Imperial College London

Department of Computing

Formalizing Generics for Pony

Author:
Paul Liétar

Supervised by:
Prof. Sophia Drossopoulou

June 2017

Abstract

Concurrent programming is generally error-prone, as care must be taken to prevent two concurrent tasks from interfering with each other's data. Pony is a new language designed for concurrent programming, based on the actor model. It's type system is designed to statically prevent data-races, by only allowing two actors to share data if neither can write to it.

Formal models for programming languages can help us better understand how they work, and increase our confidence in the guarantees provided by the language. As Pony makes strong guarantees about data-race freedom, having a model of the language has been an important part of its design. However, the existing models of the language have omitted a number of important features, most notably generics, in an attempt to reduce the complexity of these initial models.

Generics are an essential feature of modern programming languages, which allow code to be written once and reused in various contexts, applied to different types. Generics are both a powerful but complicated feature, which interacts closely with other parts of the language. In particular, Pony introduces a number of novel concepts, whose interaction with generics had not been studied carefully before.

We present $Pony^{\mathrm{PL}}$, a formalisation of the Pony language with support for generics. Our model is based on existing formalisations of the non-generic features of the Pony language. In many occasions, these existing models modify values' types to reflect operations on these values. With the introduction of generics however, types may be variables which are only replaced with non-variable types later. We introduce symbolic type operators, which encode the modification to a type, without requiring the final instantiation of type variables to be known. We also redefine a number of relations on types such that they can handle the introduced type variables and symbolic operators. We do so using partial reification, which allows us to reuse their original definitions with little change required. Finally we define a translation from $Pony^{\mathrm{PL}}$ to a non-generic version of our model, $Pony^{0}$.

While developing our model we've uncovered a number of bugs in the original design of generics or in their implementation in the Pony compiler. Many of these bugs cause unsoundness, violating the guarantees promised by the language, and could lead to data-races. We have worked closely with the authors of the compiler, fixing some of these bugs ourselves or providing suggestions on how they could be fixed.

Acknowledgements

First and foremost, I would like to thank Sophia Drossopoulou, who has been supervising my work on this project. She has provided me with extremely useful and detailed feedback on my work, through very frequent and lengthy meetings. She has also forced me to write for others to read rather for myself, and without her this report would have been nothing more than an inconsistent accumulation of personal notes.

I would also like to thank Sylvan Clebsch and the rest of the team currently working on the Pony compiler, Benoit, Joe, Sean, Theo and others. They have been very helpful in helping me learn the language as I was starting to work on this project. As I was working on the model, our weekly group phone calls have been incredibly useful in developing a better understanding of the inner details of the language, and the reasons which have led to the current design. As I uncovered various issues with the current design, our conversations were essential in finding solutions to fix these.

Finally I would like to thank my friends and family, who have been an infaillible support throughout my three years at Imperial and this project.

Contents

Chapter 1

Introduction

1.1 Motivation

While performance of computers has been rapidly improving since they have first been invented, the last decade has seen the speed of single threaded execution get closer and closer to an upper limit. Instead, modern processors have been embedding an increasing number of cores. Applications must be designed with concurrency in mind in order to take full advantage of this newly available power.

However, concurrent applications are a lot more challenging to design than traditional applications, as care must be taken to prevent two concurrent operations from interfering with each other's data, as it could lead to data-races. Many mechanisms which are widely used to prevent these data-races, such as mutual exclusion, have proven to be suboptimal. They are very error-prone as they expose programmers to many new risks, such as deadlocking. Others are safer and simpler to use, but come at the cost of runtime performance penalty.

Pony is a new high performance actor based language, which provides a natural but safe way of writing applications which are highly concurrent and can therefore take full advantage of the parallelism available on modern computers. However, unlike existing similar solutions, Pony enforces data-race freedom at compile time through a system of *reference capability.* Each reference to an object has an associated capability, which determines how the reference may be used. We describe capabilities in more details in Section 2.2.6.

In order for the language to be better defined and understood, and to verify the guarantees it makes, a first model of the Pony language was developed by [Clebsch et al., 2015]. We refer to this model as $\mathit{Pony}^{\mathrm{SC}}$. While it covers the core parts of the language, most importantly the reference capability system, it omits a large number of features which were considered less essential. This model was later improved upon by [Steed, 2016], which introduces a more principled approach to defining the capability system, as well as expands it to cover a larger number of features from the language. We refer to this model as $\mathit{Pony}^{\mathrm{GS}}$.

However, even the improved model, *Pony*$^{\mathrm{GS}}$, still omits important parts of the Pony language. Most importantly, it does not cover generics. Generics allow definitions in programs to be reused in different contexts, even if the types in use differ. It makes for faster and cleaner development by reducing code duplication, as well as facilitates maintenance and reasoning about programs. Generics in Pony interact deeply with other features unique to the language. We give a more detailed description of generics as used in Pony in Section 2.3.

1.2 Goals

The primary aim of this project is to define a new model of the Pony language, with support for generic definitions. This model should allow a better understand how generics should integrate and interact with the rest of the language. Defining this model requires extending the syntax of the language to allow definitions to be parametrized by type arguments, extending the runtime semantics of the language appropriately, and finally define new typing rules which determine whether a program is well-formed in our model.

Given this model of the Pony language, we would like to verify the soundness of our approach, ensuring that the typing rules are sufficient to uphold the data-race freedom guarantees promised by the language.

Finally, we want to evaluate how relevant our model is to the implementation of generics in the compiler, by comparing which programs are allowed by our model but not by the compiler, and vice-versa.

Chapter 2

Background

2.1 Concurrent programming

Concurrent programming allows multiple processes to execute simultaneously on a single computer. It can be used for example in servers which need handle multiple clients. If client requests were processed sequentially, a request which takes a long time to answer would block all other client requests from being served. In general the different processes executing simultaneously are not independent. They may need to share resources or data.

In the example from Figure 2.1, the `MultithreadedServer` class creates a new thread for each incoming request, and executes the `serve` method on it. This allows multiple requests to be served simultaneously. The `CounterServer` class' implementation of the `serve` method simply increments the `counter` field.

This simple example lacks any synchronisation between threads, which can lead to unpredictable and inconsistent behaviour. If two request are processed simultaneously, many scenarios are possible.

One thread might execute the `serve` method first, incrementing the counter by one, before the second thread executes the method and increments it again.

However, the two handling threads could both simultaneously read the same value out of the field, increment the value by one and write the result back, where one thread would overwrite the value written by the other. The counter would only have been increased by one even though two requests were processed.

In languages with a looser memory model such as C or C++, data races are considered undefined behaviour and a similar code could lead to any outcome [Boehm and Adve, 2008].

```
class CounterServer extends MultithreadedServer {
  int counter;

  void serve(Request r) {
    this.counter = this.counter + 1;
  }
}

abstract class MultithreadedServer {
  abstract void serve(Request r);

  void onRequest(Request r) {
    new Thread(() -> this.serve(r)).start();
  }
}
```

Figure 2.1: Multithreaded server, vulnerable to data races

2.1.1 Mutual exclusion

The traditional approach to solving this problem relies on explicit synchronisation between processes, through mutual exclusion locks. Only a single process may hold the lock at the time. Attempting to acquire a held lock will block the process until the lock is released. In the following modified example, the lock guarantees only one thread can be modifying the counter at a time.

```
class CounterServer extends MultithreadedServer {
  Lock lock;
  int counter;

  void serve(Request r) {
    this.lock.lock();
    this.counter = this.counter + 1;
    this.lock.unlock();
  }
}
```

Figure 2.2: Synchronized server using mutual exclusion

This approach has multiple disadvantages. Firstly, in most languages synchronisation is not enforced by the type system. The unsynchronised Java example from Figure 2.1 will compile without any issue, despite suffering from a data race. It therefore requires the programmer to carefully protect any access to shared data.

Mutual exclusion also introduces a risk of deadlock, where a process `A` is waiting to acquire a lock which is held by a process `B`, but the latter is also waiting to acquire a lock held by process `A`.

2.1.2 Actor-Based Programming

An alternative approach to preventing data races is to not share mutable state at all. Processes instead communicate through *asynchronous* message passing. In *actor*-based languages [Hewitt et al., 1973] each process, also called actor, is associated with a single message queue. While different actors execute concurrently, each one processes incoming messages sequentially.

Some actor languages, such as Erlang [Erlang], prevent shared mutable state by making all data immutable, or by copying any data sent to another actor, adding overhead to message passing. Others, such as Akka [Akka] or Kilim [Kilim], do not prevent data races at all, requiring on programmers' to avoid these manually.

2.2 Pony

Pony [Clebsch et al., 2015] is an high-performance actor language. Through its type system, Pony statically guarantees the absence of data races between actors,

We present in this section an overview of the language, focusing on the aspects which differentiate it from other similar languages. This section covers similar material as found, for example, on the Pony website. However we limit our description to features of the language relevant to our work. Additionally, we use the same syntax as our model, which can differ with the syntax used by the Pony language in a few places.

We intentionally avoid discussing generics here, as these are described informally in Section 2.3, and formally throughout the rest of this report.

2.2.1 Actors

Pony actors expose asynchronous methods called *behaviours*, which can be invoked by other actors. Each actor has a message queue onto which behaviour invocations are pushed. Whenever an actor is idle, the first message is dequeued and the corresponding behaviour executed. This ensures at most a single behaviour may be executing at a time for a given actor.

In the following example, two `Client` actors concurrently execute the `request` behaviour. The two clients will invoke the `done` behaviour on the `Callback` when the request is complete. If the two invocations of the `done` behaviour were executed concurrently, there would be a risk of a data race when incrementing the actor's `_count` member variable. However, since behaviours within a single actor are executed sequentially, this snippet is free of data races and `_count` can safely modified.

```
actor Client
  be request(cb: Callback) =>
    // Do some work
    cb.done()

actor Callback
  var _count: U32 = 0
  be done() =>
    this._count = this._count + 1

actor Main
  new create() =>
    let cb = Callback.create()
    let client1 = Client.create()
    let client2 = Client.create()

    client1.request(cb)
    client2.request(cb)
```

Unlike system threads, Pony actors and messages are extremely cheap. It is natural to have hundreds of thousands of actors within a program. The Pony runtime takes care of scheduling the actors onto a multiple system threads, making full use of the machine's capacity.

2.2.2 Classes and synchronous functions

Pony also offers a more traditional object oriented programming model. In addition to behaviours, actors may also have functions which are executed synchronously. Pony also allows classes, which unlike actors may only define synchronous functions. For example, the `Counter` class defined below exposes two functions, `add` and `value`.

```
class Counter
  var _count: U32 = 0
  fun box value() : U32 =>
    this._count

  fun ref add(x: U32) =>
    this._count = this._count + x
```

2.2.3 Named constructors

Pony supports *named constructors*. A single class or actor can define multiple constructors with different names. When creating a new instance, the name of the constructor used must be specified. Object constructors are executed synchronously, while actor constructors are executed asynchronously. For example the class `Counter` below defines two constructors, `create` and `default`. Lines 11 and 12 demonstrate how constructors are invoked.

```
 1  class Counter
 2    var _count: U32
 3    new create(count: U32) =>
 4      this._count = count
 5    new default() =>
 6      this._count = 0
 7
 8  actor Main
 9    new create() =>
10      let counterA = Counter.create(10)
11      let counterB = Counter.default()
```

2.2.4 Inheritance

Pony allows the definition of *abstract types*. Unlike classes and actors, methods in abstract types do not define a body. Instead the list of methods define requirements which other types must fulfill in order to implement this type. In the example below, the abstract type `Incrementable` contains a single function signature, `increment`. Because it provides such a function, the class `Counter` implements `Incrementable`.

```
interface Incrementable
  fun ref increment()

class Counter
  var _count: U32
  fun ref increment() =>
    this._count = this._count + 1
```

When a type implements an abstract type, it creates a *subtyping relation* between the two types. Because the child type provides all of the requirements of the parent, an instance of the former can be used wherever an instance of the former is expected. In the example below, even though the `increment_twice` function expects an instance of `Incrementable`, an instance of `Counter` is provided instead on line 5. This is correct because `Counter` is a subtype of `Incrementable`. On the other hand, because the `NotIncrementable` class does not provide a `increment` method, it is not a subtype of `Incrementable` and the invocation of `increment_twice` on line 9 is not allowed.

```
class NotIncrementable

actor Main
  new create() =>
    this.show(Counter.create(0))

    // error: expected Incrementable, got NotIncrementable
    // error: NotIncrementable is not a subtype of Incrementable
    this.show(NotIncrementable.create())

  fun increment_twice(inc: Incrementable) =>
    inc.increment()
    inc.increment()
```

Pony allows two sorts of abstract types. The first one, *interfaces* only require implementors to provide the required method. As soon as a type defines all the methods of the interface it becomes a subtype of that interface, which is a case of *structural subtyping*. The other sort of abstract type in Pony, *traits* impose a further requirement in implementors, which need to explicitely opt-in into implementing the trait, by listing it as a parent type. Traits thus rely on *nominal subtyping*.

For example below, all of `A`, `B` and `C` implement the `HasName` interface since they provide the `name` function. However, only `B` and `C` implement the `Named` trait, as it is not in `A`'s parent list. Note that interfaces can also be listed as parents, such as `HasName` for `C`, but doing so is not required.

```
trait Named
  fun name(): String
interface HasName
  fun name(): String

class A
  fun name(): String => "A"
class B is Named
  fun name(): String => "B"
class C is Named, HasName
  fun name(): String => "C"
```

2.2.5 Variance in method signatures

As described in the previous section, in order to implement an abstract type children types must provide all of the methods required by the parent. However, Pony allows the signature of these methods to differ, as long as the method in the child is more general than the corresponding signaure in the parent. In other words, any where the method in the parent type may be called, calling the child method must be allowed.

Consider for example the following definitions. The `Controller` trait requires child types to provide two methods, `description` and `add`. The first method must return a type which implements `Stringable`, whereas the second method receives an argument of type `Counter`. The `ref` annotatation on the argument type is required for reasons which we'll explain in section Section 2.2.6, and can be ignored for now.

```
interface Stringable
  fun box string(): String
interface Incrementable
  fun ref increment()

class String is Stringable
  fun box string(): String => this
class Counter is Incrementable
  fun ref increment() => ···

trait Controller
  fun description(): Stringable
  fun add(counter: Counter ref)
```

The `AddTwo` class, defined below, implements the `Controller` by providing the two required methods. However, the types which appear in the methods' signature differ from those appearing in the trait. However these differences are acceptable as they make the methods more general. Indeed, the `description` method below returns `String`, and because `String` implements `Stringable` it is always valid to use the return type of the method as a `Stringable`. Similarily, the `add` method accepts any instance of `Increment`, which includes any instance of `Counter` as well, making the method more general than the one from the parent.

```
class AddTwo is Controller =>
  fun description(): String => "AddTwo"

  fun increment_twice(i: Incrementabe ref) =>
    i.increment()
    i.increment()
```

Return types are said to be *covariant*, whereas argument types are *contravariant*.

2.2.6 Capabilities

In order to guarantee data race freedom without any performance costs, Pony assigns each object reference a capability. Capabilities describe what operations are allowed on other aliases of the reference. This is turn determines what operations can be performed using this reference. The six base capabilities are describe below, and summarized in Table 2.1.

Global aliases are aliases to the object from a different actor, whereas *local aliases* are aliases from the same one. For example given the heap described by the graph below, the reference `x` from actor α_1 to to the object ι has one local alias, the reference `y`, and one global alias, the reference `z` from actor α_2

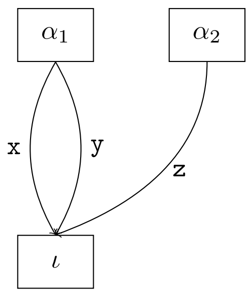

Figure 2.3: Local and Global aliases

- `iso` references deny read and write aliases, both globally and locally. The isolated reference is therefore the only path to access the pointed object in the entire program, making it possible to read and write through this reference without causing any data race.
- `trn` references deny read and write global aliases, but locally they only deny write aliases. Since the current actor is the only one to reference the object, this reference can be used to mutate the object.
- `ref` references deny read and write global aliases, but allow any local alias. Just like `trn` references, no other actor can access this object, making it possible to mutate the object safely.
- `box` references only deny global write aliases, allowing either global read aliases or local write ones. Note that these two cases are mutually exclusive Either there exists a global read alias, or there exists a local mutable one. However the `box` reference alone is not enough to determine which situation applies. It is therefore not possible to mutate an object through a `box` reference, since there *could* be a global alias to it. On the other hand, mutable aliases to that object can only exist locally, making it safe to read through the `box` reference.
- `val` references deny mutable aliases both globally and locally, but allow immutable aliases from any actor. It is therefore safe to read from such a reference, but not to write.
- Finally `tag` references allow any sort of alias, both globally and locally. It is thus not allowed to neither read nor write thorugh this reference, since another actor could be mutating it. Such a reference is therefore said to be *opaque*. Nevertheless, opaque references can be used to compare the identity of objects. Additionally, sending a message to a remote actor is an atomic operation, which can safely be performed concurrently. `tag` references can therefore be used to invoke behaviours on remote actors.

Capabilities which have the same restrictions for both local and global aliases, **`iso`**, **`val`** and **`tag`**, can be sent to other actors as arguments to behaviour invocations. The other three capabilities cannot be used in an argument to a behaviour, as there could exist an alias from the sending actor which should not be allowed anymore once the reference would have been sent to another actor.

	Deny global read/write aliases	Deny global write aliases	Allow all global aliases
Deny local read/write aliases	`iso`		
Deny local write aliases	`trn`	`val`	
Allow all local aliases	`ref`	`box`	`tag`
	Mutable	Immutable	Opaque

Table 2.1: Pony Capabilities, reproduced from [Clebsch et al., 2015]

2.2.7 Temporary references

Object fields and local variables form *stable references*. The reference can be named, and can be used multiple times. On the other hand, *temporary references* do not have a name, and can only be used once. Once used, they are destroyed and cannot be reused.

In addition to the six base capabilities, described above, Pony defines two more capabilities, `iso`○ and `trn`○, which are *ephemeral*. These have similar properties as their respective non-ephemeral counterparts, `iso` and `trn`, but can only apply to temporary references.

- A temporary reference with capability `iso`○ guarantees that no non-opaque stable reference exist pointing to the object.
- A temporary reference with capability `trn`○ guarantees that no mutable stable reference exist pointing to the object, and no global alias exists.

2.2.8 Aliasing

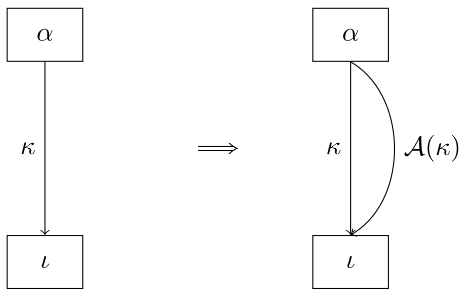

Figure 2.4: Aliasing

Aliasing happens whenever a new stable reference to an object is created from an existing one. This occurs when assigning a reference to a field or when a reference is given as an argument to a constructor or method call. The capability of the aliased reference depends on the capability of the original one. An alias cannot deny more than the original capability. However, it may need to deny less if the original capability requires so.

`ref`, `val`, `box` and `tag` all alias as themselves, since they are all locally compatible with the original capability. `iso` references deny any read or write alias both globally and locally. The only capability it can alias to is therefore `tag`. Finally, `trn` only allows immutable local aliases. However, since `trn` is mutable and `val` denies such aliases, `trn` must alias into a `box`.

Finally, aliasing ephemeral references, with capabilities `iso`∘ or `trn`∘, destroys the original reference, creating a stable one. An `iso`∘ reference therefore aliases as an `iso`, since the only non-opaque alias has just been destroyed. Similarily, `trn`∘ aliases to `trn`, as the only mutable alias has just been destroyed, but there may exist local immutable ones.

2.2.9 Destructive reads and unaliasing

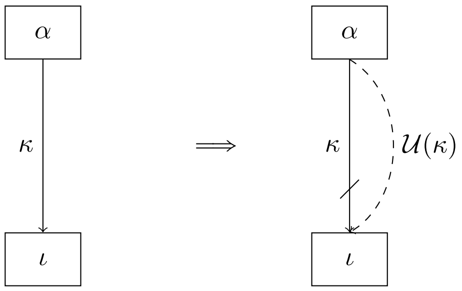

Figure 2.5: Unaliasing

Pony allows *extracting write*, which overwrite a local variable or field's value, while returning the old one as a temporary. This operation unaliases the reference, destroying a stable reference to the object. Just like aliasing, the capability of the unaliased reference depends on the original capability.

If the original capability was `iso`, then unaliasing has destroyed the only non-opaque stable reference to the object. The temporary's capability is therefore `iso`∘. Similarily, if the original capability was `trn`, then the destroyed reference was the only mutable stable reference to the object. There can however exist local immutable aliases to the object. The temporary's capability is therefore `trn`∘.

Finally, if the original capability was one of `ref`, `val`, `box` or `tag`, then despite destroying the stable reference, there could exist other aliases with the same capability. Therefore the temporary has the same capability as the original reference.

2.2.10 Viewpoint adaptation

$$\begin{tikzcd}
\boxed{\alpha} \arrow[d, "\kappa"'] \arrow[dd, bend left, dashed, "\mathcal{V}p(\kappa, \kappa')"] \\
\boxed{\iota} \arrow[d, "\kappa'"'] \\
\boxed{\iota'}
\end{tikzcd}$$

Figure 2.6: Viewpoint adaptation

Reading a field from an object creates a temporary whose capability depends on both the capability of the origin, and the capability of the field. The operator which combines the two capabilities is called *viewpoint adaptation*, as it determines the capability of the field as seen from the origin's point of view.

2.3 Generics in Pony

In this section we present an informal overview the various features introduced alongside generics in Pony. This section serves as a motivation for choices in the syntax and semantics we will introduce later.

Throughout this section, we will describe different versions of a class `Cell`, which simply holds a reference to an object. The most basic example of this class is shown below. It contains a single field of type `A ref`, a constructor which initialises the field from the constructor's argument, and a getter method which return the stored reference.

```
class A
class Cell
  var f: A ref
  new create(x: A ref) =>
    this.f = consume x
  fun ref get() : A ref =>
    this.f
```

Unfortunately this class can only be used to store references of type `A ref`. In order to use the class `Cell` with other types than `A ref`, we would have to define a different version of `Cell` for each of these types, which would lead to significant amounts of duplicated code. Instead, we would want our class to be *polymorphic*, such that it can be used with different types.

2.3.1 Polymorphism without generics

Pony already enables a form of polymorphism through subtyping. Anywhere a reference of a certain type is expected, a reference of a subtype may be used instead. In the following example, the `Cell` class uses an `Any box` reference to point to an object of any type. For instance, on lines 13 and 14, the same class is used to store references to objects of class `A` and `B`.

```
 1  class A
 2  class B
 3  class Cell
 4    var f: Any tag
 5    new create(x: Any tag) =>
 6      this.f = consume x
 7
 8    fun ref get() : Any tag =>
 9      this.f
10
11  actor Main
12    new create() =>
13      var cellA : Cell ref = Cell.create(A.create())
14      var cellB : Cell ref = Cell.create(B.create())
```

This form of polymorphism is however limited, since it loses any extra information about the type of the reference. The `get` function returns `Any tag` no matter what was the type of the reference passed to `Cell`'s constructor. For example, even though the `Cell` constructor is called on line 4 with an argument of type `A ref`, we are unable to retrieve a reference of this type on line 8.

```
1  class A
2  actor Main
3    new create() =>
4      var cell : Cell ref = Cell.create(A.create())
5
6      // error: right side must be a subtype of left side
7      // error: Any tag is not a subtype of A ref
8      var contents : A ref = x.get()
```

2.3.2 Generics

Generics allow classes and methods to have type parameters, introducing a new form of polymorphism which preserves the identity of the original type. Once defined, type variables can be used wherever a type is expected, such as field types or method signatures. In the following example, the `Cell` class is generic over a type parameter `X`.

```
class Cell[X]
  var f: X
  new create(x: X) =>
    this.f = consume x

  fun ref get() : X =>
    this.f
```

Before a generic class can be used, it must be *instantiated* by passing it a type argument. For instance, `Cell[A ref]` is the class `Cell` where all occurences of `X` have been replaced by `A ref`. The `get` function therefore returns a reference of type `A ref`, as shown below.

```
class A
actor Main
  new create() =>
    var cell : Cell[A ref] ref = Cell[A ref].create(A.create())
    var contents : A ref = x.get()
```

2.3.3 Generic Bounds

Each type parameter can have a bound associated, which restricts which types it can be instantiated with. In the example below, the `Cell` class implements the `string` method, which provides a description of the object. It does so by calling the `string` method on the contents of the cell. To ensure this method exists on type `X`, a bound `Stringable box` is added to the type parameter, as shown on line 3.

```
 1 interface Stringable
 2   fun box string(): String iso○
 3
 4 class Cell[X: Stringable box] is Stringable
 5   var f: X
 6   new create(x: X) =>
 7     this.f = consume x
 8
 9   fun box string(): String iso○ =>
10     "Cell(" + this.f.string() + ")"
```

The `Cell` class defined above can therefore only be instantiated with types which implement the `Stringable` interface. In the example below, instatiating `Cell` with `A box`, such as on line 7 is allowed, while instantiating it with `B box` is not, as shown on line 10.

```
class A is Stringable
  fun box string(): String iso○ => "A"
class B

actor Main
  new create() =>
    var cellA : Cell[A box] ref = Cell[A box].create(A.create())

    // error: B does not implement Stringable
    var cellB : Cell[B box] ref = Cell[B box].create(B.create())
```

2.3.4 Capability constraints

Unlike regular subtyping, instantiation of type variables requires the capability of the parameters to match the capability of the bound exactly. For example the type variable and bound `X: Any box` can be instantiated with `A box` but not with a `A iso`, even though `iso` is a subtype of `box`.

