }
}
```

(a) `Vertex` class

```
class Edge {
  Vertex _source;
  Vertex _sink;
  Vertex source() {
    return _source;
  }
  Vertex sink() {
    return _sink;
  }
}
```

(b) `Edge` class

Figure 3.7: Witnessing Multivariate Constraints

We have seen that constraints can be entailed, and witnessing classes must fulfill all constraints that are entailed by the target constraint. In order to witness the `Comparable` constraint from Figure 3.4, a class must also provide the methods to witness the `Eq` constraint that `Comparable` extends. Figure 3.8 shows how our `IntPair` class can be modified to witness `Comparable`.

Classes themselves can `extend` other classes, but this behaviour is orthogonal to that of extending constraints; extending a constraint simply allows the programmer to reduce code duplication where one constraint logically relies on another, whereas extending a class allows the programmer to specialise the behaviour of a class. This is similar to the difference between extending interfaces

```
class IntPair {
    int fst;
    int snd;
    bool equals(IntPair other) {
        return this.compareTo(other) == 0;
    }
    int compare(IntPair other) {
        return (this.fst + this.snd) - (other.fst + other.snd);
    }
}
```

Figure 3.8: Witnessing Entailed Constraints

and extending classes. A class has access to all the fields and methods of its superclass, and can choose to override existing methods in order to create specialised behaviour. We have omitted the `extends` clause from previous examples for brevity, but require it in our formal specification; we rely on a trivial class `Object` that is a superclass of all classes. Thus the declaration of `IntPair` should actually read '`class IntPair extends Object`'. We will continue to omit this in future examples.

In order to allow for the missing behaviour that would be provided by interfaces, we have allowed constraints to use constrained type variables. In Figure 3.9, we show two different approaches to witnessing a constraint like this; one for a concrete class, and one for a parametric class. In Figure 3.9a, `VarBox` does not instantiate the type variable T, leaving all occurrences of it from the constraint as they were. This is a valid way of witnessing the constraint; repeating the constraint on `T` in the scope of the class is sufficient to ensure the expected methods will be available at runtime. In Figure 3.9b, the variable `T` has been implicitly instantiated as `String`. Since `String` witnesses `Eq[T]`, this is also a valid method of witnessing the constraint. This is similar to Java, where the class must explicitly instantiate the type variable in the manner of '`StringBox implements Box<String>`'. Also note that both classes provide implementations for the `equals` method required by constraint entailment.

3.1.3 Methods

Methods are declared in constraints, and implemented in classes. A method in a constraint has a return type, a receiver (which has to be one of the type variables in the constraint), a sequence of parameter types, and a (possibly empty) set of generic parameters. In a class, a method consists of the above and an expression for the method body.

The generic parameters supplied to a method are in scope during the execution of the method body and must be initialised at the method call site. An example of a generic method and valid calling code is given in Figure 3.10. This is a bit of a redundant example (the same effect could achieved by dispatching `equals` to `pair.fst`), but serves as an example of how type variables can be used in generic methods.

Classes can override methods from a super class by redeclaring them in the class body. The return type and parameter types of the overriding method must be the same as in the super class.

3.1.4 Types

Types in Genus- may be either type variables or classes; if the class has generic parameters, then it must be supplied with type arguments for each of its parameters.

```
constraint Box[B, T where Eq[T]] extends Eq[B] {
    T B.get();
    void B.set(T obj);
}
```

```
class VarBox[T where Eq[T]] {
    T item;
    T get() {
        return this.item;
    }
    void set(T obj) {
        this.item = obj;
    }
    bool equals(VarBox[T] other) {
        return this.item
            .equals(other.item);
    }
}
```

(a) Parametric Class Witnessing Constraint

```
class StringBox {
    String item;
    String get() {
        return this.item;
    }
    void set(String obj) {
        this.item = obj;
    }
    bool equals(StringBox other) {
        return this.item
            .equals(other.item);
    }
}
```

(b) Concrete Class Witnessing Constraint

Figure 3.9: Two Methods Of Witnessing Interface-Style constraints

Existential and Universal Types

Genus as presented by Zhang et al. extended Java's wildcard syntax, itself a limited form of existential quantification, to more explicit and expressive existential typing. The Java type `List<? extends Printable>` is equivalent to the type-theoretic quantified type $\exists$`U` $\leq$ `Printable.List[U]`, which (assuming `Printable` is a constraint) can be written in Genus as `[some U where Printable[U]]List[U]`.

Existential types in the Genus syntax have the form $\dot{\tau} ::= [\beta]\tau$, where a type is prefixed with some generic parameters, which introduce type variables that are in scope in the quantified type. This explicit syntax for introducing existentially quantified variables is very expressive, but also adds a great deal of complexity to the system; subtyping is extended to coersion, which introduces extra computation such as existential packing. It is due to this that in Genus- we remove existential types completely.

We justify this by noting that all existentially quantified types can be encoded as universally quantified types. Consider the known equivalence $(\exists X.P(X)) \implies Q \equiv \forall X.(P(X) \implies Q)$. A method `m` with signature `A m[X where C[X]](v:X)` has type $m : \forall X.(C(X) \implies A)$, and a method `n` with signature `A n(some X where C[X].v:X)` has type $n : (\exists X.C(X)) \implies A$. These two types are equivalent, and so an existential type for a parameter can be replaced by a universal type for the function. Similar arguments can be made for other scenarios, and knowing that existential types can be encoded as universal types is enough justification for their removal.

Variance

Variance in Java (and by extension Genus) occurs at use-site, meaning that variance is defined when the type parameter is used. We define that in Genus-, as in Java, type variables in generic classes (and constraints) are invariant. However, Java introduces covariance and contravariance by using bounded wildcards, allowing a wildcard to be bounded from above or below to give covariance and contravariance respectively. As discussed in the previous section, we have opted to remove existential types, which wildcards are a limited form of. As a result of this decision, there is mechanism that introduces variance into Genus-, and thus we do not need to consider it.

```
class PairChecker {
    bool checkPair[X where Eq[X]](Pair[X] pair) {
        X fst = pair.fst();
        X snd = pair.snd()
        return fst.equals(snd);
    }
}


...
p = new PairChecker();
pair = new Pair[String](fst="123", snd="abc");
p.checkPair[String](pair); // returns false
```

Figure 3.10: Generic Method and Calling Code

3.1.5 Null

A null reference is a concept in most modern programming languages, used to indicate that the reference does not refer to a valid object in the heap. Attempting to dereference a null pointer in most programming languages will lead to a null pointer exception, and the termination of execution. However, in Genus-, we do not have `null` as an expression. Since we require that all fields on objects be instantiated with valid values at creation, we have no need for it. As a result of this, Genus- will not be able to represent cyclic types; instantiating a cyclic object will be impossible.

3.2 A Mathematical Approach to Witnessing

In the previous section we presented a number of example constraints, with classes that witness those constraints. In this section, we present a concise formulation of the relationship between classes and constraints in order to justify our choices for the design of parametric constraints. Additionally, this formulation provides a basis for justifying the removal of models as named constructs.

3.2.1 Notation

In section 3.1.1 we describe constraints as predicates on classes. We can formalise this notion by imagining the set of all classes $\mathbb{L}$. We can say that all constraints are a subset of $\mathbb{L}$ where all elements of the constraint obey a certain feature. To simplify this, we use the function *Pred* defined in Figure 3.11.

$$C : \mathit{Pred}_1(\mathbb{L}) \equiv C \in \wp(\mathbb{L})$$

$$C : \mathit{Pred}_2(\mathbb{L} \times \mathbb{L}) \equiv C \in \wp(\mathbb{L} \times \mathbb{L})$$

$$\mathit{etc.}$$

Figure 3.11: Pred Function

3.2.2 Predicates

Alternative 1

We define the constraints we have introduced as functions on $\mathbb{L}$ in Figure 3.12. We can see that constraint entailment is equivalent to a subset relation between two predicates. In this method of defining predicates, parametric constraints are predicates on every type variable, where constraints on type variables manifest as necessary and sufficient conditions for a class to be in the predicate. These definitions correspond to the syntax for constraints we have seen thus far.

$$\begin{aligned}
\texttt{Eq} &: \mathit{Pred}_1(\mathbb{L}) \\
\texttt{Graphlike} &: \mathit{Pred}_2(\mathbb{L} \times \mathbb{L}) \\
\texttt{Comparable} &: \mathit{Pred}_1(\mathbb{L}) \wedge \texttt{Comparable} \subseteq \texttt{Eq} \\
\texttt{Set} &: \mathit{Pred}_2(\mathbb{L} \times \mathbb{L}) \texttt{ where } \forall\, C' \,.\, (C \notin \texttt{Eq} \iff \texttt{Set}(C, C') = \emptyset) \\
\texttt{Box} &: \mathit{Pred}_2(\mathbb{L} \times \mathbb{L}) \texttt{ where } \forall\, C' \,.\, (C \notin \texttt{Eq} \iff \texttt{Set}(C, C') = \emptyset) \wedge \texttt{Box} \subseteq \texttt{Eq}
\end{aligned}$$

Figure 3.12: Constraints as Predicates

Alternative 2

Our other option for modelling constraints as predicates is given in Figure 3.13. The parameterised constraint `Set` is a function that take a class and gives a set of classes; this set of classes is non-empty if the argument class is in `Eq`. Parameterised constraints require a different syntax to the one discussed in Section 3.1.1, an example of which is given in Figure 3.14. `T` acts as the parameter to the function in Figure 3.13, and the constraint acts as a predicate on `S`, which represents the set itself. Since `T` is not part of the predicate, we can omit the receiver from any functions, as it is implicitly `S`.

$$\begin{aligned}
\texttt{Eq} &: \mathit{Pred}(\mathbb{L}) \\
\texttt{Graphlike} &: \mathit{Pred}(\mathbb{L} \times \mathbb{L}) \\
\texttt{Comparable} &: \mathit{Pred}(\mathbb{L}) \wedge \texttt{Comparable} \subseteq \texttt{Eq} \\
\texttt{Set} &: \mathbb{L} \to \mathit{Pred}(\mathbb{L}) \texttt{ where } \forall\, C' \,.\, (C \notin \texttt{Eq} \iff \texttt{Set}(C) = \emptyset) \\
\texttt{Box} &: \mathbb{L} \to \mathit{Pred}(\mathbb{L}) \texttt{ where } \forall\, C' \,.\, (C \notin \texttt{Eq} \iff \texttt{Box}(C) = \emptyset) \wedge \texttt{Box}(C) \subseteq \texttt{Eq}
\end{aligned}$$

Figure 3.13: Parameterised Predicates

```
constraint Set<T where Eq[T]>[S] {
    ...
    void remove(T item);
    ...
}
```

Figure 3.14: Alternate Syntax For Parametric Constraints

Both syntaxes are capable of representing all possible constraints, and so either can be used. We have chosen the syntax in A.1 to stay closer to the original Genus, but acknowledge it requires somewhat more complex well-formedness rules to ensure type variables are used appropriately.

3.2.3 Classes

There are two forms of classes; classes such as `Vertex`, `Edge`, and `IntPair` that are simply members of $\mathbb{L}$, and parametric classes such as `VarBox` that are partial functions that have arguments and return types that are members of $\mathbb{L}$. The definition of `VarBox` as a function is given in Figure 3.15. Parametric classes such as `VarBox` already exist in Genus, and we have used the same square bracket notation to denote the type variables.

$$\texttt{VarBox} : \mathbb{L} \rightharpoonup \mathbb{L} \texttt{ where } (C \notin \texttt{Eq} \iff \texttt{VarBox}(C) = \bot)$$

Figure 3.15: `VarBox` As A Function On Classes

Classes that witness predicates are in the set defined by the predicate, thus $\texttt{IntPair} \in \texttt{Eq}$ and $(\texttt{Vertex}, \texttt{Edge}) \in \texttt{GraphLike}$. We can form this as a judgement, as shown in Figure 3.16. The program structure provides the knowledge of whether a class is in the subset of a constraint, for the most part. The final judgment shows that if the necessary constraints are active on a type variable, then we know enough about that type variable to decide whether it fulfils the predicate provided by constraints.

$$\begin{aligned}
Prog &\vdash \texttt{Eq}[\texttt{IntPair}] \\
Prog &\vdash \texttt{GraphLike}[\texttt{Vertex}, \texttt{Edge}] \\
Prog &\vdash \texttt{Box}[\texttt{VarBox}[\texttt{String}], \texttt{String}] \\
Prog &\vdash \texttt{Box}[\texttt{StringBox}, \texttt{String}] \\
Prog,\ \texttt{Eq}[L] &\vdash \texttt{Box}[\texttt{VarBox}[L], L]
\end{aligned}$$

Figure 3.16: Witness Relations As Judgements

3.3 Translating Interfaces

Genus- simplifies Genus by removing models and interfaces as named constructs. Section A.2 first gives an expanded syntax that adds interfaces to Genus- in Figure A.2. Figure A.4 then provides a translation from Genus- + interfaces to Genus-. The fact that we can describe this translation shows that Genus- without interfaces is at least as expressive as Genus- with interfaces.

Translation of interface declarations is relatively straightforward; interfaces can be modelled as constraints where we introduce a type variable to represent the interface itself. Figure 3.17 shows how we would do this translation for a `Set` interface. We first show a simple implementation of a `Set` interface in Java. The Genus- translation of this interface is shown on the right. Translation introduces a type variable `S` to represent the `Set`. The Genus- constraint creates a relationship between the two classes `S` and `T`, where equality can be established on objects of type `T`. We ensure that the methods required of a `Set` are present on `S`. A parametric class such as `HashSet[T where Eq[T]]` would be able to witness this constraint along with an appropriate `T`

```
interface Set<T where Eq[T]> {
  ...
  void remove(T item);
  ...
}
```

(a) A `Set` interface in Java

```
constraint Set[S, T where Eq[T]] {
  ...
  void S.remove(T item);
  ...
}
```

(b) Translated interface

Figure 3.17: How interfaces can be translated to Genus- constraints.

Further complexity comes from where interfaces can be used as types; method return types and method parameters. Both of these problems can be solved with universal quantification.

Figure 3.18 shows how we can translate parameters that are interfaces into universally quantified type variables that are in scope on the method. The method `addAll(...)` on the `Set` interface can take anything that fulfills the contract of `Set` as a parameter. This isn't necessarily the same as the implementing class, i.e. a `HashSet<T>` class could take `TreeSet<T>` as a parameter to `addAll(...)`. To translate the parameter, we introduce a new type variable `U` which, together with the already defined variable `T`, witnesses the `Set` constraint. `U` is then used in place of the interface in the parameter list. Implementations of `Set` don't necessarily have to instantiate `U`, but any calling code would have to provide a class that obeys the constraint.

Figure 3.19 demonstrates translation of return types by introducing type variables at the constraint level. Here, we introduce universally quantified type variables in the scope of the constraint. To translate the return type `List<T>`, we introduce a new type variable `L`, in the same manner as for methods. `L` represents a list, and we introduce a constraint on `L` and `T` that represents this relationship. `L` then replaces `List<T>` as the method return type.

The translation rules in Section A.4 detail the syntactic translation that occurs, and encapsulate the approach discussed above. To translate a program consisting of a sequence of class, constraint, and interface declarations, we apply the rule `T-Prog` to translate each declaration.

Interfaces are first converted to constraints in the manner described in Figure 3.17; a new type variable is introduced to convert the interface to a predicate, and this type variable is set as the receiver for each method. The interface identifier is prepended with the characters 'c_'.

Both constraints and classes then go through two further levels of translation; the rules `T-Constraint` and `T-Class` translate any interface return types in their method signatures by introducing type variables and constraints. Each method is then translated using the `T-Meth` rule to translate any interface parameters.

```
interface Set<T> {
  ...
  void remove(T item);
  ...
  void addAll(Set<T> set);
  ...
}
```

(a) Interface as a Parameter Type

```
constraint Set[S, T where Eq[T]] {
  ...
  void S.remove(T item);
  ...
  void S.addAll[U
    where Set[U, T]](U set);
  ...
}
```

(b) Translated parameters

Figure 3.18: Translation of interface parameters.

```
interface Set<T> {
  ...
  void remove(T item);
  ...
  List<T> toList();
  ...
}
```

(a) Interface as a Return Type

```
constraint Set[S, T, L
    where Eq[T], List[L, T]] {
  ...
  void S.remove(T item);
  ...
  L S.toList();
  ...
}
```

(b) Translated Return Type

Figure 3.19: How Genus- constraints can be used in place of interfaces.

Interface translation uses the *expand* function defined in Figure A.3. *expand* takes an interface type I and recurses through its type arguments to further convert any other interface types. It returns a sequence of type variables, the head of which is the type variable that represents I. For example, $expand(Set\langle List\langle \mathtt{X}\rangle\rangle, \emptyset) = (\mathtt{U}_1 : \mathtt{U}_2, c\_Set[\mathtt{U}_1, \mathtt{U}_2] : c\_List[\mathtt{U}_2, \mathtt{X}])$. This allows the translation to convert a single interface type to the full set of type variables and constraints required to represent it in one go.

3.4 Simplifying Models

In the original Genus paper, models exist as named constructs. The major benefit of this, instead of just relying on a type's natural model, is that models allow the programmer to make a type to satisfy a constraint when the natural model doesn't exist, or has undesired behaviour. By extension of this facility, models also give Genus code reuse through model inheritance, the ability to use expanders to invoke functionality directly, and the ability to specialise the behaviour of models through the creation of parameterized models.

```
model CIEq for Eq[String] {
  bool equals(String str) {
    return equalsIgnoreCase(str);
  }
}

Set[String] s0 = ...;
Set[String with CIEq] s1 = ...;
s1 = s0; // illegal assignment: different types.
```

Figure 3.20

Figure 3.20 shows an example of creating a model in Genus to enable a use of different behaviour for a type with an existing natural model. `String` already has a structural model of equality, but `CIEq` provides a specialised model of equality that ignores the case of the string. We can see from the calling code that when `String` is used as a type argument, we can specify which model of equality we want to use. The chosen model is part of the type, so two types that are equivalent in all but model are strictly different.

We can emulate this ability using wrapper classes. Figure 3.21 demonstrates this idea. The class `CIString` is a wrapper around a `String` with `eq` method that reflects the desired behaviour of case-insensitive equality. It delegates all other method calls to its inner `String` field, as the `append` method demonstrates.

```
class CIString {
    String str;
    ...
    bool equals(CIString other) {
        return str.equalsIgnoreCase(other);
    }
    ...
    CIString append(CIString other) {
        str.append(other.str);
        return this;
    }
    ...
}
```

Figure 3.21: Wrapper Function for Specialising Behaviour

Since `CIString` correctly witnesses `Eq[T]`, it can be used as an argument to `Set` and will have the intended behaviour, including case-insensitive equality. It can not be used in a place where `String` is explicitly needed, as it is obviously a different type, and would therefore cause the same assignment error as in Figure 3.20.

We can extrapolate this idea into a strategy for recreating the expressiveness provided by models; when we wish to specialise a type's behaviour relative to a given constraint, we create a wrapper class around that type, implementing the specialised behaviour in the methods required by the constraint, and delegating all others to the inner type. The concept of model inheritance doesn't have an equivalent in this strategy, and so we lose the benefit of code reuse through inheritance. This does not impact the expressiveness of our solution, simply the brevity, as all behaviour an inherited model would provide can be replicated.

Despite this strategy, we do not attempt to replace the functionality of expanders; to do so would require us to increase syntactic and type complexity, for relatively little gain in our generic expressiveness. Genus models allow types to be "different, but analogous", where the model used by a type does not affect the underlying structure of the type. This enables method calls such as `"x".(CIEq.equals)("X")`, as the types `String` and `String with CIEq` are generally interchangeable. In the example in Figure 3.20, `String` and `CIString` are completely distinct types, and so expanders would not work here. We do not view this as much of a deficiency; expanders in Genus are important as they all the programmer to invoke methods promised by named where-clause constraints. Genus- does not have named constraints, and so has little need for expanders. The ability to specialise behaviour at a given point in the program can be recreated, albeit with less succinct code, by instantiating a wrapper type. For example, `"x".(CIEq.equals)("X")` in Genus has behaviour equivalent to `(new CIString("x").)equals(new CIString("X"))` in Genus-.