This enables the body of the generic class to exploit properties specific to `box`, such being able to alias to itself. In the example below, if `X` could be instantiated with `A iso` the `Cell[A iso]` object would hold two `trn` references to the same object, violating Pony's aliasing rules.

```
class Cell[X: Any box]
  var f1: X
  var f2: X
  new create(x: X) =>
    this.f1 = x
    this.f2 = x
```

There are however cases where we would want to allow more than a single capability. For instance, the class `Cell` from Section 2.3.3 should allow any capability as long as it is possible to call the `string` method. In addition to concrete capabilities, Pony allows *capability constraints* to be used in generic bounds. Each constraint admits a set of concrete capabilities. There are five basic constraints, as well as two ephemeral variants. They are described in Table 2.2.

The `Cell` class from section Section 2.3.3 can be modified to use the #`read` constraint in its bound, as shown below. This constraint allows any of `ref`, `val` or `box`, making it possible to call the `string` on line 6, while being more flexible than the `X: Stringable box` bound used previously.

```
1 interface Stringable
2   fun box string(): String iso○
3 class Cell[X: Stringable #read] is Stringable
4   var f: X
5   fun box string(): String iso○ =>
6     "Cell(" + this.f.string() + ")"
```

Constraint	Allowed capabilities	Description
`#any`	`iso`, `trn`, `ref`, `val`, `box`, `tag`	Any capability
`#read`	`ref`, `val`, `box`	Capabilities which can be read from, and alias as themselves
`#send`	`iso`, `val`, `tag`	Capabilities which can be sent to an actor
`#share`	`val`, `tag`	Capabilities which can be sent to more than one actor
`#alias`	`ref`, `val`, `box`, `tag`	Capabilities that alias as themselves
`#any`$\circ$	`iso`$\circ$, `trn`$\circ$, `ref`, `val`, `box`, `tag`	Any ephemeral capability
`#send`$\circ$	`iso`$\circ$, `val`, `tag`	Ephemeral capabilities which can be sent to an actor

Table 2.2: Capability Constraints

The choice of constraints and how they relate to concrete capabilities is not motivated by a specific rationale, but simply captures common patterns in Pony programs. Modifying existing constraints' definitions or adding new ones would only require small changes to our description of the language. For example, we may want in the future to modify #`read` to include `trn` and `iso`, or we could add a new constraint #`write` which would include `ref`, `trn` and `iso`.

2.3.5 Recursive bounds

Pony supports recursive bounds, also known as F-bounded polymorphism. These allow type variables to appear in their own bounds, as well as in bounds of other variables. These bounds are useful in the definition of binary operations. Consider for example the `Ordered` interface below, which defines a function `less`. This function compares the receiver with an object of type `X`.

```
interface Ordered[X]
  fun less(other: X): Bool
```

Binary operations are generally applied on two objects of the same type. The recursive bound `X: Ordered[X] #read` is used to express this, as shown below.

```
actor Main
  fun minimum[X: Ordered[X] #read](a: X, b: X) =>
    if a.less(b) then
      consume a
    else
      consume b
    end
```

2.3.6 Variance of Type Parameters

In general, types in Pony are *invariant* with respect to their type arguments. Given two types `S` and `T`, where `S` is a subtype of `T`, then `Cell[S]` is neither a subtype nor a supertype of `Cell[T]`.

In the example below, if `Cell[Any box] ref` were a subtype of `Cell[N box] ref`, one could pass the former as the argument to the `read` method. This method would be able to read the contents as an `N box`, even though it could contain any subtype of `Any box`.

Similarily, if `Cell[A box] ref` were a subtype of `Cell[N box] ref`, one could could pass the latter as the argument to the `write` method. This method writes a `B box` into the cell, even though from the caller's perspective it must contain a `A box`.

```
interface Any
trait N is Any
class A is N
class B is N

class Cell[X]
  var f: X
  fun ref get() : X =>
    this.f
  fun ref set(x: X) =>
    this.f = consume x

actor Main
  fun read(cell: Cell[N box] ref) : N box =>
    cell.get()

  fun write(cell: Cell[N box] ref) =>
    cell.set(B.create())

  new create() =>
    this.read(Cell[N box].create(A.create()))
    this.write(Cell[N box].create(A.create()))

    // error: Cell[A box] is not a subtype of Cell[N box]
    // error: Type parameters are not equal
    this.read(Cell[A box].create(A.create()))

    // error: Cell[Any box] is not a subtype of Cell[N box]
    // error: Type parameters are not equal
    this.write(Cell[Any box].create(A.create()))
```

Unfortunately, invariance of type arguments prevents uses which would otherwise be sound. For instance, passing a `Cell[A box] ref` as the argument to the `read` method or passing a `Cell[Any box] ref` as the argument to the `write` would be sound, but neither of these are allowed, as shown on lines 26 and 30 in the example above.

However, this limitation can be worked around by defining two different interfaces, `CellGet` which contains all the methods of `Cell` where `X` only appears in *contravariant* positions, and `CellSet` which contains all the methods where `X` only appears in *covariant* position, as shown below. Thanks to structural subtyping, `Cell[S]` implements `CallGet[T]` as long as `S` is a subtype of `T`, and it implements `CallSet[T]` as long as `T` is a subtype of `S`.

```
interface CellGet[X]
  fun ref get() : X
interface CellSet[X]
  fun ref set(x: X)
```

We can use these two interfaces in the signatures of the `read` and the `write` methods, as shown in the example below. This allows the two sound uses on lines 9 and 10, while still preventing the unsound ones on lines 14 and 18.

```
 1  actor Main
 2    fun read(cell: CellGet[N box] ref) : N box =>
 3      cell.get()
 4
 5    fun write(cell: CellSet[N box] ref) =>
 6      cell.set(B.create())
 7
 8    new create() =>
 9      this.read(Cell[A box].create(A.create()))
10      this.write(Cell[Any box].create(A.create()))
11
12      // error: Cell[Any box] does not implement CellGet[N box]
13      // error: Any box is not a subtype of N box
14      this.read(Cell[Any box].create(A.create()))
15
16      // error: Cell[A box] does not implement CellSet[N box]
17      // error: N box is not a subtype of A box
18      this.write(Cell[A box].create(A.create()))
```

2.3.7 Explicit viewpoint adaptation

As described before in Section 2.2.10, field access in Pony uses viewpoint adaptation to preserve deep immutability. For example, the contents of a `Cell`, when accessed through a **`box`** reference, should be immutable. This can be expressed with explicit viewpoint adaptation, such as **`box`**`->X`. This notation refers to the `X` type as seen through a **`box`** reference.

The concrete capability depends on how the type variable is instantiated, following the usual rules of viewpoint adaptation. For example, when instantiated with `A` **`ref`**, the adapted type will be

`A box`. When instantiated with `A val`, the adapted type will be `A val`.

The example below demonstrates the use of explicit viewpoint adaptation in the return type of the `get` function. In order to allow mutable access, the `get_ref` function is available, but requires the receiver to be mutable. Since viewpoint adaptation through `ref` always results the original type, a return type of `X` is sufficient.

```
class Cell[X: Any #any]
  var f: X
  new create(x: X) =>
    this.f = consume x

  fun box get() : box->X =>
    this.f

  fun ref get_ref() : X =>
    this.f
```

In this example, the `get` and `get_ref` functions only differ in their receiver capability and return type. In order to avoid such duplication, Pony allows functions to be polymorphic with respect to their receiver capability. Such a function can be called with different receiver capability, and have it reflected in the signature using the special viewpoint `this`.

In the `Cell` class, the `get` and `get_ref` functions can be unified into a single function which allows any reference capability in `#read`, as shown below. Calling the `get` function on a reference of type `Cell[A ref] box` will return a reference of type `A box`, preserving deep immutability. However, calling the same method on a reference of type `Cell[A ref] ref` will return an `A ref` reference, enabling mutability.

```
class Cell[X: Any #any]
  var f: X
  new create(x: X) =>
    this.f = consume x

  fun #read get() : this->X =>
    this.f
```

Finally, the viewpoint on the left of the arrow operator can also be another type, in order to refer to the how this type sees the type on the right-hand side of the arrow. This is illustrated below, by defining a function `get_other` which calls `get` on another instance of `Cell[X]`. The capability of the return type depends on how both `X` and `Y` are instantiated, and thus we use explicit viewpoint adaptation to express the return type `Y->X`.

```
class Cell[X: Any #any]
  var f: X
  new create(x: X) =>
    this.f = consume x

  fun #read get() : this->X =>
    this.f

  fun get_other[Y: Cell[X] #read](other: Y) : Y->X =>
    other.get()
```

2.3.8 Explicit aliasing and unaliasing

In addition to viewpoint adaptation, types in Pony may be modified through aliasing and unaliasing. Like viewpoint adaptation, the result of these operations, when applied to type variables, depends on their instantiation. Pony therefore supports special syntax for referring to the result, through the $+$ and $-$ operators, respectively for aliasing and unaliasing. The example below demonstrates the use of each operator.

```
class Cell[X: Any #any]
  var f: X
  new create(x: X) =>
    this.f = consume x

  fun ref replace(x: X): X- =>
    this.f = consume x

  fun ref clone(): Cell[X+] ref =>
    Cell[X+].create(this.f)
```

The `replace` function changes which reference is stored in the cell. Thanks to destructive field write, the old reference is returned. This reference has the same type as the contents of the cell, but with an alias removed. The unaliasing operator $-$ is used to reflect this fact. Again, the exact capability of the returned reference depends on how `X` was instantiated. For a `Cell[A box]`, the unaliasing operator has no impact and the function returns an `A box`. However, for a `Cell[A iso]` the function returns an `A iso`$\circ$, allowing the caller to alias it to an `A iso`.

The `clone` function creates a new cell pointing to the same object. The call to `Cell.create` however creates an alias of `this.f`, which is reflected in the return type `Cell[X+]`, by the $+$ operator. Calling `clone` on a `Cell[A ref] ref` would return a reference of the same type, aliasing leaves the type unmodified. However, calling the same method on a `Cell[A iso] ref` would return a `Cell[A tag] ref`, since a `iso` reference aliases into a `tag`.

The `clone` function as described above can only be used on `ref` receiver. If the function had been defined with, for example, a `box` receiver, then the expression `this.f` would have type `box->X`, and it would not be possible to call the `Cell[X+]` constructor with this value. Instead, we can combine the two features, explicit viewpoint adaptation and explicit aliasing, to write a `clone` function which can be called with any sort of readable receiver, as shown below.

```
class Cell[X: Any #any]
  var f: X
  fun #read clone(): Cell[(this->X)+] ref =>
    Cell[(this->X)+].create(this.f)
```

2.3.9 Object creation

In addition to methods, traits and interfaces can define constructors which must be provided by types which implement them. This allows constructors to be invoked on type variables bound by such a trait or interface.

The return capability of constructors is normally implicit depending on the type being constructed. Actor constructors return a **`tag`** reference, whereas class constructors return a **`ref`** one. Because interfaces could be implemented by either, the return capability must be specified explicitely, as shown below. Traits and interfaces with **`ref`** constructors, such as `Default`, can only be implemented by classes, whereas those with only **`tag`** constructors, such as `DefaultOpaque` can be implemented by both actors and classes.

```
interface Default
  new default() : ref
interface DefaultOpaque
  new default_opaque() : tag

class A is Default, DefaultOpaque
  new default() => this
  new default_opaque() => this

actor B is DefaultOpaque
  new default_opaque() => this
```

Usually, type variables represent both a type identifier and a capability. However constructors must return references of a specific capability, either **`ref`** or **`tag`**, no matter what capability the type variable's instantiation has. Pony therefore allows a special type syntax, composed of a type variable and a capability, such as `X` **`ref`**. This type uses the type identifier of the variable's instantiation, but a fixed capability. For example below, even though the class `Cell` is instantiated with `A` **`iso`**, the constructor call on line 4 returns a reference of type `X` **`ref`**. The field `f` has the same type, making the assignment valid.

```
class Cell[X: Default #any]
  var f: X ref
  new create() =>
    this.f = X.default()

class A
actor Main
  new create() =>
    let cell : Cell[A iso].create()
    let contents : A ref = cell.f
```

It is however possible to recover the result of the constructor call into an `X iso○`, which can be aliased as an `X` reference, as shown below.

```
class Cell[X: Default #any]
  var f: X
  new create() =>
    this.f = recover X.default()
```

In general, Pony allows either *concrete types*, actors and classes, or *abstract types*, traits and interfaces, to be used to instantiate generic types. The latter can be used to create heterogenous collections. For example, given a generic class `Array`, the type `Array[Stringable box]` represents an array which can contain objects of different types, as long as they all implement the `Stringable` interface, as shown below.

```
class A is Stringable
  fun box string(): String iso○ => "A"
class B is Stringable
  fun box string(): String iso○ => "A"

class Array[X]
  new create() => ···
  fun ref push(x: X) => ···
  fun box pop(): X- => ···

actor Main
  new create() =>
    let array : Array[Stringable box] ref = Array[Stringable].create()

    array.push(A.create())
    array.push(B.create())

    let element : Stringable box = array.pop()
```

However, if a type variable's bound is *constructible*, that is it defines a constructor, then it can only be instantiated by concrete types. Indeed, in the example below, if the instantiation of `Cell` with the abstract type `Default ref` was allowed,

```
class Cell[X: Default #any]
  var f: X
  new create() =>
    this.f = recover X.default()

actor Main
  new create() =>
    Cell[A ref].create()

    // error: abstract type Default ref does not satisfy
    //        constructible bound Default #any
    Cell[Default ref].create()
```

2.4 Modelling and soundness of languages

2.4.1 Modelling Java

There have been multiple formal models describing subsets of the Java language, and proving their soundness. [Drossopoulou et al., 1999] is one of the first such model and describes a significant portion of the original Java language. This model included most of the language's features, including primitive types, classes and inheritance, method overloading and overriding. It does not however describe any form of generics, as these were only added later to the language.

[Igarashi et al., 2001] introduced a very reduced version of the Java language, intended as a minimal core calculus for modelling Java's type system. Featherweight Java omits many important features of the language, including primitive types, mutability or method overloading. It is however used as the basis to describe Featherweight Generic Java, which is itself a subset of Generic Java. Generic Java is a backwards compatible extension to the Java language, allowing classes and methods to be generic [Bracha et al., 1998]. It was ultimitely used as the basis of the implementation of generics introduced in Java 5.

While no model for GJ was presented, soundness of FGJ's type system is described by [Igarashi et al., 2001] and illustrates the key features of GJ. Rather than proving soundness of FGJ's type system from scratch, it defines a transformation from FGJ to FJ. The transformation erases type arguments from class and methods, replacing all type variables by their bounds, as illustrated by Figures 2.7 and 2.8.

In order for the resulting FJ program to type check propely, *synthetic casts* are inserted in the transformed program. For instance, the expression `new Pair<A,B>(new A(), new B()).snd` is transformed into `(B)new Pair(new A(), new B()).snd`. These synthetic casts are guaranteed to succeed at runtime. This matches Java's implemention of generics.

It is shown that any valid FGJ program is transformed to a valid FJ program, and that reduction in FGJ and FJ are equivalent, thus showing soundness of FGJ's type system [Igarashi et al., 2001].

2.4.2 Unsoundness of Java's type system

Despite the various models and proofs of soundness for subsets of the Java language, no model covers the entire language. Indeed, it has been recently discovered that its type system is actually unsound [Amin and Tate, 2016].

The program reproduced in Figure 2.9 succesfully coerces an integer into a string. This program relies on different features of the language. While these features have been specified and proven sound in isolation, their interaction within in a single language hadn't been fully considered.

```java
class Pair<X extends Object, Y extends Object> extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);
  }
}
```

Figure 2.7: Pair class in FGJ, reproduced from [Igarashi et al., 2001]

```java
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

Figure 2.8: Erased pair class, reproduced from [Igarashi et al., 2001]

```java
class Unsound {
  static class Constrain<A, B extends A> {}
  static class Bind<A> {
    <B extends A> A upcast(Constrain<A, B> constrain, B b) {
      return b;
    }
  }
  static <T,U> U coerce(T t) {
    Constrain<U, ? super T> constrain = null;
    Bind<U> bind = new Bind<U>();
    return bind.upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = Unsound.<Integer,String>coerce(0);
  }
}
```

Figure 2.9: Unsound valid Java program, reproduced from [Amin and Tate, 2016]

In the example, the `Constrain` class acts as a proof that its type argument `B` is a subtype of `A`. An instance of this type can only be created if the compiler can prove this, and conversly an instance of this type can be used by the program as a proof of the subtyping relation.

While the features used by the program, wilcards and existential types, were shown to be sound by [Torgersen et al., 2005] and [Cameron et al., 2008], the subsets of the language used omitted null pointers in an attempt to simplify the models. However, in Java `null` inhabits any valid type, making it possible to construct a value of type `Constrain` with the right type parameters, without any need to prove any relation between the two types, as shown by the example. This value of `Constrain` is then incorrectly used as a proof of the subtyping relation, allowing unrelated types to be coerced.

Fortunately, the program listed in Figure 2.9 reliably terminates at runtime by throwing a `ClassCastException`. This is only because the JVM does not directly support generics, and they are instead implemented in Java by erasing type information and introducing synthetic casts. The JVM still performs runtime checks executing the synthetic casts and is able to identify this invalid case. Had Java generics been implemented by reification, this soundness issue could have led to undefined behaviour and potential security vulnerabilities.

This example demonstrates the importance of modelling the largest subset possible of the language. While larger models complicates any reasoning made about them, they are necessary to make sure all interactions between features are considered.

2.4.3 Modelling Pony

Pony makes strong guarantees about the lack of data races in valid Pony programs. This makes a formal model for the language even more important in order for the guarantees to be trusted.

Just like Java, existing models for subset of the Pony language have been described by [Clebsch et al., 2015] and [Steed, 2016]. The former, which we'll refer to as $\mathit{Pony}^{\mathrm{SC}}$, provides an initial model of the language, with its syntax, operational semantics and typing rules. It also proves soundness and data race freedom for valid Pony programs. It however focuses on the core components of the language, most notably deny capabilities. The latter, which we'll refer to as $\mathit{Pony}^{\mathrm{GS}}$, is based off $\mathit{Pony}^{\mathrm{SC}}$ but presents a more principled approach in determining some of the typing rules. It also extends the language by introducing inheritance as well as union, intersection and tuple types.

Both models however omit generics from their description of the language. As described previously, generics have a very complex interaction with the other essential features of Pony. Adding them to these models would be a significant step forward torwards modelling the entire language.

Chapter 3

Syntax

In this chapter we present the syntax for *Pony*$^{\mathrm{PL}}$, a formal model for the Pony language with support for generics. The model is an extension of *Pony*^{0}, a model of the Pony language without support for generics, which is described formally in Appendix A. Throughout this chapter, differences between *Pony*$^{\mathrm{PL}}$ and *Pony*^{0} are highlighted in grey.

3.1 Programs and definitions

Pony$^{\mathrm{PL}}$ programs consist of class, actor, trait and interface definitions. The syntax for each of these definitions is decribed in Figure 3.1.

$$
\begin{array}{rrl}
\mathtt{P} \in & \mathit{Program} & ::= \overline{\mathtt{CT}}\ \overline{\mathtt{AT}}\ \overline{\mathtt{NT}}\ \overline{\mathtt{ST}} \\
\mathtt{CT} \in & \mathit{ClassDef} & ::= \mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}:\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{F}}\ \overline{\mathtt{K}}\ \overline{\mathtt{M}} \\
\mathtt{AT} \in & \mathit{ActorDef} & ::= \mathtt{actor}\ \mathtt{A}[\overline{\mathtt{X}:\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{F}}\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\ \overline{\mathtt{B}} \\
\mathtt{NT} \in & \mathit{TraitDef} & ::= \mathtt{trait}\ \mathtt{N}[\overline{\mathtt{X}:\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{KS}}\ \overline{\mathtt{MS}}\ \overline{\mathtt{BS}} \\
\mathtt{ST} \in & \mathit{InterfaceDef} & ::= \mathtt{interface}\ \mathtt{S}[\overline{\mathtt{X}:\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{KS}}\ \overline{\mathtt{MS}}\ \overline{\mathtt{BS}}
\end{array}
$$

Figure 3.1: Syntax of programs

Classes are made of fields ($\overline{\mathtt{F}}$), named constructors ($\overline{\mathtt{K}}$) and functions ($\overline{\mathtt{M}}$). Actors are similar to classes, but may also contain behaviours ($\overline{\mathtt{B}}$).

Traits and interfaces are used to describe the constructors and methods exposed by an object. They are made up of *stubs* which only define the constructor and method signatures, without providing an implementation. In order to implement a trait or interface, objects must provide all of the constructors and methods of the parent type, with a compatible signature. Traits use nominal subtyping, whereas interfaces use structural subtyping, as explained previously in Section 2.2.4.

In order to support generics, all the definitions can be parametrized with type variables (X). In the rest of the program, the definition can only be used by *instantiating* it first by providing appropriate type arguments. Each type variable has an associated bound (BT) which describes which types are allowed as type arguments in instantiations.

Since parent types themselves may be generic, they must be instantiated when they are specified. Note that a definition's type arguments are unrelated to its parents'. A definition may have fewer or more type arguments than its parents, and the parents can be instantiated with both the definition's own type variables or with concrete types. As such, all of the definitions below are correct.

```
trait N1
trait N2[X]
class A[X] is N1
class B[X] is N2[X]
class C[X] is N2[E ref]
class D is N2[E ref]
class E
```

Type variables may however not be used directly as a parent type, only as a their type arguments. The definition below is therefore not allowed.