```
model DualGraph[V,E] for GraphLike[V,E] where GraphLike[V,E] g {
  V E.source() { return this.(g.sink)(); }
  V E.sink() { return this.(g.source)(); }
  Iterable[E] V.incomingEdges() {
    return this.(g.outgoingEdges)(); }
  Iterable[E] V.outgoingEdges() {
    return this.(g.incomingEdges)(); }
}
```

Figure 3.22: Dual Graph Parametric Model

The parametric model in Figure 3.22 specialises the behaviour of another model that witnesses the same constraint as itself. Given a model for `GraphLike[V,E]`, this model will use the behaviour the constraint guarantees to return the transpose of any graph `g` by reversing the edge orientations. While we cannot return the dual as simply and succinctly as this, the calculation of a dual graph is a simple algorithm that requires iterating over the graph and building the dual. Genus's approach gives us the benefit of reduced complexity and space required, as it only requires one graph to be stored in memory. While this is a significant weakness on the part of Genus-, it is a weakness of complexity rather than a weakness of expressiveness.

3.5 Well-Formedness

Section A.3 presents a large number of rules that judge the well-formedness of a program. These rules judge if a sequence of the syntactic elements from Figure A.1 form a valid Genus- program.

3.5.1 Environments

Figure A.1 includes a number of environments that are used in the well-formedness rules.

First, the type environment Γ represents the type variables in the current scope of judgement. Type variables can come into scope at any point where generic parameters are instantiated. We could create a vacuous constraint that acts on all type variables and define the domain of Δ to be the type variables currently in scope. However, having Γ will become very useful when we introduce deny capabilities.

Second, the constraint environment Δ is a list of all the constraints that are active in the current judgment scope. It is primarily used to determine the methods that can be assumed to be present on a type variable for witness relations and method dispatch; it allows the type system to lookup what constraints are acting upon a type variable, and which methods in the constraint have that type variable as a receiver. Given a well-formed program and a sound type system (see Section 3.8), any argument provided for this type variable at runtime will have the expected methods.

Finally, we have the standard value environment E; this provides a mapping from local variables (including `this`) to types and is used to type expressions. As such, a new value environment is created for each method call, which we discuss in Section 3.6.

3.5.2 Relations

We next present rules that ensure the validity of certain relationships in the program: subtyping, constraint entailment, and witnessing.

Subtyping

Figure A.6 gives the rules for class subtyping. A program judges a class L_1 to be a subtype of another class L_2 if the declaration of L_1 in the program extends L_2, and any type variables that occur in both the parameter list for L_1 and the arguments passed to L_2 in the class declaration are correctly mapped. This relation is defined in rule `S-Subclass`. Figure 3.23 gives an example of how this rule is invoked, with `ArrayListSet[String]` being judged to be a subtype of `Set[List[String]]`.

$$\textsc{S-Subclass}\quad \frac{\mathit{Class}(\mathit{Prog}, \mathit{ArrayListSet}) = \texttt{class}\ \mathit{ArrayListSet}[\texttt{X}]\ \texttt{extends}\ \mathit{Set}[\mathit{List}[\texttt{X}]]\ \{...\}}{\mathit{Prog} \vdash \mathit{ArrayListSet}[\texttt{String}] \leq \mathit{Set}[\mathit{List}[\texttt{X}]]\{\texttt{String/X}\}}$$

Figure 3.23: Invocation of Subtyping Rule

Subtyping is a transitive and reflexive relation, and the rules `S-Refl` and `S-Trans` ensure this is built into the model. The rule `S-Object` also lets us say that every class is a subtype of a vacuous class *Object*, allowing us to require an `extends` clause in class declarations.

Constraint Entailment

The rules for constraint entailment, given in Figure A.7, work very similarly to subtyping but applied to constraints rather than classes. A constraint C_1 is judged by a program to entail a constraint C_2 if the declaration of C_1 in the program extends C_2, and any type variables that occur in both the parameter lists for C_1 and C_2 in the constraint declaration are correctly mapped. Again, this relation is both transitive and reflexive.

Witnessing

The witness relation judges whether a sequence of types witness a constraint applied to that sequence of types. The two rules defining the judgment are presented in Figure A.8. As discussed in Section 3.1.2, classes witness a constraint structurally by implementing the methods required by the constraint. Figure 3.3 presents the multivariate constraint `GraphLike`, and Figure 3.7 shows how replacing the type variables `V` and `E` with the classes `Vertex` and `Edge` show that the classes correctly implement compatible methods to that required by the constraint. The rule `W-Class` therefore judges the relation using lookup and rewrite rules.

First we lookup the methods that are required by the constraint by consulting its declaration in the program, and by recursing to look up methods in any entailed constraints. We then replace the type variables used in the constraint with the sequence of types that we are judging.

We then lookup the methods implemented in the classes, again applying the same rewrite if a class has instantiated type variables. The type witness relation then consists of checking that the methods we looked-up on the constraint are a subset of the union of the methods implemented by the types.

In addition to classes, the sequence of type could also contain type variables that are constrained in the scope. We therefore implement a lookup rule for type variables, where we assume they contain the methods in the active constraints for which they are the receiver, as discussed in Section 3.5.1. By formulating the witness relation as a series of look-ups, extensions such as this are relatively simple, and keep complexity out of the witness rules.

The second rule `W-Subsume` describes how constraint entailment affects witnessing; if a sequence of types are a witness for a constraint, they are also a witness for any entailed constraint.

3.5.3 Programs, Constraints, and Classes

Programs are judged to be well-formed based on the constraints and classes that compose them. The rule `W-Prog` in Figure A.9 says that if all the constraints and classes that form a program are judged to be well-formed, so is that program.

There are many aspects of judging a constraint to be well-formed. A constraint of the form ‘$\texttt{constraint}\ C_1[\overline{\texttt{X}_1}\ \texttt{where}\ \overline{C_2[\overline{\texttt{X}_2}]}]\ \texttt{extends}\ C_3[\overline{\texttt{X}_3}]\ \{\overline{\tau_1\ \tau_2.m[\beta](\overline{\tau_3})}\}$‘ requires several checks. Firstly, any type variables that appear in either the constraints $\overline{C_2}$ in the type parameter list or in the entailed constraint C_3 must be declared in the original sequence of variables $\overline{\texttt{X}_1}$. All the other constraints are then checked for well-formedness by invoking the rule `W-TC` in Figure A.13. This check is relatively straight forward, as the arguments passed to each constraint are also type variables, but it performs a number of important checks, as discussed in Section 3.5.5. We then check the method declarations $\overline{\tau_1\ \tau_2.m[\beta](\overline{\tau_3})}$. This is mostly done by invoking rule `W-Methsig` to check that each method signature is well-formed, with the constraints $\overline{C_2}$ and C_3 active, but also performs an additional check that the receiver of each method is a type variable in the list $\overline{\texttt{X}_1}$, and that it is unconstrained, a requirement discussed in Section 3.1.1.

A class of the form ‘$\texttt{class}\,L_1[\overline{\texttt{X}}\texttt{where}\overline{C[\overline{\tau_1}]}]\texttt{extends}\,L_2[\overline{\tau_2}]\{\overline{\texttt{fld}}\ \overline{\tau_3\ \tau_4.m[\beta](\overline{\tau_5})\{\texttt{e}\}}\}$‘ goes through the following checks: each constraint in $C[\overline{\tau_1}]$ is checked for well-formedness using the same rule `W-TC` that we used to check the constraints in the constraint declaration. The invoked rule is the same, but the check is more complex, as the sequence of types may not just contain type variables. We then invoke rule `W-TL` to check that $L[\overline{\tau_2}]$ is a well-formed type. We then defer to rule `W-MethDecl` to perform general checks on each method declaration, and perform to more specific checks; the receiver of each method is the same type as the class, and any method that overrides as method in a super class must have invariant parameter and return types.

3.5.4 Methods

The general checks for method well-formedness are relatively simple. A well-formed method declaration ‘$\tau_1\ \tau_2.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_3\ x})\ \{\texttt{e}\}$‘ has a well-formed return type τ_1, a well-formed receiver type τ_2, and well-formed parameter types $\overline{\tau_3}$. The constraints $\overline{C[\overline{\tau}]}$ are active when checking the parameter types, and themselves must be well-formed. The method body, given by the expression `e` must have a type compatible with τ_1. A method signature has all the same checks as above, without checking the expression.

3.5.5 Types

Types in Genus- can either by a type variable, or a class with a sequence of type parameters. Checking a type variable is well-formed is as simple as checking that it is currently in scope, which is done by checking if it is in Γ.

Checking an instantiated class with type arguments of the form $L[\overline{\tau}$ involves checking that each of the type arguments are well-formed, and that there are the same number of arguments in the type as there are in the class declaration. We then invoke the same rule to check the super class from the class declaration, with a rewrite applied that replaces each type variable in $\overline{\mathtt{X}}$ with its corresponding type argument in $\overline{\tau}$. Each constraint in the class declaration is checked as well, with the same rewrite applied. Finally, we check that each type argument helps witness the constraint it is part of. We do this by taking each constraint, applying the rewrite to it's type variable list, and then checking that the rewritten variables jointly witness the constraint.

Instantiated constraints are constraints that have had a list of type arguments passed to them. They go through largely the same checks as instantiated classes; we check the number of arguments passed is the same as the number of type parameters, and that each argument is itself well-formed. We then apply the same rewrite described above to the constraints that act on the type variables and check they are well-formed, and do the same for the entailed constraint. We also check that the arguments witness the constraints applied to the type variables, in the method described above.

3.5.6 Expressions

When judging expressions, we not only decide if an expression is well-formed, we give a type for that expression. This type represents the type of any value returned by a successful execution of that expression. To type expressions, we use the value environment E, which provides a mapping from variables to types. Thus, typing variables requires simply looking them up in E, which we do in rules `var-x` and `var-this`.

Typing a field access $\mathtt{e}.f$ works in two stages; first, the target expression `e` is typed as a class with a number of type arguments. Then the field type is looked up in the class declaration. The type of the field access is the type of the field in the declaration, with a rewrite applied to replace the type parameters with their arguments.

A field assignment of the form '$\mathtt{e}_1.f = \mathtt{e}_2$' separately types $\mathtt{e}_1.f$ and $\mathtt{e}_2$ to give the same type τ, and then assigns the overall expression that type. Subsumption, discussed later in this section, removes the need for an explicit subtyping check here.

Creating a new class with a set of type arguments is obviously typed as that class with those arguments. However, we also check that the type is well-formed. If the class has any fields, they must also be initialised at creation, and so we check that each expression signed to each field has the correct type, and that all fields are present.

To type a method call we require several things. The receiving expression has to be typed to get a class to dispatch the method to. We then lookup the method on the class to find the parameter types. We evaluate the types for each of the arguments and check they match those of the parameters. Method calls can also take generic parameters, which are in scope for the parameter list; if any generic parameters are passed, we perform a substitution of the type parameters for their arguments on the parameter types. We also check that the type arguments correctly witness the constraints on the type parameters.

We also include a rule for subsumption that simplifies the application of subtyping to the other rules. If a type τ_1 is a subtype of a type τ_2, and a expression is typed as τ_1, we can safely assign the type τ_2 to that expression. For example, when performing a field assignment $\mathtt{e}_1.f = \mathtt{e}_2$, the right hand side expression $\mathtt{e}_2$ can be a subtype of the left hand side expression $\mathtt{e}_1.f$. Subsumption allows us to invoke the rule `fld-ass` when the rhs is a subtype of the lhs without having to explicitly check that in the rule.

3.6 Operational Semantics and Runtime Specification

Section A.4 given the runtime specification for Genus-, including the runtime entities and the operational semantic. In this section we give a brief overview of the differences between Genus- and a standard runtime system.

3.6.1 Runtime Environments and Resolution

In addition to the standard runtime entities such as heap and stack frame, in Genus- we have introduced a `runtime environment` as part of the execution frame. This is a mapping from the type variables that are currently in scope to concrete types - fully reified types that do not contain any type variables.

Runtime environments are created when execution enters a new object or a new method. If a class contains type parameters, any instantiation of that class must provide type arguments for those parameters. Thus, each object contains its own runtime environment with a mapping of its type parameters to the arguments that were provided at instantiation. A method with type parameters also requires type arguments to be provided upon method call. The variables and arguments form a runtime environment. Therefore, when a method is dispatched to an object, the runtime environment for the stack frame of that execution is the union of the receiver's runtime environment and the runtime environment created by the method call.

Type variables are mapped to concrete types at runtime through *resolution*. The `resolve` function in A.16 describes how resolution occurs; given a type τ and a runtime environment Ω, the `resolve` function recursively looks up any type variables in τ in Ω and replaces them with their concrete counterparts. Resolution is used to build new runtime environments, and is instrumental in our definitions of soundness and agreement. The function *resolve* is similar to the *upperBound* function described in Liétar2016.

3.6.2 Operational Semantics

Figure A.17 provides the operational semantics for Genus-. The are mostly standard execution rules, with the addition of runtime environments, so we discuss them only briefly.

`this` and variables are evaluated by looking them up in the frame; `this` is the address of the receiver (ι in the stack frame), and the value of variables are stored in `vMap` in the frame. Field lookup ($\mathtt{e}.f$) and field assignment ($\mathtt{e}.f = \mathtt{e}'$) both execute the target expression `e` to give an address ι. Field lookup then looks up the value of the field in ι's field map, and field assignment executes $\mathtt{e}'$ and writes the resulting value into the field map.

Instantiation is slightly more complex in that it requires the creation of the object's runtime environment, but it proceeds in the method we described in the previous section. Each expression being assigned to a field is executed in turn, and an object with a new ι is created containing the runtime environment as described earlier, and a field map containing the values from the executed expressions. This object is then added to the heap.

A method call again is slightly more complex than usual. The receiving expression is evaluated to an address, that of the receiving object. The method is looked-up, and the information contained in the program allows us to build the method's runtime environment and value map for the parameters. A new frame is then created, with a runtime environment as described in the previous section, and a variable map containing the evaluated parameter expressions. Execution of the method body then occurs in that frame, and the resulting value is passed back to the calling frame.

3.7 Lemmas

Before proving the soundness of the model we have presented thus far, we establish a number of lemmas that will aid us. Most are relatively straight forward, but where not, we also give a proof.

Lemma 1. *Given* $Prog, \chi \vdash \iota \triangleleft (L[\overline{\tau}], \Omega)$*,* $\chi(\iota) = (L', \Omega', \_)$*,* $Class(Prog, L) = L[\overline{\mathtt{X}}]$ *..., then* $\forall i \,.\, resolve(\tau_i, \Omega) = \Omega'(\mathtt{X}_i)$

The stack-frame's Ω and each object's Ω agree with each other.

Lemma 2. *Given* $Prog \vdash \tau_1 \leq \tau_2$*, then* $\forall\, \Omega, Prog \vdash resolve(\tau_1, \Omega) \leq resolve(\tau_2, \Omega)$

Proof.

1. Assume $\tau_1 = L_1[\overline{\tau_1}]$ and $Class(Prog, L_1) = \mathtt{class}\ L_1[\overline{\mathtt{X}}]\ \mathtt{extends}\ L_2[\overline{\tau_2}]$ ...
2. By givens and (1), $\tau_2 = L_2[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]$
3. $resolve(L_1[\overline{\tau_1}], \Omega) = L_1[\overline{resolve(\tau_1, \Omega)}]$
4. $resolve(L_2[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}], \Omega) = L_2[\overline{resolve(\tau_2, \Omega)}\{\overline{resolve(\tau_1, \Omega)}/\overline{\mathtt{X}}\}]$
5. Let $\overline{resolve(\tau_1, \Omega)} = \overline{\sigma_1}$, and $\overline{resolve(\tau_2, \Omega)} = \overline{\sigma_2}$
6. (3) gives $L_1[\overline{\sigma_1}]$ and (4) gives $L_2[\overline{\sigma_2}\{\overline{\sigma_1}/\overline{\mathtt{X}}\}$
7. By assumption, (6), and definition of `S-Concrete`, $Prog \vdash resolve(\tau_1, \Omega) \leq resolve(\tau_2, \Omega)$

$\square$

Resolution preserves the subtype relationship.

Lemma 3. *For* $\Omega' = (\overline{\mathtt{X} \mapsto resolve(\tau, \Omega)})$*,* $resolve(L[\overline{\tau}], \Omega) \equiv resolve(L[\overline{\mathtt{X}}], \Omega')$

Transitivity of map application.

Lemma 4. *Given* $Class(Prog, L) = \mathtt{class}\ L[\overline{\mathtt{X}}]$ *..., then* $Field(Prog, L, f) \equiv Field(Prog, L[\overline{\mathtt{X}}], f)$

The identity element for rewriting.

Lemma 5. *Given* $Class(Prog, L) = \mathtt{class}\ L[\overline{\mathtt{X}}]$ *..., then* $resolve(Field(Prog, L, f), (\overline{\mathtt{X} \mapsto resolve(\tau, \Omega)})) \equiv resolve(Field(Prog, L[\overline{\tau}], f), \Omega)$

How resolution relates to field lookup.

Lemma 6. $FV(\tau_2) \subseteq \overline{\mathtt{X}} \implies resolve(\tau_2, \overline{\mathtt{X} \mapsto resolve(\tau_1, \Omega)}) \equiv resolve(\tau_2\{\overline{\tau_1}/\overline{\mathtt{X}}\}, \Omega)$

Equivalence of rewrite and resolution.

Lemma 7. *Given* $Prog; \Delta \vdash \overline{\tau} :: C[\overline{\tau}]$*, and* $\overline{\sigma} = \overline{resolve(\tau, \Omega)}$*, then* $Prog; \Delta \vdash \overline{\sigma} :: C[\overline{\sigma}]$

Resolution preserves witnessing.

Lemma 8. *Given* $Prog, \chi \vdash \iota \triangleleft resolve(\tau, \Omega)$ *and* $Prog, \chi \vdash v \triangleleft resolve(Field(Prog, \tau, f), \Omega)$*, then* $Prog, \chi[\iota, f \mapsto v] \vdash \iota \triangleleft resolve(\tau, \Omega)$

Agreement on field write preserves well-formedness.

Lemma 9. $FV(\tau_1) \subseteq \overline{\mathtt{X}} \implies \tau_1\{\overline{\tau_2}/\overline{\mathtt{X}}\} = resolve(\tau_1, (\overline{\mathtt{X} \mapsto \tau_2}))$

Further equivalence of rewrite and resolution.

Lemma 10. *Given*

- $Prog, \chi \vdash \iota \triangleleft resolve(\tau_1, \Omega)$,
- $\chi(\iota) = (L, \Omega', \_)$,
- $Func(Prog, \Delta, \tau_1, m) = \tau_3\ \tau_1.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau_5}]}](\overline{\tau_4})$,
- $Meth(Prog, L, m) = \tau'\ \tau''.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''})\{\,e\,\}$

then

- $resolve(\tau_3, \Omega) = resolve(\tau', \Omega)$,
- $resolve(\tau_1, \Omega) \geq resolve(\tau'', \Omega)$,
- $\forall i, j \ . \ resolve((\tau_5)_{ij}, \Omega) = resolve((\tau''')_{ij}, (\overline{\mathtt{X} \mapsto resolve(\mathtt{X}, \Omega)}, \Omega')$,
- $\forall k \ . \ resolve((\tau_4)_k, \Omega) = resolve((\tau'''')_k, (\overline{\mathtt{X} \mapsto resolve(\mathtt{X}, \Omega)}, \Omega')$

How resolution relates to static and dynamic types on method call.

3.8 Soundness

Now we present a proof of soundness through structural induction on the operational semantics from Section 3.6. Our theorem of soundness is given Figure A.21, and states that given a well-formed program, heap, and stack, a valid expression $\mathtt{e}$ of type τ, and a valid evaluation rule that gives a new heap and a value v, the value v agrees with the type τ, and the new heap is also well-formed. These two things respectively show subject reduction and progress, and therefore show the model to be sound. The following sections present the proof by structural induction on a case-by-case basis, using the lemmas we have define previously.

3.8.1 Case THIS

Given:

1) $\vdash \mathit{Prog} : \mathtt{ok}$
2) $\mathit{Prog}; \Gamma; \Delta; E \vdash \mathtt{this} : E(\mathtt{this})$
3) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi \diamond$
4) $\mathtt{this}, (\iota, \Omega, \mathtt{vMap}), \chi \rightsquigarrow_P \iota, \chi$

Show:

5) $\mathit{Prog}, \chi \vdash \iota \triangleleft \mathit{resolve}(E(\mathtt{this}), \Omega)$
6) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi \diamond$

Proof. Definition of 3) gives 5) and 6) □

3.8.2 Case VAR

Given:

1) $\vdash \mathit{Prog} : \mathtt{ok}$
2) $\mathit{Prog}; \Gamma; \Delta; E \vdash x : E(x)$
3) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi \diamond$
4) $x, (\iota, \Omega, \mathtt{vMap}), \chi \rightsquigarrow_P \mathtt{vMap}(x), \chi$

Show:

5) $\mathit{Prog}, \chi \vdash \mathtt{vMap}(x) \triangleleft \mathit{resolve}(E(x), \Omega)$
6) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi \diamond$

Proof. Definition of 3) gives 5) and 6) □

3.8.3 Case FLD

Given:

1) $\vdash \mathit{Prog} : \mathtt{ok}$
2) $\mathit{Prog}; \Gamma; \Delta; E \vdash \mathtt{e}.f : \tau$
3) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi \diamond$
4) $\mathtt{e}.f, (\iota, \Omega, \mathtt{vMap}), \chi \rightsquigarrow_P v, \chi'$

Show:

5) $\mathit{Prog}, \chi' \vdash v \triangleleft \mathit{resolve}(\tau, \Omega)$
6) $\mathit{Prog}; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi' \diamond$

Proof. 2) gives:

7) $Prog; \Gamma; \Delta; E \vdash \texttt{e} : L[\overline{\tau_1}]$

8) $Class(Prog, L) = \texttt{class}\ L[\overline{\texttt{X}}]\ ...\ \{...\ \tau'\ f\ ...\}$, $FV(\tau') \subseteq \overline{\texttt{X}}$, and $\tau = \tau'\{\overline{\tau_1}/\overline{\texttt{X}}\}$

4) gives:

9) $\texttt{e}, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P \iota', \chi'$

10) $v = \chi'(\iota') \downarrow_3 (f)$

7), 9) and IH gives:

11) $Prog, \chi' \vdash \iota' \lhd resolve(L[\overline{\tau_1}], \Omega)$

12) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi' \diamond$

Assume $\chi'(\iota') = (L', \Omega', \_)$. Then, by 11) and Lemma 1:

13) $\forall i\ .\ resolve((\tau_1)_i, \Omega) = \Omega'(\texttt{X}_i)$

By definition of judgement at 11) and 8) we get:

14) $Prog, \chi' \vdash \chi'(\iota') \downarrow_3 (f) \lhd resolve(\tau', \Omega')$

Which by 10) is equivalent to:

15) $Prog, \chi' \vdash v \lhd resolve(\tau', \Omega')$

By 8) and Lemma 6 we have that:

16) $resolve(\tau, \Omega) = resolve(\tau'\{\overline{\tau_1}/\overline{\texttt{X}}\}, \Omega) = resolve(\tau', (\overline{\texttt{X} \mapsto resolve(\tau_1, \Omega)}))$

By 13):

17) $resolve(\tau', (\overline{\texttt{X} \mapsto resolve(\tau_1, \Omega)})) = resolve(\tau', (\overline{\texttt{X} \mapsto \Omega'(\texttt{X})})) = resolve(\tau', \Omega')$

15), 16), and 17) give 5), and 12) gives 6) □

3.8.4 Case FLD-ASS

Given:

1) $\vdash Prog : \texttt{ok}$

2) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_1.f = \texttt{e}_2 : \tau_2$

3) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi \diamond$

4) $\texttt{e}_1.f = \texttt{e}_2, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P v, \chi'$

Show:

5) $Prog, \chi' \vdash v \lhd resolve(\tau_2, \Omega)$

6) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi' \diamond$

Proof. 2) gives:

7) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_1.f : \tau_1$

8) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_2 : \tau_2$

9) $Prog \vdash \tau_2 \leq \tau_1$

4) gives:

10) $\texttt{e}_1, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P \iota_1, \chi''$

11) $\texttt{e}_2, (\iota, \Omega, \texttt{vMap}), \chi'' \rightsquigarrow_P v, \chi'''$

12) $\chi' = \chi'''[\iota_1, f \mapsto v]$

We can rewrite 12) as:

13) $v = \chi'(\iota_1) \downarrow_3 (f)$

By 7), 10) and IH:

14) $Prog, \chi'' \vdash \chi''(\iota_1) \downarrow_3 (f) \triangleleft resolve(\tau_1, \Omega)$

15) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi'' \diamond$

By 8), 11) and IH:

16) $Prog, \chi''' \vdash v \triangleleft resolve(\tau_2, \Omega)$

17) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi''' \diamond$

By 9) and Lemma 2 we have:

18) $resolve(\tau_2, \Omega) \leq resolve(\tau_1, \Omega)$

16), 18) and the definition of `SA-SUB` gives:

19) $Prog, \chi''' \vdash v \triangleleft resolve(\tau_1, \Omega)$

By definition of 7) and definition of rule `L-F-T`:

20) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_1 : L[\overline{\tau}]$

21) $\tau_1 = Field(Prog, L[\overline{\tau}], f)$

21) makes 19) equivalent to:

22) $Prog, \chi''' \vdash v \triangleleft resolve(Field(Prog, L[\overline{\tau}], f), \Omega)$

10), 20), and IH gives:

23) $Prog, \chi''' \vdash \iota_1 \triangleleft resolve(L[\overline{\tau}], \Omega)$

22), 23), and Lemma 8 gives:

24) $Prog, \chi'''[\iota_1, f \mapsto v] \vdash \iota_1 \triangleleft resolve(L[\overline{\tau}], \Omega)$

By 17) and 24), we get 6).
To show 5), we consider 3 cases:

1. $v = \texttt{null}$: trivial
2. $v = \iota' \neq \iota_1$: given by 16)
3. $v = \iota_1$: given by 16) and 19)

□

3.8.5 Case NEW

Given:

1) $\vdash Prog : \texttt{ok}$

2) $Prog; \Gamma; \Delta; E \vdash \texttt{new } L[\overline{\tau_1}](\overline{f = \texttt{e}}) : L[\overline{\tau_1}]$

3) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi \diamond$

4) $\texttt{new } L[\overline{\tau_1}](\overline{f = \texttt{e}}), (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P \iota', \chi'$

Show:

5) $Prog, \chi' \vdash \iota' \triangleleft resolve(L[\overline{\tau_1}], \Omega)$

6) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi' \diamond$

Proof. 2) gives:

7) $Prog; \Gamma \vdash L[\overline{\tau_1}] : \texttt{ok}$

8) $\forall i \;.\; Prog; \Gamma; \Delta; E \vdash \texttt{e}_i : Field(Prog, L[\overline{\tau_1}], f_i)$

9) $\overline{f} = Fields(Prog, L)$

4) gives:

10) $\texttt{e}_i, (\iota, \Omega, \texttt{vMap}), \chi_i \rightsquigarrow_P v_i, \chi_{i+1} \quad (i = 0, ..., n-1)$

11) $Class(Prog, L) = \texttt{class}\ L[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau_2}]}]\ ...$

12) for new ι', $\chi' = \chi_n[\iota' \mapsto (L, (\overline{\texttt{X} \mapsto resolve(\tau_1, \Omega)}), (\overline{f \mapsto v}))]$

By 8), 10), and IH:

13) $\forall i \;.\; Prog, \chi' \vdash v_i \triangleleft resolve(Field(Prog, L[\overline{\tau_1}], f_i), \Omega)$

14) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi_i \;\diamond \quad (i = 1, .., n)$

For simplicity, we rewrite 12) as:

15) $\chi'(\iota') = (L, \Omega', (\overline{f \mapsto v}))$, where $\Omega' = (\overline{\texttt{X} \mapsto resolve(\tau_1, \Omega)})$

11), 15), and Lemma 3 give:

16) $Prog, \chi' \vdash \iota' <: resolve(L[\overline{\tau_1}], \Omega)$

13), 15), and Lemma 5 give:

17) $\forall i \;.\; Prog, \chi' \vdash \chi'(\iota') \downarrow_3 (f_i) \triangleleft resolve(Field(Prog, L, f_i), \Omega')$

By definition of the judgement at 7):

18) $\forall j \;.\; Prog; \Gamma; \Delta \vdash (\overline{\tau_2\{\overline{\tau_1}/\overline{\texttt{X}}\}})_j :: C[\overline{\tau_2\{\overline{\tau_1}/\overline{\texttt{X}}\}}]_j$

Given 11) and the shorthand from 15), application of Lemmas 7 and 6 to 18) gives:

19) $\forall j \;.\; Prog; \Gamma; \Delta \vdash (\overline{resolve(\tau_2, \Omega')})_j :: C[\overline{resolve(\tau_2, \Omega')}]_j$

15), 16), 17), and 19) give 5). 5) and 14) give 6). □

3.8.6 Case M-CALL

Given:

1) $\vdash Prog : \texttt{ok}$

2) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_1.m[\overline{\tau_2}](\overline{\texttt{e}_4}) : \tau_3$

3) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi \diamond$

4) $\texttt{e}_1.m[\overline{\tau_2}](\overline{\texttt{e}_4}), (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P v, \chi'$

Show:

5) $Prog, \chi' \vdash v \triangleleft resolve(\tau_3, \Omega)$

6) $Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi' \diamond$

Proof. 2) gives:

7) $Prog; \Gamma; \Delta; E \vdash \texttt{e}_1 : \tau_1$

8) $\forall i \;.\; Prog; \Gamma \vdash (\tau_2)_i : \texttt{ok}$

9) $Func(Prog, \Delta, \tau_1, m) = \tau_3\ \tau_1.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau_5}]}](\overline{\tau_4})$

10) $Prog; \Gamma; \Delta; E \vdash (\mathtt{e}_4)_i : (\tau_4\{\overline{\tau_2/\mathtt{X}}\}) \quad (i = 0, ..., n-1)$

11) $Prog; \Gamma; \Delta \vdash (\overline{\tau_5}\{\overline{\tau_2/\mathtt{X}}\})_i :: C[\overline{\tau_5}\{\overline{\tau_2/\mathtt{X}}\}]_i \quad (i = 0, ..., n-1)$

4) gives:

12) $\mathtt{e}_1, (\iota, \Omega, \mathtt{vMap}), \chi \rightsquigarrow_P \iota', \chi_0$

13) Assume $\chi_0(\iota') = (L, \Omega', \_)$

14) $(\mathtt{e}_4)_i, (\iota, \Omega, \mathtt{vMap}), \chi_i \rightsquigarrow_P v_i, \chi_{i+1} \quad (i = 0, ..., n-1)$

15) $Meth(Prog, L, m) = \tau'\ \tau''.m[\overline{\mathtt{Y}}\ \mathtt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''\ x})\{\mathtt{e}'\}$

16) $\mathtt{e}', (\iota', (\overline{x \mapsto v}), (\overline{\mathtt{Y} \mapsto resolve(\tau_2, \Omega)} \cup \Omega')), \chi_n \rightsquigarrow v, \chi'$

7), 12), and IH:

17) $Prog, \chi_0 \vdash \iota' \lhd resolve(\tau_1, \Omega)$

18) $Prog; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi_0 \diamond$

10), 14), and IH:

19) $Prog, \chi' \vdash v_i \lhd resolve((\tau_4\{\overline{\tau_2/\mathtt{X}}\})_i, \Omega) \quad (i = 0, ..., n-1)$

20) $Prog; \Delta; E \vdash (\iota, \Omega, \mathtt{vMap}), \chi_{i+1} \diamond \quad (i = 0, ..., n-1)$

By 17), 13), 9), and 15):

21) $resolve(\tau_3, \Omega) = resolve(\tau', \Omega')$

22) $resolve(\tau_1, \Omega) \geq resolve(\tau'', \Omega')$

23) $\forall i, j\ .\ resolve((\tau_5)_{ij}, \Omega) = resolve((\tau''')_{ij}, (\overline{\mathtt{X} \mapsto resolve(\mathtt{X}, \Omega)} \cup \Omega'))$

24) $\forall i\ .\ resolve((\tau_4)_i, \Omega) = resolve((\tau'''')_i, (\overline{\mathtt{X} \mapsto resolve(\mathtt{X}, \Omega)} \cup \Omega'))$

By 17), 22), $FV(\tau'') \cap \overline{\mathtt{X}} = \emptyset$ and subsumption of agreement:

25) $Prog, \chi_n \vdash \iota' \lhd resolve(\tau'', (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'))$

By 19), 24), and rules of rewrite

26) $\forall i\ .\ Prog, \chi_n \vdash v_i \lhd resolve((\tau'''')_i, (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'))$

By 11), 23), and Lemma 7

27) $\forall i\ .\ Prog, \Delta \vdash (\overline{resolve((\tau'''), (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'))})_i :: C[\overline{resolve((\tau'''), (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'))}]_i$

By 20), 25), 26), 27)

28) $Prog; \Delta, \overline{C[\overline{\tau'''}]}; \mathtt{this} : \tau'', \overline{x : \tau''''} \vdash (\iota', (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'), (\overline{x \mapsto v})), \chi_n \diamond$

By 1) and 15)

29) $Prog; \Delta, \overline{C[\overline{\tau'''}]}; \mathtt{this} : \tau'', \overline{x : \tau''''} \vdash \mathtt{e}' : \tau'$

By 29), 28), 16) and IH

30) $Prog, \chi' \vdash V \lhd resolve(\tau', (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'))$

31) $Prog; \Delta, \overline{C[\overline{\tau'''}]}; \mathtt{this} : \tau'', \overline{x : \tau''''} \vdash (\iota', (\overline{\mathtt{X} \mapsto resolve(\tau_2, \Omega)} \cup \Omega'), (\overline{x \mapsto v})), \chi' \diamond$

By 30) and $FV(\tau') \cap \overline{\mathtt{X}} = \emptyset$

32) $Prog, \chi' \vdash v \lhd resolve(\tau', \Omega')$

From 21) and 32) we get 5), and by 20) and 31) we get 6). $\square$

4. DeGen

In this chapter, we introduce DeGen (**De**ny Capability **Gen**us). DeGen takes the Genus- model we have detailed in Chapter 3 and applies deny capabilities to it. We first begin with a study into how we would represent some of our Genus- examples in Pony, and then show how these are easily translated across to DeGen. We then give a syntax and type system for Degen updated from Genus-, and discuss how we would show that the proof of soundness still holds. We then introduce the concept of well-formed visibility, and discuss how we could show that it holds for DeGen.

Our main focus when designing DeGen was the end goal of a concise formal system which can stand as an alternative form of generics to those that currently exist in Pony. By pursuing the design strategy of representing Genus- concepts in Pony first, we were empowered by the existing Pony compiler, which guided our decisions concerning how capabilities should be used and applied to Genus-. As such, all Pony examples in the section below as valid Pony programs that compile and run.

4.1 Pony

4.1.1 Classes and Constraints

```
interface Eq[T: Eq[T] #read]
  fun box eq(other: T): Bool

class IntPair
  var fst: U32 val
  var snd: U32 val

  new box create(fst': U32, snd': U32) =>
    this.fst = fst'
    this.snd = snd'

  fun box eq(other: IntPair box) : Bool =>
    (this.fst == other.fst) and (this.snd == other.snd)

actor Main
  new create(env: Env) =>
    var pairA : IntPair box = IntPair.create(1, 2)
    var pairB : IntPair box = IntPair.create(1, 2)
    env.out.print(check[IntPair box](pairA, pairB)) // "true"

  fun box check[T : Eq[T] #read](a : T, b : T) : String val =>
    if a.eq(b) then
      "true"
    else
      "false"
    end
```

Figure 4.1: `Eq`, `IntPair` and calling code in Pony

Figure 4.1 shows how we can represent `IntPair` and its equality constraint in Pony. We can see that instead of using a constraint to require equality as in Genus-, we must resort back to the using an interface. The interface has a single type parameter `T`, which is bounded using f-bounded polymorphism. The `#read` capability constraint in the generic parameters says that not only must `T` implement `Eq[T]`, it must also have a capability that allows it to be read from, i.e. it must be *box,* *val*, or *ref*. The capability of `T` will be passed as part of the type argument, so we do not specify what capability `other` will have - it will be the same as the type argument. Requiring some form of readability is sensible here, as we want to allow our equality method to be able to freely read fields from the parameter, however we do not want anyone who implements the interface to be able to mutate the parameter. The equality method is a *box* method, meaning that the receiver must have a *box* capability, or a capability that is a subcapability of *box*. *box* references are immutable, and so if the receiver is a *box*, the method cannot mutate any state.

Moving down to `IntPair`, we see immediately that it does not explicitly implement anything. In Pony, interfaces are determined by structural typing, much the same as Genus- constraints. Therefore, as long as it implements all the methods required by `Eq`, it can be used anywhere that interface is expected. The fields `fst` and `snd` both have *val* capabilities, meaning there can be no mutable references to them; including the capability here is unnecessary, because U32 is a primitive type, and so no reference of any kind can exist to the fields.

Constructors in Pony use capabilities differently to normal methods; rather than specifying the required capability of the receiver, it specifies the capability of the reference to the new object that is returned to the calling code. If omitted, class constructors default to *ref* and actor constructors default to *tag*. However, we can override this default by explicitly specifying which capability we want the reference to have. In the code above, we have nothing which mutates any reference to an `IntPair`, so we specify we want an `IntPair` *box* from the constructor.

The method `check` is a *box* method because we want to ensure it does not mutate any code. It takes a type parameter `T` with a `#read` capability constraint; we want `check` to be able to take any type parameter that allows us to call its equality method while also guaranteeing we won't mutate it. The calling code for `check` gives `IntPair box` as the argument, which is a valid type.

```
constraint Eq[T #any] {
  box equals(T other) : Bool
}

class IntPair {
  int fst val;
  int snd val;
  box equals(IntPair box other) : bool {
    return this.fst == other.fst && this.snd == other.snd;
  }
}

actor Main {
  main() : void {
    pairA = new IntPair(fst=1, snd=2) box;
    pairB = new IntPair(fst=1, snd=2) box;
    this.check[IntPair](pairA, pairB) // "true"
  }

  box check[T where Eq[T]](T a, T b) : String val {
      if (a.eq(b)) {
        return "true"
      } else {
        return "false"
      }
  }
}
```

Figure 4.2: `Eq`, `IntPair` and calling code in DeGen

Figure 4.2 shows how we can apply ideas from the Pony implementation of `IntPair` to Genus-to give us a DeGen respresentation. `Eq` and `IntPair` have the same implementation as in the previous chapter, with the addition of capabilities. We have also changed the method syntax to avoid confusion between the method capability and the return type capability.

`Eq` is not a direct counterpart of the interface in Figure 4.1; the interface has a type parameter `T` which is used to give the type of `other` in the `eq` method, whereas the parameter `T` in the constraint in Figure 4.2 instead represents the class that witnesses the constraint. This difference is subtle, but says that, in order for a type to witness the `Eq` constraint, not only must it be structurally correct, it must also be readable. The parameter passed must also be readable by extension. In addition to this requirement, it also means we cannot test equality against two objects of different classes, which is discussed more when we discuss Figure 4.6.

When creating objects, we copy the behaviour of Pony that we can specify the capability of the new reference. We do this by placing the capability after the type. We create two `IntPair` `box`, as we did in Pony.

4.1.2 Multivariate Constraints

```
use "collections"

  interface AbstractVertex[
      EdgeType :
        AbstractEdge[ActualVertexType, EdgeType],
      ActualVertexType :
        AbstractVertex[EdgeType, ActualVertexType]]
    fun box outgoingEdges() : this->List[EdgeType]
    fun box incomingEdges() : this->List[EdgeType]

  interface AbstractEdge[
      VertexType :
        AbstractVertex[ActualEdgeType, VertexType],
      ActualEdgeType :
        AbstractEdge[VertexType, ActualEdgeType]]
    fun box source() : this->VertexType
    fun box sink() : this->VertexType

  class Vertex
    var _out : List[Edge] ref
    var _in : List[Edge] ref
    fun box outgoingEdges() : this->List[Edge]  => _out
    fun box incomingEdges() : this->List[Edge] => _in
    new ref create(outList : List[Edge], inList : List[Edge]) =>
      _out = outList
      _in = inList

  class Edge
    var _source : Vertex ref
    var _sink : Vertex ref
    fun box source() : this->Vertex => _source
    fun box sink() : this->Vertex => _sink
    new ref create(source' : Vertex, sink' : Vertex) =>
      _source = source'
      _sink = sink'

  actor Main
    new create(env: Env) =>
      var v1 : Vertex ref = Vertex.create(List[Edge].create(),
          List[Edge].create())
      var v2 : Vertex ref = Vertex.create(List[Edge].create(),
          List[Edge].create())
      var e : Edge ref = Edge.create(v1, v2)
      env.out.print(check[Edge, Vertex](e, v1, v2)) // "done"

    fun box check[E : AbstractEdge[V, E] ref,
        V : AbstractVertex[E, V] ref](e : E, v1 : V, v2 : V) : String val =>
      v1.outgoingEdges().push(e)
      v2.incomingEdges().push(e)
      "done"
```

Figure 4.3: `Vertex`, `Edge`, and calling code in Pony

In Figure 4.3 we attempt to recreate the behaviour of the `GraphLike` constraint. The constraint is intended to create a predicate on the behaviour of two classes; together they provide the behaviours required to represent a graph structure in code. To replicate this guarantee in Pony, which uses f-bounded polymorphism, we need very complex type bounds. Take `AbstractVertex` for example. It has 2 type parameters, `EdgeType` and `ActualVertexType`. `EdgeType` the type of the in- and out-going edges from the vertex, is bounded by `AbstractEdge[ActualVertexType, EdgeType]`, and `ActualVertexType` is bounded by`AbstractVertex[EdgeType, ActualVertexType]`. This means that `AbstractVertex` can use any `AbstractEdge` that takes its `ActualVertexType` and itself as arguments, and its `ActualVertexType` must be an `AbstractVertex` taking the same arguments. This is known as "generic clutter", and is required to express the mutual dependency between the vertex and edge types. `AbstractEdge` is defined in similar terms.

We don't provide capability constraints to any of the type parameters in the two interfaces, thus they get the default constraint of `#any`. We could safely give the type paramters a `#read` constraint, but chose to omit this in preference of not making the definitions even harder to read.

The classes `Vertex` and `Edge` structurally implement the interfaces `AbstractVertex` and `AbstractEdge` respectively, and the codependency they require; `Edge` implements `AbstractEdge[Vertex, Edge]`, and `Vertex` implements `AbstractVertex[Edge, Vertex]`. Since the implementation, and thus the codependency, is structural, the classes themselves are relatively simple, with each directly using the other for field and return types.

Another feature this example introduces is the viewpoint adaptation `this->`. The getters for the lists of edges and vertexes are all *box* methods. This means that not only can it be called with a *box* receiver, it can be called with a *ref* or *val* receiver as well (due to subcapabilities). The viewpoint `this->Vertex` in `Edge` therefore means "a `Vertex` as viewed by `this`". Since the default capability is *ref* (as none is specified in the class declaration), this means the return type is a `Vertex` with the capability `this->`*ref*. Thus, we return a *box* when `this` is a *box*, a *val* when `this` is a *val*, and a *ref* when `this` is a *ref*. This pairwise type equality is required for us to be able to return a readable field when the receiver is readable.

```
constraint GraphLike[V #any, E #any] {
  box V.outgoingEdges() : this->List[E];
  box V.incomingEdges() : this->List[E];
  box E.source() : this->V;
  box E.sink() : this->V;
}

class Vertex {
  List[Edge] _out ref;
  List[Edge] _in ref;
  box outgoingEdges() : this->List[Edge] {
    return _out;
  }
  box incomingEdges()  : this->List[Edge] {
    return _in;
  }
}

class Edge {
  Vertex _source ref;
  Vertex _sink ref;
  box source() : this->Vertex {
    return _source;
  }
  box sink() this->Vertex {
    return _sink;
  }
}

actor Main {
  main() : void {
    v1 = new Vertex(_out={}, _in={}) ref;
    v2 = new Vertex(_out={}, _in={}) ref;
    e = new Edge(_source=v1, _sink=v2) ref;
    this.do[Vertex, Edge](e, v1, v2) // "done"
  }

  box check[V ref, E ref where Graphlike[V, E]](E e, V v1, V v2) : String val {
    v1.outgoingEdges().add(e);
    v2.incomingEdge().add(e);
    return "done";
  }
}
```

Figure 4.4: `GraphLike`, `Vertex`, `Edge`, and calling code in DeGen

It is in Figure 4.4 that the true beauty of constraints is realised; we can replace several lines of generic clutter with a single succinct, multivariate constraint. The design is far cleaner, and the way in which it works is far more obvious. We include the `#read` capability constraints, which no longer provide additional clutter. The methods are defined similarly to in the above interfaces, with the same method capabilities and viewpoint adapted return types.

In this example, we take a liberty with the use of `List`, referencing it freely without defining it. We assume it to be a class with a single type parameter to represent the type of its elements, and that `{}` represents an empty list.

We can also see that in the type parameters, we use *ref* as a capability constraint; this is valid, and all other capabilities can also be used in place of a constraint. A valid argument is any capability that is a subcapability of the constraint, be it a singular constraint such as *ref*, or a grouped constraint such as `#read`.

4.1.3 Parametric Constraints and Classes

```
interface Eq[T: Eq[T] #read]
  fun eq(other: T): Bool

interface Box[T : Eq[T] #read] is Eq[Box[T]]
  fun box get() : this->T
  fun ref set(item : T) : T

class Num
  var x : U32 val
  fun box eq(other : Num) : Bool =>
    x == other.x
  new ref create(x' : U32) =>
    x = x'

class VarBox[T : Eq[T] #read] is Box[T]
  var item : T

  new ref create(item' : T) =>
    item = item'

  fun box eq(other: Box[T] ref): Bool =>
    item.eq(other.get())

  fun box get() : this->T => item
  fun ref set(item' : T) : T =>
    item = item'

class NumBox is Box[Num]
  var item : Num ref
  new create(item' : Num ref) =>
    item = item'

  fun eq(other: Box[Num]): Bool =>
    item.eq(other.get())

  fun box get() : this->Num => item
  fun ref set(item' : Num) : Num =>
    item = item'

actor Main
  new create(env: Env) =>
    var vbox : VarBox[Num] = VarBox[Num].create(Num.create(14))
    var nbox : NumBox = NumBox.create(Num.create(15))
    env.out.print(check[Box[Num]](vbox, nbox)) // "false"

  fun check[T : Eq[T] #read](a : T, b : T) : String val =>
    if a.eq(b) then
      "true"
    else
      "false"
    end
```

Figure 4.5: `Box`, `VarBox`, `NumBox`, and calling code in Pony

In Figure 4.5 we have adapted the `Box` [1] constraint and its related classes for Pony. However, instead of `StringBox` as previously, we have instead created a class `Num` and a wrapper for `U32`s, as `String` is an in-built type in Pony, and creating our own class allows us to implement our own version of equality.

To build the `Box` interface whilst ensuring we can test equality on any `Box`, we have used Pony's interface inheritance to ensure that classes that implement `Box` also implement `Eq`. However, since we specify that `Box[T] is Eq[Box[T]]`, the parameter of the `eq` method in any class that implements `Box[T]` must be of type `Box[T]`. For example, if we were to make `other` in `VarBox[T]`'s `eq` method type `VarBox[T]`, the compiler would complain that it doesn't correctly implement `Box[T]`. Likewise, `NumBox` must have the parameter type as `Box[Num]`.

We have two classes, `VarBox` and `NumBox` that both implement `Box` in some form. `NumBox` specifies explicitly that it is a `Box` with type parameter `Num`, and as such provides all the methods to structurally conform to `Box[Num]`. `VarBox`, on the other hand, implements `Box[T]`, and so does not replace any occurrences of the type parameter in the `Box` interface. It uses the parameter `T` as the type of its field, also omitting the capability that will be passed with the type argument. Note that when the `VarBox` is initialised in `Main`, the capability is omitted. This means the capability defaults to *ref*. When `vbox` is created, `Num` is passed as the type parameter to `VarBox`, meaning `vbox` implements `Box[Num]`. As a result, `check` requires only a single type parameter, which we pass `Box[Num]` to, as both `vbox` and `nbox` implement this interface, meaning we can compare equality on two 'different' classes[2].

We can see that the setter method on `Box` is a *ref* method, which it needs to be because it mutates state. We also make the `item` field in `NumBox` *ref*, to ensure the pairwise type equality for the viewpoints that we discussed earlier.

[1]Not to be confused with the *box* capability.

[2]'Different' is in quotes here because the difference is syntactic; passing `Num` as type argument for `T` in `VarBox` makes is semantically identical to `NumBox`. However, either could contain additional behaviour unrelated to `Box` and that would not affect the comparison.