```
// error: type variable X cannot be used as parent type
class A[X] is X
```

Compared to *Pony*^{0}, traits and interfaces in *Pony*$^{\mathrm{PL}}$ may also contain constructor stubs (KS). These were not needed before the introduction of generics, since methods on traits and interfaces were always invoked through an instance of that type. In *Pony*$^{\mathrm{PL}}$ however, constructors can be called directly on type variables, provided they are defined in the variable's bound.

Differences with the Pony language

The Pony language allows the bound of type variables to be omitted, in which case the bound `Any #any` type is used instead, as it is a supertype for all types. For conciseness, we also omit the bounds in some of our examples.

3.2 Items

We refer to the contents of definitions as *items*. Their syntax is described in Figure 3.2.

$$
\begin{array}{rrl}
\mathtt{F} \in & \textit{Field} & ::= \mathtt{var}\ \mathtt{f} : \mathtt{T} \\
\mathtt{K} \in & \textit{Ctor} & ::= \mathtt{new}\ \mathtt{k}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{M} \in & \textit{Func} & ::= \mathtt{fun}\ \nu\ \mathtt{m}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{T} \Rightarrow \mathtt{e} \\
\mathtt{B} \in & \textit{Behv} & ::= \mathtt{be}\ \mathtt{b}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{KS} \in & \textit{CtorStub} & ::= \mathtt{new}\ \mathtt{k}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \kappa \\
\mathtt{MS} \in & \textit{FuncStub} & ::= \mathtt{fun}\ \nu\ \mathtt{m}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{T} \\
\mathtt{BS} \in & \textit{BehvStub} & ::= \mathtt{be}\ \mathtt{b}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}})
\end{array}
$$

Figure 3.2: Syntax of items

Constructors, functions and behaviours share a common syntax. They receive arguments ($\overline{\mathtt{x}} : \overline{\mathtt{T}}$) and have an associated body (e). Even though method bodies are a single expression, expressions can be composed together using a semicolon, forming a new expression. This can be used to define methods with many expressions which are evaluated sequentially. We describe the syntax of expressions in detail in Section 3.5. In order to support generic constructors and methods, their syntax is extended in *Pony*$^{\text{PL}}$ to include type arguments, following the same syntax ($\overline{\mathtt{X}} : \overline{\mathtt{BT}}$) used previously in Section 3.1.

Functions have a receiver capability, which determines what capability is required to call the function on a reference, and a return type. Invoking a behaviour can be done using any reference to the actor, even opaque. The receiver type is therefore implicitely `tag`. Additionally they cannot return a value since they are executed asynchronously.

As explained in Section 2.3.7, a method can be polymorphic over its receiver capability, allowing it to be used on references of different capability, and have it reflected in the signature by using the special **`this`** viewpoint. Compared to *Pony*^{0}, we therefore allow methods to specify a capability bound (ν), such as #`read`, rather than a concrete capability.

Since constructors create a new object rather than receive an existing one, they do not have any receiver capability and their return type is implicitely the type they are constructing. The capability of the return type however depends on the kind of object being creates. For classes, the return type has capability **`ref`**, whereas it has capability **`tag`** for actors. Constructor stubs however are contained in traits and interfaces, which can be implemented by either actors or classes.

A trait or interface with a **`tag`** constructor can be implemented by either actors or classes, since **`ref`** is a subtype of **`tag`** and, as explained in Section 5.12, the return capability is covariant. On the other hand, a **`ref`** constructor cannot be implemented by actors, since these only have **`tag`** ones.

Differences with the Pony language

The Pony language places the return capability of constructor stubs between the `new` keyword and the constructor name, similar to how receiver capabilities are placed, whereas we've placed it to the right of the signature, where return types are usually found. The two following two constructor stub are therefore equivalent in their respective language.

```
new ref create() // Pony
new create() : ref // Pony^PL
```

We've found that placing the capability in the return position better reflects its purpose. The difference with receiver capabilities becomes apparent when we define method subtyping in Section 5.12, as function receiver capabilities are contravariant but constructor return capabilities are covariant. In other words, a `ref` constructor for example would be a subtype of a `tag` one, whereas a `tag` function is a subtype of a `ref` one.

Because all existing constraints other than `#read` allow `tag` to be used as the receiver, they have very little practical as a receiver capability. Therefore, the Pony language does not allow capability constraints to be used as a receiver capability, but treats `box` as if they were `#read`. Additionally, a `box` receiver capability can always be replaced by a `#read` one. The following two methods are equivalent in their respective languages.

```
fun box get() : this▷X => ··· // Pony
fun #read get() : this▷X => ··· // Pony^PL
```

3.3 Types

Types are used in field declarations, as method parameter and return types, and as arguments for instantiations of generic types and methods. Their syntax is described in Figure 3.3.

$$
\begin{array}{rcl}
\mathtt{T} \in & \textit{Type} & ::= \mathtt{DS}[\overline{\mathtt{T}}]\ \kappa \mid \mathtt{X} \mid \mathtt{X}\ \kappa \mid \mathtt{T}+ \mid \mathtt{T}- \mid \mathtt{VP} \rhd \mathtt{T} \mid \mathtt{recover\ T} \\
\mathtt{DS} \in & \textit{TypeID} & ::= \mathtt{A} \mid \mathtt{C} \mid \mathtt{N} \mid \mathtt{S} \\
\mathtt{RS} \in & \textit{RuntimeTypeID} & ::= \mathtt{A} \mid \mathtt{C} \\
\mathtt{I} \in & \textit{AbstractTypeID} & ::= \mathtt{N} \mid \mathtt{S} \\
\kappa \in & \textit{Cap} & ::= \mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref} \mid \mathtt{val} \mid \mathtt{box} \mid \mathtt{tag} \mid \mathtt{iso}\circ \mid \mathtt{trn}\circ \\
\mathtt{VP} \in & \textit{Viewpoint} & ::= \kappa \mid \mathtt{T} \mid \mathtt{this} \\
\mathtt{KT} \in & \textit{CtorType} & ::= \mathtt{RS}[\overline{\mathtt{T}}] \mid \mathtt{X}
\end{array}
$$

Figure 3.3: Syntax of types

The main syntax for types consists of a type identifier (`DS`) with an associated capability (κ). Additionally, if the type identifier refers to a generic definition, then it must be instantiated with the right number of type arguments. For example, given the definitions below, **`A ref`** and **`Cell[A iso] ref`** and are valid types, whereas **`Cell ref`** is not.

```
class A
class Cell[X]
class Pair[X, Y]

actor Main
  var a : A ref = ···
  var cell : Cell[A iso] ref = ···
  var pair : Pair[A iso, A iso] ref = ···

  // error: wrong number of type arguments for class A, expected 0 got 1.
  var a' : A[A iso] ref = ···

  // error: wrong number of type arguments for class Cell, expected 1 got 0.
  var cell' : Cell ref = ···

  // error: wrong number of type arguments for class Pair, expected 2 got 1.
  var pair' : Pair[A iso] ref = ···
```

Inside a generic definition, any type variable (`X`) in scope can be used wherever a type is expected. When instantiating the definition, the type variable will be replaced with its corresponding instantiation. In the following example, the field `f` of class `Cell` has type `X`. However, when considering the instantiation **`Cell[A ref]`**, the field has type **`A ref`**.

```
class A
class Cell[X]
  var f: X
```

Type variables are generally used without specifying a capability, as in the example above, since their instantiation already includes one. However, when calling a constructor on a type variable, the reference returned by the constructor will have the type of the variable's instantiation, but the capability will be one of **`ref`** or **`tag`**, depending on whether an actor or a class is being constructed. The form `X` κ is used to refer to these types. In example below, the constructor call on line 8 has type **`X ref`**, even if `X` is instantiated with a type of a different capability. Calling `make` on an object of type **`Factory[A tag]`** would return a reference of type **`A ref`**.

```
trait Default
  new create() : ref
class A is Default
  new create() => this

class Factory[X: Default #any]
  fun make() : X ref =>
    X.create()
```

We introduce three operators which are use to modify as type's capability, aliasing (+), unaliasing (−) and viewpoint adaptation (▷). The first two are postfix operators applied to a type, while the latter is an infix operator which accepts a viewpoint (`VP`) on the left and another type on the right. These operators encode these operations symbolically. For example, `(A iso)+` and `A tag` are two syntactically distinct types, even though they are semantically equivalent. A fourth operator, `recover`, is used to represent the type from recovering an expression. It is not meant to be used in programs, but rather as a detail of the type system.

Viewpoints may be a capability, another type or the special viewpoint `this`. When a type is used as a receiver, its type identifier is ignored and only its capability is used. However it can be used when the type's capability is not known yet, such as a type variable. The viewpoint `this` is used in functions with polymorphic receivers, and refer to the receiver capability.

Constructor types (`KT`) are used as the base for constructor calls. Because traits and interfaces cannot be constructed, constructor types can only be a runtime type (`RS`) or a type variable. In the latter case, the typing rules will ensure the variable can only be instantiated with a runtime type.

The two ephemeral capabilities, `iso`∘ and `trn`∘, are recognizable by their trailing circle symbol. These capabilities should generally only be used in return types, as it does not make sense for a field or argument, which are stable references, to have an ephemeral type. However, enforcing this restriction adds complexity to the model for little benefit. It is much simpler to allow any type, ephemeral or not, to be used in any position. Since it is impossible to create a stable reference with an ephemeral type, a method with an ephemeral argument type will be impossible to call, and a field with such a type will be unassignable. Similarily, while ephemeral capabilities can be used as method receiver capabilities, such method can never be called.

Differences with the Pony language

The Pony language does not define any general aliasing and unaliasing operator, nor does it define the ephemeral capabilities `iso`∘ and `trn`∘.

Instead, both concepts are replaced by modifiers (ϕ), for aliasing (!) and unaliasing ($\wedge$). While these have similar semantics as the *Pony*$^{\text{PL}}$ operators, they can only be applied to type identifiers, using the form `DS[`$\overline{\texttt{T}}$`]` κ ϕ, or to type variables, using the form `X` ϕ, but not to any type like *Pony*$^{\text{PL}}$ operators can be. Figure 3.4 describes the syntax of types used by the compiler, ignoring other features which are not covered by our model.

$$\begin{aligned} \mathtt{T} \in \mathit{Type} &::= \mathtt{DS}[\overline{\mathtt{T}}]\ \kappa\ \phi \mid \mathtt{X}\ \phi \mid \mathtt{VP} \rhd \mathtt{T} \\ \kappa \in \mathit{Cap} &::= \mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref} \mid \mathtt{val} \mid \mathtt{box} \mid \mathtt{tag} \\ \phi \in \mathit{Mod} &::= \,! \mid \wedge \mid \epsilon \end{aligned}$$

Figure 3.4: Syntax of types used by the Pony compiler (simplified)

We've found that the syntax used by the compiler, which only allows modifiers on certain types, does not make it possible to represent the type of certain expressions. For example in *Pony*$^{\text{PL}}$, the reference created by consuming a variable of type `val->(X+)` has type `(val->(X+))-`, as shown below on lines 3 and 4. However, in the Pony language, the type of the reference created by consuming a variable of type `val->(X!)` cannot be expressed at all, as shown on lines 8 and 9.

```
1 // PonyPL
2 actor Main
3   fun m[X](x: val->(X+)) : (val->(X+))- =>
4     consume x
5
6 // Pony
7 actor Main
8   fun m[X](x: val->(X!)) : /* this type cannot be expressed */ =>
9     consume x
```

At the time of writing, the compiler represents this type by unaliasing the right hand side of the arrow, keeping the viewpoint unchanged. For instance, the compiler would unalias the type `val->(X!)` into `val->X`, as shown below. However, we've found this to be unsound.

```
// Pony
actor Main
  fun m[X](x: val->(X!)) : val->X =>
    consume x
```

Finally we've used the + and − symbols, borrowed from [Steed, 2016], as we've found them to be more evocative than the ! and ∧ symbols, since they represent respectively the addition and removal of an alias. These symbols are not used in the Pony language as they already used as arithmetic operators.

3.3.1 Reified and Ground Types

We also define restricted syntaxes for types, shown below in Figure 3.5. These are subsets of the general type syntax described in Figure 3.3, and are exclusively used by our definition of the typing rules for *Pony*$^{\text{PL}}$ in Chapter 5.

$$\begin{aligned}
\mathtt{RT} \in \quad & \mathit{ReifiedType} ::= \mathtt{BRT}\ \kappa \\
\mathtt{BRT} \in \quad & \mathit{BasicReifiedType} ::= \mathtt{DS}[\,\overline{\mathtt{RT}}\,] \mid \mathtt{X} \\
\mathtt{GT} \in \quad & \mathit{GroundType} ::= \mathtt{BGT}\ \kappa \\
\mathtt{BGT} \in \quad & \mathit{BasicGroundType} ::= \mathtt{DS}[\,\overline{\mathtt{GT}}\,]
\end{aligned}$$

Figure 3.5: Restricted syntaxes of types

Reified types are a restricted syntax of types, and are the result of partial reification, described in Section 5.5. All type variables appearing in a reified type must have a capability associated. Additionally, reified types may not contain type operators, since these are reduced during reification. Unlike general types, reified types can be decomposed into their base, a *basic reified type*, and a capability. Note that the restriction is deep, type arguments which appear in a reified type must be reified themselves.

Ground types are a further restriction of reified types, where type variables cannot appear at all. Similarily, they can be decomposed into a *basic ground type* and a capability. Again, the restriction is deep, type arguments must be ground types themselves.

3.4 Type bounds

$$\begin{aligned}
\mathtt{BT} \in \quad & \mathit{TypeBound} ::= \mathtt{DS}[\,\overline{\mathtt{T}}\,]\ \nu \mid \mathtt{X} \mid \mathtt{BT}+ \mid \mathtt{BT}- \\
\nu \in \quad & \mathit{CapBound} ::= \kappa \mid \#\mathtt{any} \mid \#\mathtt{read} \mid \#\mathtt{send} \mid \#\mathtt{share} \mid \#\mathtt{alias} \mid \#\mathtt{any}\circ \mid \#\mathtt{send}\circ \\
\mathtt{RB} \in \quad & \mathit{ReifiedBound} ::= \mathtt{BRT}\ \nu
\end{aligned}$$

Figure 3.6: Syntax of bounds

Type bounds are used to constrain what types can be used to instantiate a type variable. Their syntax is similar to that of types, with the addition of capability constraints, as explained in Section 2.3.4.

Explicit viewpoint adaptation is not allowed in bounds, because capability constraints are not closed under viewpoint adaptation. For instance, the bound $\mathtt{box} \rhd \mathtt{N}\ \#\mathtt{any}$ should allow capabilities `val`, `box`, and `tag`, but there aren't any constraints which allow exactly these three capabilities.

Differences with the Pony language

Originally, the Pony compiler considered type arguments inside bounds as bounds themselves. In other words, bounds had the following syntax

$$\mathtt{BT} \in \mathit{TypeBound} ::= \mathtt{DS}[\,\overline{\mathtt{BT}}\,]\ \nu \mid \cdots$$

This allowed the use of capability constraints in these arguments, such as `X: Cell[A #any] box`. However we've noticed this is almost never has the expected behaviour since bounds are in general invariant over type parameters. This means even a type such as `Cell[A iso] box` would not be allowed by this bound. Additionally, it prevents explicit viewpoint adaptation to be used within type arguments of bounds, such as `X: Cell[box▷A #any] box`.

We've also found that supporting such bounds makes the model more complicated by propagating capability constraints into other parts of the type system, making it possible for an expression's type to have a capability constraints.

Instead, we've defined type arguments in bounds to be regular types. This prevents the use of capability constraints, but these uses can be rewritten under the form `X: Cell[Y], Y: A #any`, which has the expected behaviour. On the other hand, this allows the use of viewpoint adaptation in the type arguments, such as `X: Cell[box▷Y], Y: A #any`.

We've proposed this change to the Pony language, which has been integrated in version 0.13.2 of the compiler. When implementing this change we've found this form of bounds was never used in the standard library, conforting us in the idea it had just been an oversight in the compiler's original implementation.

3.5 Expressions

$$\begin{aligned}
\mathtt{e} \in \quad \mathit{Expr} \quad &::= \mathtt{this} \mid \mathtt{null} \mid \mathtt{e};\mathtt{e} \\
&\mid \mathtt{x} \mid \mathtt{x} = \mathtt{e} \\
&\mid \mathtt{e.f} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{recover\ e} \\
&\mid \mathtt{e.n}[\,\mathtt{T};\overline{\mathtt{T}}\,](\overline{\mathtt{e}}) \mid \mathtt{KT.k}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{e}}) \\
\mathtt{E}\langle\cdot\rangle \in \mathit{ExprHole} &::= (\ \cdot\ ) \mid \mathtt{x} = \mathtt{E}\langle\cdot\rangle \mid \mathtt{E}\langle\cdot\rangle;\mathtt{e} \mid \mathtt{E}\langle\cdot\rangle.\mathtt{f} \\
&\mid \mathtt{e.f} = \mathtt{E}\langle\cdot\rangle \mid \mathtt{E}\langle\cdot\rangle.\mathtt{f} = \mathtt{t} \mid \mathtt{recover}\ \mathtt{E}\langle\cdot\rangle \\
&\mid \mathtt{E}\langle\cdot\rangle.\mathtt{n}[\,\mathtt{T};\overline{\mathtt{T}}\,](\overline{\mathtt{t}}) \mid \mathtt{e.n}[\,\mathtt{T};\overline{\mathtt{T}}\,](\overline{\mathtt{t}},\mathtt{E}\langle\cdot\rangle,\overline{\mathtt{e}}) \\
&\mid \mathtt{KT.k}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{t}},\mathtt{E}\langle\cdot\rangle,\overline{\mathtt{e}})
\end{aligned}$$

Figure 3.7: Syntax of expressions

Most of the forms of expressions described in Figure 3.7 are typical of most language and should not come as a suprise.

Assignment to local variables (`x = e`), and to fields (`e.f = e`) extract the old value, through a extracting read, described in Section 2.2.9.

In order to support generics, in addition to the usual parameters method calls also accept a receiver type and type arguments, which are separated by a semicolon. The former is used to determine the receiver capability used by the method. For brevity, we omit it in examples when the method's receiver is not polymorphic. Constructor calls also accept type arguments.

Differences with the Pony language

The Pony language supports many more forms of expressions. These include control flow expressions, such as conditionals and loops, and arithmetic operators. While these are essential for a general purpose language, their semantics are similar to that of other imperative languages. Adding them to our model would make it more complicated while not adding much value.

The Pony language does not allow the receiver type to be specified. Instead it is inferrred by the compiler from the receiver's expression's type. In section , we define a translation of generics into non-generic code, which depends on the receiver type. Making the receiver type explicit allows this translation to be independent of typing.

Null references

The Pony language does not allow the use of null references, since it is a frequent source of bugs in programs. This requires the compiler to check that constructors initialise all of their fields before returning. In *Pony*$^{\text{PL}}$ we avoid the extra complexity by initialising all fields to null.

Additionally, null pointers allow us to express the `consume` expression of the Pony language, described in section Section 6.1, in terms of a destructive read by assigning a null pointer to the variable, as shown below. We also use this more evocative notation in our examples.

```
class A
actor Main
  fun m(x: A iso) : A iso○ =>
    consume x // Pony

  fun n(x: A iso) : A iso○ =>
    x = null // PonyPL
```

Just like it ensures constructors must initialise all fields, the Pony compiler ensures a consumed variable cannot be reused. Our model however does not enforce this, as it would be adding more complexity.

In both cases, while null pointers can be a source of bugs in user programs, they do not have any impact on the soundness of the language. Dereferencing a null reference will simply cause the execution to be stuck.

3.6 Identifiers

$$
\begin{array}{rlrl}
\mathtt{C} \in & \mathit{ClassID} & \mathtt{this}, \mathtt{x} \in & \mathit{SourceID} \\
\mathtt{A} \in & \mathit{ActorID} & \mathtt{t} \in & \mathit{TempID} \\
\mathtt{N} \in & \mathit{TraitID} & \mathtt{k} \in & \mathit{CtorID} \\
\mathtt{S} \in & \mathit{InterfaceID} & \mathtt{m} \in & \mathit{FuncID} \\
\mathtt{X} \in & \mathit{TypeVarID} & \mathtt{b} \in & \mathit{BehvID} \\
\mathtt{f} \in & \mathit{FieldID} & \mathtt{n} \in & \mathit{MethID} = \mathit{CtorID} \cup \mathit{FuncID} \cup \mathit{BehvID}
\end{array}
$$

Figure 3.8: Identifiers

We describe in Figure 3.8 the identifiers used in $\mathit{Pony}^{\mathrm{PL}}$. Most of these are identical to Pony^{0}, and were borrowed from [Steed, 2016]. The metavariable `X` was chosen as it establishes a parallel between type variables and local variables.

Chapter 4

Operational Semantics

We present in this chapter the operational semantics of *Pony*$^{\mathrm{PL}}$. Most of the semantics are identical to *Pony*^{0}, and have been borrowed from [Steed, 2016].

Pony$^{\mathrm{PL}}$ allows constructors to be called on type variables, as shown below. This requires information about type variables' instantiation to be maintained at runtime, in order to determine which type to construct. In the example below, on line 15, the `Factory` class is instantiated with `A ref`, and the identity of the type variable is used at runtime when executing the `make` method, invoked from line 17.

```
 1  trait Default
 2    new default(): ref
 3
 4  class Factory[X: Default #any]
 5    new create() => None
 6
 7    fun make(): X ref =>
 8      X.default()
 9
10  class A is Default
11    new default() => ···
12
13  actor Main
14    new create() =>
15      let factory : Factory[A ref] = Factory[A ref].create()
16
17      let a : A ref = factory.make()
```

4.1 Runtime entities

$$
\begin{array}{lcll}
\chi & \in & \mathit{Heap} & = \mathit{Addr} \rightarrow (\mathit{Actor} \cup \mathit{Object}) \\
 & & \mathit{Actor} & = \mathit{ActorID} \times \mathit{RuntimeEnv} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \times \overline{\mathit{Message}} \times \mathit{Stack} \times \mathit{Expr} \\
 & & \mathit{Object} & = \mathit{ClassID} \times \mathit{RuntimeEnv} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\
\mu & \in & \mathit{Message} & = \mathit{MethID} \times \mathit{RuntimeEnv} \times \overline{\mathit{Value}} \\
\sigma & \in & \mathit{Stack} & = \mathit{ActorAddr} \cdot \overline{\mathit{Frame}} \\
\varphi & \in & \mathit{Frame} & = \mathit{MethID} \times \mathit{RuntimeEnv} \times (\mathit{LocalID} \rightarrow \mathit{Value}) \times \mathit{ExprHole} \\
\Delta & \in & \mathit{RuntimeEnv} & = \mathit{TypeVarId} \rightarrow (\mathit{TypeID} \times \mathit{RuntimeEnv}) \\
 & & \mathit{LocalID} & = \mathit{SourceID} \cup \mathit{TempID} \\
v & \in & \mathit{Value} & = \mathit{Addr} \cup \{\texttt{null}\} \\
\iota & \in & \mathit{Addr} & = \mathit{ActorAddr} \cup \mathit{ObjectAddr} \\
\alpha & \in & \mathit{ActorAddr} & \\
\omega & \in & \mathit{ObjectAddr} &
\end{array}
$$

Figure 4.1: Runtime entities

Figure 4.1 shows the runtime entities used by *Pony*$^{\mathrm{PL}}$. Differences with *Pony*^{0} are highlighted in grey.

The state of a program's execution is represented by a heap, which maps memory addresses to individual actors and objects. Each object or actor contains a map storing the current values of their fields. Values can be either an address to other actors and objects, or null. Compared to objects, actors also carry a sequence of pending messages ($\overline{\mu}$) sent by other actors, as well as a stack (σ) composed of individual frames (φ).

Compared to *Pony*^{0}, we introduce runtime environments (Δ), which map type variables to the runtime representation of how they were instantiated at the object's creation. Each object and actor in the heap carries the environment of its instantiation, binding each of the type variables of the class.

Additionally, each frame in an actor's stack contain a runtime environment, binding all the type variables in scope. These type variables can be originating from either from type parameters of the class definition or from type parameters of the executing method. On method calls, the environment of the object is merged with a new environment representing the method's type arguments. Similarily, messages to actors contain the runtime environment to be used when executing the behaviour.

For example, given the program below, an object of type `C[A ref, C[A ref, B ref]]` would have the following runtime environment

$$\Delta = [\,\mathtt{X} \mapsto (\mathtt{A}, \varnothing), \mathtt{Y} \mapsto (\mathtt{C}, [\,\mathtt{X} \mapsto (\mathtt{A}, \varnothing), \mathtt{Y} \mapsto (\mathtt{B}, \varnothing)\,])\,]$$

Invoking the method `m[B]` on this object would create a new frame on the currently executing actor, with a runtime environment $\Delta' = \Delta \circ [\,\mathtt{Z} \mapsto \mathtt{B}\,]$.