```
constraint Eq[T #any] {
  box equals(T other) : bool
}
constraint Box[B #any, T #read where Eq[T]] extends Eq[B] {
  box B.get() : this->T;
  ref B.set(T obj) : void;
}

class Num {
  int val x;
  box equals(Num other) : bool {
    return x == other.x
  }
}

class VarBox[T #read where Eq[T]] {
  T item;
  box get() this->T {
    return this.item;
  }
  ref set(T obj) : set {
    this.item = obj;
  }
  box equals(VarBox[T] other) : bool {
    return this.item.equals(other.item);
  }
}

class NumBox {
  Num item ref;
  box get() : this->Num {
    return this.item;
  }
  ref set(Num obj) : void {
    this.item = obj;
  }
  box equals(NumBox other) : bool {
    return this.item.equals(other.item);
  }
}

actor Main {
  void main() {
    vbox = new VarBox[Num ref](item = new Num(x=14) ref) ref;
    nbox = new NumBox(item = new Num(x=15) ref) ref;

    this.check[Num ref](vbox.get(), nbox.get()); // "false"
  }
  String check[T #read where Eq[T]](T a, T b) {
      if (a.eq(b)) {
        return "true"
      } else {
        return "false"
      }
  }
}
```

Figure 4.6: Box, VarBox, NumBox and calling code in DeGen

When translating this behaviour to DeGen we use entailed constraints to represent the relationship between `Eq` and `Box` - any class that witnesses `Box` must also witness `Eq`. `Box` has a self parameter `B`, and the constraint requires a witness would have the `get` and `set` method as above. It also has a type parameter `T` that is constrained with equality, and represents the type of `Box`'s field.

Again, we have a class `Num`, and all capability choices are the same as in the previous example. The interesting point of this example is the limitation of DeGen that it exposes. When comparing equality on `vbox` and `nbox`, we cannot directly compare the two and invoke their `eq` methods on each other. Instead we must use their `get` methods to test equality on their items. This is due to the lack of interfaces in DeGen; constraints cannot be used directly as types and instead constrain type variables. We cannot have a parameter type be `Box[T]` as this isn't a valid type. Therefore, we cannot invoke the equals method of `VarBox[Num]` on `NumBox`, despite them being the essentially the same. While this is a limitation in the expressiveness of DeGen, we can argue in its favour. When comparing equality of two distinct classes, the only way such an operation would make sense is if the classes had some aspect that is the same (the `Num` field in this case). It is therefore obvious that we don't want to compare the classes themselves, but instead we wish to compare the aspects of them which are the same. Thus, we obtain the same behaviour by extracting and comparing the fields directly.

4.2 Syntax

4.2.1 Actors

$$\begin{array}{lrl}
\textit{Programs} & \texttt{P} ::= & \overline{\texttt{CO}}\ \overline{\texttt{CL}}\ \overline{\texttt{A}} \\
\textit{Actors} & \texttt{A} ::= & \texttt{actor}\ A[\beta]\ \{\overline{\texttt{fld}}\ \overline{\texttt{methSig}}\ \overline{\texttt{behave}}\} \\
\textit{Constraints} & \texttt{CO} ::= & \texttt{constraint}\ C_1[\overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C_2[\overline{\texttt{Y}}]}] \\
 & & \quad \texttt{extends}\ C_3[\overline{\texttt{Z}}]\ \{\overline{\texttt{methSig}}\ \overline{\texttt{behaveSig}}\} \\
\textit{Behaviour Signature} & \texttt{behaveSig} ::= & \tau_1.b[\beta](\overline{\tau_2}) \\
\textit{Behaviour} & \texttt{behave} ::= & \texttt{behaveSig}\ \{\texttt{e}\} \\
\textit{Method Signature} & \texttt{methSig} ::= & \tau_2.m[\beta](\overline{\tau_3}) : \tau_1 \\
\textit{Generic Parameters} & \beta ::= & \overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C[\overline{\tau}]}
\end{array}$$

$$\begin{array}{llll}
\textit{Actor Names} & A & \textit{Behaviour Names} & b
\end{array}$$

Figure 4.7: Syntax

DeGen introduces actors into the Genus- model, and Figure 4.7 gives the syntax for them. We have highlighted new additions.

Programs now consist of constraint, class, and actor definitions. Actors have generics parameters, fields, and methods the same classes, but have an additional sequence of definitions: behaviours. Note that actors do not have an `extends` clause, and as a result there is no subtype relation between actors.

Behaviours are similar to functions, except that behaviour calls are asynchronous; calling a behaviour on any actor will execute it at some indeterminate point in the future, as the actor processes its message list. Behaviour signatures therefore do not have return types, and instead consist solely of a receiver type, a behaviour name, generic parameters, and parameter types. We have also altered the method signature syntax to reflect the changes discussed in the previous section.

Actors can witness constraints in much the same way as classes, so constraints also contain a sequence of behaviour signatures in addition to method signatures.

The generic parameters of classes, constraints, actors, and methods now require capability constraints on the type variables, represented by v.

4.2.2 Syntax of Expressions

$$\begin{array}{lrl}
\textit{Expressions} & \texttt{e} ::= & x \mid \texttt{e}.f \mid \texttt{e}.f\ =\ \texttt{e} \mid \texttt{this} \mid \texttt{null} \mid \texttt{new}\ L[\overline{\tau}](\overline{f = \texttt{e}})\ \kappa \mid \\
 & & \texttt{new}\ A[\overline{\tau}](\overline{f = \texttt{e}}) \mid \texttt{e}.m[\overline{\tau}](\overline{\texttt{e}}) \mid \texttt{e}.b[\overline{\tau}](\overline{\texttt{e}}) \mid \texttt{recover e}
\end{array}$$

Figure 4.8: Syntax of Expressions

In Figure 4.8 we give an expanded syntax for expressions to incorporate the new behaviour capabilities has introduced. We have added a capability to the expression `new` to represent the capability of the returned reference. We have created syntax for behaviour calls, which is nearly identical to that of method calls. Finally, we have `recover`, which allows us to recover mutable types back to isolated references (or any capability).

We have also introduced `null`. Previously, we had no need for `null` in Genus-, as we required that all fields on an object be initialised at creation, and so at no point would any reference be a null pointer. However, now we are working with actors, we have weakened this constraint, for reasons we will discuss in later.

In addition to the new and updated expressions, the behaviour of field write is modified to support destructive reads. Instead of a field write returning the written expression, it returns the previous value of the field before the write. This value is unaliased at this point, and so allows us to do things such as send *iso*s between actors by performing a destructive read when passing the *iso* as a parameter to a behaviour.

4.2.3 Syntax of Types

$$
\begin{array}{lrcl}
\textit{Types} & \tau & ::= & \mathtt{X} \mid D[\overline{\tau}]\ \kappa \mid \tau+ \mid \tau- \mid \mathtt{V} \triangleright \tau \mid \mathtt{V} \triangleright\!\!\!- \tau \mid \textit{recover}\ \tau \\
\textit{Type Identifiers} & D & ::= & L \mid A \\
\textit{Capabilities} & \kappa & ::= & \textit{iso} \mid \textit{trn} \mid \textit{ref} \mid \textit{val} \mid \textit{box} \mid \textit{tag} \mid \textit{iso}^{-} \mid \textit{trn}^{-} \\
\textit{Capability Constraints} & \upsilon & ::= & \#\textit{any} \mid \#\textit{read} \mid \#\textit{send} \mid \#\textit{share} \mid \#\textit{alias} \mid \#\textit{any}^{-} \mid \#\textit{send}^{-} \\
\textit{Viewpoints} & \mathtt{V} & ::= & \kappa \mid \tau \mid \mathtt{this} \\
\textit{Type Environments} & \Gamma & ::= & \overline{\mathtt{X}\ \upsilon}
\end{array}
$$

Figure 4.9: Syntax of Types

Each type in DeGen has an associated capability, which specify how a reference of that type may be used. Classes are the same as previously, with each type argument also having an associated capability.

Type variables can still be used in place of any type where they are in scope. As a result, type variables do not have capabilities, but their arguments are required to. We saw in Figure 4.6 that instantiating a `VarBox[T]` with `Num ref` replaces each instance of `T` with `Num ref`.

We also define 5 operators that can act on types. Aliasing (+) and unaliasing (-) represent their associated operations; aliasing an *iso* would result in a *tag*, and so `(IntPair iso)+` and `IntPair tag` are semantically equivalent. Unaliasing an *iso* would result, unsurprisingly, in an $\textit{iso}^{-}$. The next two operators are adaptations: viewpoint adaptation $\triangleright$ and extracting adaptation $\triangleright\!\!\!-$. We have seen viewpoint adaptation in previous examples as `->`; it is used in cases where the reference capability of the origin is unknown. The extracting adaptation $\triangleright\!\!\!-$, introduced by Steed201X, gives the capability of a value returned following a destructive read. Finally, we have an operator *recover*, where *recover* τ is the type we get after recovering from an expression.

The capabilities we define are the expected deny capabilities described earlier in this report, and in previous models of deny capability systems, including Pony. The capability constraints are as expected also, but with two additional constraints $\#\textit{any}^{-}$ and $\#\textit{send}^{-}$ to represent any ephemeral (unaliased) capability and any sendable ephemeral capability respectively.

Viewpoints can be either another capability, a type, or `this`. When using a type we use only the capability of that type; `IntPair->ref` would give `ref->ref`, as the default capability of `IntPair` is *ref*. We can also use a type variable here, where the capability is not yet known, but will be known at runtime. Using `this` as a viewpoint refers to the receiver capability, as discussed previously.

Type environments now keep track of not only what type variables are in scope, but also each type variable's respective capability constraint.

4.3 Type System

4.3.1 Subcapabilities and Capability Constraints

Here we specify a number of relations on capabilities that will allow us to update previous rules from Genus-. Firstly, the subcapability relation, reproduced from Steed2016, is shown in Figure 4.10.

$$\frac{}{\mathit{iso}^- \leqslant \{\mathit{iso}, \mathit{trn}^-\}} \qquad \frac{}{\mathit{trn}^- \leqslant \{\mathit{trn}, \mathit{ref}, \mathit{val}\}} \qquad \frac{}{\{\mathit{trn}, \mathit{ref}, \mathit{val}\} \leqslant \mathit{box}} \qquad \frac{}{\{\mathit{iso}, \mathit{box}\} \leqslant \mathit{tag}}$$

$$\frac{}{\kappa \leqslant \kappa} \qquad \frac{\kappa \leqslant \kappa' \qquad \kappa' \leqslant \kappa''}{\kappa \leqslant \kappa''}$$

Figure 4.10: Subtyping of Capabilities, reproduced from Steed2016

The shorthand $\kappa \leqslant \{\kappa', \kappa''\}$ means κ is a subcapability of both κ' and κ''. From this relation we can build the graphical model of subcapabilities, shown in Figure 4.11.

$$\begin{tikzcd}
\mathit{tag} & & & \\
& \mathit{box} \arrow[lu] & & \\
& \mathit{trn} \arrow[u] & \mathit{ref} \arrow[lu] & \mathit{val} \arrow[llu] \\
\mathit{iso} \arrow[uuu] & \mathit{trn}^- \arrow[u] \arrow[ru] \arrow[rru] & & \\
\mathit{iso}^- \arrow[u] \arrow[ru] & & &
\end{tikzcd}$$

Figure 4.11: Subtyping of Capabilities, reproduced from Steed2016

We also define how capabilities comply with capability constraints. We show the compliance relation $\ll$ in Figure 4.12.

$$\frac{}{\{\mathit{iso},\ \mathit{trn},\ \mathit{ref},\ \mathit{val},\ \mathit{box},\ \mathit{tag}\} \ll \#\mathit{any}} \qquad \frac{}{\{\mathit{iso},\ \mathit{val},\ \mathit{tag}\} \ll \#\mathit{send}}$$

$$\frac{}{\{\mathit{iso}^-, \mathit{trn}^-, \mathit{ref},\ \mathit{val},\ \mathit{box},\ \mathit{tag}\} \ll \#\mathit{any}^-} \qquad \frac{}{\{\mathit{iso}^-, \mathit{val},\ \mathit{tag}\} \ll \#\mathit{send}^-}$$

$$\frac{}{\{\mathit{ref},\ \mathit{val},\ \mathit{box},\ \mathit{tag}\} \ll \#\mathit{alias}} \qquad \frac{}{\{\mathit{ref},\ \mathit{val},\ \mathit{box}\} \ll \#\mathit{read}} \qquad \frac{}{\{\mathit{val},\ \mathit{tag}\} \ll \#\mathit{share}} \qquad \frac{}{\kappa \ll \kappa}$$

Figure 4.12: Capability Constraint Compliance

The compliance relation between capabilities and capability constraints allows us to determine whether a type is a valid argument to a type parameter; the argument must witness any constraints on the the parameter, and also have a compliant capability.

$$\frac{}{\{\#read,\ \#alias,\ \#share, \#send^{-}\} \prec \#any^{-}} \qquad \frac{}{\{\#read,\ \#alias,\ \#share, \#send\} \prec \#any}$$

$$\frac{}{\#share \prec \{\#send, \#send^{-}\}} \qquad \frac{\kappa \ll \upsilon}{\kappa \prec \upsilon}$$

Figure 4.13: Subcompliance

4.3.2 Aliasing and Unaliasing

The aliasing operator + is well-formed if $\kappa+$ is the minimum compatible capability for a new alias to an object, given that the capability of the original path is κ. Figure 4.14 gives the complete well-formed definition of aliasing.

$$\kappa+ = \begin{cases} iso & \kappa = iso^{-} \\ trn & \kappa = trn^{-} \\ tag & \kappa = iso \\ box & \kappa = trn \\ \kappa & otherwise \end{cases}$$

Figure 4.14: Well-formed Aliasing

The capabilities *ref, val, box*, and *tag* permit the programmer to create infinitely many aliases, and they are all locally compatible with themselves. As a result, they all alias as themselves. Aliasing an *iso* results in a *tag*, because an *iso* guarantees it is the only non-opaque reference to the object in the entire program. Aliasing a *trn* gives an immutable reference, *box*, since *trn* guarantees to be the only mutable reference to the object in the entire program. Aliasing an ephemeral capability gives its stable counterpart.

The unaliasing operator - is defined in Figure 4.15.

$$\kappa- = \begin{cases} iso^{-} & \kappa = iso \\ trn^{-} & \kappa = trn \\ \kappa & otherwise \end{cases}$$

Figure 4.15: Well-formed Unaliasing

As described above, the capabilities *ref, val, box*, and *tag* allow the programmer to make as many aliases to an object as they want, and so removing a single reference to the object does not give us any stronger guarantees than the capability already does. Thus, unaliasing any of these capabilities results in the same capability. Unaliasing either *iso* or *trn* gives the ephemeral counterpart each capability, for reasons discussed in Section 2.8.5

4.3.3 Adaptations

Unlike previous models, we choose not to explicitly define the two capability adaptations. We note that Steed2016 defined a series of requirements that each viewpoint must observe to be judged to be correct, and then aimed to find potential solutions to each. We instead choose to omit the solutions to each adaptations, instead focusing on the requirement each present, and showing how these alone can be used make guarantees about the system. The solutions from Steed2016 are given in Appendix B.

First, we need a number of auxiliary definitions. These are shown in Figure 4.16. We also define local and global compatibility, in Tables 4.1 and 4.2 respectively.

$$\frac{\kappa \in \{\mathit{iso},\ \mathit{val},\ \mathit{tag}\}}{\mathit{Sendable}(\kappa)} \qquad \frac{\kappa \in \{\mathit{val},\ \mathit{box}\}}{\mathit{Immutable}(\kappa)}$$

Figure 4.16: Auxiliary Definitions

$\kappa \sim_l \kappa'$		κ'					
		iso	*trn*	*ref*	*val*	*box*	*tag*
κ	*iso*						✓
	trn					✓	✓
	ref			✓		✓	✓
	val				✓	✓	✓
	box		✓	✓	✓	✓	✓
	tag	✓	✓	✓	✓	✓	✓

Table 4.1: Local Compatibility

$\kappa \sim_g \kappa'$		κ'					
		iso	*trn*	*ref*	*val*	*box*	*tag*
κ	*iso*						✓
	trn						✓
	ref						✓
	val				✓	✓	✓
	box				✓	✓	✓
	tag	✓	✓	✓	✓	✓	✓

Table 4.2: Global Compatibility

Sendable and *Immutable* are simple definitions that will be useful later. Local and global compatibility define what capabilities can coexist as aliases to the same object. Local compatibility is between two paths to an object from the same actor, and global compatibility is two paths to the same object from different actors. Each follows from the definition of the individual capabilities.

We now have enough to define well-formedness for both viewpoint adaptation and extracting adaptation. Following the convention in Steed2016, we use λ to represent the capability the actor has of the object, and κ to represent the capability the object has of its field. Figure 4.17 gives the well-formedness rule for viewpoint adaptation.

$$\frac{\begin{array}{c} \forall \lambda', \lambda'', \lambda''', \kappa', \kappa'' \ .\ \lambda- \leqslant \lambda'' \wedge (\kappa \sim_l \kappa' \vee \kappa = \kappa') \implies (\\ (\mathit{Immutable}(\lambda) \vee \mathit{Immutable}(\kappa) \implies \mathit{Immutable}(\lambda \triangleright \kappa)) \wedge \\ (\kappa \sim_g \kappa' \implies (\lambda \triangleright \kappa)+ \sim_g \kappa') \wedge \\ ((\lambda \sim_l \lambda' \vee \lambda = \lambda' = \kappa'') \implies (\lambda \triangleright \kappa)+ \sim_l \lambda' \triangleright \kappa') \wedge \\ (\lambda \sim_g \lambda' \implies (\lambda'' \triangleright \kappa)+ \sim_g \lambda' \triangleright \kappa') \wedge \\ (\mathit{Sendable}(\lambda) \implies (\lambda \triangleright \kappa)+ \sim_g \lambda'' \triangleright \kappa')) \end{array}}{\lambda \triangleright \kappa\ :\ \mathtt{ok}}$$

Figure 4.17: Well-formed Viewpoint Adaptation

This definition of well-formedness maintains the 5 requirements for a correct viewpoint adaptation put forward in Steed2016.

Firstly, it ensures preservation of immutability; if either λ or κ are immutable, the resulting viewpoint must preserve this immutability.

The 2nd requirement is that the operator preserves global field compatibility; if κ is already globally compatible with κ', creating an alias of our viewpoint of κ must preserve that global compatibility.

The 3rd requirement preserves local compatibility; if we have an alternative path to the same field , aliasing our viewpoint of that field must preserve local compatibility with the alternative path.

Similar to the 2nd requirement, the 4th requirement of viewpoint adaptation is that it preserves object global compatibility; given two globally compatible references to an object, our viewpoint must be globally compatible with the path through the alternative reference.