```
class A
class B
class C[X, Y]
  fun m[Z]() => ···
```

In order to determine the real type of a type variable from a runtime environment, we define the *resolve* function described below. For type variables, it simply looks up its value in the environment. For non-variable types, it returns the type name and the runtime environment corresponding to the applied type parameters, by recursively using the function on the arguments.

$$\begin{aligned}
&\mathit{resolve} :: (\mathit{CtorType} \cup \mathit{Type}) \to (\mathit{TypeID} \times \mathit{RuntimeEnv})\\
&\mathit{resolve}(\mathtt{X}, \Delta) = \Delta(\mathtt{X})\\
&\mathit{resolve}\big(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \Delta\big) = \Big(\mathtt{DS}, \overline{\mathcal{T}p(\mathtt{DS})} \mapsto \overline{\mathit{resolve}(\mathtt{T}, \Delta)}\Big)
\end{aligned}$$

The definition of *resolve* is also extended in a straightforward way to include other type syntaxes, by ignoring capabilities.

$$\begin{aligned}
&\mathit{resolve}\big(\mathtt{DS}[\,\overline{\mathtt{T}}\,]\ \kappa, \Delta\big) = \mathit{resolve}\big(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \Delta\big)\\
&\mathit{resolve}(\mathtt{T}\ +, \Delta) = \mathit{resolve}(\mathtt{T}, \Delta)\\
&\mathit{resolve}(\mathtt{T}\ -, \Delta) = \mathit{resolve}(\mathtt{T}, \Delta)\\
&\mathit{resolve}(\mathtt{VP} \rhd \mathtt{T}, \Delta) = \mathit{resolve}(\mathtt{T}, \Delta)\\
&\mathit{resolve}(\mathtt{recover}\ \mathtt{T}, \Delta) = \mathit{resolve}(\mathtt{T}, \Delta)
\end{aligned}$$

4.2 Execution

We describe in Figure 4.2 describe the operational semantics of *Pony*$^{\text{PL}}$.

The rules Ctor and Ator must use *resolve* to determine the runtime type being constructed, from the constructor type `KT`. *resolve* also returns a runtime environment for the type arguments applied to the type. This runtime environment is stored in the constructed object.

All method calls create a new runtime environment Δ' from the type parameters of the call, using *resolve* to remove occurences of type variable in the arguments. This runtime environment is combined with the receiver object's runtime environment Δ, and passed to called method.

For synchronous calls (Ctor and Sync), the combined runtime environment is placed in the newly created frame. For asynchronous calls (Ator and Async), it is placed in the message sent to the remote actor. Upon reception of the message, the Behave rule copies the runtime environment from the received message into the created frame.

Other rules are identical to those of *Pony*$^{\mathrm{GS}}$ and have been reproduced in Figure 4.3.

$$\frac{\begin{array}{c} \iota = \varphi(\mathtt{t}) \\ \mathtt{RS} = \chi(\iota)\downarrow_1 \qquad \Delta = \chi(\iota)\downarrow_2 \\ (\overline{\mathtt{X}}, \overline{\mathtt{x}}, \mathtt{e}) = \mathcal{M}r(\mathtt{RS}, \mathtt{m}) \\ \Delta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathit{resolve}(\mathtt{T}, \varphi\downarrow_2)}\,] \\ \varphi'' = (\mathtt{m}, \Delta \circ \Delta', [\,\mathtt{this} \mapsto \iota, \overline{\mathtt{x}} \mapsto \overline{\varphi(\mathtt{t})}\,], \cdot) \\ \varphi' = (\varphi\downarrow_1, \varphi\downarrow_2, \varphi\downarrow_3, \mathtt{E}\langle\cdot\rangle) \end{array}}{\chi, \sigma \cdot \varphi, \mathtt{E}\left\langle \mathtt{t.m_T}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{t}}) \right\rangle \rightsquigarrow \chi, \sigma \cdot \varphi' \cdot \varphi'', \mathtt{e}} \ \textsc{Sync}
\qquad
\frac{\begin{array}{c} \omega \notin \mathit{dom}(\chi) \\ (\mathtt{C}, \Delta) = \mathit{resolve}(\mathtt{KT}, \varphi\downarrow_2) \\ (\overline{\mathtt{X}}, \overline{\mathtt{x}}, \mathtt{e}) = \mathcal{M}r(\mathtt{C}, \mathtt{k}) \\ \Delta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathit{resolve}(\mathtt{T}, \varphi\downarrow_2)}\,] \\ \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{C}) \\ \chi' = \chi[\,\omega \mapsto (\mathtt{C}, \Delta, \overline{\mathtt{f}} \mapsto \mathtt{null})\,] \\ \varphi'' = (\mathtt{k}, \Delta \circ \Delta', [\,\mathtt{this} \mapsto \omega, \overline{\mathtt{x}} \mapsto \overline{\varphi(\mathtt{t})}\,], \cdot) \\ \varphi' = (\varphi\downarrow_1, \varphi\downarrow_2, \varphi\downarrow_3, \mathtt{E}\langle\cdot\rangle) \end{array}}{\chi, \sigma \cdot \varphi, \mathtt{E}\left\langle \mathtt{KT.k}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{t}}) \right\rangle \rightsquigarrow \chi', \sigma \cdot \varphi' \cdot \varphi'', \mathtt{e}} \ \textsc{Ctor}$$

$$\frac{\begin{array}{c} \alpha = \varphi(\mathtt{t}) \\ \mathtt{A} = \chi(\alpha)\downarrow_1 \qquad \Delta = \chi(\alpha)\downarrow_2 \\ (\overline{\mathtt{X}}, \overline{\mathtt{x}}, \mathtt{e}) = \mathcal{M}r(\mathtt{A}, \mathtt{b}) \\ \Delta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathit{resolve}(\mathtt{T}, \varphi\downarrow_2)}\,] \\ \overline{\mu} = \chi(\alpha)\downarrow_4 \qquad \mu = (\mathtt{b}, \Delta \circ \Delta', \overline{\varphi(\mathtt{t})}) \\ \chi' = \chi[\,\alpha \mapsto \overline{\mu} \cdot \mu\,] \end{array}}{\chi, \sigma \cdot \varphi, \mathtt{t.b_T}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{t}}) \rightsquigarrow \chi, \sigma \cdot \varphi, \mathtt{t}} \ \textsc{Async}
\qquad
\frac{\begin{array}{c} \alpha \notin \mathit{dom}(\chi) \\ (\mathtt{A}, \Delta) = \mathit{resolve}(\mathtt{KT}, \varphi\downarrow_2) \\ (\overline{\mathtt{X}}, \overline{\mathtt{x}}, \mathtt{e}) = \mathcal{M}r(\mathtt{A}, \mathtt{k}) \\ \Delta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathit{resolve}(\mathtt{T}, \varphi\downarrow_2)}\,] \\ \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{A}) \qquad \mu = (\mathtt{k}, \Delta \circ \Delta, \overline{\varphi(\mathtt{t})}) \\ \chi' = \chi[\alpha \mapsto (\mathtt{A}, \Delta, \overline{\mathtt{f}} \mapsto \mathtt{null}, \mu, \alpha, \epsilon)] \\ \mathtt{t} \notin \varphi \qquad \varphi' = \varphi[\mathtt{t} \mapsto \alpha] \end{array}}{\chi, \sigma \cdot \varphi, \mathtt{KT.k}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{t}}) \rightsquigarrow \chi', \sigma \cdot \varphi', \mathtt{t}} \ \textsc{Ator}$$

$$\frac{\begin{array}{c} \mathtt{A} = \chi(\alpha)\downarrow_1 \qquad (\mathtt{n}, \Delta, \overline{v}) \cdot \overline{\mu} = \chi(\alpha)\downarrow_4 \\ (\overline{\mathtt{X}}, \overline{\mathtt{x}}, \mathtt{e}) = \mathcal{M}r(\mathtt{A}, \mathtt{n}) \\ \varphi = (\mathtt{n}, \Delta, [\,\mathtt{this} \mapsto \alpha, \overline{\mathtt{x}} \mapsto \overline{v}, \cdot\,]) \end{array}}{\chi, \alpha, \epsilon \rightsquigarrow \chi[\,\alpha \mapsto \overline{\mu}\,], \alpha \cdot \varphi, \mathtt{e}} \ \textsc{Behave}$$

Figure 4.2: Execution

$$\frac{\chi, \sigma \cdot \varphi, \texttt{e} \rightsquigarrow \chi', \sigma \cdot \varphi', \texttt{e}'}{\chi, \sigma \cdot \varphi, \texttt{E}\langle \texttt{e} \rangle \rightsquigarrow \chi', \sigma \cdot \varphi', \texttt{E}\langle \texttt{e}' \rangle}\ \textsc{ExprHole} \qquad \frac{\chi, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5 \rightsquigarrow \chi', \sigma, \texttt{e}}{\chi \rightarrow \chi'[\alpha \mapsto (\sigma, \texttt{e})]}\ \textsc{Global}$$

$$\frac{}{\chi, \sigma \cdot \varphi, \texttt{t}; \texttt{e} \rightsquigarrow \chi, \sigma \cdot \varphi, \texttt{e}}\ \textsc{Seq}$$

$$\frac{\texttt{t} \notin \varphi \qquad \varphi' = \varphi[\texttt{t} \mapsto \varphi(\texttt{x})]}{\chi, \sigma \cdot \varphi, \texttt{x} \rightsquigarrow \chi, \sigma \cdot \varphi', \texttt{t}}\ \textsc{Local} \qquad \frac{\begin{array}{c} \texttt{t}' \notin \varphi \\ \varphi' = \varphi[\texttt{x} \mapsto \varphi(\texttt{t}), \texttt{t}' \mapsto \varphi(\texttt{x})] \end{array}}{\chi, \sigma \cdot \varphi, \texttt{x} = \texttt{t} \rightsquigarrow \chi, \sigma \cdot \varphi', \texttt{t}'}\ \textsc{AsnLocal}$$

$$\frac{\texttt{t}' \notin \varphi \qquad \varphi' = \varphi[\texttt{t}' \mapsto \chi(\varphi(\texttt{t}), \texttt{f})]}{\chi, \sigma \cdot \varphi, \texttt{t.f} \rightsquigarrow \chi, \sigma \cdot \varphi', \texttt{t}'}\ \textsc{Fld} \qquad \frac{\begin{array}{c} \texttt{t}'' \notin \varphi \qquad \varphi' = \varphi[\texttt{t}'' \mapsto \chi(\varphi(\texttt{t}), \texttt{f})] \\ \chi' = \chi[\varphi(\texttt{t}), \texttt{f} \mapsto \varphi(\texttt{t}')] \end{array}}{\chi, \sigma \cdot \varphi, \texttt{t.f} = \texttt{t}' \rightsquigarrow \chi', \sigma \cdot \varphi', \texttt{t}''}\ \textsc{AsnFld}$$

$$\frac{\texttt{t} \notin \varphi \qquad \varphi' = \varphi[\texttt{t} \mapsto \texttt{null}]}{\chi, \sigma \cdot \varphi, \texttt{null} \rightsquigarrow \chi, \sigma \cdot \varphi', \texttt{t}}\ \textsc{Null} \qquad \frac{\varphi(\texttt{t}) = \texttt{null}}{\begin{array}{c} \chi, \sigma \cdot \varphi, \texttt{t.f} \rightsquigarrow \chi, \sigma \cdot \varphi, \texttt{t} \\ \chi, \sigma \cdot \varphi, \texttt{t.f} = \texttt{t}', \rightsquigarrow \chi, \sigma \cdot \varphi, \texttt{t} \\ \chi, \sigma \cdot \varphi, \texttt{t.n}(\overline{\texttt{t}}) \rightsquigarrow \chi, \sigma \cdot \varphi, \texttt{t} \end{array}}\ \textsc{Except}$$

$$\frac{\begin{array}{c} \texttt{E}\langle \cdot \rangle = \varphi \downarrow_3 \qquad \texttt{t}' \notin \varphi \\ \varphi'' = (\varphi \downarrow_1, \varphi \downarrow_2 [\texttt{t}' \mapsto \varphi'(t)], \cdot) \end{array}}{\chi, \sigma \cdot \varphi \cdot \varphi', \texttt{t} \rightsquigarrow \chi, \sigma \cdot \varphi'', \texttt{E}\langle \texttt{t}' \rangle}\ \textsc{Return} \qquad \frac{}{\chi, \alpha \cdot \sigma, \texttt{t} \rightsquigarrow \chi, \alpha, \epsilon}\ \textsc{ReturnBe}$$

Figure 4.3: Execution (continued)

4.3 Implementation

In order to confirm that our semantics match the expected behaviour, we have implemented an interpreter which follows our semantics. The interpreter is written in Prolog, as the logic programming paradigm of Prolog allows the implementation to follow a similar structure as the rules described above.

We've reproduced below a couple of excerpts from the implementation. The full implementation is available in the source archive accompanying this report.

The exceprt below shows the main execution loop. The `run` predicate executes the actors contained in a heap until termination, which happens when there are no more active actors. On each iteration it uses the `step` predicate to perform a single execution step, and dumps the current state of the heap to the console. The `step` predicate corresponds to the Global execution rule. It picks a random actor from the list of active actors, and executes a single step in that actor, using the `eval` predicate. The modified stack and expression is then written back to the heap.

```
run(_Program, Heap, Heap) :-
    heap:active_actors(Heap, []).

run(Program, Heap0, HeapFinal) :-
    step(Program, Heap0, Heap1),
    heap:dump(Heap1), nl,
    run(Program, Heap1, HeapFinal).

% Global
step(Program, Heap0, Heap2) :-
    heap:active_actors(Heap0, ActiveActors),
    heap:random_member(ActorAddr-Actor0, ActiveActors),
    Actor0 = (actor, ActorId, _, _, _, Stack0, Expr0),

    eval(Program, (Heap0, Stack0, Expr0), (Heap1, Stack1, Expr1)),

    heap:update_actor_state(Heap1, ActorAddr, Stack1, Expr1, Heap2).
```

The `eval` predicate executes a single step in a given actor, potentially modifying the heap, the actor's stack and expression. There is a separate rule for the predicate for each rule of our semantics. For example, the excerpt below show the ASYNC rule, which invokes a behaviour on a remote actor. The implementation follows very closely the rule as described in the previous section.

```
% Async
eval(Program, (Heap0, [Frame|Stack], expr(call_method(term(ActorTerm),
                                                        MethodId,
                                                        TyValues,
                                                        Args))),
              (Heap, [Frame|Stack], term(ActorTerm))) :-
    frame:get(Frame, ActorTerm, ActorAddr),
    heap:get(Heap0, ActorAddr, Actor),
    Actor = (actor, ActorId, _, _, _, _, _),

    program:method(Program, ActorId, MethodId, (actor, be, _, _, TyArgs)),
    maplist(\T^V^(T-R)^(frame:resolve(Frame, Program, V, R)),
            TyArgs, TyValues, EnvList),
    list_to_assoc(EnvList, MethodTyEnv),

    maplist(\term(Term)^V^(frame:get(Frame, Term, V)), Args, ArgValues),

    heap:queue(Heap0, ActorAddr, (MethodId, MethodTyEnv, ArgValues), Heap).
```

Example trace

The interpreter can be used to produce an execution trace of the program. We can use the following program, which makes use of invocaton of constructors on type variables.

```
actor A
  var f: Main
  new create(x: Main) =>
    this.f = x ; this

class B[X]
  new create(x: Main) =>
    X.create(x) ; this

actor Main
  new create() =>
    B[A].create(this) ; this
```

The initial program state is shown below, and consist of a single actor of type `Main` with a single frame executing the `create` constructor.

```
#0 : actor Main ()
 *  create
    - this = #0
```

After a number of steps, the program has created an instance of the class `B[A]`, whose constructor is executing on the main actor. The runtime environment of the object is show in brackets on line 8 below.

```
1  #0 : actor Main ()
2   *  create
3      - this = #1
4      - "x" = #0
5   *  create
6      - 0 = #0
7      - this = #0
8  #1 : class B[X="A"]
```

The object's constructor will execute the `X.create(x)` expression, which resolves the type to construct through its environment, and creates a new actor of type `A`, and sends it a message to invoke its constructor.

```
#0 : actor Main ()
 *  create
    - 0 = #0
    - 1 = #2
    - this = #1
```

```
        - "x" = #0
  *   create
        - 0 = #0
        - this = #0
#1 : class B[X="A"]
#2 : actor A (create([#0]))
-   f = null
```

Eventually, all the behaviours and methods will have terminated, and the program reaches the following expected state:

```
#0 : actor Main ()
#1 : class B[X="A"]
#2 : actor A ()
-   f = #0
```

Chapter 5

Typing rules

5.1 Approach

Before we describe the typing rules for *Pony*$^{\text{PL}}$, we first give in this section two properties we want from our rules, and present a few approaches to typing generics.

5.1.1 Desired properties

Extensibility

The first property we want from our typing rules is extensibility of programs. Given a well formed program, adding new definitions should not cause any of the preexisting ones to become ill-formed. This is important to ensure programs can be written incrementally.

There are various levels of extensibility, depending on what kind of definitions can be added and how they have to be added in order for existing definitions to continue to be well typed. There is a tradeoff between how many programs the rules accept, and how extensible these programs are.

For instance, programs written in the Rust language are only partially extensible, as there are situations where adding a new definition can cause existing ones to become ill-formed. In Rust, implementations of traits are separate from the definition of the types. However, implementations must not *overlap*, that is there must be at most one implementation for each pair of type and trait.

In the example below, on line 5 we provide an implementation of `N1` for `A`, and on line 6 we provide one for any type `X` which already implements `N2`. The two implementations do not overlap only because `A` does not implement `N2`.

```
struct A {}
trait N1 {}
trait N2 {}

impl N1 for A {}
impl<X: N2> N1 for X {}
```

However, extending the program with an implementation of `N2` for `A`, as shown below on line 8, causes the two implementations on lines 6 and 7 to overlap for `A`, making the program ill-formed.

```
1 struct A {}
2 trait N1 {}
3 trait N2 {}
4
5 // error: overlapping implementations of A for N1
6 impl N1 for A {}
7 impl<X: N2> N1 for X {}
8
9 impl N2 for A {}
```

The Rust language mitigates this through *coherence* rules which restrict how and where trait implementations may be defined. Through these rules, adding new implementations can only create overlaps in the current package, not in others.

Modularity

We also want our system to allow for modularity, wherei changing a method's body should not cause any existing, well formed uses of the method to become ill-formed, as long as the signature of the method is not modified. This allows different parts of a program to be developed independently, as long as the signature of methods has been agreed upon.

Templates in C++ are an example of poor modularity. A use of a function template is correct only if after replacing the type variables with their instantiation, the function is well-formed. For example below, the call to the `baz` method on line 10 is only correct because, when replacing `X` by `A`, the function is well-formed.

```
 1  class A {
 2    void foo() {}
 3  }
 4  template<typename X> baz(X x) {
 5    x.foo()
 6  }
 7
 8  void main() {
 9    A a;
10    baz<A>(a);
11  }
```

It is however impossible to determine what requirements exist on the type argument without looking into the body of the `baz` function, and these may change without modifications to the signature. For example, if the `baz` function is changed to call the `bar` method on `x`, then its use on line 10 becomes ill-formed since the type `A` does not provide such a method.

```
 1  class A {
 2    void foo() {}
 3  }
 4  template<typename X> baz(X x) {
 5    x.bar()
 6  }
 7
 8  void main() {
 9    A a;
10    baz<A>(a); // error: no method named foo in class A
11  }
```

The *concepts* proposal for the C++ language adds a way to define better modularity on templates.

5.1.2 Delayed typing

The first approach to typing generics we present is inspired by the design of C++ and D templates. Under this model, generic definitions are always well formed, as long as there are syntactically correct of course. Instead, typing is delayed until the definition is instantiated. After instantiation, all occurences of type variables have been replaced by concrete types, and the typing rules from *Pony*^{0} can be reused with just a few changes.

In the example below, the definition of `A` is always well formed. However, while instantiating it with `B ref` is allowed, it cannot be instantiating with `C ref`, as shown on lines 11 and 14.

```
class A[X]
  new create() => ···
  fun m(x: X) => x.foo()
class B
  fun foo() => ···
class C
  fun bar() => ···

class Main
  new create() =>
    let ab : A[B ref] = A[B ref].create()

    // error: no method named foo in class C
    let ac : A[C ref] = A[C ref].create()
```

While this approach is both simple to formalise and is the one which allows the most programs, it does not allow modularity between different definitions, just like C++ and D templates. While the instantiation `A[B ref]` is allowed in the above, it may not be if the body of `A` is changed, even without changing signatures.

5.1.3 Exhaustive typing

In order to allow good modularity, we want to use bounds on type variables, and ensure any instantiation which fits these bounds is well-formed. One approach in doing so is to instantiate the definition exhaustively and make sure it is always well-formed.

In the program below, the generic class `A` would be well-formed is for every possible instantiation of `X`, the type of the method `m`'s body is a subtype of the return type. The only possible instantiations of `X` are `N` κ, `B` κ and `A[T]` κ for any type `T` and capability κ, since these are the only types defined in the program. Because these are all subtypes of `N tag`, the class `A` would be well-formed under this approach.

```
trait N
class A[X] is N
  fun m(x: X) : N tag => x
class B is N
```

This approach has two major inconvenients. First of all there may exist an infinite number of instantiations, such as `B ref`, `A[B ref] ref`, `A[A[B ref] ref] ref`, ... While, in this example, it is easy to show they are all subtypes of `N tag`, it may not always be the case making it impossible to check all instantiations exhaustively.

Additionally, this approach does not provide the extensibility of programs we desire. Indeed, extending the program with a new class `C` which does not implement `N`, as shown below, creates an instantiation of `A[X]` which is not well-formed.

```
trait N
class A[X] is N
  // error: possible instantiation C tag of X is not a subtype of N tag
  fun m(x: X) : N tag => x
class B is N
class C
```

5.1.4 Abstract typing

This approach treats type variables in an abstract way, without instantiating them. This way the well-formness of a definition cannot be affected by adding new types to the program, preserving extensibility.

Unlike the approaches we've described earlier, this one needs to be able determine well-formness of uninstantiated definitions. This requires new typing rules which to handle type variables. For example, we need to define method and field lookup on expression whose type is a variable, and define subtyping in the presence of variable.

In the example below, in order for the `foo` method in line 4 to be well-formed, the type of the body, `X`, must be a subtype of the return type, `N ref`. Similarily, in order for the `bar` method on line 5 to be well-formed, the `X` type must expose a `m` method.

```
1 trait N
2   fun m()
3 class A[X: N #read]
4   fun foo(x: X) : N box => x
5   fun bar(x: X) => x.m()
```

In both cases, under this approach, these are allowed because of the bounds on the type variable `X`. A type variable is a subtype of its bound, and lookup of methods on type variables is performed on their bound. In the previous example, both methods are ill-formed if the bound on `X` is removed, as shown below.

```
trait N
  fun m()
class A[X]
  // error: X is not a subtype of N box
  fun foo(x: X) : N box => x
  // error: no method m defined on type X
  fun bar(x: X) => x.m()
```

This approach is used successfully by other languages such as Java and C#. However, the presence of type operators in *Pony*$^{\mathrm{PL}}$ make it hard to define subtyping in an abstract way. While we can easily establish in the example above that `X` is a subtype of `N box` based on its bound, it is less straightforward to determine whether for example `this▷X+` should be a subtype `N box`, and under which conditions.

While it would probably be possible to find such rules, there would need to be a large number of them in order to handle all possible shapes of types while still being both sound and flexible.

5.1.5 Partially reified typing

We introduce a new approach to typing generics based on partial reification, which is a combination of the abstract and the exhaustive approaches described previously. Compared to the other approaches, it preserves both extensibility and allow modularity, while being much simpler to formulate than abstract typing.

This approach is similar to the abstract typing one. Type variables are kept abstract, and their bounds are used to perform method and field lookup. The abstract typing approach however required subtyping to be defined for type variables as well, which would have been hard to define in the presence of type operators.

Concrete types in Pony are comprised of a type identifier and a capability. Partial reification of type variables keeps the type identifier abstract, but assigns concrete capabilities exhaustively based on the variable's bound. For example, a type variable `X` bound by `N #read` would be reified into `X box`, `X val` and `X ref`. Because type operators can be reduced away when all the capability are known, it is not necessary to define subtyping rules to operate directly on the operators. Instead they can be defined in terms of subtyping on the reified and reduced types, which is much simpler.

For example, in the type `this▷X+`, where `X` is bounded by `N #send`, `this` can be reified into any of `box`, `val` or `ref`. Similarily, `X` can be reified into any of `X tag`, `X val` or `X iso`. Applying these reifications to the original type and reducing the operators results in only two types, `X tag` and `X val`.

While partial reification exhaustively assigns capabilities to type variables, there is only a fixed, small number of capabilities. It is therefore trivial to enumerate the set of possible reifications, unlike the exhaustive typing approach which required enumerating the potentially infinite set of possible instantiations. Additionally, partial reification preserves extensibility of programs since adding new types does not create new reifications which must be taken into account.

On the other hand, partial reification may make the language itself, as opposed to programs, less extensible than under the abstract typing approach. Indeed, introducing new capabilities would introduce new possible reifications, and previously well-formed definitions may not be so anymore afterwards. Adding new types of reference capabilities to the language is however a much more fundamental change than simply adding new definitions to a program, and it seems reasonable that such a change may cause existing programs to become ill-formed.

5.2 Introduction

We will introduce, in Section 5.3, the typing rules of expressions for $Pony^{\mathrm{PL}}$. These rules however depend on a number of functions and relations for which we first give an overview. We will come back to these and define them formally later in this chapter.

Typing contexts and environments

The typing of expressions is dependent on the types of the local variables in scope. We represent these with as a typing context, which we note Γ. It is a finite mapping from variable names to the corresponding type.

$$\Gamma \in \mathit{TypeCtx} = \mathit{LocalId} \rightarrow \mathit{Type}$$

With the introduction of generics, typing of expressions is also dependent on a typing environment β, which maps type variables to their bounds, and within methods maps the `this` viewpoint to a capability bound.

$$\begin{gathered}\beta \in \mathit{TypeEnv} = \mathit{TypeVarID} \cup \{\mathtt{this}\} \rightarrow \mathit{TypeBound} \cup \mathit{CapBound} \\ \forall \mathtt{X}.\ \mathtt{X} \in \beta \rightarrow \beta(\mathtt{X}) \in \mathit{TypeBound} \\ \mathtt{this} \in \beta \rightarrow \beta(\mathtt{this}) \in \mathit{CapBound}\end{gathered}$$

Upper bound

In order to determine the type of field accesses, method calls and constructor invocations, we must first lookup their signature in their type's definition, using the lookup rules presented in section Section 5.14. However, the type used to lookup may be a type variable, such as in the example below.

```
1 class A[X: Y+, Y: B ref]
2   fun m(x: X) =>
3     x.n()
4 class B
5   fun n() => ···
```

We call the upper bound of a type variable the non-variable basic type which transitively bounds the variable. When typing a method call for instance, its signature is looked up on this upper bound instead. For example, on line 3 above, the method's signature is looked on the upper bound of `X`, which is `B`.

$$\mathit{upperBound}_{\beta} :: \mathit{Type} \rightarrow (\mathit{TypeID} \times \overline{\mathit{BasicType}})$$

Subtyping

Pony supports *subsumption*, which allows a value of type `T` to be used where a value of type `T′`, as long as `T` has the same properties as `T′`, in which case we call `T` a subtype of `T′`. We use the notation

$$\beta \vdash \mathtt{T} \leqslant \mathtt{T'}$$

to indicate that `T` is a subtype of `T′`, in an environment β. The judgment's shape reflects the dependency of subtyping on the type environment.

In the example below, the `bar` method on line 8 expects an argument of type `B box`. The `foo` method calls the `bar` method by providing `x` as an argument. Even though `x` has type `A ref`, this method call is well-formed because `A ref` is a subtype of `B box`. Indeed the class `A` implements the trait `B`, and a reference with a `ref` capability provides strictly more guaranteed than a `box` reference does.

```
 1  class A is B
 2  trait B
 3
 4  class Main
 5    fun foo(x: A ref) =>
 6      this.bar(x)
 7
 8    fun bar(x: B box) => ···
```

Bound compliance

Type variables in Pony have an associated bound which determines what type they may be instantiated with. We say a type `T` is compliant with a bound `BT` if it can be used to instantiate a type variable bound by it.

In the example below, the class `A` can only be instantiated with a type compliant with the bound `B #read`. The instantiation with type `C ref` on line 9 is correct, whereas the one on the following line are not as neither `C tag` nor `D ref` are compliant with the bound.

```
 1  class A[X: B #read]
 2    new create() => ···
 3  trait B
 4  class C is B
 5  class D
 6
 7  class Main
 8    new create() =>
 9      A[C ref].create()
10      A[C tag].create() // error: C tag is not compliant with bound B #read
11      A[D ref].create() // error: D ref is not compliant with bound B #read
```

Since both the type and the bound may contain type variables, the relation is dependent on the type environment. We use the notation

$$\beta \vdash \mathtt{T} \lll \mathtt{BT}$$

if T is compliant with BT given the environment β.

5.3 Expression Typing

Expression typing depends directly on the typing context to determine the type of variables, and indirectly on the typing environment through the relations presented in the previous section. We therefore use the following judgment to express that e has type T in the type context Γ and environment β

$$\Gamma; \beta \vdash \mathtt{e} : \mathtt{T}$$

The typing rules for expressions are given in Figure 5.1.

- The base type for method calls and field access may be a type variable. To account for this, the corresponding rules make use of *upperBound* when looking up fields and methods of a type.
- In *Pony*^{0}, the rules T-AsnLocal, T-Fld, T-AsnFld and T-Alias each modify the type's capability. In *Pony*$^{\mathrm{PL}}$ however, types' capabilities are not be known until all type variables are instantiated. Instead, the new syntaxes we've introduced for type operators are used to construct types which encode the desired modification to capabilities. These operators are only reduced once all the type variables are instantiated through reification, as described in Section 5.5
- The rules T-Call and T-Ctor must ensure type arguments, and the receiver type in the case of method calls, are compliant with their bound. Note that these do not use regular subtyping $\leqslant$, but a different relation $\lll$ which does not allow capability subtyping, as explained in Section 2.3.4.
- The field access, method call and constructor call rules pass type arguments to the corresponding lookup rules, which are responsible for replaces occurences of type variables in the signatures with the given type arguments.

5.4 Upper bound

The function *upperBound*$_{\beta}$, introduced in Section 5.2, determines a type's upper bound and is defined below. The upper bound is used to lookup signatures of fields and methods.

$$\frac{x \in \Gamma}{\Gamma; \beta \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \textsc{T-Local} \qquad \frac{\Gamma(\mathtt{x}) = \mathtt{T} \qquad \Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T}}{\Gamma; \beta \vdash \mathtt{x} = \mathtt{e} : \mathtt{T}-}\ \textsc{T-AsnLocal}$$

$$\frac{\begin{array}{c}\Gamma; \beta \vdash \mathtt{e} : \mathtt{T} \\ \mathcal{F}(\mathit{upperBound}_\beta(\mathtt{T}), \mathtt{f}) = \mathtt{T}'\end{array}}{\Gamma; \beta \vdash \mathtt{e.f} : \mathtt{T} \rhd \mathtt{T}'}\ \textsc{T-Fld} \qquad \frac{\begin{array}{c}\Gamma; \beta \vdash \mathtt{e} : \mathtt{T} \\ \mathcal{F}(\mathit{upperBound}_\beta(\mathtt{T}), \mathtt{f}) = \mathtt{T}' \\ \Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e}' : \mathtt{T}'' \qquad \beta \vdash \mathtt{T}'' \leqslant \mathtt{T}' \\ \beta \vdash \mathtt{T} \lhd \mathtt{T}''\end{array}}{\Gamma; \beta \vdash \mathtt{e.f} = \mathtt{e}' : (\mathtt{T} \rhd \mathtt{T}')-}\ \textsc{T-AsnFld}$$

$$\frac{\Gamma; \beta \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma; \beta \vdash \mathtt{e}' : \mathtt{T}'}{\Gamma; \beta \vdash \mathtt{e}; \mathtt{e}' : \mathtt{T}'}\ \textsc{T-Seq}$$

$$\frac{\begin{array}{c}\Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T} \\ \mathcal{M}d(\mathit{upperBound}_\beta(\mathtt{T}), \mathtt{n}[\mathtt{T}; \overline{\mathtt{T}}]) = (\mathtt{BT}, \overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{x}} : \overline{\mathtt{T}'}, \mathtt{T}') \\ \beta \vdash \mathtt{T} \lll \mathtt{BT} \qquad \beta \vdash \mathtt{T}_i \lll \mathtt{BT}_i \qquad \Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e}_i : \mathtt{T}'_i\end{array}}{\Gamma; \beta \vdash \mathtt{e.n}[\mathtt{T}; \overline{\mathtt{T}}](\overline{\mathtt{e}}) : \mathtt{T}'}\ \textsc{T-Call}$$

$$\frac{\begin{array}{c}\mathcal{K}d(\mathit{upperBound}_\beta(\mathtt{KT}), \mathtt{k}[\overline{\mathtt{T}}]) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{x}} : \overline{\mathtt{T}'}, \kappa) \\ \beta \vdash \mathtt{T}_i \preceq \mathtt{BT}_i \qquad \Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e}_i : \mathtt{T}'_i\end{array}}{\Gamma; \beta \vdash \mathtt{KT.k}[\overline{\mathtt{T}}](\overline{\mathtt{e}}) : \mathtt{KT}\ \kappa}\ \textsc{T-Ctor}$$

$$\frac{\Gamma; \beta \vdash \mathtt{e} : \mathtt{T}' \qquad \beta \vdash \mathtt{T}' \leqslant \mathtt{T}}{\Gamma; \beta \vdash_{\mathcal{S}} \mathtt{e} : \mathtt{T}}\ \textsc{T-Subsume} \qquad \frac{\Gamma; \beta \vdash_{\mathcal{S}} \mathtt{e} : \mathtt{T}}{\Gamma; \beta \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T}+}\ \textsc{T-Alias}$$

$$\frac{\mathtt{DS} \in \mathtt{P} \qquad \beta \vdash (\mathtt{DS}[\overline{\mathtt{T}}]\ \mathtt{iso}\circ)\ \diamond}{\Gamma; \beta \vdash \mathtt{null} : \mathtt{DS}[\overline{\mathtt{T}}]\ \mathtt{iso}\circ}\ \textsc{T-Null}$$

Figure 5.1: Expression typing

The $\mathit{upperBound}_\beta$ function recursively replaces type variables with their bound, until a non-variable type is reached. Capabilities and type operators, as these are irrelevant to method lookup.

$$\mathit{upperBound}_\beta :: \mathit{Type} \rightarrow \mathit{RawType}$$
$$\mathit{upperBound}_\beta(\mathtt{X}) = \mathit{upperBound}_\beta(\beta(\mathtt{X}))$$

The definition of *upperBound* is extended to include other type syntaxes, by ignoring their capability.

$$\begin{aligned}
&\mathit{upperBound}_\beta(\mathtt{DS}[\overline{\mathtt{T}}]\ \nu) = \mathtt{DS}[\overline{\mathtt{T}}]\\
&\mathit{upperBound}_\beta(\mathtt{X}\ \kappa) = \mathit{upperBound}_\beta(\mathtt{X})\\
&\mathit{upperBound}_\beta(\mathtt{T}+) = \mathit{upperBound}_\beta(\mathtt{T})\\
&\mathit{upperBound}_\beta(\mathtt{T}-) = \mathit{upperBound}_\beta(\mathtt{T})\\
&\mathit{upperBound}_\beta(\mathtt{VP} \rhd \mathtt{T}) = \mathit{upperBound}_\beta(\mathtt{T})\\
&\mathit{upperBound}_\beta(\mathtt{recover}\ \mathtt{T}) = \mathit{upperBound}_\beta(\mathtt{T})
\end{aligned}$$

If a type variable's bounds form a cycle, such as in the example below, then the upper bound is undefined. Without an upper bound, it is not possible to lookup field and method signatures on the type, making any field access or method call on an expression of this type invalid

```
class A[X: Y, Y: X]
  fun m(x: X) =>
    // error: type X has no upper bound
    x.n()

    // error: type X has no upper bound
    x.f
```

5.5 Reification

In Sections 5.7 to 5.11, we will define a numer of relations and properties on types, such as subtyping, bound compliance, or sendability. These are similar relations and properties as introduced by [Clebsch et al., 2015] and [Steed, 2016]. Both of these model have a single form of type, $\mathtt{DS}\ \kappa$, comprised of a type identifier and a capability.

In *Pony*$^{\mathrm{PL}}$ however, we've introduced more forms of types, through the addition of type variables and type operators. These complex form of types make it more complicated to define many of the relations and definitions directly. This would involve a lot of different rules to handle the various forms. Additionally, these numerous rules would make a proof of soundness more complicated.

We've instead followed a different approach, which assigns concrete capability to each type variable. Since each type variable admits multiple capabilities there can be multiple such assignments. Given a specific assignment, it is possible to reduce the type operators can be reduced as these only represent manipulation of capabilities, which are known thanks to the assignment. Reduction leaves the types into a form `BRT` κ. The relations we are interested in are much easier to define on types of this form than on the form general of types. A relation holds on some types only if holds on the reduced types for all acceptable assignments.

For example, behaviours require their arguments to be sendable from one actor to actor to an other. In order for the actor `A` below to be well-formed, the type of the argument of the `b` behaviour, **`val`**`⊳X`, must be sendable.

```
trait N
actor A[X: N #alias]
  be m(x: val⊳X) => ···
```

Given the environment $\beta = [$ `X` $\mapsto$ `N #alias` $]$, the type variable `X` accepts as capabilities any of **`tag`**, **`box`**, **`val`** and **`ref`**. The behaviour's argument's type **`val`**`⊳X` is therefore rewritten as **`val`**`⊳(X` **`tag`**`)`, **`val`**`⊳(X` **`box`**`)`, **`val`**`⊳(X` **`val`**`)` and **`val`**`⊳(X` **`ref`**`)`. Following the rules we define in Section 5.5.1, these reduce into either `X` **`tag`** or `X` **`val`**, which are both sendable types. The type **`val`**`⊳X` is thus sendable.

5.5.1 Type Reduction

We call a *partial reification* π a mapping which assigns each type variable a concrete capability. The partial reification may also map **`this`** to a concrete capability.

$$\pi \in \mathit{TypeVarID} \cup \{\texttt{this}\} \to \mathit{Cap}$$

Given a partial reification π, type reduction annotates type variables with the capability assigned in π. Reduction also replaces type operators using their corresponding definitions, which we describe in Figures 5.4 and 5.5. This is possible since all the operators represent a manipulation of capabilities, and the partial reification has given a value to all of them.

The reduction rules are described below in Figure 5.2. We use the notation

$$\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa$$

if the type `T` reduces to `BRT` κ given the partial reification π, where `BRT` is a *basic reified type*, as described in Section 3.3.1.

$$\frac{\pi(\texttt{X}) = \kappa}{\pi \vdash \texttt{X} \Downarrow \texttt{X}\ \kappa} \qquad \frac{}{\pi \vdash \texttt{X}\ \kappa \Downarrow \texttt{X}\ \kappa} \qquad \frac{\pi \vdash \overline{\texttt{T}} \Downarrow \overline{\texttt{BRT}\ \kappa}}{\pi \vdash \texttt{DS[}\overline{\texttt{T}}\texttt{]}\ \kappa \Downarrow \texttt{DS[}\overline{\texttt{BRT}\ \kappa}\texttt{]}\ \kappa}$$

$$\frac{\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa}{\pi \vdash \texttt{T+} \Downarrow \texttt{BRT}\ \mathcal{A}(\kappa)} \qquad \frac{\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa}{\pi \vdash \texttt{T}- \Downarrow \texttt{BRT}\ \mathcal{U}(\kappa)} \qquad \frac{\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa}{\pi \vdash \texttt{recover T} \Downarrow \texttt{BRT}\ \mathcal{R}(\kappa)}$$

$$\frac{\begin{array}{c}\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa \\ \pi \vdash \texttt{T}' \Downarrow \texttt{BRT}'\ \kappa'\end{array}}{\pi \vdash \texttt{T} \rhd \texttt{T}' \Downarrow \texttt{BRT}'\ \mathcal{V}p(\kappa, \kappa')} \qquad \frac{\pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa'}{\pi \vdash \kappa \rhd \texttt{T} \Downarrow \texttt{BRT}\ \mathcal{V}p(\kappa, \kappa')} \qquad \frac{\begin{array}{c}\pi(\texttt{this}) = \kappa \\ \pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa'\end{array}}{\pi \vdash \texttt{this} \rhd \texttt{T} \Downarrow \texttt{BRT}\ \mathcal{V}p(\kappa, \kappa)}$$

Figure 5.2: Reduction of types with a partial reification

For example, given the partial reification $\pi = [\texttt{X} \mapsto \textbf{ref}, \textbf{this} \mapsto \textbf{box}]$, then **this**$\rhd$X reduces to X **box**. Not that reduction is deep and applies to type arguments as well, so for example A[**this**$\rhd$X] **ref** reduces to A[X **box**] **ref**.

Note that viewpoint adaptation is not defined when the viewpoint is tag, making reduction of types where such a viewpoint occurs undefined. For example, given $\pi = [\texttt{X} \mapsto \texttt{tag}]$, neither of **tag**$\rhd$A **ref** nor X$\rhd$A **ref** are reducible.

We overload reduction in Figure 5.3 to be defined on bounds as well.

$$\frac{\pi(\texttt{X}) = \kappa}{\pi \vdash \texttt{X} \Downarrow \texttt{X}\ \kappa} \qquad \frac{\pi \vdash \overline{\texttt{T}} \Downarrow \overline{\texttt{BRT}\ \kappa}}{\pi \vdash \texttt{DS[}\overline{\texttt{T}}\texttt{]}\ \nu \Downarrow \texttt{DS[}\overline{\texttt{BRT}\ \kappa}\texttt{]}\ \nu}$$

$$\frac{\pi \vdash \texttt{BT} \Downarrow \texttt{BRT}\ \nu}{\pi \vdash \texttt{BT+} \Downarrow \texttt{BRT}\ \mathcal{A}(\nu)} \qquad \frac{\pi \vdash \texttt{BT} \Downarrow \texttt{BRT}\ \nu}{\pi \vdash \texttt{BT}- \Downarrow \texttt{BRT}\ \mathcal{U}(\nu)}$$

Figure 5.3: Reduction of bounds with a partial reification

Since the reduction rules are syntax directed, we can define the function $reduce_\pi$ below. The function is overloaded for both types and bounds.

$$\begin{array}{l} reduce_\pi :: Type \rightarrow ReifiedType \\ reduce_\pi(\texttt{T}) = \texttt{BRT}\ \kappa \quad \textit{iff}\ \pi \vdash \texttt{T} \Downarrow \texttt{BRT}\ \kappa \\ \\ reduce_\pi :: Bound \rightarrow ReifiedBound \\ reduce_\pi(\texttt{BT}) = \texttt{BRT}\ \nu \quad \textit{iff}\ \pi \vdash \texttt{BT} \Downarrow \texttt{BRT}\ \nu \end{array}$$

The $reduce_\pi$ function is partial, and is undefined if the type cannot be reduced due to tag being used as a viewpoint. For example, if $\pi = [\mathtt{X} \mapsto \mathtt{tag}]$ then $\mathit{reduce}_\pi(\mathtt{X} \rhd \mathtt{A}\ \mathtt{ref})$ is undefined.

Aliasing and unaliasing of capabilities

$$\mathcal{A}(\nu) = \begin{cases} \mathtt{tag} & \textit{iff } \nu = \mathtt{iso} \\ \mathtt{box} & \textit{iff } \nu = \mathtt{trn} \\ \mathtt{iso} & \textit{iff } \nu = \mathtt{iso}\circ \\ \mathtt{trn} & \textit{iff } \nu = \mathtt{trn}\circ \\ \#\mathtt{alias} & \textit{iff } \nu = \#\mathtt{any} \\ \#\mathtt{share} & \textit{iff } \nu = \#\mathtt{send} \\ \#\mathtt{any} & \textit{iff } \nu = \#\mathtt{any}\circ \\ \#\mathtt{send} & \textit{iff } \nu = \#\mathtt{send}\circ \\ \nu & \textit{otherwise} \end{cases} \qquad \mathcal{U}(\nu) = \begin{cases} \mathtt{iso}\circ & \textit{iff } \nu = \mathtt{iso} \\ \mathtt{trn}\circ & \textit{iff } \nu = \mathtt{trn} \\ \#\mathtt{any}\circ & \textit{iff } \nu = \#\mathtt{any} \\ \#\mathtt{send}\circ & \textit{iff } \nu = \#\mathtt{send} \\ \nu & \textit{otherwise} \end{cases}$$

Figure 5.4: Aliasing and unaliasing of capabilities

We extend the definition of aliasing and unaliasing from $Pony^{\text{GS}}$ to support capability constraints. For each constraint, these definition are obtained by aliasing or unaliasing individual concrete capabilities it allows. For example, #`send` allows conrete capabilities `iso`, `val` and `tag`, which respectively alias as `tag`, `val` and `tag`. The alias of #`send` is therefore #`share`, since it allows exactly `tag` and `val`. The same reasoning is used to obtain the aliasing and unaliasing of each constraint, which are enumerated in Figure 5.4.

Viewpoint adaptation

$\mathcal{V}p(\kappa, \kappa')$	κ'					
κ	`iso`	`trn`	`ref`	`val`	`box`	`tag`
`iso`	`iso`	`tag`	`tag`	`val`	`tag`	`tag`
`trn`	`iso`	`trn`	`box`	`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`	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 5.5: Viewpoint Adaptation

Since bounds do not allow viewpoint adaptation, we do not need to define viewpoint adaptation on capability constraints, unlike aliasing and unaliasing. The definition is therefore identical to the one used in $Pony^{\mathrm{GS}}$, and is reproduced in Figure 5.5.

5.5.2 Reification set

In order to define relation on types, we are interested in finding all the reifications of types. This requires finding all the reifications allowed by an environment.

The $capBound_{\pi,\beta}$ function, defined below, determines which capabilities a type variable may take. Because type variables may be bounded by other type variables, the capability bound depends on the partial reification π in addition to environment β.

$$\begin{aligned}&capBound_{\pi,\beta} :: TypeVar \to CapBoundMod\\ &capBound_{\pi,\beta}(\texttt{X}) = \nu \quad \textit{iff } reduce_{\pi}(\beta(\texttt{X})) = \texttt{BRT}\ \nu\end{aligned}$$

For example, given the following type environment and partial reification,

$$\begin{aligned}\beta &= [\,\texttt{X} \mapsto \texttt{X}, \texttt{Y} \mapsto \texttt{Z+}, \texttt{Z} \mapsto \texttt{Any \#any}\,]\\ \pi &= [\,\texttt{X} \mapsto \texttt{X ref}, \texttt{Y} \mapsto \texttt{Y box}, \texttt{Z} \mapsto \texttt{Z trn}\,]\end{aligned}$$

then the capability bounds of `X`, `Y` and `Z` are

$$capBound_{\pi,\beta}(\texttt{X}) = \texttt{ref} \qquad capBound_{\pi,\beta}(\texttt{Y}) = \texttt{box} \qquad capBound_{\pi,\beta}(\texttt{Z}) = \texttt{\#any}$$

Since `X` and `Y`'s bounds depend on type variables, their capability bound is a concrete capability, based on the other variable's assigned capability in π. On the other hand, since `Z`'s in bound with a capability constraint, it's capability bound is thus independent of π.

A partial reification π is well-formed with respect to a type environment β if it assigns to each type variable of the environment a capability compatible with the variable's bound. Additionally, the partial reification must map `this` to a capability allowed by its bound in β, if it enusts. In this case, we say π is a partial reification of β, and we note it $\beta \vdash \pi$. The relation used to determine if a capability is allowed by a capability ($\lll$) is the same as used by bound compliance, and is described in section Section 5.8.

$$\frac{\begin{array}{c}dom(\pi) = dom(\beta)\\ \forall \texttt{X} \in \beta.\ \pi(\texttt{X}) \lll capBound_{\pi,\beta}(\texttt{X})\\ \texttt{this} \in \beta \to \pi(\texttt{this}) \lll \beta(\texttt{this})\end{array}}{\beta \vdash \pi}$$

For example, the type environment $\beta = [\,\texttt{X} \mapsto \texttt{A \#share}, \texttt{this} \mapsto \texttt{\#read}\,]$ allows the following six different partial reifications,

$$\begin{aligned}\pi_1 &= [\,\texttt{X} \mapsto \texttt{tag}, \texttt{this} \mapsto \texttt{box}\,] &\qquad \pi_2 &= [\,\texttt{X} \mapsto \texttt{val}, \texttt{this} \mapsto \texttt{box}\,]\\ \pi_3 &= [\,\texttt{X} \mapsto \texttt{tag}, \texttt{this} \mapsto \texttt{val}\,] & \pi_4 &= [\,\texttt{X} \mapsto \texttt{val}, \texttt{this} \mapsto \texttt{val}\,]\\ \pi_5 &= [\,\texttt{X} \mapsto \texttt{tag}, \texttt{this} \mapsto \texttt{ref}\,] & \pi_6 &= [\,\texttt{X} \mapsto \texttt{val}, \texttt{this} \mapsto \texttt{ref}\,]\end{aligned}$$

Because the *capBound* function depends on the reification itself, it cannot be used to find all well-formed reifications directly. However, given a reification it is possible to check whether it is well-formed. Since for a set of type variables there can only enust a finite number of reifications, the set of well-formed reifications for a given environment is obtained by enumerating all reifications and eliminating the ones which aren't well-formed.

The reify_β function returns the set of reified types obtained by reducing a given type with each of the environment's well-formed reifications.

$$\begin{aligned}&\mathit{reify}_\beta :: \mathit{Type} \rightarrow \mathbb{P}(\mathit{ReifiedType})\\&\mathit{reify}_\beta(\mathtt{T}) = \{(\mathit{reduce}_\pi(\mathtt{T}) \mid \beta \vdash \pi\}\end{aligned}$$

For example, considering the example from the start of Section 5.5, given $\beta = [\,\mathtt{X} \mapsto \mathtt{N}\ \#\mathtt{alias}\,]$, reifying the type $\mathtt{val} \rhd \mathtt{X}$ produces the set of type

$$\mathit{reify}_\beta(\mathtt{val} \rhd \mathtt{X}) = \{\mathtt{X\ tag},\, \mathtt{X\ val}\}$$

We also define the $\mathit{reifyPair}_\beta$ function below to perform pairwise reification, by applying reifications to two types simultaneously. The function is overloaded to work with different combinations of types and bounds.

$$\begin{aligned}&\mathit{reifyPair}_\beta :: \mathit{Type} \times \mathit{Type} \rightarrow \mathbb{P}(\mathit{ReifiedType} \times \mathit{ReifiedType})\\&\mathit{reifyPair}_\beta(\mathtt{T}, \mathtt{T}') = \{(\mathit{reduce}_\pi(\mathtt{T}), \mathit{reduce}_\pi(\mathtt{T}')) \mid \beta \vdash \pi\}\end{aligned}$$

$$\begin{aligned}&\mathit{reifyPair}_\beta :: \mathit{Type} \times \mathit{Bound} \rightarrow \mathbb{P}(\mathit{ReifiedType} \times \mathit{ReifiedBound})\\&\mathit{reifyPair}_\beta(\mathtt{T}, \mathtt{BT}) = \{(\mathit{reduce}_\pi(\mathtt{T}), \mathit{reduce}_\pi(\mathtt{BT})) \mid \beta \vdash \pi\}\end{aligned}$$

$$\begin{aligned}&\mathit{reifyPair}_\beta :: \mathit{Bound} \times \mathit{Bound} \rightarrow \mathbb{P}(\mathit{ReifiedBound} \times \mathit{ReifiedBound})\\&\mathit{reifyPair}_\beta(\mathtt{BT}, \mathtt{BT}') = \{(\mathit{reduce}_\pi(\mathtt{BT}), \mathit{reduce}_\pi(\mathtt{BT}')) \mid \beta \vdash \pi\}\end{aligned}$$

Pairwise reification is used when defining relations between two types. For example, checking whether $\mathtt{X}$ is a subtype of $\mathtt{val}{\rhd}\mathtt{X}$ with $\pi = [\,\mathtt{X} \mapsto \#\mathtt{send}\,]$ involves reifying the two types simultaneously, as shown below.

$$\mathit{reifyPair}_\beta(\mathtt{X}, \mathtt{val} \rhd \mathtt{X}) = \{(\mathtt{X\ tag}, \mathtt{X\ tag}),\ (\mathtt{X\ val}, \mathtt{X\ val}),\ (\mathtt{X\ iso}, \mathtt{X\ val})\,\}$$

Both the reify_β and the $\mathit{reifyPair}_\beta$ function are undefined if reducing the types fails with any of the well-formed partial reifications. For example, given $\beta = [\,\mathtt{X} \mapsto \mathtt{N}\ \#\mathtt{alias}\,]$, reifying the type $\mathtt{X} \rhd \mathtt{A\ ref}$ fails since it cannot be reduced with $\pi = [\,\mathtt{X} \mapsto \mathtt{tag}\,]$, even though there are other partial reifications of β for which the reduction is defined.

5.6 Inheritance ($\sqsubseteq$)

$$\frac{\beta \vdash \texttt{RT} \sqsubseteq \texttt{RT}'' \qquad \beta \vdash \texttt{RT}'' \sqsubseteq \texttt{RT}'}{\beta \vdash \texttt{RT} \sqsubseteq \texttt{RT}'}\ \textsc{I-Trans}$$

$$\frac{\beta \vdash \texttt{T}_i \leqslant \texttt{T}'_i \qquad \beta \vdash \texttt{T}'_i \leqslant \texttt{T}_i}{\beta \vdash \texttt{DS}[\overline{\texttt{T}}] \sqsubseteq \texttt{DS}[\overline{\texttt{T}'}]}\ \textsc{I-Refl1} \qquad \frac{}{\beta \vdash \texttt{X} \sqsubseteq \texttt{X}}\ \textsc{I-Refl2}$$

$$\frac{\beta \vdash \mathit{implements}(\texttt{DS}[\overline{\texttt{T}}], \texttt{S}[\overline{\texttt{T}'}])}{\beta \vdash \texttt{DS}[\overline{\texttt{T}}] \sqsubseteq \texttt{S}[\overline{\texttt{T}'}]}\ \textsc{I-Struct} \qquad \frac{\beta \vdash \texttt{I}[\overline{\texttt{T}'}] \in \mathcal{I}s(\texttt{DS}[\overline{\texttt{T}}])}{\beta \vdash \texttt{DS}[\overline{\texttt{T}}] \sqsubseteq \texttt{I}[\overline{\texttt{T}'}]}\ \textsc{I-Nominal}$$

$$\frac{\texttt{X} \in \beta}{\beta \vdash \texttt{X} \sqsubseteq \mathit{basicBound}(\beta(\texttt{X}))}\ \textsc{I-Bound}$$

Figure 5.6: Inheritance

We define inheritance, in Figure 5.6, as a relation between basic types. Inheritance is used in sections Sections 5.7 and 5.8 to define subtyping and bound compliance. We use the notation

$$\beta \vdash \texttt{RT} \sqsubseteq \texttt{RT}'$$

when RT inherits from $\texttt{RT}'$ given a type environment β. Inheritance is transitive by the I-Trans rule, and reflexive by the I-Refl1 and I-Refl2 rules.

As explained in Section 2.3.6, type arguments are generally invariant, which means a type `A[T]` does not necessarily inherit from `A[T']`, even if `T` is a subtype of `T`. However, the I-Refl1 allows type arguments to be replaced by *equivalent* arguments. Two types `T` and `T'` are equivalent if they are both subtypes of the other, that is $\beta \vdash \texttt{T} \leqslant \texttt{T}'$ and $\beta \vdash \texttt{T}' \leqslant \texttt{T}$. This can happen for example when two interfaces require the same set of methods, such as in the example below. Since the types `N1 box` and `N2 box` are equivalent, `A[N1 box]` inherits from `A[N2 box]` and vice-versa.