Finally, we want to ensure that sendable capabilities behave correctly in the presence of subtyping; if we assume an alias is created through a subtype capability and then the original is sent to another actor, the two must be globally compatible.

Figure 4.18 gives the definition of well-formed extracting adaptation.

$$\frac{\begin{array}{c}\forall \lambda', \lambda'', \kappa', \kappa'' \;. \\ (\kappa \sim_g \kappa' \implies (\lambda \mathrel{\ooalign{$\triangleright$\cr$+$}} \kappa)+ \sim_g \kappa') \wedge \\ (((\lambda \sim_l \lambda' \vee \lambda = \lambda' = \kappa'') \wedge \kappa \sim_l \kappa') \implies (\lambda \mathrel{\ooalign{$\triangleright$\cr$+$}} \kappa)+ \sim_l (\lambda'-) \triangleright \kappa')\end{array}}{\lambda \mathrel{\ooalign{$\triangleright$\cr$+$}} \kappa \;:\; \texttt{ok}}$$

Figure 4.18: Well-formed Extracting Adaptation

The requirements for an extracting adaptation are much simpler. First we ensure global compatibility is preserved by saying that in the case where there are two globally compatible references to an object, a destructive read (the equivalent of aliasing the extracting adaptation) of one reference is still globally compatible with the other.

In addition to preserving global compatibility, we seek to ensure local compatibility. Assuming we have multiple, locally compatible paths to an object, we ensure that aliasing the extracting viewpoint is locally compatible with the remaining paths.

4.3.4 Recovery

Recovering a capability allows us to change the capability of a reference to one with stronger local guarantees. By interacting with the capability within a code block that only allows access to sendable variables, we can make guarantees about how the reference has been used within the block.

We can guarantee that no other aliases to a mutable capability will exist after leaving the recover block, and so we can recover any of *iso, trn, or ref* to an *iso*$^-$. A *box* can be safely recovered to a *val*, since it cannot be an alias of anything mutable by warrant of the fact that it is not compatible with *iso*, and we can reference any other mutable aliases within the recover block. *val* and *tag* both already have the strongest guarantees possible, and so recover to themselves.

The full definition of recovery is given in Figure 4.19.

$$
recover\ \kappa = \begin{cases} iso^- & \kappa \in \{iso^-, iso, trn^-, trn, ref\} \\ val & \kappa \in \{val, box\} \\ tag & \kappa = tag \end{cases}
$$

Figure 4.19: Well-formed Recovery

4.3.5 Safe-to-write

Having a mutable reference to an object does not guarantee to us that we can write any other reference into that object; for example, it would not be safe to write a *ref* variable into a *ref* field on an *iso* object, because we could send the *iso* to another actor, and then we would have two mutable aliases in two different actors, which violates global compatibility. We show the safe-to-write relation ($\downarrow$) in Table 4.3, which tells us when it is safe to write a reference with capability κ into an object with capability κ'.

$\kappa \downarrow \kappa'$		κ'					
		iso	*trn*	*ref*	*val*	*box*	*tag*
κ	iso^-	✓	✓	✓	✓	✓	✓
	iso	✓			✓		✓
	trn^-	✓	✓	✓	✓	✓	✓
	trn	✓	✓		✓		✓
	ref	✓	✓	✓	✓	✓	✓
	val						
	box						
	tag						

Table 4.3: Safe-to-write

4.3.6 Extension to Types

Each of the operations we have discussed can be extended to types by in each case unpacking the capability from the type identifier and applying the operation to the capability. Figure 4.20 shows this.

$$
\begin{aligned}
(D\ \kappa)+ &= D\ (\kappa)+ \\
(D\ \kappa)- &= D\ (\kappa)- \\
D\ (\kappa \triangleright \tau) &= \kappa \triangleright \tau \\
\kappa \triangleright D\ \kappa' &= D\ (\kappa \triangleright \kappa') \\
D\ (\kappa \mathrel{-\!\!\triangleright} \tau) &= \kappa \mathrel{-\!\!\triangleright} \tau \\
\kappa \mathrel{-\!\!\triangleright} D\ \kappa' &= D\ (\kappa \mathrel{-\!\!\triangleright} \kappa') \\
recover\ (D\ \kappa) &= D\ (recover\ \kappa) \\
\kappa \downarrow (D\ \kappa') &\iff \kappa \downarrow \kappa'
\end{aligned}
$$

Figure 4.20: Extending Capability Operators to Types

4.4 Well-Formedness

Here we present a number of updated and new rules for well-formedness. The introduction of capabilities renders a large number of the rules from the previous system inadequate, and requires us to augment them to achieve a system in which we will be able to prove soundness. Like previous sections, we have highlighted new additions and new rules.

Note we deliberately omit rules where the only changes are updating method syntax and/or adding capability constraints to generic parameters. The omitted rules are those for constraint entailment, class declarations, and well-formed methods.

Figure 4.21 presents a number of new look-up rules that will be useful later. These are not the only new look-up rules that are required, but they are the only interesting ones; the remaining look-up rules can be found in Appendix C.

L-CapCon
$$\frac{\mathit{Constraint}(\mathit{Prog}, C) = \texttt{constraint}\ C[\overline{\texttt{X}'\ v'},\ \texttt{X}\ v,\ \overline{\texttt{X}''\ v''}\ ...]\ ...}{\mathit{CapCon}(\mathit{Prog}, C, \texttt{X}) = v}$$

L-CapCon
$$\frac{\mathit{Class}(\mathit{Prog}, L) = \texttt{class}\ L[\overline{\texttt{X}'\ v'},\ \texttt{X}\ v,\ \overline{\texttt{X}''\ v''}\ ...]\ ...}{\mathit{CapCon}(\mathit{Prog}, L, \texttt{X}) = v}$$

L-CapCon
$$\frac{\mathit{Actor}(\mathit{Prog}, A) = \texttt{constraint}\ A[\overline{\texttt{X}'\ v'},\ \texttt{X}\ v,\ \overline{\texttt{X}''\ v''}\ ...]\ ...}{\mathit{CapCon}(\mathit{Prog}, A, \texttt{X}) = v}$$

Caps
$$\frac{}{\mathit{caps}(\Gamma, D[\overline{\tau}]\ \kappa) = \kappa}$$

Caps
$$\frac{\Gamma = \overline{\texttt{X}'\ v'},\ \texttt{X}\ v,\ \overline{\texttt{X}''\ v''} \qquad \overline{\kappa} = \{\kappa \mid \kappa \ll v\}}{\mathit{caps}(\Gamma, \texttt{X}) = \overline{\kappa}}$$

Caps
$$\frac{\mathit{caps}(\Gamma, \tau) = \overline{\kappa}}{\mathit{caps}(\Gamma, \tau+) = \overline{\kappa+}}$$

Caps
$$\frac{\mathit{caps}(\Gamma, \tau) = \overline{\kappa}}{\mathit{caps}(\Gamma, \tau-) = \overline{\kappa-}}$$

Caps
$$\frac{\mathit{caps}(\Gamma, \tau) = \overline{\kappa}}{\mathit{caps}(\Gamma, \mathit{recover}\ \tau) = \overline{\mathit{recover}\ \kappa}}$$

Caps
$$\frac{\overline{\kappa} = \{\kappa \triangleright \kappa' \mid \kappa \in \mathit{caps}(\Gamma, \texttt{V}), \kappa' \in \mathit{caps}(\Gamma, \tau)\}}{\mathit{caps}(\Gamma, \texttt{V} \triangleright \tau) = \overline{\kappa}}$$

Caps
$$\frac{\overline{\kappa} = \{\kappa \mathrel{+\!\!\triangleright} \kappa' \mid \kappa \in \mathit{caps}(\Gamma, \texttt{V}), \kappa' \in \mathit{caps}(\Gamma, \tau)\}}{\mathit{caps}(\Gamma, \texttt{V} \mathrel{+\!\!\triangleright} \tau) = \overline{\kappa}}$$

Caps
$$\frac{}{\mathit{caps}(\Gamma, \kappa) = \kappa}$$

Caps
$$\frac{\overline{\kappa} = \{\kappa \mid \kappa \ll \#\mathit{any}\}}{\mathit{caps}(\Gamma, \texttt{this}) = \overline{\kappa}}$$

Figure 4.21: Lookup Rules

The first new look-up rule we have created is `L-CapCon`, which enables us to get the capability constraint of a given type variable within a declaration with generic parameters, be it a class, a constraint, or an actor. This function is useful during witnessing, as we will describe later.

The second function, `Caps`, takes a type environment Γ and some type τ, and returns all possible capabilities that type could have. It is defined recursively over the structure of types, and applies the correct capability operations to generate the list of capabilities. For a declared type (either a class or an actor), it simply returns the associated capability that makes up part of the type. For aliased, unaliased, and recovered types, it recurses to find the capability list for the operand, and then applies the operator to the list in an element-wise fashion. For the binary viewpoint and extracting adaptations, it recurses to find lists for both operands, and then builds a complete list from every combination of arguments.

If τ is a type variable, it potentially could have many capabilities based on its constraint. In all cases when type checking a type expression involving a type variable, we want to ensure that the expression is well-formed for all possible values the variable could take. Thus, `Caps` returns all capabilities that comply with the type variable's capability constraint.

$$\textsc{S-Unpack}\quad \frac{\kappa \preccurlyeq \kappa' \qquad Prog \vdash L[\overline{\tau_1}] \leq L[\overline{\tau_2}]}{Prog \vdash L[\overline{\tau_1}]\ \kappa \leq L[\overline{\tau_2}]\ \kappa'}$$

Figure 4.22: Subtyping

Figure 4.22 extends the subtype relation from Genus- to incorporate capabilities in the same way we did when extending capability operators to work on types. We unpack the capabilities from each type and check the subcapability relation holds, and then we check subtyping on the declared type as in Genus-. Note the two operators are different, with $\preccurlyeq$ denoting subcapability and $\leq$ denoting subtyping.

$$\textsc{W-Class}\quad \frac{\begin{array}{c} Funcs(Prog, \Delta, C[\overline{\tau}]) \cup Behavs(Prog, \Delta, C[\overline{\tau}]) \\ \subseteq \bigcup_i Funcs(Prog, \Delta, \tau_i) \cup \bigcup_i Behavs(Prog, \Delta, \tau_i) \\ Constraint(Prog, C) = \texttt{constraint}\ C[\overline{\texttt{X}\ v}\ ...]\ ... \\ \forall i\ .\ \tau_i \ll CapCon(Prog, C, \texttt{X}_i) \end{array}}{Prog; \Delta \vdash \overline{\tau} :: C[\overline{\tau}]}$$

Figure 4.23: Witnessing

Figure 4.23 shows the new witness relation in DeGen. For a sequence of types to correctly witness a constraint, the capability of each type must comply with the capability constraint of its respective type variable. Additionally, actors can now be part of the sequence of types witnessing constraints, and so we check that union of the functions and behaviours defined on the constraint is a subset of the union of all functions and behaviours defined on the sequence of types.

$$\textsc{W-Prog}\quad \frac{\begin{array}{c} \forall C \in Constraints(Prog)\ .\ Prog \vdash C : \texttt{ok} \\ \forall L \in Classes(Prog)\ .\ Prog \vdash L : \texttt{ok} \\ \forall A \in Actors(Prog)\ .\ Prog \vdash A : \texttt{ok} \end{array}}{\vdash Prog : \texttt{ok}}$$

Figure 4.24: Well-formed Programs

$$\textsc{W-CDecl}\quad \dfrac{\begin{array}{c} \overline{\texttt{X}_2} \subseteq \overline{\texttt{X}_1} \qquad \overline{\texttt{X}_3} \subseteq \overline{\texttt{X}_1} \\ \forall i\ .\ Prog;\overline{\texttt{X}_1};\emptyset \vdash \overline{C_2[\overline{\texttt{X}_2}]}_i : \texttt{ok} \qquad Prog;\overline{\texttt{X}_1};\emptyset \vdash C_3[\overline{\texttt{X}_3}] : \texttt{ok} \\ \forall i\ .\ Prog;\ \overline{\texttt{X}_1};\overline{C_2[\overline{\texttt{X}_2}]}, C_3[\overline{\texttt{X}_3}] \vdash (\tau_1\ \tau_2.m[\beta](\overline{\tau_3}))_i : \texttt{ok} \qquad \forall i.\exists j\ .\ (\tau_2)_i = (\texttt{X}_1)_j \wedge (\texttt{X}_1)_j \notin \overline{\texttt{X}_2} \\ \forall i\ .\ Prog;\ \overline{\texttt{X}_1};\overline{C_2[\overline{\texttt{X}_2}]}, C_3[\overline{\texttt{X}_3}] \vdash (\tau_4.b[\beta'](\overline{\tau_5}))_i : \texttt{ok} \qquad \forall i.\exists j\ .\ (\tau_4)_i = (\texttt{X}_1)_j \wedge (\texttt{X}_1)_j \notin \overline{\texttt{X}_2} \end{array}}{Prog \vdash\ \texttt{constraint}\ C_1[\overline{\texttt{X}_1}\ \texttt{where}\ \overline{C_2[\overline{\texttt{X}_2}]}]\ \texttt{extends}\ C_3[\overline{\texttt{X}_3}]\ \{\overline{\tau_1\ \tau_2.m[\beta](\overline{\tau_3})}\ \ \overline{\tau_4.b[\beta'](\overline{\tau_5})}\ \} : \texttt{ok}}$$

Figure 4.25: Well-Formed Constraint Declarations

$$\textsc{W-ADecl}\quad \dfrac{\begin{array}{c} \forall i\ .\ Prog;\overline{\texttt{X}};\emptyset \vdash C[\overline{\tau_1}]_i : \texttt{ok} \qquad Prog;\overline{\texttt{X}};\overline{C[\overline{\tau}]} \vdash \tau_2 : \texttt{ok} \\ \forall i\ .\ Prog;\overline{\texttt{X}};\overline{C[\overline{\tau_1}]} \vdash (\tau_4.m[\beta_1](\overline{\tau_5}) : \tau_3\ \{\texttt{e}_1\})_i : \texttt{ok} \qquad \forall i\ .\ Prog;\overline{\texttt{X}};\overline{C[\overline{\tau_1}]} \vdash (\tau_4)_i : A[\overline{\texttt{X}}]\ \mathit{ref} \\ \forall i\ .\ Prog;\overline{\texttt{X}};\overline{C[\overline{\tau_1}]} \vdash (\tau_6.m[\beta_2](\overline{\tau_7})\ \{\texttt{e}_2\})_i : \texttt{ok} \qquad \forall i\ .\ Prog;\overline{\texttt{X}};\overline{C[\overline{\tau_1}]} \vdash (\tau_6)_i : A[\overline{\texttt{X}}]\ \mathit{ref} \end{array}}{Prog \vdash\ \texttt{actor}\ A[\overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C[\overline{\tau_1}]}]\ \{\overline{f\ \tau_2}\ \overline{\tau_4.m[\beta_1](\overline{\tau_5})\ :\ \tau_3\ \{\texttt{e}_1\}}\ \overline{\tau_6.b[\beta_2](\overline{\tau_7})\ \{\texttt{e}_2\}}\} : \texttt{ok}}$$

Figure 4.26: Well-Formed Actor Declarations

Figure 4.24 gives the additional requirement that all actor declarations to be correct for programs to be well-formed. Figure 4.25 shows the additional requirements on constraints relating to behaviours. These are very similar to the requirements on methods; the behaviour declaration is well-formed, the receiver is a type variable in the constraint's scope, and the receiver is not itself constrained.

Figure 4.26 shows the requirements for a well-formed actor declaration. Actors have generic parameters, a list of fields, a list of methods, and a list of behaviours, and the usual checks to these apply. The generic parameters must be well-formed, as must all the constraints that act upon them. The methods must be well-formed and their receivers must have the same type as the class, with the same requirement on behaviours.

$$\textsc{W-BehavSig}\quad \dfrac{\begin{array}{c} Prog;\Gamma;\Delta \vdash \tau_1 : \texttt{ok} \\ \forall i\ .\ Prog;\Gamma,\overline{\texttt{X}\ v};\Delta,\overline{C[\overline{\tau}]} \vdash (\tau_2)_i : \texttt{ok} \\ \forall i\ .\ Prog;\Gamma,\overline{\texttt{X}\ v};\Delta \vdash (C[\overline{\tau}])_i : \texttt{ok} \end{array}}{Prog;\Gamma;\Delta \vdash \tau_1.b[\overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_2})}$$

$$\textsc{W-BehavDecl}\quad \dfrac{\begin{array}{c} Prog;\Gamma;\Delta \vdash \tau_1.b[\overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_2}) : \texttt{ok} \\ Prog;\Gamma,\overline{\texttt{X}\ v};\Delta,\overline{C[\overline{\tau}]};\mathit{this} : \tau_1,\overline{x : \tau_2} : \texttt{e} : \tau_e \end{array}}{Prog;\Gamma;\Delta \vdash \tau_1.b[\overline{\texttt{X}\ v}\ \texttt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_2\ x})\{\texttt{e}\} : \texttt{ok}}$$

Figure 4.27: Well-Formed Behaviours

Figure 4.27 gives the requirements for well-formed behaviours. They are much the same as methods, with the exception of requiring a specific return type; behaviours are asynchronous and no value is returned, so while we need to know the behaviour body has a type, it is not required to be any specific one. As expected, we require that the receiver, parameters, and constraints are all also well-formed. We include two separate rules for signature and declaration, as we only need to check the signature when a behaviour is required by a constraint.

W-TL

$$\begin{array}{c} Class(Prog, L) = \mathtt{class}\ L[\overline{\mathtt{X}\ v}\ \mathtt{where}\ \overline{C[\overline{\tau_2}]}]\ \mathtt{extends}\ L_2[\overline{\tau_3}]\{...\} \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash (\tau_1)_i : \mathtt{ok} \qquad \#(\overline{\tau_1}) = \#(\overline{\mathtt{X}}) \\ Prog; \Gamma, \overline{\mathtt{X}}; \Delta \vdash L_2[\overline{\tau_3}\{\overline{\tau_1}/\overline{\mathtt{X}}\}] : \mathtt{ok} \qquad \forall i\ .\ Prog; \Gamma, \overline{\mathtt{X}}; \Delta \vdash C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i : \mathtt{ok} \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash (\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\})_i :: C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i \\ \forall i\ .\ \forall j\ .\ caps((\tau_1)_i)_j \ll v_i \\ \hline Prog; \Gamma; \Delta \vdash L[\overline{\tau_1}] : \mathtt{ok} \end{array}$$

W-TC

$$\begin{array}{c} Constraint(Prog, C) = \mathtt{constraint}\ C[\overline{\mathtt{X}\ v}\ \mathtt{where}\ \overline{C_2[\overline{\mathtt{Y}}]}]\ \mathtt{extends}\ C_3[\overline{\mathtt{Z}}]\{...\} \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash \tau_i : \mathtt{ok} \qquad \#(\overline{\tau}) = \#(\overline{\mathtt{X}}) \\ \forall i\ .\ Prog; \Gamma, \overline{\mathtt{X}}; \Delta \vdash C_2[\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\}]_i : \mathtt{ok} \qquad Prog; \Gamma, \overline{\mathtt{X}}; \Delta \vdash C_3[\overline{\mathtt{Z}}\{\overline{\tau}/\overline{\mathtt{X}}\}] : \mathtt{ok} \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash (\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\})_i :: C_2[\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\}]_i \\ \forall i\ .\ \forall j\ .\ caps(\tau_i)_j \ll v_i \\ \hline Prog; \Gamma; \Delta \vdash C[\overline{\tau}] : \mathtt{ok} \end{array}$$

W-TA

$$\begin{array}{c} Actor(Prog, A) = \mathtt{actor}\ A[\overline{\mathtt{X}\ v}\ \mathtt{where}\ \overline{C[\overline{\tau_2}]}]\ ... \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash (\tau_1)_i : \mathtt{ok} \qquad \#(\overline{\tau_1}) = \#(\overline{\mathtt{X}}) \\ \forall i\ .\ Prog; \Gamma, \overline{\mathtt{X}}; \Delta \vdash C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i : \mathtt{ok} \\ \forall i\ .\ Prog; \Gamma; \Delta \vdash (\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\})_i :: C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i \\ \forall i\ .\ \forall j\ .\ caps((\tau_1)_i)_j \ll v_i \\ \hline Prog; \Gamma; \Delta \vdash A[\overline{\tau_1}] : \mathtt{ok} \end{array}$$

W-TAlias

$$\begin{array}{c} Prog; \Gamma; \Delta \vdash \tau : \mathtt{ok} \\ \hline Prog; \Gamma; \Delta \vdash \tau+ : \mathtt{ok} \end{array}$$

W-TUnalias

$$\begin{array}{c} Prog; \Gamma; \Delta \vdash \tau : \mathtt{ok} \\ \hline Prog; \Gamma; \Delta \vdash \tau- : \mathtt{ok} \end{array}$$

W-TRecover