```
interface N1
  fun m()
interface N2
  fun m()

class A[X]

// N1 box and N2 box are equivalent
// A[N1 box] inherits from A[N2 box]
// A[N2 box] inherits from A[N1 box]
```

5.6.1 Nominal Inheritance

$$\frac{\beta \vdash \mathtt{I}[\,\overline{\mathtt{T}'}\,] \in \mathcal{I}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,])}{\beta \vdash \mathtt{DS}[\,\overline{\mathtt{T}}\,] \sqsubseteq \mathtt{I}[\,\overline{\mathtt{T}'}\,]}\ \textsc{I-Nominal}$$

The rule I-Nominal handles nominal inheritance. A basic type $\mathtt{DS}[\overline{\mathtt{T}}]$ inherits an abstract type $\mathtt{I}[\overline{\mathtt{T}'}]$ if the parent is a member of the set of types which the child explicitely inherits from. This set is determined by the lookup rule $\mathcal{I}s$, defined in section Section 5.14. The lookup rule substitutes type variables with their instantiations. For example, given the definitions below, the set of parent types of `A[B]` is $\mathcal{I}(\mathtt{A}[\mathtt{B}]) = \{\mathtt{N1}, \mathtt{N2}[\mathtt{A}]\}$.

```
trait N1
trait N2[X]

class A[Y] is N1, N2[Y]
class B

// A[B] inherits nominally from N1 and N2[B]
```

5.6.2 Structural Inheritance

$$\frac{\beta \vdash \mathit{implements}(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{S}[\,\overline{\mathtt{T}'}\,])}{\beta \vdash \mathtt{DS}[\,\overline{\mathtt{T}}\,] \sqsubseteq \mathtt{S}[\,\overline{\mathtt{T}'}\,]}\ \textsc{I-Struct}$$

$$\frac{\begin{array}{c}\forall \mathtt{KS}' \in \mathcal{K}s(\mathtt{S}[\,\overline{\mathtt{T}'}\,]).\ \exists \mathtt{KS} \in \mathcal{K}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,]).\ \beta \vdash \mathtt{KS} \leqslant \mathtt{KS}' \\ \forall \mathtt{MS}' \in \mathcal{M}s(\mathtt{S}[\,\overline{\mathtt{T}'}\,]).\ \exists \mathtt{MS} \in \mathcal{M}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,]).\ \beta \vdash \mathtt{MS} \leqslant \mathtt{MS}' \\ \forall \mathtt{BS}' \in \mathcal{B}s(\mathtt{S}[\,\overline{\mathtt{T}'}\,]).\ \exists \mathtt{BS} \in \mathcal{B}s(\mathtt{DS}[\,\overline{\mathtt{T}'}\,]).\ \beta \vdash \mathtt{BS} \leqslant \mathtt{BS}'\end{array}}{\beta \vdash \mathit{implements}(\mathtt{DS}[\,\overline{\mathtt{T}'}\,], \mathtt{S}[\,\overline{\mathtt{T}}\,])}$$

Structural inheritance is allowed by the rule I-Struct. This rule makes a basic type $\mathtt{DS}[\overline{\mathtt{T}}]$ inherit the interface $\mathtt{S}[\overline{\mathtt{T}'}]$ if it *implements* the interface, by providing all of the methods required by the parent, with compatible signatures.

For example, given the definitions below, the type `A[B]` implements, and therefore inherits, the interfaces `S1` and `S2[B]`, but not `S2[C]` since, after replacing the type variables, the signatures of the `m1` method are not compatible. The type `A[C]` inherits `S2[C]` but not from `S1` nor `S2[B]` for similar reasons. Finally, since structural inheritance is restricted to interfaces, neither `A[B]` nor `A[C]` inherit from the trait `N`, even though they both implement it.

```
interface S1
  fun m1(x: B)
interface S2[X]
  fun m2(x: X)
trait N
  fun m3()

class A[Y]
  fun m1(x: Y) => ···
  fun m2(x: Y) => ···
  fun m3() => ···

class B
class C

// A[B] inherits structurally from S1 and S2[B].
// A[C] inherits structurally from S2[C]
```

5.6.3 Bound Inheritance

$$\frac{\mathtt{X} \in \beta}{\beta \vdash \mathtt{X} \sqsubseteq \mathit{basicBound}(\beta(\mathtt{X}))}\ \textsc{I-Bound}$$

Finally, a type variables inherits from its bound, by the I-Bound rule. This comes from the fact that it can only be instantiated by types which inherit themselves from the bound. Inheritance is defined on basic types, whereas the type environment maps type variables to bounds. We therefore define the recursive function *basicBound*, which strips capabilities and type operators from the bound to determine the underlying basic type.

$$\begin{aligned}
&\mathit{basicBound} :: \mathit{TypeBound} \rightarrow \mathit{BasicType}\\
&\mathit{basicBound}(\mathtt{DS}[\overline{\mathtt{T}}]\ \nu) = \mathtt{DS}[\overline{\mathtt{T}}]\\
&\mathit{basicBound}(\mathtt{X}) = \mathtt{X}\\
&\mathit{basicBound}(\mathtt{BT}+) = \mathit{basicBound}(\mathtt{BT})\\
&\mathit{basicBound}(\mathtt{BT}-) = \mathit{basicBound}(\mathtt{BT})
\end{aligned}$$

For example, given the definitions below, the type variable `X` inherits from `N`, which is the result of removing capabilities and type operators from its bound, `N` **`box`**, through the application of *basicBound*. Similarily, `Y` inherits from `X` and `Z` inherits from `B[X]`. Finally, while the rule I-Bound only applies to immediate bounds, inheritance is transitive through the I-Trans, thus `X` also inherits from `Y`.

```
class A[X: N box, Y: X+]
  // X inherits from its basic bound, N.
  // Y inherits from its basic bound, X, and transitively from N.

  fun m[Z: B[X] iso]() => ···
    // Z inherits from its basic bound, B[X].

trait N
class B[W]
```

5.7 Subtyping ($\leqslant$)

$$\frac{}{\texttt{iso}\circ \leqslant \{\texttt{iso}, \texttt{trn}\circ\}} \qquad \frac{}{\texttt{trn}\circ \leqslant \{\texttt{trn}, \texttt{ref}, \texttt{val}\}} \qquad \frac{}{\{\texttt{trn}, \texttt{ref}, \texttt{val}\} \leqslant \texttt{box}}$$

$$\frac{}{\{\texttt{iso}, \texttt{box}\} \leqslant \texttt{tag}} \qquad \frac{}{\kappa \leqslant \kappa} \qquad \frac{\kappa \leqslant \kappa'' \qquad \kappa'' \leqslant \kappa'}{\kappa \leqslant \kappa'}$$

Figure 5.7: Subtyping of capabilities
Reproduced from [Steed, 2016]

The subtyping relation for capabilities is presented in Figure 5.7. This definition of capability subtyping is borrowed from [Steed, 2016]. We use the notation $\kappa \leqslant \kappa'$ if the capability κ is a subtype of κ', as well as the shorthand $\{\kappa, \kappa'\} \leqslant \kappa''$ if both κ and κ' are subtypes of κ''. Figure 5.8 presents the same relation using a graphical representation.

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

Figure 5.8: Subtyping of capabilities
Reproduced from [Steed, 2016]

We extend the definition of subtyping to reified types as the $\leqslant^{reified}$ relation, in Figure 5.9. Subtyping of reified types allows both capability subtyping and inheritance of the basic types.

$$\frac{\kappa \leqslant \kappa' \qquad \beta \vdash \texttt{BRT} \sqsubseteq \texttt{BRT}'}{\beta \vdash \texttt{BRT}\ \kappa \leqslant^{reified} \texttt{BRT}'\ \kappa'}$$

Figure 5.9: Reified subtyping

Finally we extend the definition of subtyping to full types through pairwise reification, as shown in Figure 5.10

$$\frac{\forall (\texttt{RT}, \texttt{RT}') \in \mathit{reifyPair}_{\beta}(\texttt{T}, \texttt{T}').\ \beta \vdash \texttt{RT} \leqslant^{reified} \texttt{RT}}{\beta \vdash \texttt{T} \leqslant \texttt{T}'}$$

Figure 5.10: Reified subtyping

5.8 Bound compliance ($\lll$)

The bound compliance relation is used to determine whether a type can be used to instantiate a type variable, given the variable's bound.

We first define bound compliance for capabilities, in Figure 5.11. We use the notation $\kappa \lll \nu$ if capability κ is compliant with bound ν. Capability bounds can be either a concrete capability κ, or a capability constraint, recognizable by the # symbol in their names. If a concrete capability is used as a bound, it only allows itself as a compliant capability. On the other hand, capability constraints allow a set of concrete capabilities.

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

$$\frac{}{\{\texttt{iso}\circ, \texttt{trn}\circ, \texttt{ref}, \texttt{val}, \texttt{box}, \texttt{tag}\} \lll \#\texttt{any}\circ} \qquad \frac{}{\{\texttt{iso}\circ, \texttt{val}, \texttt{tag}\} \lll \#\texttt{send}\circ}$$

$$\frac{}{\{\texttt{ref}, \texttt{val}, \texttt{box}, \texttt{tag}\} \lll \#\texttt{alias}} \qquad \frac{}{\{\texttt{ref}, \texttt{val}, \texttt{box}\} \lll \#\texttt{read}} \qquad \frac{}{\{\texttt{val}, \texttt{tag}\} \lll \#\texttt{share}}$$

$$\frac{}{\kappa \lll \kappa}$$

Figure 5.11: Capability bound compliance

We extend the definition of bound compliance to reified types and bounds as the $\lll^{\mathit{reified}}$ relation, in Figure 5.12. A reified type is compliant with a reified bound if its basic type `BRT` inherits from the bound's basic type `BRT'`, and its capability κ is compliant with the bound's capability ν.

$$\frac{\kappa \lll \nu \qquad \beta \vdash \texttt{BRT} \sqsubseteq \texttt{BRT}' \qquad \beta \vdash \mathit{constructible}(\texttt{BRT}') \rightarrow \texttt{BRT} \notin (\mathit{AbstractTypeID} \times \overline{\mathit{ReifiedType}})}{\beta \vdash \texttt{BRT}\ \kappa \lll^{\mathit{reified}} \texttt{BRT}'\ \nu}$$

$$\frac{\mathcal{K}s(\mathit{upperBound}_{\beta}(\texttt{BRT})) \neq \varnothing}{\beta \vdash \mathit{constructible}(\texttt{BRT})}$$

Figure 5.12: Reified bound compliance

Note that compared to subtyping, bound compliance imposes an additional requirement. If the bound is *constructible*, that is if it has at least one constructor, then abstract types cannot be compliant with this bound. This prevents cases such as in the example below, where the generic class invokes a constructor on a type variable, but that type variable would have been instantiated with an abstract type.

```
interface Default
  new default() : ref
class Cell[X: Default #any]
  var f: X ref
  new create() =>
    this.f = X.default()

actor Main
  new create() =>
    Cell[A ref].create()

    // error: abstract type Default ref does not satisfy
    //        constructible bound Default #any
    Cell[Default ref].create()
```

Finally we extend the definition of bound compliance to full types through pairwise reification, as shown in Figure 5.13

$$\frac{\forall(\mathtt{RT},\mathtt{RB}) \in \mathit{reifyPair}_\beta(\mathtt{T},\mathtt{BT}).\ \beta \vdash \mathtt{RT} \lll^{\mathit{reified}} \mathtt{RB}}{\beta \vdash \mathtt{T} \lll \mathtt{BT}}$$

Figure 5.13: Bound compliance

5.9 Sub-bound (⪯)

We define a new relation, sub-bound, which is the equivalent of subtyping but applied to bounds instead. A bound BT is a sub-bound of BT′, which we denote $\beta \vdash \mathtt{BT} \preceq \mathtt{BT}'$ if all types compliant with BT′ are also compliant with BT.

We define this new relation the same way we've defined the other relations, first by defining it on capabilities in Figure 5.14, then extending it to reified types as the $\preceq^{\mathit{reified}}$ relation, and finally extending it to full types through pairwise reification, as demonstrated in Figure 5.15.

$$\frac{}{\{\#\texttt{read}, \#\texttt{alias}, \#\texttt{share}, \#\texttt{send}\circ\} \preceq \#\texttt{any}\circ} \qquad \frac{}{\{\#\texttt{read}, \#\texttt{alias}, \#\texttt{share}, \#\texttt{send}\} \preceq \#\texttt{any}}$$

$$\frac{}{\#\texttt{share} \preceq \{\#\texttt{send}, \#\texttt{send}\circ\}} \qquad \frac{\kappa \lll \nu}{\kappa \preceq \nu}$$

Figure 5.14: Sub-bound of capabilities

$$\frac{\nu \preceq \nu' \qquad \beta \vdash \texttt{BRT} \sqsubseteq \texttt{BRT}'}{\beta \vdash \texttt{BRT}\ \nu \leq^{reified} \texttt{BRT}'\ \nu'} \qquad \frac{\forall(\texttt{RT}, \texttt{RT}') \in \mathit{reifyPair}_\beta(\texttt{BT}, \texttt{BT}').\ \beta \vdash \texttt{RT} \leq^{reified} \texttt{RT}}{\beta \vdash \texttt{BT} \leq \texttt{BT}'}$$

Figure 5.15: Sub-bound

5.10 Safe-to-Write (◁)

It may not always be safe to write a reference into a object, even through a mutable reference to the containing object. For instance if we could write a **ref** into an **iso** one, we could already have another local alias to the inner object. The safe-to-write relation determines whether it is safe to write a reference of a given type `T` into an object of type `T'`.

$\kappa \lhd \kappa$	iso	trn	ref	val	box	tag
iso∘	✓	✓	✓	✓	✓	✓
iso	✓			✓		✓
trn∘	✓	✓	✓	✓	✓	✓
trn	✓	✓		✓		✓
ref	✓	✓	✓	✓	✓	✓
val						
box						
tag						

Figure 5.16: Safe-to-write capabilities
Reproduced from [Steed, 2016]

$$\frac{\kappa \lhd \kappa'}{\beta \vdash \mathtt{BRT}\ \kappa \lhd^{reified} \mathtt{BRT'}\ \kappa'} \qquad \frac{\forall (\mathtt{RT}, \mathtt{RT'}) \in \mathit{reifyPair}_\beta(\mathtt{T}, \mathtt{T'}).\ \beta \vdash \mathtt{RT} \lhd^{reified} \mathtt{RT}}{\beta \vdash \mathtt{T} \lhd \mathtt{T'}}$$

5.11 Sendable Types

Sendable types' capability must be one of `iso`, `val` or `tag`, as these deny the same aliases locally and globally. Figure 5.17 defines a type as sendable if all of its reifications are sendable.

$$\frac{\forall (\mathtt{BRT}\ \kappa) \in \mathit{reify}_\beta(\mathtt{T}).\ \kappa \in \{\mathtt{iso}, \mathtt{val}, \mathtt{tag}\}}{\beta \vdash \mathit{Sendable}(\mathtt{T})}$$

Figure 5.17: Sendable types

5.12 Method subtyping

$$\frac{\begin{array}{c}\beta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{BT'}}\,] \\ \beta \cup \beta' \vdash \mathtt{BT'}_i \preceq \mathtt{BT}_i \qquad \beta \cup \beta' \vdash \mathtt{T'}_i \leqslant \mathtt{T}_i \qquad \kappa \leqslant \kappa'\end{array}}{\beta \vdash \big(\mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT}}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \kappa\big) \leqslant \big(\mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT'}}\,](\overline{\mathtt{x'}} : \overline{\mathtt{T'}}) : \kappa'\big)}$$

$$\frac{\begin{array}{c}\beta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{BT'}}, \mathtt{this} \mapsto \nu'\,] \\ \nu' \preceq \nu \qquad \beta \cup \beta' \vdash \mathtt{BT'}_i \preceq \mathtt{BT}_i \qquad \beta \cup \beta' \vdash \mathtt{T'}_i \leqslant \mathtt{T}_i \qquad \beta \cup \beta' \vdash \mathtt{T} \leqslant \mathtt{T'}\end{array}}{\beta \vdash \Big(\mathtt{fun}\ \nu\ \mathtt{m}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT}}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{T}\Big) \leqslant \Big(\mathtt{fun}\ \nu'\ \mathtt{m}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT'}}\,](\overline{\mathtt{x'}} : \overline{\mathtt{T'}}) : \mathtt{T'}\Big)}$$

$$\frac{\begin{array}{c}\beta' = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{BT'}}\,] \\ \beta \cup \beta' \vdash \mathtt{BT'}_i \preceq \mathtt{BT}_i \qquad \beta \cup \beta' \vdash \mathtt{T'}_i \leqslant \mathtt{T}_i\end{array}}{\beta \vdash \Big(\mathtt{be}\ \mathtt{b}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT}}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}})\Big) \leqslant \Big(\mathtt{be}\ \mathtt{b}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT'}}\,](\overline{\mathtt{x'}} : \overline{\mathtt{T'}})\Big)}$$

Figure 5.18: Method subtyping

We define in Figure 5.18 rules for method subtyping. These rules allow for contra-variant receiver capability, type argument bounds and formal argument types, and they allow for co-variant return capability and type.

5.13 Implementation

We have implemented parts of the type system using Coq, a proof assistant. Our implementation covers the operational semantics of *Pony*^{0} and most of its typing rules. We have started implementing the typing rules of *Pony*$^{\mathrm{PL}}$. So far we have implemented the reduction of types for a given partial reification, as well as a number of auxiliary functions, such as *upperBound*.

We use Coq's extraction feature [Letouzey, 2002] to generate a Haskell version of our implementation. We have also written a parser for Pony programs in Haskell, which combined with the extracted Coq source provides an interactive prompt which can be used to type check and interpret Pony expressions.

A excerpt of the *Pony*^{0} type checker is included as B. The rest of the source is included in the archive accompanying this report. Our implementation reuses many general purpose data structures from [Krebbers and Wiedijk, 2015].

5.14 Lookup rules

$$\frac{\mathtt{P} = \overline{\mathtt{NT}}\ \overline{\mathtt{ST}}\ \overline{\mathtt{CT}}\ \overline{\mathtt{AT}} \qquad \mathtt{class\ C}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{F}}\ \overline{\mathtt{K}}\ \overline{\mathtt{M}} \in \mathtt{CT}}{\mathcal{P}(\mathtt{C}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \epsilon)}$$

$$\frac{\mathtt{P} = \overline{\mathtt{NT}}\ \overline{\mathtt{ST}}\ \overline{\mathtt{CT}}\ \overline{\mathtt{AT}} \qquad \mathtt{actor\ A}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{F}}\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\ \overline{\mathtt{B}} \in \mathtt{AT}}{\mathcal{P}(\mathtt{B}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}})}$$

$$\frac{\mathtt{P} = \overline{\mathtt{NT}}\ \overline{\mathtt{ST}}\ \overline{\mathtt{CT}}\ \overline{\mathtt{AT}} \qquad \mathtt{trait\ N}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{KS}}\ \overline{\mathtt{MS}}\ \overline{\mathtt{BS}} \in \mathtt{NT}}{\mathcal{P}(\mathtt{N}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{KS}}, \overline{\mathtt{MS}}, \overline{\mathtt{BS}})}$$

$$\frac{\mathtt{P} = \overline{\mathtt{NT}}\ \overline{\mathtt{ST}}\ \overline{\mathtt{CT}}\ \overline{\mathtt{AT}} \qquad \mathtt{interface\ S}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}]\ \overline{\mathtt{I}[\overline{\mathtt{T}}]}\ \overline{\mathtt{KS}}\ \overline{\mathtt{MS}}\ \overline{\mathtt{BS}} \in \mathtt{ST}}{\mathcal{P}(\mathtt{S}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{KS}}, \overline{\mathtt{MS}}, \overline{\mathtt{BS}})}$$

Figure 5.19: Program lookup

$$\frac{\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}})}{\mathcal{F}r(\mathtt{RS}) = \{\mathtt{f} \mid \mathtt{var\ f} : \mathtt{T} \in \overline{\mathtt{F}}\}}
\qquad
\frac{\begin{array}{c}\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ (\mathtt{new\ k}[\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}\end{array}}{\mathcal{M}r(\mathtt{RS}, \mathtt{k}) = (\overline{\mathtt{X}'}, \overline{\mathtt{x}}, \mathtt{e})}$$

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ (\mathtt{fun}\ \nu\ \mathtt{m}[\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{T} \Rightarrow \mathtt{e}) \in \overline{\mathtt{M}}\end{array}}{\mathcal{M}r(\mathtt{RS}, \mathtt{m}) = (\overline{\mathtt{X}'}, \overline{\mathtt{x}}, \mathtt{e})}
\qquad
\frac{\begin{array}{c}\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ (\mathtt{be\ b}[\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{B}}\end{array}}{\mathcal{M}r(\mathtt{RS}, \mathtt{k}) = (\overline{\mathtt{X}'}, \overline{\mathtt{x}}, \mathtt{e})}$$

Figure 5.20: Runtime lookup

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ \mathtt{var\ f} : \mathtt{T} \in \overline{\mathtt{F}}\end{array}}{\mathcal{F}(\mathtt{RS}[\overline{\mathtt{T}}], \mathtt{f}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{T}}$$

Figure 5.21: Field lookup

$$\frac{\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}})}{\mathcal{M}s(\mathtt{RS}[\overline{\mathtt{T}}]) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\{\mathtt{MS} \mid \mathtt{MS} \Rightarrow \mathtt{e} \in \overline{\mathtt{M}}\}}
\qquad
\frac{\mathcal{P}(\mathtt{I}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{KS}}, \overline{\mathtt{MS}}, \overline{\mathtt{BS}})}{\mathcal{M}s(\mathtt{I}[\overline{\mathtt{T}}]) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{MS}}}$$

$$\frac{\mathcal{P}(\mathtt{RS}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}})}{\mathcal{B}s(\mathtt{RS}[\overline{\mathtt{T}}]) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\{\mathtt{BS} \mid \mathtt{BS} \Rightarrow \mathtt{e} \in \overline{\mathtt{B}}\}}
\qquad
\frac{\mathcal{P}(\mathtt{I}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\overline{\mathtt{T}}]}, \overline{\mathtt{KS}}, \overline{\mathtt{MS}}, \overline{\mathtt{BS}})}{\mathcal{B}s(\mathtt{I}[\overline{\mathtt{T}}]) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{BS}}}$$

$$\frac{(\mathtt{fun}\ \nu\ \mathtt{m}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}''}) : \mathtt{T}') \in \mathcal{M}s(\mathtt{DS}[\overline{\mathtt{T}}])}{\mathcal{M}d(\mathtt{DS}[\overline{\mathtt{T}}], \mathtt{m}[\mathtt{T}; \overline{\mathtt{T}'}]) = [\mathtt{this} \mapsto \mathtt{T}, \overline{\mathtt{X}} \mapsto \overline{\mathtt{T}'}](\mathtt{DS}[\overline{\mathtt{T}}]\ \nu, \overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{x}} : \overline{\mathtt{T}''}, \mathtt{T}')}$$

$$\frac{(\mathtt{be\ b}[\overline{\mathtt{X}} : \overline{\mathtt{BT}}](\overline{\mathtt{x}} : \overline{\mathtt{T}''})) \in \mathcal{B}s(\mathtt{DS}[\overline{\mathtt{T}}])}{\mathcal{M}d(\mathtt{DS}[\overline{\mathtt{T}}], \mathtt{b}[\mathtt{T}; \overline{\mathtt{T}'}]) = [\mathtt{this} \mapsto \mathtt{T}, \overline{\mathtt{X}} \mapsto \overline{\mathtt{T}'}](\mathtt{DS}[\overline{\mathtt{T}}]\ \mathtt{tag}, \overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{x}} : \overline{\mathtt{T}''}, \mathtt{DS}[\overline{\mathtt{T}}]\ \mathtt{tag})}$$

Figure 5.22: Method lookup

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{A}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\,\overline{\mathtt{T}}\,]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ \overline{\mathtt{KS}} = \{\ \mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}'}) : \mathtt{tag} \mid (\mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}'}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}\ \}\end{array}}{\mathcal{K}s(\mathtt{A}[\,\overline{\mathtt{T}}\,]) = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}\,]\overline{\mathtt{KS}}}$$

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{C}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\,\overline{\mathtt{T}}\,]}, \overline{\mathtt{F}}, \overline{\mathtt{K}}, \overline{\mathtt{M}}, \overline{\mathtt{B}}) \\ \overline{\mathtt{KS}} = \{\ \mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}'}) : \mathtt{ref} \mid (\mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}'} : \overline{\mathtt{BT}'}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}'}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}\ \}\end{array}}{\mathcal{K}s(\mathtt{C}[\,\overline{\mathtt{T}}\,]) = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}\,]\overline{\mathtt{KS}}}$$

$$\frac{\mathcal{P}(\mathtt{I}) = (\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{I}[\,\overline{\mathtt{T}}\,]}, \overline{\mathtt{KS}}, \overline{\mathtt{MS}}, \overline{\mathtt{BS}})}{\mathcal{K}s(\mathtt{I}[\,\overline{\mathtt{T}}\,]) = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}\,]\overline{\mathtt{KS}}}$$

$$\frac{(\mathtt{new}\ \mathtt{k}[\,\overline{\mathtt{X}} : \overline{\mathtt{BT}}\,](\overline{\mathtt{x}} : \overline{\mathtt{T}''})) \in \mathcal{K}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,]) : \kappa}{\mathcal{K}d(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{k}[\,\overline{\mathtt{T}'}\,]) = [\,\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}'}\,](\overline{\mathtt{X}} : \overline{\mathtt{BT}}, \overline{\mathtt{x}} : \overline{\mathtt{T}''}, \kappa)}$$

Figure 5.23: Constructor lookup

Chapter 6

Soundness

We have presented in Chapters 3 to 5 the syntax, semantics and typing rules of *Pony*$^{\text{PL}}$, our new model of the Pony language. In this chapter we present our initial work towards a demonstration of soundness for our model.

Pony^0 is a restricted version of our model without generics. *Pony*^{0} is mostly just a reformulation of *Pony*$^{\text{GS}}$, as presented by [Steed, 2016], with minor changes in order to match the syntax and definitions from *Pony*$^{\text{PL}}$. The syntax, semantics and typing rules for *Pony*^{0} can be found in Appendix A.

In order to argue about the soundness of our new model, we define a translation of programs from *Pony*$^{\text{PL}}$ to *Pony*^{0}. We would have wished to prove the soundness of the translation, by showing that any valid *Pony*$^{\text{PL}}$ is translated to a *Pony*^{0} program where both programs have equivalent runtime behaviour. Together with the soundness of *Pony*$^{\text{GS}}$, presented in [Steed, 2016], this would form a demonstration of the soundness of *Pony*$^{\text{PL}}$.

Unfortunately, due to lack of time, we were not able to complete our work on formulating and proving the soundness of our translation. We will therefore only be defining the translation, without arguing about its soundness.

6.1 Translation of *Pony*$^{\text{PL}}$ programs

6.1.1 Overview

We define the translation of programs from *Pony*$^{\text{PL}}$ to *Pony*^{0} through *full reification* of generics. Generic types and methods in the source *Pony*$^{\text{PL}}$ program are translated on-demand, by replacing each occurence of type parameters by their instantiation. Each instantiation of a generic class used by the program requires a distinct reified *Pony*^{0} copy.

Consider the following example, which defines a class `Cell` generic over a type argument `X`, inspired by the example from section . Furthermore, the `get` function is polymorphic over its receiver capability, and this capability is reflected in the return type through the **`this`** viewpoint.

```
class Cell[X: Any #any]
  var f: X
  new create(x: X) => this.f = consume x
  fun #read get() : this->X => this.f

class A
class B
actor Main
  new create() =>
    var x : Cell[A ref] box = Cell[A ref].create(A)
    var y : Cell[A ref] ref = Cell[A ref].create(A)
    var z : Cell[B iso] ref = Cell[B iso].create(B)

    var a : A box = x.get[Cell[A ref] box]()
    var a' : A ref = y.get[Cell[A ref] ref]()
    var b : B iso = z.get[Cell[B iso] ref]()
```

The *Pony*^{0} translation of the example above is shown below. The `Main` actor from the *Pony*$^{\mathrm{PL}}$ example referenced two different instantiations of the `Cell` class, `Cell[A ref]` and `Cell[B iso]`. Therefore, the translated program defines two distinct versions of the class, `Cell_Aref` and `Cell_Biso`. In each class, the type parameter `X` has been replaced with its instantiation, repectively `A ref` and `B iso`. Note that we use *mangling* to encode the type parameters as part of the class names, such that the two version. We describe mangling in more details in Section 2.3.

Additionally, distinct versions of the receiver-polymorphic `get` method must be created in class `Cell_Aref`, in order to reflect the receiver capability in the signature. Similarily, the name of the translated methods are mangled to reflect the capability of the receiver.

```
class Cell_Aref
  var f: A ref
  new create(x: A ref) => this.f = consume x
  fun ref get_ref() : A ref => this.f
  fun box get_box() : A box => this.f

class Cell_Biso
  var f: B iso
  new create(x: B iso) => this.f = consume x
  fun ref get_ref() : B iso => this.f

class A
class B
actor Main
  new create() =>
    var x : Cell_Aref box = Cell_Aref.create(A)
    var y : Cell_Aref ref = Cell_Aref.create(A)
    var z : Cell_Biso ref = Cell_Biso.create(B)

    var a : A box = x.get_box()
    var a' : A ref = y.get_ref()
    var b : B ref = z.get_ref()
```

Finally note that we only translate instances of generic types and methods which are needed by the program. For example above, the translated program does not contain a translation of the `Cell[A box]` class as it is not used by the rest of program. Similarily, the instance of `get` method from the `Cell[B iso]` class with receiver capability `box` does not need to be included in the translated program, as it does not get used.

6.1.2 Name mangling

In order to prevent the names of the multiple instances of the same type or method to conflict in the translated program, their names are *mangled* to include the type parameters of the instantiation. Three different mangling functions are used, depending on what kind of name needs to be mangled, with the following signatures:

$$\begin{aligned}
&\mathit{mangle} :: \mathit{TypeId} \times \overline{\mathit{GroundType}} \rightarrow \mathit{TypeId}\\
&\mathit{mangle} :: \mathit{MethodID} \times \mathit{Cap} \times \overline{\mathit{GroundType}} \rightarrow \mathit{MethodID}\\
&\mathit{mangle} :: \mathit{CtorID} \times \overline{\mathit{GroundType}} \rightarrow \mathit{CtorID}
\end{aligned}$$

The arguments to the functions correspond to the original name in the $\mathit{Pony}^{\mathrm{PL}}$ program, along with potential receiver capability and type arguments.

We leave the mangling functions unspecified. We only require them to form bijections In our examples we use a simple and intuitive mangling based on concatenation separated by underscores.

6.1.3 Translation contexts

We call a translation context Π a mapping from type variables to ground types and from **`this`** to a concrete capability.

$$\Pi \in \mathit{TypeVarID} \cup \{\mathtt{this}\} \rightarrow \mathit{GroundType} \cup \mathit{Cap}$$

where:

$$\begin{array}{l} \forall \mathtt{X} \in \Pi.\ \Pi(\mathtt{X}) \in \mathit{GroundType} \\ \mathtt{this} \in \Pi \rightarrow \Pi(\mathtt{this}) \in \mathit{Cap} \end{array}$$

The translation context will be used to determine how type variables and the **`this`** viewpoint should be replaced in the translated program.

6.1.4 Translation of types

Translation of types is similar to the partial reification we defined in section 5.5. Indeed, we overload the notation

$$\Pi \vdash \mathtt{T} \Downarrow \mathtt{GT}$$

to denote that `T` reduces to `GT` given the translation context Π. We define the rules of translation below in Figure 6.1.

$$\frac{\Pi(\mathtt{X}) = \mathtt{GT}}{\Pi \vdash \mathtt{X} \Downarrow \mathtt{GT}} \qquad \frac{\Pi(\mathtt{X}) = \mathtt{BGT}\ \kappa'}{\Pi \vdash \mathtt{X}\ \kappa \Downarrow \mathtt{BGT}\ \kappa} \qquad \frac{\Pi \vdash \overline{\mathtt{T}} \Downarrow \overline{\mathtt{BGT}\ \kappa}}{\Pi \vdash \mathtt{DS}[\,\overline{\mathtt{T}}\,]\ \kappa \Downarrow \mathtt{DS}[\,\overline{\mathtt{BGT}\ \kappa}\,]\ \kappa}$$

$$\frac{\Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa}{\Pi \vdash \mathtt{T+} \Downarrow \mathtt{BGT}\ \mathcal{A}(\kappa)} \qquad \frac{\Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa}{\Pi \vdash \mathtt{T-} \Downarrow \mathtt{BGT}\ \mathcal{U}(\kappa)} \qquad \frac{\Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa}{\Pi \vdash \mathtt{recover}\ \mathtt{T} \Downarrow \mathtt{BGT}\ \mathcal{R}(\kappa)}$$

$$\frac{\begin{array}{c}\Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa \\ \Pi \vdash \mathtt{T}' \Downarrow \mathtt{BGT}'\ \kappa'\end{array}}{\Pi \vdash \mathtt{T} \rhd \mathtt{T}' \Downarrow \mathtt{BGT}'\ \mathcal{V}p(\kappa, \kappa')} \qquad \frac{\Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa'}{\Pi \vdash \kappa \rhd \mathtt{T} \Downarrow \mathtt{BGT}\ \mathcal{V}p(\kappa, \kappa')} \qquad \frac{\begin{array}{c}\Pi(\mathtt{this}) = \kappa \\ \Pi \vdash \mathtt{T} \Downarrow \mathtt{BGT}\ \kappa'\end{array}}{\Pi \vdash \kappa \rhd \mathtt{T} \Downarrow \mathtt{BGT}\ \mathcal{V}p(\kappa, \kappa')}$$

Figure 6.1: Reduction of types with a translation context

We denote $|\mathtt{T}|_\Pi$ the translation of a type `T` in a translation context Π. It is defined below by mangling the result of the reduction of `T`.

$$|\mathtt{T}|_\Pi = \mathit{mangle}(\mathtt{DS}, \overline{\mathtt{GT}})\ \kappa \quad \mathit{iff}\ \Pi \vdash \mathtt{T} \Downarrow \mathtt{DS}[\,\overline{\mathtt{GT}}\,]\ \kappa$$

6.1.5 Translation of expressions

Translation of expressions other than method and constructors, as shown below, simply uses the direct *Pony*^{0} equivalent.

$$|\texttt{this}|_\Pi = \texttt{this} \qquad |\texttt{x}|_\Pi = \texttt{x}$$
$$|\texttt{null}|_\Pi = \texttt{null} \qquad |\texttt{x} = \texttt{e}|_\Pi = \texttt{x} = |\texttt{e}|_\Pi$$
$$|\texttt{e}; \texttt{e}'|_\Pi = |\texttt{e}|_\Pi; |\texttt{e}'|_\Pi \qquad |\texttt{e.f}|_\Pi = (|\texttt{e}|_\Pi).\texttt{f}$$
$$|\texttt{recover e}|_\Pi = \texttt{recover } (|\texttt{e}|_\Pi) \qquad |\texttt{e.f} = \texttt{e}'|_\Pi = (|\texttt{e}|_\Pi).\texttt{f} = |\texttt{e}'|_\Pi$$

Translation of method calls requires mangling the name of the function using the receiver's capability and the specified type arguments. The receiver capability is extracted from the reducing the receiver type.

$$\frac{\Pi \vdash \texttt{T} \Downarrow \texttt{DS}[\,\overline{\texttt{GT}}\,]\ \kappa \qquad \overline{\Pi \vdash \texttt{T} \Downarrow \texttt{GT}'} \qquad \texttt{n}' = \mathit{mangle}(\texttt{n}, \kappa, \overline{\texttt{GT}'})}{|\texttt{e.n}[\,\texttt{T}; \overline{\texttt{T}}\,](\overline{\texttt{e}})|_\Pi = (|\texttt{e}|_\Pi).\texttt{n}'(\overline{|\texttt{e}|_\Pi})}$$

Finally, translation of constructor calls is defined by mangling both the name of the type being constructed and the name of the constructor. We handle the two cases, constructing a known type, and constructing an object through a type variable separately.

$$\frac{\overline{\Pi \vdash \texttt{T} \Downarrow \texttt{GT}} \qquad \texttt{RS}' = \mathit{mangle}(\texttt{RS}, \overline{\texttt{GT}}) \qquad \overline{\Pi \vdash \texttt{T}' \Downarrow \texttt{GT}'} \qquad \texttt{k}' = \mathit{mangle}(\texttt{k}, \overline{\texttt{GT}'})}{|\texttt{RS}[\,\overline{\texttt{T}}\,].\texttt{k}[\,\overline{\texttt{T}'}\,](\overline{\texttt{e}})|_\Pi = \texttt{RS}'.\texttt{k}'(\overline{|\texttt{e}|_\Pi})}$$

$$\frac{\Pi \vdash \texttt{X} \Downarrow \texttt{RS}[\,\overline{\texttt{GT}}\,]\ \kappa \qquad \texttt{RS}' = \mathit{mangle}(\texttt{RS}, \overline{\texttt{GT}}) \qquad \overline{\Pi \vdash \texttt{T}' \Downarrow \texttt{GT}'} \qquad \texttt{k}' = \mathit{mangle}(\texttt{k}, \overline{\texttt{GT}'})}{|\texttt{X.k}[\,\overline{\texttt{T}}\,](\overline{\texttt{e}})|_\Pi = \texttt{RS}'.\texttt{k}'(\overline{|\texttt{e}|_\Pi})}$$

6.1.6 Reachability

As we mentioned in section Section 6.1.1, we only want to include in the translated program classes and and methods required by the rest of the program. We therefore define the notion of *reachability* of an expression, which is the set of types and methods required for this expression to be well-formed in the translated program. When translating a whole program, we decided on an *entrypoint expression* for the program, generally `Main.create()`. The translated program will contain a translation of all the types reachable by this expression.

Formally, we define the reachability of an expression through the following four relations:

$$\begin{aligned}
\mathit{reachableExpr}_{\Pi} &:: \mathit{Expr} \times \mathit{Expr}\\
\mathit{reachableType}_{\Pi} &:: \mathit{Expr} \times \mathit{BasicGroundType}\\
\mathit{reachableMethod}_{\Pi} &:: \mathit{Expr} \times (\mathit{BasicGroundType} \times \mathit{MethodID} \times \mathit{Cap} \times \overline{\mathit{GroundType}})\\
\mathit{reachableCtor}_{\Pi} &:: \mathit{Expr} \times (\mathit{BasicGroundType} \times \mathit{CtorID} \times \overline{\mathit{GroundType}})
\end{aligned}$$

For instance, $\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{BGT})$ holds if the type BGT is reachable from the expression e.

We now define the rules used to determine these relations. First of all, method calls require the receiver type and the corresponding method on this type to be reachable.

$$\frac{\begin{array}{c}\mathtt{e} = \mathtt{e}'.\mathtt{n}[\,\mathtt{T};\overline{\mathtt{T}}\,](\overline{\mathtt{e}})\\ \Pi \vdash \mathtt{T} \Downarrow \mathtt{DS}[\,\overline{\mathtt{GT}}\,]\ \kappa \qquad \overline{\Pi \vdash \mathtt{T} \Downarrow \mathtt{GT}'}\end{array}}{\begin{array}{c}\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{DS}[\,\overline{\mathtt{GT}}\,])\\ \mathit{reachableMethod}_{\Pi}(\mathtt{e}, (\mathtt{DS}[\,\overline{\mathtt{GT}}\,], \mathtt{n}, \kappa, \overline{\mathtt{GT}'}))\end{array}}$$

When translating a reachable method, it is necessary that the argument and return types are also translated. We therefore define those as reachable as well.

$$\frac{\begin{array}{c}\mathit{reachableMethod}_{\Pi}(\mathtt{e},\ (\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{n}, \kappa, \overline{\mathtt{T}'}))\\ \mathcal{M}d(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{n}[\,\mathtt{DS}[\,\overline{\mathtt{T}'}\,]\ \kappa; \overline{\mathtt{T}}\,]) = (\_, \_, \overline{\mathtt{x} : \mathtt{DS}[\,\overline{\mathtt{T}}\,]\ \kappa}, \mathtt{DS}'[\,\overline{\mathtt{T}''}\,]\ \kappa')\end{array}}{\begin{array}{c}\overline{\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{DS}[\,\overline{\mathtt{T}}\,])}\\ \mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{DS}'[\,\overline{\mathtt{T}''}\,])\end{array}}$$

The translation of reachable types requires their parents to be included in the program as well. In order for the program to be well formed, the child type must define all of its parent's reachable methods.

Note that unlike $\mathit{Pony}^{\mathrm{PL}}$, traits and interfaces in Pony^{0} do not have any constructors, and thus it is not required for child types to implement all constructors of its parent types. In fact, only classes and actors have reachable constructors.

$$\frac{\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{DS}[\,\overline{\mathtt{T}}\,]) \qquad \mathtt{I}[\,\overline{\mathtt{T}'}\,] \in \mathcal{I}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,])}{\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{I}[\,\overline{\mathtt{T}'}\,])}$$

$$\frac{\begin{array}{c}\mathit{reachableType}_{\Pi}(\mathtt{e}, \mathtt{DS}[\,\overline{\mathtt{T}}\,]) \qquad \mathtt{I}[\,\overline{\mathtt{T}'}\,] \in \mathcal{I}s(\mathtt{DS}[\,\overline{\mathtt{T}}\,])\\ \mathit{reachableMethod}_{\Pi}(\mathtt{e},\ (\mathtt{I}[\,\overline{\mathtt{T}'}\,], \mathtt{n}, \kappa, \overline{\mathtt{T}}))\end{array}}{\mathit{reachableMethod}_{\Pi}(\mathtt{e},\ (\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{n}, \kappa, \overline{\mathtt{T}}))}$$

In addition to nominal subclassing, we also want structural subclassing to be preserved by translation. That is for each pair of trait and child type, where both types are reachable, if the child type implements the trait in the original $\mathit{Pony}^{\mathrm{PL}}$ program, then it must implement it in the translated Pony^{0} program.

This is achieved by making all reachable methods in the trait also reachable in the child, so they are included during translation. Again, traits in *Pony*^{0} do not contain any contructors, hence including the parent's methods is enough.

In the rule below, since the type arguments applied to `DS` and `S` do not contain any type variables, an empty type environment is used when testing is `DS` implements `DS`.

$$\frac{\begin{array}{c} \mathit{reachableType}_\Pi(\mathtt{e}, \mathtt{DS}[\,\overline{\mathtt{T}}\,]) \qquad \mathit{reachableType}_\Pi(\mathtt{e}, \mathtt{S}[\,\overline{\mathtt{T'}}\,]) \\ \varnothing \vdash \mathit{implements}(\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{S}[\,\overline{\mathtt{T'}}\,]) \\ \mathit{reachableMethod}_\Pi(\mathtt{e},\ (\mathtt{S}[\,\overline{\mathtt{T'}}\,], \mathtt{n}, \kappa, \overline{\mathtt{T''}})) \end{array}}{\mathit{reachableMethod}_\Pi(\mathtt{e},\ (\mathtt{DS}[\,\overline{\mathtt{T}}\,], \mathtt{n}, \kappa, \overline{\mathtt{T''}}))}$$

Note that we did not explicitely require type arguments to be reachable. For example even if `A[B ref]` is reachable, `B` might not be. Indeed it is not necessary required by the translated program, for example if it is unused by the original program or only used as a type variable bound, such as in the following program,