$$\begin{array}{c} Prog; \Gamma; \Delta \vdash \tau : \mathtt{ok} \\ \hline Prog; \Gamma; \Delta \vdash recover\ \tau : \mathtt{ok} \end{array}$$

W-TView

$$\begin{array}{c} Prog; \Gamma; \Delta \vdash \tau : \mathtt{ok} \\ \forall \kappa \in caps(\Gamma, \mathtt{V}), \kappa' \in caps(\Gamma, \tau)\ .\ \kappa \triangleright \kappa' : \mathtt{ok} \\ \hline Prog; \Gamma; \Delta \vdash \mathtt{V} \triangleright \tau : \mathtt{ok} \end{array}$$

W-TExt

$$\begin{array}{c} Prog; \Gamma; \Delta \vdash \tau : \mathtt{ok} \\ \forall \kappa \in caps(\mathtt{V}), \kappa' \in caps(\tau)\ .\ \kappa \Rrightarrow \kappa' : \mathtt{ok} \\ \hline Prog; \Gamma; \Delta \vdash \mathtt{V} \Rrightarrow \tau : \mathtt{ok} \end{array}$$

Figure 4.28: Well-Formed Types and Constraints

In Figure 4.28 we give an expanded set of rules for judging types and constraints in use-cases. Firstly, we present updated rules for classes and constraints with type arguments. In addition to witnessing the constraints on the type parameters they replace, the arguments must comply with the capability constraints of each type parameter. We then give the rules for actor types, which are nearly identical to that of classes, again checking that all type arguments are valid, the constraints remains well-formed and are witnessed correctly, and each argument complies with its respective capability constraints.

We then give a number of rules that check type expressions are well-formed. For the unary operators `+`, `-` and *recover*, this simply requires checking the operand is well-formed, as each operation is defined on all capabilities. For the two adaptations, we find all possible combinations of capabilities from both operands using the `Cap` function we defined in Figure 4.21, and then check every combination is well-formed using the rules described in Figures 4.17 and 4.18.

ALIAS
$$\frac{Prog;\Gamma;\Delta;E \vdash \texttt{e} : \tau}{Prog;\Gamma;\Delta;E \vdash_a \texttt{e} : \tau+}$$

FLD
$$\frac{Prog;\Gamma;\Delta;E \vdash e : D[\overline{\tau}]\ \kappa \qquad Field(Prog, D[\overline{\tau}], f) = \tau}{Prog;\Gamma;\Delta;E \vdash e.f : \kappa \triangleright \tau}$$

FLD-ASS
$$\frac{Prog;\Gamma;\Delta;E \vdash e_1 : D[\overline{\tau}]\ \kappa \qquad Prog;\Gamma;\Delta;E \vdash_a e_2 : \tau \qquad Field(Prog, D[\overline{\tau}], f) = \tau \qquad \kappa \downarrow \tau}{Prog;\Gamma;\Delta;E \vdash e_1.f = e_2 : \kappa \blacktriangleright \tau}$$

NEW-L
$$\frac{Prog;\Gamma \vdash L[\overline{\tau_1}]\ \kappa : \texttt{ok} \qquad \forall i\ .\ Prog;\Gamma;\Delta;E \vdash \texttt{e}_i : \tau_i \qquad \forall i\ .\ Field(Prog, L[\overline{\tau_1}], f_i) = \tau_i \qquad \forall i\ .\ \kappa \downarrow \tau_i \qquad \overline{f} = Fields(Prog, L)}{Prog;\Gamma;\Delta;E \vdash \texttt{new}\ L[\overline{\tau_1}](\overline{f = \texttt{e}})\ \kappa : L[\overline{\tau_1}]\ \kappa}$$

M-CALL
$$\frac{Prog;\Gamma;\Delta;E \vdash_a \texttt{e}_1 : \tau_1 \qquad \forall i\ .\ Prog;\Gamma \vdash (\tau_2)_i : \texttt{ok} \qquad Func(Prog, \Delta, \tau_1, m) = \tau_1.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau_5}]}](\overline{\tau_4}) : \tau_3 \qquad \forall i\ .\ Prog;\Gamma;\Delta;E \vdash_a (\texttt{e}_4)_i : (\tau_4\{\overline{\tau_2}/\overline{\texttt{X}}\})_i \qquad \forall i\ .\ Prog;\Gamma;\Delta \vdash (\overline{\tau_5}\{\overline{\tau_2}/\overline{\texttt{X}}\})_i :: C[\overline{\tau_5}\{\overline{\tau_2}/\overline{\texttt{X}}\}]_i}{Prog;\Gamma;\Delta;E \vdash \texttt{e}_1.m[\overline{\tau_2}](\overline{\texttt{e}_4}) : \tau_3}$$

NEW-A
$$\frac{Prog;\Gamma \vdash A[\overline{\tau}]\ tag : \texttt{ok}}{Prog;\Gamma;\Delta;E \vdash \texttt{new}\ A[\overline{\tau}]\ \kappa : A[\overline{\tau}]\ tag}$$

B-CALL
$$\frac{Prog;\Gamma;\Delta;E \vdash_a \texttt{e} : A[\overline{\tau_2}]\ tag \qquad \forall i\ .\ Prog;\Gamma \vdash (\tau_1)_i : \texttt{ok} \qquad Behav(Prog, \Delta, A[\overline{\tau_2}], b) = A[\overline{\tau}].b[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau_3}]}](\overline{\tau_4}) \qquad \forall i\ .\ Prog;\Gamma;\Delta;E \vdash_a \texttt{e}_i : (\tau_4\{\overline{\tau_2}/\overline{\texttt{X}}\})_i \qquad \forall i\ .\ Prog;\Gamma;\Delta \vdash (\overline{\tau_3}\{\overline{\tau_2}/\overline{\texttt{X}}\})_i :: C[\overline{\tau_3}\{\overline{\tau_2}/\overline{\texttt{X}}\}]_i}{Prog;\Gamma;\Delta;E \vdash \texttt{e}.b[\overline{\tau_2}](\overline{\texttt{e}}) : A[\overline{\tau_2}]\ tag}$$

RECOVER
$$\frac{Prog;\Gamma;\Delta;E\backslash\{x \mid \neg Sendable(E(x))\} \vdash \texttt{e} : \tau}{Prog;\Gamma;\Delta;E \vdash \texttt{recover}\ \ \texttt{e} : recover\ \tau}$$

NULL
$$\frac{D \in \mathsf{P} \qquad Prog;\Gamma;\Delta;E \vdash D[\overline{\tau}] : \texttt{ok}}{Prog;\Gamma;\Delta;E \vdash \texttt{null} : D[\overline{\tau}]\ iso^{-}}$$

Figure 4.29: Well-Formed Expressions

Finally, Figure 4.8 gives all the rules for typing expressions in DeGen.

We introduced a new judgement to aid us - the alias judgement, denoted $\vdash_a$. The alias judgement allows us to determine what type an expression will alias as by typing the expression, then applying the alias operator to the resulting type.

The type of a field access is now the type of the field, as viewed through the capability of the target expression. Thus we determine the type of the target expression and use its capability as the viewpoint for a viewpoint adaptation on the field type.

Field assignment now uses the extracting adaptation to determine the type of the assignee, using the target expression's capability as the viewpoint. Note that we use the alias judgement to determine the type of the assignee, and ensure that it is safe to write the type into the target capability.

Creating a new object now takes a capability as part of the expression. In addition to previous checks, we also check the type and its intended capability are jointly well-formed, and then check it is safe to write the type of each field expression into the object. The determined type, assuming these requirements hold, is the declared type with the expected capability.

Creating a new actor works differently to creating a new object. Since all actors view other actors as *tag*s, the type of the expression must have capability *tag*. Since *tag* references are opaque, we cannot write to the new actor's fields, and therefore do not allow instantiating them as part of this expression. Assuming the actor type with capability *tag* is well-formed, the `new` expression is also well-formed.

Typing a method call expression works very similarly to before. When calling a method, the receiver and arguments are all passed into the new frame used to evaluate the expression, which requires them to be aliased, and so we determine their types using the aliasing judgement.

Typing a behaviour is similar to typing a method call. The receiving expression must have tag capability, and as with a method call the receiver and arguments are aliased in the message sent to the receiving actor, and so we use the aliasing judgement to determine their types.

A recover expression ensures that the inner expression can be typed using only sendable variables from the value context. If this requirement holds, the recover expression has the same type as recovering the type of the inner expression.

A null expression can be given any valid type, with a capability of iso^{-}. This is because `null` represents any reference that is not pointing to an object on the heap, and so can take the type of any object that could possibly exist in the heap. The capability of iso^{-} aliases as *iso*,

4.5 Well-Formed Heaps

Now that we have introduced capabilities and actors, we can expand our ideas of well-formed heaps to more than just preservation of soundness. We can additionally make judgments about whether the execution of expressions preserves well-formed visibility, which allows us to check data-race freedom statically.

4.5.1 Soundness

Show how the previous proof of soundness is preserved in this system.

Introducing actors into DeGen altered the method of executing expressions, and so the previous operational semantics from Genus- are no longer valid. Each actor executes individually and sequentially, but sharing the same heap. Actors contain a message queue which the process sequentially, reading the message at the top of the queue and executing the behaviour body entirely before processing the next one. The only way for actors to interact is by sending messages (i.e. dispatching behaviours) to one another. Messages contain the name of the behaviour to execute, a sequence of values for arguments, and a runtime environment.

Capabilities are erased at runtime, and so don't impact the runtime specification or operational semantics. The runtime environment Ω still maps type variables to concrete types, and does not contain any information about capabilities. Runtime environments combine similarly to in Genus-; executing a behaviour uses the runtime environment of the message that caused the behaviour to be executed and the runtime environment of the actor. Executing methods declared in an actor works the same as previously, and methods dispatched to an object use the object's runtime environment in addition to that of the method and actor.

The runtime specification, though not discussed in this report, will largely resemble that of $Pony^{PL}$. Each rule from Genus- will be adapted to actor execution, and a number of new rules will be introduced for the new expressions. Additionally, a rule is required for actors reading behaviours from their message queue, and also a global rule that when an actor executes an expression, that actor can change and all others stay the same.

Since we envision a runtime specification similar to previous deny capability models, built on top of a sound generics system, we expect that DeGen is also sound. However, we do not give a proof of this in this report.

Our theorem of soundness for DeGen is the same as that for Genus-, given in Figure A.21, and we would prove soundness in a similar way. Capabilities are erased at runtime, and so don't impact the execution of expressions. However, DeGen introduces actors, so the method of executing expression has been altered.

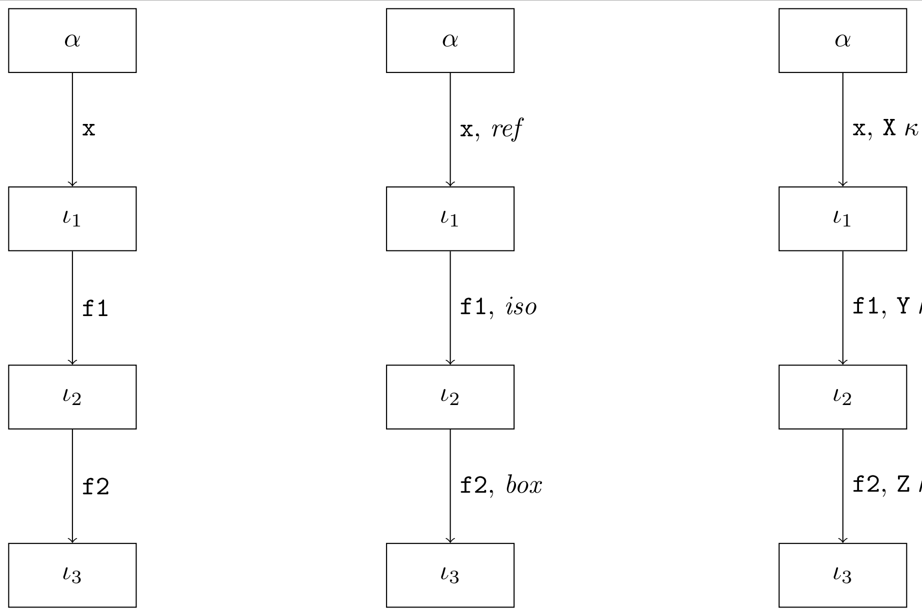
4.5.2 Well-Formed Visibility

The visibility of a path is the capability resulting from the series of field accesses and writes that form that path, and can be obtained by repeatedly applying the viewpoint and extracting adaptations. Given this notion of path visibilities, well-formed visibility states that all path visibilities are compatible in the sense that no data races can occur.

Well-formed visibility has been formally stated and proven for both the original Pony model and $Pony^{GS}$, but we just give an informal, intuitive reasoning here. Given that well-formed visibility deals only with capabilities, and our system uses the same formalisation of capabilities as $Pony^{GS}$, it follows that well-formed visibility holds for DeGen, but we do not explicitly prove this. Note, in the section below we are using the adaptation solutions from $Pony^{GS}$, given in Appendix B.

Visibility

We have already seen that a field access of the form `x.f` has capability $E(\texttt{x}) \triangleright \tau_f$, where τ_f is the field type as declared in the class. This is a slight simplification; the capability should actually be $\texttt{this} \triangleright E(\texttt{x}) \triangleright \tau_f$. However, all actors see themselves as *ref*, which does not change a capability when used as a viewpoint. A path `x.f1.f2` therefore has capability $(E(\texttt{x}) \triangleright \tau_{f1}) \triangleright \tau_{f2}$. This heap corresponds to the diagram in Figure 4.30a.



(a) Chained Field Accesses (b) Heap with Explicit Capabilities (c) Heap with Implicit Capabilities

If the variables and field have the capabilities shown in 4.30b, the resulting capability of the path `x.f1.f2` would be $\mathit{ref} \triangleright \mathit{iso} \triangleright \mathit{box}$, giving us a *tag*. However the path `(x.f1 = y).f2` would have the capability $\mathit{ref} \mathrel{-\!\!\triangleright} \mathit{iso} \triangleright \mathit{box}$, which is the same as *val*.

Due to the existence of type variables in DeGen, we may not have the capability explicitly available when trying to determine the visibility of a path. For the heap in Figure 4.30c, where the types of all references are symbolic, we can determine that the path `x.f1.f2` would have a visibility of $\chi(\alpha) \downarrow_2 (\texttt{X}) \triangleright \chi(\iota_1) \downarrow_2 (\texttt{Y}) \triangleright \chi(\iota_2) \downarrow_2 (\texttt{Z})$. Note that the runtime environments do not actually contain capability information, and we reference them here as something of a shorthand. When determining the visibility of type variable paths, effort must be taken to determine the resulting capability statically.

Given the heap and capabilities in 4.31, the path `x.f1.f` has capability *iso*, as does the path `x.f2.f`. These path visibilities are compatible, since *ref* allows multiple local mutable references, and there is only one *iso* to ι_1. In fact, this heap has well-formed visibility, so any pair of paths is compatible. Note also that the path `(x = y).f1.f2` has capability $\texttt{this} \mathrel{-\!\!\triangleright} \mathit{iso} \triangleright \mathit{ref} \triangleright \mathit{ref}$, giving us a visibility of iso^-, due to the fact that `this` always has capability *ref* in an actor.

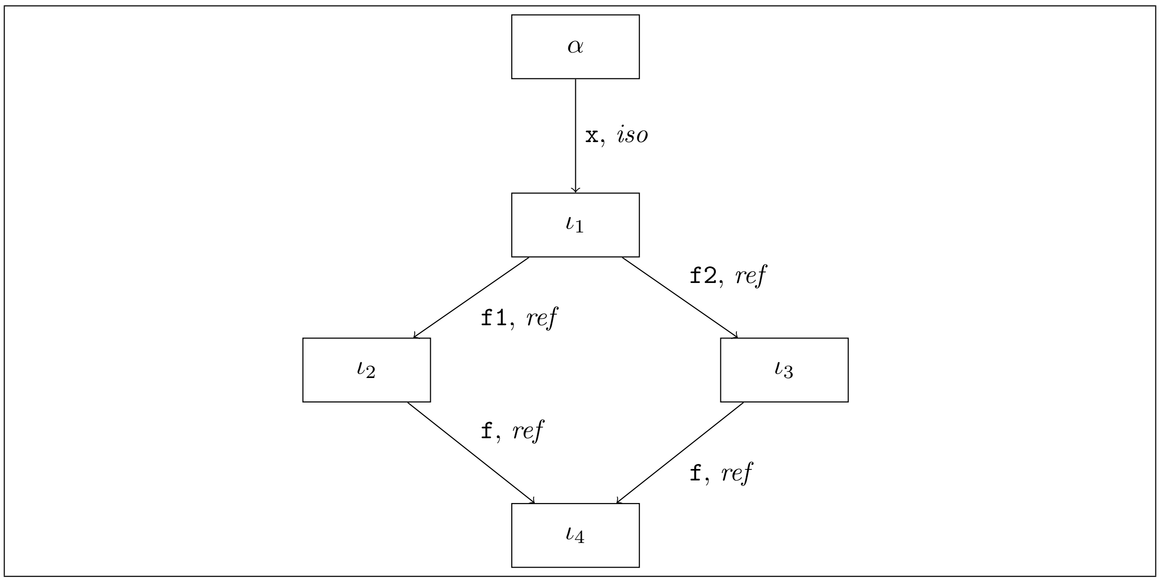


Figure 4.31: Caption

Well-formed visibility is a judgment on the heap, and so we require that all possible pairs of paths are compatible, with each permutation of field access, write, aliasing and unaliasing being valid and well-formed.

Proving Well-Formed Visibility

We would prove that well-formed visibility is preserved much like we would prove soundness, through structural induction over operational semantics; given a heap satisfying well-formed visibility, performing a single execution of a well-typed expression results in a new heap satisfying well-formed visibility, and the type of the expression. By proving it in this manner, we know that the execution of any valid DeGen program has well-formed visibility at all stages of execution, and by extension all well-formed programs are data-race free.

5. Evaluation and Conclusion

In this section we evaluate DeGen and Genus- relative to the two systems that inspired them, Pony and Genus, and discuss their merits as formal programming models. We discuss some aspects of future work that could be pursued.

5.1 In Relation to Pony

DeGen is a successor to $Pony^{PL}$, the formal model of how generics currently exist in Pony. DeGen is designed as an alternative to $Pony^{PL}$, showing how a new mechanic for generics can interact with deny capabilities. Since we designed our generic system first, and then applied deny capabilities to it, we have nearly no features that are unrelated to those two aspects. This means that $Pony^{PL}$ contains far more features taken from Pony, such as named constructors. It also contains a full runtime description, which DeGen lacks.

To evaluate how DeGen compares to Pony, we have taken a portion of the `List` class from the Pony Standard Library, and recreated it using DeGen. The original portion of `List` that we are considering is given in Appendix D. Evaluating DeGen in this manner serves a dual purpose; we gain an understanding of how well DeGen can represent Pony programs, and we can also see important features that DeGen lacks. To this end, we have highlighted where it has been necessary to enhance DeGen to accurately recreate the behaviour of `list`.

It is immediate to see that although DeGen represents quite a large subset of Pony features, some necessary mechanisms are still absent. In `index`, a function which gets the list element at a particular index, Pony uses type coercion to cast `node` to the desired return type. Viewpoint adaptation always returns a subtype of the operand type, and so here `this->ListNode[A]` is guaranteed to be compatible with `ListNode[A]`, the type of `_head`. As a result, the method would type check correctly in DeGen even without the cast, but we highlight it here as it is a significant feature that is missing from DeGen. Also missing are various programming constructs such as loops and if-statements, which are highlighted in this example.

The function `map` takes a lambda as a parameter and applies it to every element in a list. Obviously this is an advanced feature of the language, and is not something we would want to model at this stage, but it serves to demonstrate a level of functionality that DeGen cannot achieve; there is no way to create higher-order functions like `map` in DeGen. Additionally, it is also interesting to consider the capability the function should have. Since it is being applied in a *box* context, we would expect that it would likely also be a *box* function. The capabilities of its parameter and return types come from the type arguments for `A` and `B`, and the type operators that are applied to them.

The highlighted elements of Figure 5.1 suggest aspects of Pony that we would ideally like to have in DeGen. In addition to these, Appendix D demonstrates a number of features of Pony that are interesting, but unnecessary for developing the type system of Genus. There are numerous uses of '`?`' to indicate partial functions, another advanced language feature that allows Pony to handle errors in a succinct manner. For clarity, we have omitted all error handling from Figure 5.1. The syntax '`.>`' indicates method chaining, where calls to the same object can be combined for brevity. This allows Pony to have a much neater implementation of `pop`, which in DeGen requires a series of destructive reads to ensure the returned type aliases correctly.

5.2 In Relation to Genus

Genus-, and DeGen by extension, present a minimisation of Genus, and so are less expressive by design. However, through this minimisation, we have established a subset of Genus features that we know to be sound.

```
class List[A #any] {
  ListNode[A] _head;
  ListNode[A] _tail;
  int _size;

  ...

  box index(int i) : this->ListNode[A] {
    node = _head as this->ListNode[A] ;

    for (int i = 0; i < _size; i++) {
      node = node.next() as this->ListNode[A] ;
    }

    return node;
  }

  box head() : this->ListNode[A] {
    return _head as this->ListNode[A] ;
  }

  ref push(A a) : void {
    append_node(new ListNode[A](_item = a = null) ref);
  }

  ref pop() : A- {
    _size = _size - 1;
    t = _tail = null;
    _tail = t._prev = null;
    return _t = null;
  }

  box clone() : List[this->A+]- {
    out = new List[A](_head = null, _tail = null, _size = 0) ref;

    for (int i = 0; i < _size; i++) {
      out.push(this.index(i));
    }

    return out;
  }

  box map[B #any]({(this->A+) :  B-} func) : List[B]- {
    return _map[B](_head,
                   func,
                   new List[B](_head = null,
                               _tail = null,
                               _size = 0)
                   ref);
  }

  box _map[B #any](
      this->ListNode[A] last,
      {(this->A+) :  B-} func,
      List[B] acc) : List[B]- {
    acc.push(func(ln));
    if (ln.next() != null) {
      _map[B](ln.next() as this->ListNode[A], func, acc);
    } else {
      return acc;
    }
  }
  ...
}
```

Figure 5.1: `list` in DeGen

- Our model of Genus- is an imperative model, describing how the state is mutated by the execution of each expression. The formal definition of Genus is not imperative, and so our model is the first description of these behaviours.
- Genus does not have a proof of soundness. While the original authors expressed their belief that the system is sound, this project has shown for certain that our distilled formalization is sound.
- Genus- does not contain models as named constructs, instead only using the natural (structural) model of each type. While we have shown in Section 3.4 that our approach to removing models and simplifying Genus is generally valid, we do lose conciseness and some instances of expressiveness, such as expanders and models that take other models as parameters.
- Genus- as a programming language is far less expressive system than Genus; Genus, being built on top of Java, is a far more feature-rich system. It also introduces a number of features that do not relate to its mode of generics that increase expressiveness, full existential types being a prime example. We have justified excluding these, but it would be a nice-to-have feature if DeGen is to be developed into a fully fledged programming language.

5.3 Feature Set

Table 5.1 shows how DeGen compares to previous models of Pony and Genus in terms of features. While DeGen does lack a lot of mechanics common to many programming languages, it represents a large subset of relevant features from both Genus and Pony. We note that for both $Pony^{PL}$ and DeGen, soundness and data-race freedom are assumed, but not proven; $Pony^{PL}$ uses a translation to a non-generic base model, but this translation hasn't been proven sound, and DeGen assumes soundness from Genus- and data-race freedom from the formulation of capabilities in $Pony^{GS}$.

	$Pony^{PL}$	$Pony^{GS}$	Genus	DeGen
Sound	◐	●		◐
Data-Race Free (Well-Formed Visibility)	◐	●		◐
Actors	●	●		●
Capabilities	●	●		●
Destructive Read	●	●		●
Recovery	●	●		●
Union, Intersection, and Tuple types		●		
Existential Typing			●	
Generics	●		●	●
Multiparamter Constraints			●	●
Multiple Constraints	●		●	●
Natural Models			●	●
Parametric Models			●	
Retroactive Modelling			●	
Model Inheritance			●	

Table 5.1: Feature Comparison for $Pony^{PL}$, $Pony^{GS}$, Genus, and DeGen

5.4 Design Choices

Throughout the process of minimising Genus and designing DeGen, we made a number of choices that impacted the resulting systems. Here we discuss the choices we made, why we decided to make them, and what impact those choices had.

5.4.1 From Genus...

From the very beginning, our intention when designing Genus- was to create as simple a system as possible that had the same generic mechanism as Genus, while also maintaining enough expressiveness to allow us to reintroduce deny capabilities.

We opted to create an imperative model rather than just a standalone type system. With our intention being to prove well-formed visibility, a judgement on the heap, when we reintroduced capabilities, we knew the model would need a runtime specification eventually. Upon reflection, it was not necessary to introduce the runtime system while we still only had Genus-; Genus itself is not an imperative model, and a large part of the early stages on this project focused on creating this imperative model and proving its soundness. Had we instead focused on cherry-picking the aspects of Genus we wanted and then introduced the runtime specification when we introduce capabilities, we may have had time to fully develop the imperative model of DeGen.

A significant first step to our simplification of Genus was the decision to remove interfaces from the language entirely. Interfaces are used in examples in [1], but the formal specification does not reference them at all. Our intuition told us they were unnecessary, due to the fact that interfaces in most programming languages and constraints in Genus play nearly equivalent roles. We created a translation, described in 3.3 that showed this to be true, and that any interface declaration can be replaced by an equivalent constraint. However, there was a slight oversight in our reasoning that is demonstrated in Figure 4.6; unlike interfaces in other languages, we do not allow constraints to be used as types, and so they cannot be used as type arguments. This affects the case where we wish to apply binary functions to two different types which obey the same constraint.

Once we decided we were going to remove interfaces, the next step was deciding how we would replace the behaviour they represent; we knew from our translation that we could replace interfaces as constraints, but it required us to augment the behaviour of constraints in a non-intuitive way. Interfaces such as `Set[A]` can only be represented by constraints in a meaningful way if we allowed the type variables that constraints act on to themselves be constrained. However, for constraints to properly act as predicates, constrained type variables can't be used as receivers. Section 3.2 gives an approach to witnessing that enabled us to reason about how we can manifest these requirements, with parametric constraints becoming either functions returning constraints (in the manner that parametric classes are commonly thought of), or multivariate predicates that impose conditions on their type variables. We opted to keep a single list of type variables for constraints and more complicated type rules, but upon reflection, it would have been more desirable to slightly simplify the type system, even at the cost of a messier syntax.

The most significant decision we made concerning the minimisation of Genus was to remove models. Models supply a large part of the expressiveness of Genus, and so we carefully considered the implications of removing them; by the formulation in Section 3.2, we knew the natural model of each type was sufficient to witness a constraint, and so we could . However, since a significant motivation for reformulating generics using Genus was reducing programmer burden, we wanted to ensure that removing models did not greatly increase the complexity of using Genus. Section 3.4 represents a principled analysis of the features that we lose as a result of removing models, and how we can replace that behaviour. Not all behaviour can be recreated succinctly, such as parametric models, but there is not a significant increase in overall complexity.

We initially started developing Genus- with existentially qualified types, but chose to remove them based on the complexity they added to the type system. Genus expands the functionality of use-site genericity far beyond what Java wildcards are capable of. However, the complexity of existential packing and capture conversion that existential type require outweighed their usefulness. Section 3.1.4 provides justification for their removal with reference to the equivalence between universal and existential types.

5.4.2 ...to DeGen

After establishing Genus- as a sound base upon which we could further develop a more complex type system, the next step was to design how we would incorporate deny capabilities into Genus-. Since we intended DeGen to be an illustrative exploration of an alternate system of generics to the one that currently exists in Pony, we knew that DeGen must have the same system of concurrency as Pony, that of actors and messaging. Again for simplicity, we chose to model the smallest subset of features that accurately represent the deny capability system. We therefore don't introduce any features of Pony that do not relate to either actor-based execution or well-formed visibility.

We undertook a design process to understand how capabilities interact with type variables in Pony by recreating the behaviour of a number of Genus- constraints, which we detail in Section 4.1. As a result of this principled analysis, which allowed us to reason about and understand the required capabilities of a number of constraints, we developed an idea of how capabilities should interact with constraints and type variables.

The aspect of the design that required the most consideration was how we apply capabilities to constraints. We discovered that the simplest solution was also the most intuitive; constraints act as predicates on types that define the expected behaviour of those types. By extending this idea, it becomes clear that constraints can be used not only to say what a type can do, but also what we can do to the type. Type variables, therefore, should define what the program is permitted to do to a reference of that type. For example, the constraint `Box[B, T]` as defined in Figure 4.6 requires that a witness must have `get` and `set` methods, and that its type argument `T` is immutable. We also allow capability constraints to be placed on the 'self' type variables. This design choice allows the programmer to require that the witnesses of certain constraints are, for example, sendable, if that was the intention of the constraint.

6. Conclusion

This project has shown that deny capabilities and constraint-based generics can be combined into a succinct formal specification that is likely sound and data-race free. This system alleviates the programmer burden caused by f-bounded polymorphism and empowers them to use generics in a simple, intuitive way.

Over the course of the project, we have focused our efforts on ensuring the systems we are building are well designed. We have justified each design decision, and they have resulted in two concise, simple models. Being guided by this principled design approach allowed us to iterate quickly on design ideas, and, in nearly all cases, know quickly when we were pursuing ineffective avenues of design.

6.1 Future Work

DeGen is currently a very simple system, and so there are many possible avenues of future work, many of which we discussed in the previous chapter. We mention just some of these here:

- Expand DeGen into a full formal model, including runtime specification and operational semantics, and show that it preserves both soundness and well-formed visibility. Once these properties have been proven, we will have established DeGen as a valid proof of concept for constraint bounded generics in an actor-based language.
- Both Pony and Genus support features that DeGen does not. A key aspect to developing DeGen further would be to reintroduce some of these features in order to add greater expressiveness and power to the system. For example, adding Genus models as named constructs to DeGen would allow us to fully utilise the expressive power of using constraints and models in conjunction.
- A DeGen compiler would allow DeGen to form the basis for future programming language implementations, whether it be a new major version of Pony supporting constraint generics, or a spiritual successor to Pony.

Bibliography

[1] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*. ACM Press, 2015.

[2] Benjamin C Pierce. *Types and programming languages*. MIT Press, Cambridge, Mass., 2002.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, dec 1985.

[4] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA 89*. ACM Press, 1989.

[5] Bjarne Stroustrup. Bjarne stroustrup's c++ glossary. http://www.stroustrup.com/glossary.html, October 2012. Accessed 13/01/2019.

[6] Andrew Kennedy and Benjamin Pierce. Abstract on decidability of nominal subtyping with variance. *ACM*, 12 2008.

[7] Findbugs. http://findbugs.sourceforge.net/. Accessed 12/04/19.

[8] Ana Bove and Peter Dybjer. Dependent types at work. In *Language Engineering and Rigorous Software Development*, pages 57–99. Springer Berlin Heidelberg, 2009.

[9] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *ACM SIGPLAN Notices*, 49(10):233–249, October 2014.

[10] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, nov 1994.

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

[12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA 12*. ACM Press, 2012.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] John Boyland, James Noble, and William Retert. Capabilities for sharing. In *ECOOP 2001 — Object-Oriented Programming*, pages 2–27. Springer Berlin Heidelberg, 2001.

[15] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008 – Object-Oriented Programming*, pages 104–128. Springer Berlin Heidelberg, 2008.

[16] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE! 2015*. ACM Press, 2015.

[17] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe types for topology and encapsulation. In *Formal Methods for Components and Objects*, pages 72–112. Springer Berlin Heidelberg, 2008.

[18] George Steed and Sophia Drossopoulou. A principled design of capabilities in pony. Master's thesis, Imperial College London, 2016.

[19] What is pony? https://www.ponylang.io/discover. Accessed 23/01/2019.

[20] Pony tutorial. tutorial.ponylang.io. Accessed 23/01/2019.

[21] Paul Liétar and Sophia Drossopoulou. Formalizing generics for pony. Master's thesis, Imperial College London, 2017.

A. Genus- Model

A.1 Syntax

Programs	$\texttt{P} ::= \overline{\texttt{CO}}\ \overline{\texttt{CL}}$
Constraints	$\texttt{CO} ::= \texttt{constraint}\ C_1[\overline{\texttt{X}}\ \texttt{where}\ \overline{C_2[\overline{\texttt{Y}}]}]\ \texttt{extends}\ C_3[\overline{\texttt{Z}}]\ \{\overline{\texttt{methSig}}\}$
Classes	$\texttt{CL} ::= \texttt{class}\ L_1[\beta]\ \texttt{extends}\ L_2[\overline{\tau}]\ \{\overline{\texttt{fld}}\ \overline{\texttt{meth}}\}$
Method Signature	$\texttt{methSig} ::= \tau_1\ \tau_2.m[\beta](\overline{\tau_3})$
Method Declaration	$\texttt{meth} ::= \texttt{methSig}\ \{\texttt{e}\}$
Field Declaration	$\texttt{fld} ::= \tau\ f$
Generic Parameters	$\beta ::= \overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau}]}$
Types	$\tau ::= \texttt{X} \mid L[\overline{\tau}]$
Expressions	$\texttt{e} ::= x \mid \texttt{e}.f \mid \texttt{e}.f\ =\ \texttt{e} \mid \texttt{this} \mid \texttt{new}\ L[\overline{\tau}](\overline{f = \texttt{e}}) \mid \texttt{e}.m[\overline{\tau_2}](\overline{\texttt{e}})$
Type Environments	$\Gamma ::= \overline{\texttt{X}}$
Constraint Environments	$\Delta ::= \overline{C[\overline{\texttt{X}}]}$
Value Environments	$E ::= \emptyset \mid E, x : \tau \mid E, \texttt{this} : \tau$

Constraint Names	C	*Class Names*	L
Method Names	m	*Field Names*	f
Variable Names	x	*Type Variables*	$\texttt{X}$

Figure A.1: Syntax

A.2 Interface to Constraint Translation

$$
\begin{array}{lr}
\textit{Interface Declaration} & \mathtt{I} ::= \mathtt{interface}\ I\langle \mathtt{X}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}\rangle\{\overline{\dot{\tau}_1\ m[\beta](\overline{\dot{\tau}_2})}\} \\
\textit{Programs} & \mathtt{P} ::= \overline{\mathtt{CO}}\ \overline{\mathtt{CL}}\ \overline{\mathtt{I}} \\
\textit{Interface Types} & \dot{\tau} ::= \tau \mid I\langle\overline{\dot{\tau}}\rangle \\
\textit{Method Signature} & \mathtt{methSig} ::= \dot{\tau}_1\ \tau_2.m[\beta](\overline{\dot{\tau}_3})
\end{array}
$$

Figure A.2: Interface Syntax

$$
\begin{array}{l}
\textit{expand} :: I\langle\overline{\tau}\rangle \times \Gamma\ \rightarrow\ \overline{\mathtt{X}} \times \overline{C[\overline{\tau}]} \\
\textit{expand}(I\langle\overline{\tau}\rangle, \Gamma) = (\mathtt{U},\ C_I[\mathtt{U}, \overline{\tau}])\quad \mathtt{where\ U\ new\ in}\ \Gamma \\
\textit{expand}(I\langle\overline{\dot{\tau}}\rangle, \Gamma) = (\mathtt{U} : \overline{\mathtt{U}'} : \overline{us},\ C_I[\mathtt{U}, \overline{\mathtt{U}'}] : \overline{cs}) \\
\qquad \mathtt{where\ U\ new\ in}\ \Gamma \wedge \forall i\ .\ \textit{expand}(\dot{\tau}_i, \Gamma : \mathtt{U}) = (\mathtt{U}'_i : us_i,\ cs_i) \\
\qquad us :: \overline{\mathtt{X}},\ cs :: \overline{C[\overline{\tau}]}
\end{array}
$$

Figure A.3: Definition of *expand* function

T-Prog

$$
\frac{\begin{array}{c}\forall i\ .\ \mathtt{CO}_i \rightsquigarrow^T \mathtt{CO}'_i \\ \forall j\ .\ \mathtt{CL}_j \rightsquigarrow^T \mathtt{CL}'_j \\ \forall k\ .\ \mathtt{I}_k \rightsquigarrow^T \mathtt{CO}'_{i+k}\end{array}}{\mathtt{P} \rightsquigarrow^T \overline{\mathtt{CO}'}\ \overline{\mathtt{CL}'}}
$$

T-Interface

$$
\frac{\begin{array}{c} I \rightsquigarrow^T C_I \\ \mathtt{constraint}\ C_I[\mathtt{S}, \overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}]\{\overline{\dot{\tau}_1\ \mathtt{S}.m[\beta](\overline{\dot{\tau}_3})}\} \rightsquigarrow^T c\end{array}}{\mathtt{interface}\ I\langle\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}\rangle\{\overline{\dot{\tau}_1\ m[\beta](\overline{\dot{\tau}_2})}\} \rightsquigarrow^T c}
$$

T-Ident

$$
\frac{C_I = \text{`}c\_\text{`} + I}{I \rightsquigarrow^T C_I}
$$

T-Meth

$$
\frac{\forall i\ .\ \textit{expand}(I\langle\overline{\dot{\tau}}\rangle_i, \Gamma : \overline{\mathtt{X}}) = (\mathtt{U}_i : us_i, cs_i)}{\tau_1\ \tau_2.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}](\overline{\tau_3}, \overline{I\langle\overline{\tau}\rangle}), \Gamma \rightsquigarrow^T \tau_1\ \tau_2.m[\overline{\mathtt{X}}, \overline{\mathtt{U}}, \overline{us}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}, \overline{cs}](\overline{\tau_3}, \overline{\mathtt{U}})}
$$

T-Constraint

$$
\frac{\begin{array}{c}\forall i\ .\ \textit{expand}(I\langle\overline{\dot{\tau}}\rangle_i, \overline{\mathtt{X}}) = (\mathtt{U}_i : us_i, cs_i) \\ \forall i\ .\ (\mathtt{U}_i\ (\tau_2.m[\beta](\overline{\dot{\tau}_3}))_i, \overline{\mathtt{X}} : \overline{\mathtt{U}} : \overline{us} \rightsquigarrow^T \mathtt{meth}_i) \\ \forall j\ .\ ((\tau_4\ \tau_3.m[\beta](\overline{\dot{\tau}_6}))_j, \overline{\mathtt{X}} : \overline{\mathtt{U}} : \overline{us} \rightsquigarrow^T \mathtt{meth}_{i+j})\end{array}}{\begin{array}{c}\mathtt{constraint}\ C_1[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}]\ \mathtt{extends}\ C_2[\overline{\mathtt{Z}}]\ \{\overline{I\langle\overline{\dot{\tau}}\rangle\ \tau_2.m[\beta](\overline{\dot{\tau}_3})}, \overline{\tau_4\ \tau_5.m[\beta](\overline{\dot{\tau}_6})}\} \rightsquigarrow^T \\ \mathtt{constraint}\ C_1[\overline{\mathtt{X}}, \overline{\mathtt{U}}, \overline{us}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}, \overline{cs}]\ \mathtt{extends}\ C_2[\overline{\mathtt{Z}}]\ \{\overline{\mathtt{meth}}\}\end{array}}
$$

T-Class

$$
\frac{\begin{array}{c}\forall i\ .\ \textit{expand}(I\langle\overline{\dot{\tau}}\rangle_i, \overline{\mathtt{X}}) = (\mathtt{U}_i : us_i, cs_i) \\ \forall i\ .\ (\mathtt{U}_i\ (\tau_2.m[\beta](\overline{\dot{\tau}_3})\ \{\mathtt{e}\})_i, \overline{\mathtt{X}} : \overline{\mathtt{U}} : \overline{us} \rightsquigarrow^T \mathtt{meth}_i) \\ \forall j\ .\ ((\tau_4\ \tau_3.m[\beta](\overline{\dot{\tau}_6})\ \{\mathtt{e}\})_j, \overline{\mathtt{X}} : \overline{\mathtt{U}} : \overline{us} \rightsquigarrow^T \mathtt{meth}_{i+j})\end{array}}{\begin{array}{c}\mathtt{class}\ L_1[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}]\ \mathtt{extends}\ L_2[\overline{\tau}]\ \{\overline{\mathtt{fld}},\ \overline{I\langle\overline{\dot{\tau}}\rangle\ \tau_2.m[\beta](\overline{\dot{\tau}_3})\ \{\mathtt{e}\}}, \overline{\tau_4\ \tau_5.m[\beta](\overline{\dot{\tau}_6})\ \{\mathtt{e}\}}\} \rightsquigarrow^T \\ \mathtt{class}\ L_1[\overline{\mathtt{X}}, \overline{\mathtt{U}}, \overline{us}\ \mathtt{where}\ \overline{C[\overline{\mathtt{Y}}]}, \overline{cs}]\ \mathtt{extends}\ L_2[\overline{\tau}]\ \{\overline{\mathtt{fld}},\ \overline{\mathtt{meth}}\}\end{array}}
$$

T-MethDecl

$$
\frac{\mathtt{meth}, \Gamma \rightsquigarrow \mathtt{meth}'}{\mathtt{meth}\ \{\mathtt{e}\}, \Gamma \rightsquigarrow \mathtt{meth}'\ \{\mathtt{e}\}}
$$

Figure A.4: Translation Rules

A.3 Well-formed Programs

L-Cons

$$\frac{\mathit{Prog} = \overline{\texttt{CO}}\ \overline{\texttt{CL}}}{\mathit{Constraints}(\mathit{Prog}) = \overline{\texttt{CO}}}$$

L-Classes

$$\frac{\mathit{Prog} = \overline{\texttt{CO}}\ \overline{\texttt{CL}}}{\mathit{Constraints}(\mathit{Prog}) = \overline{\texttt{CL}}}$$

L-Con

$$\frac{\mathit{Constraints}(\mathit{Prog})_i = \texttt{constraint}\ C\ ...\ \{...\}}{\mathit{Constraint}(\mathit{Prog}, C) = \mathit{Constraints}(\mathit{Prog})_i}$$

L-Class

$$\frac{\mathit{Classes}(\mathit{Prog})_i = \texttt{class}\ L\ ...\ \{...\}}{\mathit{Class}(\mathit{Prog}, L) = \mathit{Classes}(\mathit{Prog})_i}$$

L-F-L

$$\frac{\mathit{Class}(\mathit{Prog}, L) = \texttt{class}\ L[\overline{\texttt{X}}\ ...]\ ...\ \{...\ \tau\ f\ ...\}}{\mathit{Field}(\mathit{Prog}, L[\overline{\tau}], f) = \tau\{\overline{\tau}/\overline{\texttt{X}}\}}$$

L-Fs-L

$$\frac{\mathit{Class}(\mathit{Prog}, L) = \texttt{class}\ L\ ...\ \{\overline{\tau\ f}\ ...\}}{\mathit{Fields}(\mathit{Prog}, L) = \overline{f}}$$

L-Funcs-Con