```
class A[X]
  fun m[Y: X](y: Y)
```

If the type argument appears as a method or return argument type, such that it appears in the translated program, then the rules described above already make it reachable.

Chapter 7

Conclusion

7.1 Challenges

In this section we discuss a few challenges we faced during our work which we had not expected. We had to spend significant amounts of time overcoming those, we had not planned for initially.

Before we could start defining a model for generics in Pony, if was important for us to have a good understanding of how generics were being handled by the Pony compiler. However, while most other parts of the language have been extensively documented, there has been very little material covering generics, making it hard for us to do so.

We therefore had to fallback to simply trying the compiler with a large number of various examples, until we could have a better knowledge of how generics are implemented. However, the implementation of generics had received little attention, and most uses of generics in the Pony standard library or in existing applications only use them in simple and straightforward way. While developing our model, we've had to consider many edge cases, and we frequently ran into various compiler bugs and crashes. While, together with the authors of the compiler, we were able to fix a large portion of these, which contributed torwards improving the compiler, this distracted us from our main goal of developing a formal model.

Finally, the interaction between type variables on the one hand and aliasing, unaliasing and viewpoint adaptation on the other proved to be much more complicated than we initially envisioned. It took us a lot of time and iterations to reach our final design.

7.2 Contributions

In Section 2.3, we have given an extensive informal description of generics, through a simple example of a generic class `Cell`. While generics are available in many languages, and most programmers will be familiar with the overall concept, generics in Pony also include a number of novel features. These include capability constraints, explicit aliasing, unaliasing and viewpoint adaptation, and **`this`**-based viewpoint adaptation. However, until now, there were no resources available describing these features in details, making it hard for new users to learn about them. We are planning on contributing this section to the official online *Pony Tutorial*, replacing the current existing but sparse documentation on generics.

Our main contribution, *Pony*$^{\text{PL}}$, is a new formal model for the Pony language. The core of our model is largely inspired by the existing models *Pony*$^{\text{SC}}$ and *Pony*$^{\text{GS}}$, to which we have added support for generics. Our model has allowed a better understanding of how generics should behave, and how they interact with other features, such as viewpoint adaptation, aliasing and unaliasing.

While defining typing rules for *Pony*$^{\text{PL}}$, we introduced the concept of symbolic type operators, which encode modifications to types' capabilities. In previous models, the modifications could be applied directly on types, as these were always known in their normal form. Symbolic operators can be used whatever form the underlying types have, including variables whose capability is not known.

The introduction of type variables and symbolic type operators has required us to redefine a number of relations on types, such as subtyping and safe-to-write, which already existed in the previous models. We've done so by first defining partial reification, which determines a set reified, and thus fully normalized, types from any form of type. Thanks to partial reification, most relations on types can be defined in a straightforward way in terms of the same relation applied to reified types. Since reified types are fully normalized and have a similar form as the types used by earlier models, we are able to reuse these definitions, only requiring minor changes. We therefore expect most lemmas which hold on these relations in the previous models to hold trivially in our new model.

We have also identified and defined new relations specific to generics, such as bound compliance and sub-bounds. We have defined these following the same pattern as existing relations, by first defining the relations on reified types and extending them to full types by partial reification.

As a first step towards a proof of soundness of our model, we have defined a translation of programs from *Pony*$^{\text{PL}}$ to *Pony*^{0}, a model of the Pony language without generics. *Pony*^{0} is mostly just a reformulation of *Pony*$^{\text{GS}}$ in order to match the syntax and definitions from *Pony*$^{\text{PL}}$. Our translation is based on reification, by creating distinct copies of generic types and methods, each instantiated with different type arguments.

Finally, while developing *Pony*$^{\text{PL}}$, we have identified a number of issues in the implementation of generics in the compiler. These issues range from simple oversights or mishandle edge cases to fundamental unsound design decisions in the typing rules. We have worked closely with the authors of the compiler, by reporting all the issues found upstream, with associated minimal reproducible examples, and where applicable, insight on how these bugs could be used to trigger data-races.

Whenever we could, we have either contributed a fix for issues ourselves or helped the compiler authors by providing suggestions on possible solutions. There are still a number of unfixed issues, which we plan on addressing in the near future.

7.3 Further Work

There are various ways our work presented here could be expanded upon:

- During our work, we've uncovered a number of bugs in the compiler, ranging from compiler crashes to soundness issues in the type system. While some of these have already fixed, many others still need to be resolved. In some cases, our proposed solution requires changes to how types are represented internally, which affects a large portion of the codebase.
- The existing model presented by [Steed, 2016], $Pony^{\mathrm{GS}}$, supports extensions which we did not consider in $Pony^{\mathrm{PL}}$, namely union, intersection and tuple types. We did not include these features in our own model as they do not interact directly with generics, and we wanted to keep our initial model for generics as simple as possible. Future work could integrate these features back from $Pony^{\mathrm{GS}}$ to $Pony^{\mathrm{PL}}$.
- Just like designing a model of the current system has enabled us to find a number of flaws in the implementation, modelling future features could help prevent these mistakes from being made in the first place, by identifying potential pitfalls early. Our model could therefore serve as a basis to design new features around generics.
- Finally, it is currently unclear how representive of the compiler our model is. We had originally envisioned implementing our model in a standalone typechecker, and running both our own typechecker and the Pony compiler on a corpus of Pony programs making use of various features of the language, comparing results. There are two main corpuses of Pony available, the standard library as well as the compiler's test suite. This would have allowed us to identify what programs are allowed by one but not by the other. Unfortunately, we did not have time to investigate this path further.

Bibliography

[Akka] Akka. http://akka.io/. Accessed March 1st, 2017.

[Amin and Tate, 2016] Amin, N. and Tate, R. (2016). Java and scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 838–848. ACM.

[Boehm and Adve, 2008] Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM.

[Bracha et al., 1998] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: Adding genericity to the Java programming language. *Acm sigplan notices*, 33(10):183–200.

[Cameron et al., 2008] Cameron, N., Drossopoulou, S., and Ernst, E. (2008). A model for Java with wildcards. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer.

[Clebsch et al., 2015] Clebsch, S., Drossopoulou, S., Blessing, S., and McNeil, A. (2015). Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM.

[Drossopoulou et al., 1999] Drossopoulou, S., Eisenbach, S., and Khurshid, S. (1999). Is the Java type system sound? *TAPOS*, 5(1):3–24.

[Erlang] Erlang. http://www.erlang.org/. Accessed March 1st, 2017.

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute.

[Igarashi et al., 2001] Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450.

[Kilim] Kilim. http://www.malhar.net/sriram/kilim/. Accessed March 1st, 2017.

[Krebbers and Wiedijk, 2015] Krebbers, R. and Wiedijk, F. (2015). A typed c11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 15–27. ACM.

[Letouzey, 2002] Letouzey, P. (2002). A new extraction for Coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer.

[Steed, 2016] Steed, G. (2016). A principled design of capabilities in Pony. Master's thesis.

[Torgersen et al., 2005] Torgersen, M., Ernst, E., and Hansen, C. P. (2005). Wild FJ. *Proceedings of Fool 12*.

Appendix A

Pony^0

A.1 Syntax

$$
\begin{array}{rcl}
\texttt{P} \in & \textit{Program} & ::= \overline{\texttt{CT}}\ \overline{\texttt{AT}}\ \overline{\texttt{NT}}\ \overline{\texttt{ST}} \\
\texttt{CT} \in & \textit{ClassDef} & ::= \texttt{class C}\ \overline{\texttt{I}}\ \overline{\texttt{F}}\ \overline{\texttt{K}}\ \overline{\texttt{M}} \\
\texttt{AT} \in & \textit{ActorDef} & ::= \texttt{actor A}\ \overline{\texttt{I}}\ \overline{\texttt{F}}\ \overline{\texttt{K}}\ \overline{\texttt{M}}\ \overline{\texttt{B}} \\
\texttt{NT} \in & \textit{TraitDef} & ::= \texttt{trait N}\ \overline{\texttt{I}}\ \overline{\texttt{MS}}\ \overline{\texttt{BS}} \\
\texttt{ST} \in & \textit{InterfaceDef} & ::= \texttt{interface S}\ \overline{\texttt{I}}\ \overline{\texttt{MS}}\ \overline{\texttt{BS}}
\end{array}
$$

Figure A.1: Syntax of programs

$$
\begin{array}{rcl}
\texttt{F} \in & \textit{Field} & ::= \texttt{var f} : \texttt{T} \\
\texttt{K} \in & \textit{Ctor} & ::= \texttt{new k}\big(\overline{\texttt{x}} : \overline{\texttt{T}}\big) \Rightarrow \texttt{e} \\
\texttt{M} \in & \textit{Func} & ::= \texttt{fun}\ \kappa\ \texttt{m}\big(\overline{\texttt{x}} : \overline{\texttt{T}}\big) : \texttt{T} \Rightarrow \texttt{e} \\
\texttt{B} \in & \textit{Behv} & ::= \texttt{be b}\big(\overline{\texttt{x}} : \overline{\texttt{T}}\big) \Rightarrow \texttt{e} \\
\texttt{MS} \in & \textit{FuncStub} & ::= \texttt{fun}\ \kappa\ \texttt{m}\big(\overline{\texttt{x}} : \overline{\texttt{T}}\big) : \texttt{T} \\
\texttt{BS} \in & \textit{BehvStub} & ::= \texttt{be b}\big(\overline{\texttt{x}} : \overline{\texttt{T}}\big)
\end{array}
$$

Figure A.2: Syntax of items

$$
\begin{array}{rcl}
\mathtt{T} \in & \textit{Type} & ::= \mathtt{DS}\ \kappa \\
\mathtt{DS} \in & \textit{TypeID} & ::= \mathtt{A} \mid \mathtt{C} \mid \mathtt{N} \mid \mathtt{S} \\
\mathtt{RS} \in & \textit{RuntimeTypeID} & ::= \mathtt{A} \mid \mathtt{C} \\
\mathtt{I} \in & \textit{AbstractTypeID} & ::= \mathtt{N} \mid \mathtt{S} \\
\kappa \in & \textit{Cap} & ::= \mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref} \mid \mathtt{val} \mid \mathtt{box} \mid \mathtt{tag} \mid \mathtt{iso}\circ \mid \mathtt{trn}\circ
\end{array}
$$

Figure A.3: Syntax of types

$$
\begin{array}{rcl}
\mathtt{e} \in & \textit{Expr} & ::= \mathtt{this} \mid \mathtt{null} \mid \mathtt{e};\mathtt{e} \\
 & & \mid \mathtt{x} \mid \mathtt{x} = \mathtt{e} \\
 & & \mid \mathtt{e.f} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{recover}\ \mathtt{e} \\
 & & \mid \mathtt{e.n}(\overline{\mathtt{e}}) \mid \mathtt{KT.k}(\overline{\mathtt{e}}) \\
\mathtt{E}\langle\cdot\rangle \in & \textit{ExprHole} & ::= (\ \cdot\ ) \mid \mathtt{x} = \mathtt{E}\langle\cdot\rangle \mid \mathtt{E}\langle\cdot\rangle;\mathtt{e} \mid \mathtt{E}\langle\cdot\rangle\mathtt{.f} \\
 & & \mid \mathtt{e.f} = \mathtt{E}\langle\cdot\rangle \mid \mathtt{E}\langle\cdot\rangle\mathtt{.f} = \mathtt{t} \mid \mathtt{recover}\ \mathtt{E}\langle\cdot\rangle \\
 & & \mid \mathtt{E}\langle\cdot\rangle\mathtt{.n}(\overline{\mathtt{t}}) \mid \mathtt{e.n}(\overline{\mathtt{t}}, \mathtt{E}\langle\cdot\rangle, \overline{\mathtt{e}}) \\
 & & \mid \mathtt{KT.k}(\overline{\mathtt{t}}, \mathtt{E}\langle\cdot\rangle, \overline{\mathtt{e}})
\end{array}
$$

Figure A.4: Syntax of expressions

$$
\begin{array}{ll}
 & \mathtt{this}, \mathtt{x} \in \textit{SourceID} \\
\mathtt{C} \in \textit{ClassID} & \mathtt{t} \in \textit{TempID} \\
\mathtt{A} \in \textit{ActorID} & \mathtt{k} \in \textit{CtorID} \\
\mathtt{N} \in \textit{TraitID} & \mathtt{m} \in \textit{FuncID} \\
\mathtt{S} \in \textit{InterfaceID} & \mathtt{b} \in \textit{BehvID} \\
\mathtt{f} \in \textit{FieldID} & \mathtt{n} \in \textit{MethID} = \textit{CtorID} \cup \textit{FuncID} \cup \textit{BehvID}
\end{array}
$$

Figure A.5: Identifiers

A.2 Operational Semantics

$$
\begin{array}{lll l}
\chi & \in & \mathit{Heap} & = \mathit{Addr} \rightarrow (\mathit{Actor} \cup \mathit{Object}) \\
 & & \mathit{Actor} & = \mathit{ActorID} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \times \overline{\mathit{Message}} \times \mathit{Stack} \times \mathit{Expr} \\
 & & \mathit{Object} & = \mathit{ClassID} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\
\mu & \in & \mathit{Message} & = \mathit{MethID} \times \overline{\mathit{Value}} \\
\sigma & \in & \mathit{Stack} & = \mathit{ActorAddr} \cdot \overline{\mathit{Frame}} \\
\varphi & \in & \mathit{Frame} & = \mathit{MethID} \times (\mathit{LocalID} \rightarrow \mathit{Value}) \times \mathit{ExprHole} \\
 & & \mathit{LocalID} & = \mathit{SourceID} \cup \mathit{TempID} \\
v & \in & \mathit{Value} & = \mathit{Addr} \cup \{\texttt{null}\} \\
\iota & \in & \mathit{Addr} & = \mathit{ActorAddr} \cup \mathit{ObjectAddr} \\
\alpha & \in & \mathit{ActorAddr} & \\
\omega & \in & \mathit{ObjectAddr} &
\end{array}
$$

Figure A.6: Runtime entities

$$\frac{\chi, \sigma\cdot\varphi, \mathtt{e} \rightsquigarrow \chi', \sigma\cdot\varphi', \mathtt{e}'}{\chi,\sigma\cdot\varphi, \mathtt{E}\langle\mathtt{e}\rangle \rightsquigarrow \chi',\sigma\cdot\varphi',\mathtt{E}\langle\mathtt{e}'\rangle}\ \textsc{ExprHole} \qquad \frac{\chi, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5 \rightsquigarrow \chi',\sigma,\mathtt{e}}{\chi \rightarrow \chi'[\alpha\mapsto(\sigma,\mathtt{e})]}\ \textsc{Global}$$

$$\frac{}{\chi,\sigma\cdot\varphi,\mathtt{t};\mathtt{e} \rightsquigarrow \chi,\sigma\cdot\varphi,\mathtt{e}}\ \textsc{Seq}$$

$$\frac{\mathtt{t}\notin\varphi \qquad \varphi' = \varphi[\mathtt{t}\mapsto\varphi(\mathtt{x})]}{\chi,\sigma\cdot\varphi,\mathtt{x} \rightsquigarrow \chi,\sigma\cdot\varphi',\mathtt{t}}\ \textsc{Local} \qquad \frac{\begin{array}{c}\mathtt{t}'\notin\varphi \\ \varphi' = \varphi[\mathtt{x}\mapsto\varphi(\mathtt{t}),\mathtt{t}'\mapsto\varphi(\mathtt{x})]\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{x}=\mathtt{t} \rightsquigarrow \chi,\sigma\cdot\varphi',\mathtt{t}'}\ \textsc{AsnLocal}$$

$$\frac{\mathtt{t}'\notin\varphi \qquad \varphi' = \varphi[\mathtt{t}'\mapsto\chi(\varphi(\mathtt{t}),\mathtt{f})]}{\chi,\sigma\cdot\varphi,\mathtt{t.f} \rightsquigarrow \chi,\sigma\cdot\varphi',\mathtt{t}'}\ \textsc{Fld} \qquad \frac{\begin{array}{c}\mathtt{t}''\notin\varphi \qquad \varphi' = \varphi[\mathtt{t}''\mapsto\chi(\varphi(\mathtt{t}),\mathtt{f})] \\ \chi' = \chi[\varphi(\mathtt{t}),\mathtt{f}\mapsto\varphi(\mathtt{t}')]\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{t.f}=\mathtt{t}' \rightsquigarrow \chi',\sigma\cdot\varphi',\mathtt{t}''}\ \textsc{AsnFld}$$

$$\frac{\begin{array}{c}\iota = \varphi(\mathtt{t}) \\ \mathtt{RS} = \chi(\iota)\downarrow_1 \qquad (\overline{\mathtt{x}},\mathtt{e}) = \mathcal{M}r(\mathtt{RS},\mathtt{m}) \\ \varphi'' = (\mathtt{m},[\,\mathtt{this}\mapsto\iota,\overline{\mathtt{x}}\mapsto\overline{\varphi(\mathtt{t})}\,],\cdot) \\ \varphi' = (\varphi\downarrow_1,\varphi\downarrow_2,\mathtt{E}\langle\cdot\rangle)\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{E}\langle\mathtt{t.m}(\overline{\mathtt{t}})\rangle \rightsquigarrow \chi,\sigma\cdot\varphi'\cdot\varphi'',\mathtt{e}}\ \textsc{Sync} \qquad \frac{\begin{array}{c}\mathtt{E}\langle\cdot\rangle = \varphi\downarrow_3 \qquad \mathtt{t}'\notin\varphi \\ \varphi'' = (\varphi\downarrow_1,\varphi\downarrow_2[\mathtt{t}'\mapsto\varphi'(t)],\cdot)\end{array}}{\chi,\sigma\cdot\varphi\cdot\varphi',\mathtt{t} \rightsquigarrow \chi,\sigma\cdot\varphi'',\mathtt{E}\langle\mathtt{t}'\rangle}\ \textsc{Return}$$

$$\frac{\begin{array}{c}\alpha = \varphi(\mathtt{t}) \\ \overline{\mu} = \chi(\alpha)\downarrow_3 \qquad \mu = (\mathtt{b},\overline{\varphi(\mathtt{t})}) \\ \chi' = \chi[\,\alpha\mapsto\overline{\mu}\cdot\mu\,]\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{t.b}(\overline{\mathtt{t}}) \rightsquigarrow \chi,\sigma\cdot\varphi,\mathtt{t}}\ \textsc{Async} \qquad \frac{\begin{array}{c}\mathtt{A} = \chi(\alpha)\downarrow_1 \qquad (\mathtt{n},\overline{v})\cdot\overline{\mu} = \chi(\alpha)\downarrow_3 \\ (\overline{\mathtt{x}},\mathtt{e}) = \mathcal{M}r(\mathtt{A},\mathtt{n}) \\ \varphi = (\mathtt{n},[\,\mathtt{this}\mapsto\alpha,\overline{\mathtt{x}}\mapsto\overline{v},\cdot\,])\end{array}}{\chi,\alpha,\epsilon \rightsquigarrow \chi[\,\alpha\mapsto\overline{\mu}\,],\alpha\cdot\varphi,\mathtt{e}}\ \textsc{Behave}$$

$$\frac{\begin{array}{c}\omega\notin dom(\chi) \\ (\overline{\mathtt{x}},\mathtt{e}) = \mathcal{M}r(\mathtt{C},\mathtt{k}) \\ \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{C}) \\ \chi' = \chi[\,\omega\mapsto(\mathtt{C},\overline{\mathtt{f}\mapsto\mathtt{null}})\,] \\ \varphi'' = (\mathtt{k},[\,\mathtt{this}\mapsto\omega,\overline{\mathtt{x}}\mapsto\overline{\varphi(\mathtt{t})}\,],\cdot) \\ \varphi' = (\varphi\downarrow_1,\varphi\downarrow_2,\mathtt{E}\langle\cdot\rangle)\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{E}\langle\mathtt{C.k}(\overline{\mathtt{t}})\rangle \rightsquigarrow \chi',\sigma\cdot\varphi'\cdot\varphi'',\mathtt{e}}\ \textsc{Ctor} \qquad \frac{\begin{array}{c}\alpha\notin dom(\chi) \\ \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{A}) \qquad \mu = (\mathtt{k},\overline{\varphi(\mathtt{t})}) \\ \chi' = \chi[\alpha\mapsto(\mathtt{A},\overline{\mathtt{f}\mapsto\mathtt{null}},\mu,\alpha,\epsilon)] \\ \mathtt{t}\notin\varphi \qquad \varphi' = \varphi[\mathtt{t}\mapsto\alpha]\end{array}}{\chi,\sigma\cdot\varphi,\mathtt{A.k}(\overline{\mathtt{t}}) \rightsquigarrow \chi',\sigma\cdot\varphi',\mathtt{t}}\ \textsc{Ator}$$

$$\frac{\mathtt{t}\notin\varphi \qquad \varphi' = \varphi[\mathtt{t}\mapsto\mathtt{null}]}{\chi,\sigma\cdot\varphi,\mathtt{null} \rightsquigarrow \chi,\sigma\cdot\varphi',\mathtt{t}}\ \textsc{Null} \qquad \frac{\varphi(\mathtt{t}) = \mathtt{null}}{\begin{array}{c}\chi,\sigma\cdot\varphi,\mathtt{t.f} \rightsquigarrow \chi,\sigma\cdot\varphi,\mathtt{t} \\ \chi,\sigma\cdot\varphi,\mathtt{t.f}=\mathtt{t}', \rightsquigarrow \chi,\sigma\cdot\varphi,\mathtt{t} \\ \chi,\sigma\cdot\varphi,\mathtt{t.n}(\overline{\mathtt{t}}) \rightsquigarrow \chi,\sigma\cdot\varphi,\mathtt{t}\end{array}}\ \textsc{Except}$$

$$\frac{}{\chi,\alpha\cdot\sigma,\mathtt{t} \rightsquigarrow \chi,\alpha,\epsilon}\ \textsc{ReturnBe} \qquad \frac{}{\chi,\sigma,\mathtt{recover}\ \mathtt{t} \rightsquigarrow \chi,\sigma,\mathtt{t}}\ \textsc{Rec}$$

Figure A.7: Execution

Appendix B

Coq implementation of the typechecker

This in a excerpt of our implementation of the typechecker in Coq. The rest of the source is provided in the archive.

```
Section checker.
Context (P: program).

Fixpoint ck_expr (Γ : ty_ctx) (e: expr) : option ty :=
  let ck_alias (e: expr) (expected: ty) : option unit :=
    ety ← ck_expr Γ e;
    subtype_ty P ety (unalias expected)
  in

  match e with
  | expr_null ⇒ Some ty_null

  | expr_seq e1 e2 ⇒
      ty1 ← ck_expr Γ e1;
      ty2 ← ck_expr Γ e2;
      Some ty2

  | expr_local x ⇒ Γ !! x

  | expr_assign_local x e ⇒
      lhs_ty ← Γ !! x;
      _ ← ck_alias e lhs_ty;
      Some (unalias lhs_ty)

  | expr_field e f ⇒
      base_ty ← ck_expr Γ e;
      match base_ty with
      | ty_name ds cap ⇒
          field_ty ← lookup_F P ds f;
          cap ⊳ field_ty
      | ty_null ⇒ None
      end

  | expr_assign_field e f e' ⇒
      base_ty ← ck_expr Γ e;
      match base_ty with
      | ty_name ds cap ⇒
          field_ty ← lookup_F P ds f;
          _ ← ck_alias e' field_ty;
          cap ⊳ field_ty
      | ty_null ⇒ None
      end

  | expr_ctor kt k es ⇒
      '( _, args, retty) ← lookup_Md P kt k;
      _ ← ck_args Γ es (map snd args);

      Some retty

  | expr_call e0 m es ⇒
      baset ← ck_expr Γ e0;
      match baset with
      | ty_name ds cap ⇒
          '( _, args, retty) ← lookup_Md P ds m;
          _ ← ck_args Γ es (map snd args);

          Some retty
      | ty_null ⇒ None
      end

  | _ ⇒ None
  end
with
  ck_args (Γ : ty_ctx) (es: list_expr) (args: list ty) : option unit :=
    let ck_alias (e: expr) (expected: ty) : option unit :=
      ety ← ck_expr Γ e;
      subtype_ty P ety (unalias expected)
    in

    match es, args with
    | list_expr_nil, nil ⇒ Some ()
    | list_expr_cons e es', arg :: args' ⇒
        _ ← ck_alias e arg;
        ck_args Γ es' args'
    | _, _ ⇒ None
    end
```

```
.

End checker.
```