$$\frac{\begin{array}{c}\mathit{Constraint}(\mathit{Prog}, C) = \texttt{constraint}\ C[\overline{\texttt{X}}\ ...]\ \texttt{extends}\ C'[...]\ \{\overline{\texttt{methSig}}\} \\ \mathit{Funcs}(\mathit{Prog}, C') = \overline{m}\end{array}}{\mathit{Funcs}(\mathit{Prog}, \Delta, C[\overline{\tau}]) = \overline{\texttt{methSig}}\{\overline{\tau}/\overline{\texttt{X}}\}, \overline{m}\{\overline{\tau}/\overline{\texttt{X}}\}}$$

L-Funcs-Class

$$\frac{\begin{array}{c}\mathit{Class}(\mathit{Prog}, L) = \texttt{class}\ L[\overline{\texttt{X}}\ ...]\ \texttt{extends}\ L'[...]\ \{...\ \overline{\texttt{methSig}\ \{\texttt{e}\}}\} \\ \mathit{Funcs}(\mathit{Prog}, L') = \overline{m}\end{array}}{\mathit{Funcs}(\mathit{Prog}, \Delta, L[\overline{\tau}]) = \overline{\texttt{methSig}}\{\overline{\tau}/\overline{\texttt{X}}\}, \overline{m}\{\overline{\tau}/\overline{\texttt{X}}\}}$$

L-Funcs-Var

$$\frac{\begin{array}{c}\overline{C} = \{C | C[\overline{\texttt{X'}}, \texttt{X}, \overline{\texttt{X''}}] \in \Delta\} \\ \overline{\texttt{mSig}} = \{\texttt{mSig}\ |\ \exists C \in \overline{C}, \tau, \overline{\tau}\ .\ \texttt{mSig} \in \mathit{Func}(\mathit{Prog}, \Delta, C) \wedge \texttt{mSig} = \tau\ \texttt{X}.m(\overline{\tau})\}\end{array}}{\mathit{Funcs}(\mathit{Prog}, \Delta, \texttt{X}) = \overline{\texttt{mSig}}}$$

L-Func

$$\frac{\begin{array}{c}\mathit{Funcs}(\mathit{Prog}, \Delta, \tau) = \overline{m} \\ m_i = \tau_1\ \tau_2.m[\beta](\overline{\tau_3})\end{array}}{\mathit{Func}(\mathit{Prog}, \Delta, \tau, m) = m_i}$$

L-M-Body

$$\frac{\mathit{Class}(\mathit{Prog}, L) = \texttt{class}\ L[...]\ ...\ \{...\ \tau'\ \tau''.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''\ x})\{\texttt{e}\}\ ...\}}{\mathit{Meth}(\mathit{Prog}, L, m) = \tau'\ \tau''.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''\ x})\{\texttt{e}\}}$$

Figure A.5: Lookup Functions

S-Refl

$$\frac{}{\mathit{Prog} \vdash \tau \leq \tau}$$

S-Trans

$$\frac{\mathit{Prog} \vdash \tau_1 \leq \tau_2 \qquad \mathit{Prog} \vdash \tau_2 \leq \tau_3}{\mathit{Prog} \vdash \tau_1 \leq \tau_3}$$

S-Subclass

$$\frac{\mathit{Class}(\mathit{Prog}, L_1) = \texttt{class}\ L_1[\overline{\texttt{X}}]\ \texttt{extends}\ L_2[\overline{\tau_2}]\ \{...\}}{\mathit{Prog} \vdash L_1[\overline{\tau_1}] \leq L_2[\overline{\tau_2}\{\overline{\tau_1}/\overline{\texttt{X}}\}]}$$

S-Object

$$\frac{}{\mathit{Prog}; \Gamma \vdash \tau \leq \mathit{Object}}$$

Figure A.6: Subtyping Rules

E-Prereq

$$\frac{\mathit{Constraint}(\mathit{Prog}, C_1) = \texttt{constraint}\ C_1[\overline{\texttt{X}_1}]\ \texttt{extends}\ C_2[\overline{\texttt{X}_2}]\ \{...\}}{\mathit{Prog} \vdash C_1[\overline{\tau}] \preceq C_2[\overline{\texttt{X}_2}\{\overline{\tau}/\overline{\texttt{X}_1}\}]}$$

E-Trans

$$\frac{\mathit{Prog} \vdash C_1[\overline{\tau_1}] \preceq C_2[\overline{\tau_2}] \qquad \mathit{Prog} \vdash C_1[\overline{\tau_2}] \preceq C_2[\overline{\tau_3}]}{\mathit{Prog} \vdash C_1[\overline{\tau_1}] \preceq C_2[\overline{\tau_3}]}$$

E-Reflex

$$\frac{}{\mathit{Prog} \vdash C[\overline{\tau}] \preceq C[\overline{\tau}]}$$

Figure A.7: Constraint Entailment

W-Subsume

$$\frac{\mathit{Prog}; \Delta \vdash \overline{\tau} :: C_1[\overline{\tau_1}] \qquad \mathit{Prog} \vdash C_1[\overline{\tau_1}] \preceq C_2[\overline{\tau_2}]}{\mathit{Prog}; \Delta \vdash \overline{\tau} :: C_2[\overline{\tau_2}]}$$

W-Class

$$\frac{\mathit{Funcs}(\mathit{Prog}, \Delta, C[\overline{\tau}]) \subseteq \bigcup_i \mathit{Funcs}(\mathit{Prog}, \Delta, \tau_i)}{\mathit{Prog}; \Delta \vdash \overline{\tau} :: C[\overline{\tau}]}$$

Figure A.8: Witness Relation

W-Prog

$$\frac{\begin{array}{c}\forall C \in \mathit{Constraints}(\mathit{Prog})\ .\ \mathit{Prog} \vdash C : \texttt{ok} \\ \forall L \in \mathit{Classes}(\mathit{Prog})\ .\ \mathit{Prog} \vdash L : \texttt{ok}\end{array}}{\vdash \mathit{Prog} : \texttt{ok}}$$

Figure A.9: Well-formed Programs

W-CDecl

$$\frac{\begin{array}{c}\overline{\texttt{X}_2} \subseteq \overline{\texttt{X}_1} \qquad \overline{\texttt{X}_3} \subseteq \overline{\texttt{X}_1} \\ \forall i\ .\ \mathit{Prog}; \overline{\texttt{X}_1}; \emptyset \vdash C_2[\overline{\texttt{X}_2}]_i : \texttt{ok} \qquad \mathit{Prog}; \overline{\texttt{X}_1}; \emptyset \vdash C_3[\overline{\texttt{X}_3}] : \texttt{ok} \\ \forall i\ .\ \mathit{Prog};\ \overline{\texttt{X}_1}; \overline{C_2[\overline{\texttt{X}_2}]}, C_3[\overline{\texttt{X}_3}] \vdash (\tau_1\ \tau_2.m[\beta](\overline{\tau_3}))_i : \texttt{ok} \qquad \forall i. \exists j\ .\ (\tau_2)_i = (\texttt{X}_1)_j \wedge (\texttt{X}_1)_j \notin \overline{\texttt{X}_2}\end{array}}{\mathit{Prog} \vdash\ \texttt{constraint}\ C_1[\overline{\texttt{X}_1}\ \texttt{where}\ \overline{C_2[\overline{\texttt{X}_2}]}]\ \texttt{extends}\ C_3[\overline{\texttt{X}_3}]\ \{\overline{\tau_1\ \tau_2.m[\beta](\overline{\tau_3})}\} : \texttt{ok}}$$

Figure A.10: Well-Formed Constraint Declarations

W-LDecl

$$\frac{\begin{array}{c}\forall i\ .\ \mathit{Prog}; \overline{\texttt{X}}; \emptyset \vdash C[\overline{\tau_1}]_i : \texttt{ok} \qquad \mathit{Prog}; \overline{\texttt{X}}; \emptyset \vdash L[\overline{\tau_2}] : \texttt{ok} \qquad \forall i\ .\ \mathit{Prog}; \overline{\texttt{X}}; \overline{C[\overline{\tau_1}]} \vdash (\tau_f)_i : \texttt{ok} \\ \forall i\ .\ \mathit{Prog}; \overline{\texttt{X}}; \overline{C[\overline{\tau_1}]} \vdash (\tau_3\ \tau_4.m[\beta](\overline{\tau_5})\{\texttt{e}\})_i : \texttt{ok} \qquad \forall i\ .\ \mathit{Prog}; \overline{\texttt{X}}; \overline{C[\overline{\tau_1}]} \vdash (\tau_4)_i : L_1[\overline{\texttt{X}}] \\ (\exists j\ .\ \mathit{Funcs}(\mathit{Prog}, L_2)_j = \tau_3'\ \tau_4'.m[\beta](\overline{\tau_5'})) \implies \tau_3 = \tau_3' \wedge \forall k\ .\ (\tau_5)_k = (\tau_5')_k\end{array}}{\mathit{Prog} \vdash\ \texttt{class}\ L_1[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau_1}]}]\ \texttt{extends}\ L_2[\overline{\tau_2}]\{\overline{\tau_f\ f}\ \overline{\tau_3\ \tau_4.m[\beta](\overline{\tau_5})\{\texttt{e}\}}\} : \texttt{ok}}$$

Figure A.11: Well-Formed Class Declarations

W-MethSig

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta \vdash \tau_1 : \mathtt{ok} \qquad \mathit{Prog};\Gamma;\Delta \vdash \tau_2 : \mathtt{ok} \\ \forall i \;.\; \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta,\overline{C[\overline{\tau}]} \vdash (\tau_3)_i : \mathtt{ok} \qquad \forall i \;.\; \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta \vdash (C[\overline{\tau}])_i : \mathtt{ok} \end{array}}{\mathit{Prog};\Gamma;\Delta \vdash \tau_1\ \tau_2.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_3}) : \mathtt{ok}}$$

W-MethDecl

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta \vdash \tau_1\ \tau_2.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_3}) : \mathtt{ok} \\ \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta,\overline{C[\overline{\tau}]};\mathtt{this} : \tau_2,\overline{x : \tau_3} \vdash \mathtt{e} : \tau_1 \end{array}}{\mathit{Prog};\Gamma;\Delta \vdash \tau_1\ \tau_2.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau}]}](\overline{\tau_3\ x})\ \{\mathtt{e}\} : \mathtt{ok}}$$

Figure A.12: Well-Formed Methods

W-TVar

$$\dfrac{\mathtt{X} \in \Gamma}{\mathit{Prog};\Gamma;\Delta \vdash \mathtt{X} : \mathtt{ok}}$$

W-TL

$$\dfrac{\begin{array}{c} \mathit{Class}(\mathit{Prog},L) = \mathtt{class}\ L[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau_2}]}]\ \mathtt{extends}\ L_2[\overline{\tau_3}]\{...\} \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta \vdash (\tau_1)_i : \mathtt{ok} \qquad \#(\overline{\tau_1}) = \#(\overline{\mathtt{X}}) \\ \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta \vdash L_2[\overline{\tau_3}\{\overline{\tau_1}/\overline{\mathtt{X}}\}] : \mathtt{ok} \qquad \forall i \;.\; \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta \vdash C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i : \mathtt{ok} \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta \vdash (\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\})_i :: C[\overline{\tau_2}\{\overline{\tau_1}/\overline{\mathtt{X}}\}]_i \end{array}}{\mathit{Prog};\Gamma;\Delta \vdash L[\overline{\tau_1}] : \mathtt{ok}}$$

W-TC

$$\dfrac{\begin{array}{c} \mathit{Constraint}(\mathit{Prog},C) = \mathtt{constraint}\ C[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C_2[\overline{\mathtt{Y}}]}]\ \mathtt{extends}\ C_3[\overline{\mathtt{Z}}]\{...\} \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta \vdash \tau_i : \mathtt{ok} \qquad \#(\overline{\tau}) = \#(\overline{\mathtt{X}}) \\ \forall i \;.\; \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta \vdash C_2[\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\}]_i : \mathtt{ok} \qquad \mathit{Prog};\Gamma,\overline{\mathtt{X}};\Delta \vdash C_3[\overline{\mathtt{Z}}\{\overline{\tau}/\overline{\mathtt{X}}\}] : \mathtt{ok} \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta \vdash (\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\})_i :: C_2[\overline{\mathtt{Y}}\{\overline{\tau}/\overline{\mathtt{X}}\}]_i \end{array}}{\mathit{Prog};\Gamma;\Delta \vdash C[\overline{\tau}] : \mathtt{ok}}$$

Figure A.13: Well-Formed Types and Constraints

VAR-X

$$\dfrac{}{\mathit{Prog};\Gamma;\Delta;E \vdash x : E(x)}$$

VAR-THIS

$$\dfrac{}{\mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{this} : E(\mathtt{this})}$$

FLD

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta;E \vdash e : L[\overline{\tau_1}] \\ \mathit{Class}(\mathit{Prog},L) = \mathtt{class}\ L[\overline{\mathtt{X}}]\ ...\ \{...\ \tau_2\ f\ ...\} \end{array}}{\mathit{Prog};\Gamma;\Delta;E \vdash e.f : \tau_2\{\overline{\tau_1}/\overline{\mathtt{X}}\}}$$

FLD-ASS

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta;E \vdash e_1.f : \tau \\ \mathit{Prog};\Gamma;\Delta;E, \vdash e_2 : \tau \end{array}}{\mathit{Prog};\Gamma;\Delta;E \vdash e_1.f = e_2 : \tau}$$

NEW

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma \vdash L[\overline{\tau_1}] : \mathtt{ok} \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{e}_i : \mathit{Field}(\mathit{Prog},L[\overline{\tau_1}],f_i) \\ \overline{f} = \mathit{Fields}(\mathit{Prog},L) \end{array}}{\mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{new}\ L[\overline{\tau_1}](\overline{f = \mathtt{e}}) : L[\overline{\tau_1}]}$$

M-CALL

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{e}_1 : \tau_1 \qquad \forall i \;.\; \mathit{Prog};\Gamma \vdash (\tau_2)_i : \mathtt{ok} \\ \mathit{Func}(\mathit{Prog},\Delta,\tau_1,m) = \tau_3\ \tau_1.m[\overline{\mathtt{X}}\ \mathtt{where}\ \overline{C[\overline{\tau_5}]}](\overline{\tau_4}) \\ \forall i \;.\; \mathit{Prog};\Gamma;\Delta;E \vdash (\mathtt{e}_4)_i : (\tau_4\{\overline{\tau_2}/\overline{\mathtt{X}}\})_i \\ \forall i \;.\; \mathit{Prog};\Delta \vdash (\overline{\tau_5}\{\overline{\tau_2}/\overline{\mathtt{X}}\})_i :: C[\overline{\tau_5}\{\overline{\tau_2}/\overline{\mathtt{X}}\}]_i \end{array}}{\mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{e}_1.m[\overline{\tau_2}](\overline{\mathtt{e}_4}) : \tau_3}$$

Subsumption

$$\dfrac{\begin{array}{c} \mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{e} : \tau_1 \\ \mathit{Prog} \vdash \tau_1 \leq \tau_2 \end{array}}{\mathit{Prog};\Gamma;\Delta;E \vdash \mathtt{e} : \tau_2}$$

Figure A.14: Well-Formed Expressions

A.4 Runtime Specification

$$
\begin{array}{llll}
\chi & \in & \mathit{Heap} & = \mathit{Addr} \rightarrow \mathit{Object} \\
\phi & \in & \mathit{StackFrame} & = \mathit{Addr} \times \Omega \times (\mathit{VarId} \mapsto \mathit{Value}) \\
\iota & \in & \mathit{Addr} & \\
v & \in & \mathit{Value} & = \mathit{Addr} \\
 & & \mathit{Object} & = \mathit{ClassId} \times \Omega \times (\mathit{FieldId} \mapsto \mathit{Value}) \\
\Omega & \in & \mathit{RuntimeEnv} & = \texttt{X} \mapsto \sigma \\
\sigma & \in & \mathit{ConcreteType} & = L[\overline{\sigma}]
\end{array}
$$

Figure A.15: Runtime Entities

$$
\begin{array}{l}
\mathit{resolve} :: \tau \times \Omega \rightarrow \sigma \\
\mathit{resolve}(\texttt{X}, \Omega) = \Omega(\texttt{X}) \\
\mathit{resolve}(L[\overline{\tau}], \Omega) = L[\overline{\mathit{resolve}(\tau)}]
\end{array}
$$

Figure A.16: Resolve Function

$$
\textsc{val}\quad \frac{v \in \mathit{Value}}{v, \phi, \chi \rightsquigarrow_P v, \chi}
\qquad
\textsc{this}\quad \frac{}{\texttt{this}, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P \iota, \chi}
\qquad
\textsc{var}\quad \frac{\texttt{vMap}(x) = v}{x, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P v, \chi}
$$

$$
\textsc{fld}\quad \frac{\texttt{e}, \phi, \chi \rightsquigarrow_P \iota, \chi'}{\texttt{e}.f, \phi, \chi \rightsquigarrow_P \chi'(\iota, f), \chi'}
\qquad
\textsc{fldAss}\quad \frac{\begin{array}{c}\texttt{e}, \phi, \chi \rightsquigarrow_P \iota, \chi'' \\ \texttt{e}', \phi, \chi'' \rightsquigarrow_P v, \chi''' \\ \chi' = \chi'''[\iota, f \mapsto v]\end{array}}{\texttt{e}.f = \texttt{e}', \phi, \chi \rightsquigarrow_P v, \chi'}
$$

$$
\textsc{new}\quad \frac{\begin{array}{c}\texttt{e}_i, \phi, \chi_i \rightsquigarrow_P v_i, \chi_{i+1} \quad (i = 0, ..., n-1) \\ \iota' \texttt{ new in } \chi_n \\ \mathit{Class}(\mathit{Prog}, L) = \texttt{class } [\overline{\texttt{X}}] \; ... \\ \chi' = \chi_n[\iota' \mapsto (L, (\overline{\texttt{X} \mapsto \mathit{resolve}(\tau, \Omega)}), (\overline{f \mapsto v}))]\end{array}}{\texttt{new } L[\overline{\tau}](f_0 = \texttt{e}_0, ..., f_{n-1} = \texttt{e}_{n-1}), (\iota, \Omega, \texttt{vMap}), \chi_0 \rightsquigarrow_P \iota, \chi'}
$$

$$
\textsc{mCall}\quad \frac{\begin{array}{c}\texttt{e}, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P \iota', \chi_0 \\ \chi_0(\iota') = (L, \Omega', \_) \\ \texttt{e}_i, \phi, \chi_i \rightsquigarrow_P v_i, \chi_{i+1} \quad (i = 0, ..., k-1) \\ \tau' \; \overline{\tau''.m[\overline{\texttt{X}}](\overline{\tau''' \; x})\{\texttt{e}'\}} = \mathit{Meth}(\mathit{Prog}, L, m) \\ \texttt{e}', (\iota', (\overline{\texttt{X} \mapsto \mathit{resolve}(\tau, \Omega)} \cup \Omega'), (\overline{x \mapsto v}), \chi_k \rightsquigarrow_P v, \chi'\end{array}}{\texttt{e}.m[\tau_1, ..., \tau_n](\texttt{e}_1, ..., \texttt{e}_k), (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P v, \chi'}
$$

Figure A.17: Operational Semantics

A.5 Soundness

$$\textsc{S-Concrete}\quad \frac{Class(Prog, L_1) = \texttt{class}\ L_1[\overline{\texttt{X}}]\ \texttt{extends}\ L_2[\overline{\tau}]\{...\}}{Prog \vdash L_1[\overline{\sigma_1}] \leq L_2[\overline{\sigma_2}\{\overline{\sigma_1}/\overline{\texttt{X}}\}]}$$

$$\textsc{S-Concrete-Trans}\quad \frac{Prog \vdash \sigma_1 \leq \sigma_2 \qquad Prog \vdash \sigma_2 \leq \sigma_3}{Prog \vdash \sigma_1 \leq \sigma_3}$$

$$\textsc{S-Concrete-Refl}\quad \frac{}{Prog \vdash \sigma \leq \sigma}$$

Figure A.18: Extension of Rules to Concrete Types

$$\textsc{A-Null}\quad \frac{Class(Prog, L) \neq \bot}{Prog, \chi \vdash \texttt{null} <: L[\overline{\sigma}]}$$

$$\textsc{A-Addr}\quad \frac{\chi(\iota) = (L, \Omega, \_) \qquad Class(Prog, L) = \texttt{class}\ L[\overline{\texttt{X}}]\ ... \qquad \sigma \equiv resolve(L[\overline{\texttt{X}}], \Omega)}{Prog, \chi \vdash \iota <: \sigma}$$

$$\textsc{A-Sub}\quad \frac{Prog, \chi \vdash v <: \sigma' \qquad Prog \vdash \sigma' \leq \sigma}{Prog, \chi \vdash v <: \sigma}$$

$$\textsc{SA-Null}\quad \frac{Prog, \chi \vdash \texttt{null} <: \sigma}{Prog, \chi \vdash \texttt{null} \lhd \sigma}$$

$$\textsc{SA-Addr}\quad \frac{\begin{array}{c}\chi(\iota) = (L, \Omega, \_) \qquad Prog, \chi \vdash \iota <: \sigma \qquad Class(Prog, L) = \texttt{class}\ L[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau}]}]\ ... \\ \forall f\ .\ Field(Prog, L, f) = \tau \implies Prog, \chi \vdash \chi(\iota) \downarrow_3 (f) \lhd resolve(\tau, \Omega) \\ \forall i\ .\ Prog; \emptyset \vdash \overline{resolve(\tau, \Omega)} :: C[\overline{resolve(\tau, \Omega)}]_i\end{array}}{Prog, \chi \vdash \iota \lhd \sigma}$$

Figure A.19: Agreement

$$\frac{\begin{array}{c}Prog, \chi \vdash \iota \lhd resolve(E(\texttt{this}), \Omega) \\ \forall x \in dom(\texttt{vMap})\ .\ Prog, \chi \vdash \texttt{vMap}(x) \lhd resolve(E(x), \Omega) \\ \forall \iota' \in dom(\chi)\ .\ (\chi(\iota') = (L, \Omega', \_) \wedge Class(Prog, L) = \texttt{class}\ L[\overline{\texttt{X}}] \\ \implies Prog, \chi \vdash \iota' \lhd L[\overline{resolve(\texttt{X}, \Omega')}]) \\ \forall i\ .\ C[\overline{\tau}]_i = \Delta_i \implies Prog; \Delta \vdash \overline{resolve(\tau, \Omega)} :: \overline{C[\overline{resolve(\tau, \Omega)}]}\end{array}}{Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi \diamond}$$

Figure A.20: Well-formed Heap and Stack

$$\begin{array}{l}\forall\ Prog, \Gamma, \Delta, E, \texttt{e}, \chi, \iota, \Omega, \texttt{vMap}, \tau\ .\ (\ \vdash Prog : \texttt{ok} \wedge Prog; \Gamma; \Delta; E \vdash \texttt{e} : \tau \\ \qquad\qquad \wedge\ Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi\ \diamond \wedge\ \texttt{e}, (\iota, \Omega, \texttt{vMap}), \chi \rightsquigarrow_P v, \chi' \\ \qquad\qquad\qquad \implies Prog, \chi' \vdash v \lhd resolve(\tau, \Omega) \wedge Prog; \Delta; E \vdash (\iota, \Omega, \texttt{vMap}), \chi' \diamond)\end{array}$$

Figure A.21: Theorem of Soundness

B. Adaptation Solutions

$\kappa \triangleright \kappa'$		κ'					
		iso	*trn*	*ref*	*val*	*box*	*tag*
κ	iso^-	iso^-	iso^-	iso^-	*val*	*val*	*tag*
	iso	*iso*	*iso*	*iso*	*val*	*tag*	*tag*
	trn^-	iso^-	trn^-	trn^-	*val*	*val*	*tag*
	trn	*iso*	*trn*	*trn*	*val*	*box*	*tag*
	ref	*iso*	*trn*	*ref*	*val*	*box*	*tag*
	val	*val*	*val*	*val*	*val*	*val*	*tag*
	box	*tag*	*box*	*box*	*val*	*box*	*tag*
	tag	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table B.1: Solution for Viewpoint Adaptation
Reproduced from Steed2016

$\kappa \mathrel{\triangleright\!\!\!\!-} \kappa'$		κ'					
		iso	*trn*	*ref*	*val*	*box*	*tag*
κ	iso^-	iso^-	iso^-	iso^-	*val*	*val*	*tag*
	iso	iso^-	*val*	*tag*	*val*	*tag*	*tag*
	trn^-	iso^-	trn^-	trn^-	*val*	*val*	*tag*
	trn	iso^-	*val*	*box*	*val*	*box*	*tag*
	ref	iso^-	trn^-	*ref*	*val*	*box*	*tag*
	val	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
	box	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
	tag	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table B.2: Optimal Solution for Extracting Adaptation
Reproduced from Steed2016

C. DeGen Look-Up Rules

$$\textsc{L-Actors}\quad \frac{Prog = \overline{\texttt{CO}}\ \overline{\texttt{CL}}\ \overline{\texttt{A}}}{Actors(Prog) = \overline{\texttt{A}}} \qquad \textsc{L-Actor}\quad \frac{Actors(Prog)_i = \texttt{actor}\ A\ ...}{Actor(Prog, A) = Actors(Prog)_i}$$

$$\textsc{L-F-A}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A[\overline{\texttt{X}}\ ...]\ ...\{...\ \tau\ f\ ...\}}{Field(Prog, A[\overline{\tau}], f) = \tau\{\overline{\tau}/\overline{\texttt{X}}\}} \qquad \textsc{L-Fs-A}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A\ ...\ \{\overline{\tau\ f}\ ...\}}{Fields(Prog, A) = \overline{f}}$$

$$\textsc{L-Funcs-Actor}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A[...]\{...\ \overline{\texttt{methSig}\ \{\texttt{e}\}}\ ...\}}{Funcs(Prog, A) = \overline{\texttt{methSig}}, \overline{m}}$$

$$\textsc{L-M-Body}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A[...]\ ...\ \{...\ \tau'\ \tau''.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''\ x})\{\texttt{e}\}\ ...\}}{Meth(Prog, A, m) = \tau'\ \tau''.m[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau'''}]}](\overline{\tau''''\ x})\{\texttt{e}\}}$$

$$\textsc{L-Behavs-Con}\quad \frac{\begin{array}{c} Constraint(Prog, C) = \texttt{constraint}\ C[\overline{\texttt{X}}\ ...]\ \texttt{extends}\ C'[...]\ \{...\ \overline{\texttt{behavSig}}\} \\ Behavs(Prog, \Delta, C') = \overline{b} \end{array}}{Behavs(Prog, \Delta, C) = \overline{\texttt{behavSig}}\{\overline{\tau}/\overline{\texttt{X}}\}, \overline{b}\{\overline{\tau}/\overline{\texttt{X}}\}}$$

$$\textsc{L-Behavs-Class}\quad \frac{}{Behavs(Prog, \Delta, L[\overline{\tau}] = \emptyset} \qquad \textsc{L-Behavs-Actor}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A[\overline{\texttt{X}}\ ...]\{...\ \overline{\texttt{behaveSig}\{\texttt{e}\}}\}}{Behavs(Prog, \Delta, A[\overline{\tau}]) = \overline{\texttt{behaveSig}}\{\overline{\tau}/\overline{\texttt{X}}\}}$$

$$\textsc{L-Behavs-Var}\quad \frac{\begin{array}{c} \overline{C} = \{C \mid C[\overline{\texttt{X}'}, \texttt{X}, \overline{\texttt{X''}}] \in \Delta\} \\ \overline{\texttt{bSig}} = \{\texttt{bSig} \mid \exists C \in \overline{C}, \overline{\tau}\ .\ \texttt{bSig} \in Behavs(Prog, \Delta, C) \land \texttt{bSig} = \texttt{X}.m(\overline{\tau})\} \end{array}}{Behavs(Prog, \Delta, \texttt{X}) = \overline{\texttt{bSig}}}$$

$$\textsc{L-Behav}\quad \frac{\begin{array}{c} Behavs(Prog, \Delta, \tau) = \overline{b} \\ b_i = \tau_1.b[\beta](\overline{\tau_2}) \end{array}}{Behav(Prog, \Delta, \tau, b) = b_i}$$

$$\textsc{L-B-Body}\quad \frac{Actor(Prog, A) = \texttt{actor}\ A[...]\ ...\ \{...\ \tau'.b[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau''}]}](\overline{\tau'''\ x})\{\texttt{e}\}\ ...\}}{BehavBody(Prog, A, b) = \tau'.b[\overline{\texttt{X}}\ \texttt{where}\ \overline{C[\overline{\tau''}]}](\overline{\tau'''\ x})\{\texttt{e}\}}$$

Figure C.1: DeGen Look-Up Rules

D. Pony List

```
class List[A] is Seq[A]

  ...

  new create(len: USize = 0) =>
    None

  fun index(i: USize): this->ListNode[A] ? =>
    if i >= _size then
      error
    end

    var node = _head as this->ListNode[A]
    var j = USize(0)

    while j < i do
      node = node.next() as this->ListNode[A]
      j = j + 1
    end

    node

  fun head(): this->ListNode[A] ? =>
    _head as this->ListNode[A]

  fun ref push(a: A) =>
    append_node(ListNode[A](consume a))

  fun ref pop(): A^ ? =>
    tail()? .> remove().pop()?

  fun clone(): List[this->A!]^ =>
    let out = List[this->A!]

    for v in values() do
      out.push(v)
    end
    out

  fun map[B](f: {(this->A!): B^} box): List[B]^ =>
    try
      _map[B](head()?, f, List[B])
    else
      List[B]
    end

  fun _map[B](
    ln: this->ListNode[A],
    f: {(this->A!): B^} box,
    acc: List[B])
    : List[B]^
  =>
    try acc.push(f(ln()?)) end

    try
      _map[B](ln.next() as this->ListNode[A], f, acc)
    else
      acc
    end

    ...
end
```