Imperial College London

Department of Computing

Imperial College of Science, Technology and Medicine

Strengthening Pony's capabilities logic for verification

Author:
Isaac van Bakel

Supervisor:
Sophia Drossopoulou

June 17, 2020

Submitted in partial fulfillment of the requirements for the JMC MEng of Imperial College London

Acknowledgements

I would like to thank Professor Sophia Drossopoulou for supervising this project, for guiding the writing of this report, and for putting up with a someties erratic schedule. Her technical brilliance, and her ability to adapt to many abuses of good notation that were involved in the drafts of this document and its ideas, were invaluable.

I would also like to thank my parents, both for the obvious reasons and for taking the time and effort to read my drafts, discuss my project, and help advise my writing. Without them, this document would be very different, and much more unreadable.

Contents

Abstract

This project presents a new formal model for describing and modelling the *Pony* programming language, an actor-based concurrent language. Designed around strong static concurrency guarantees, Pony uses a permission system to control *mutable access to data*; checking the correctness of this system has been the topic of previous models.

The new model aims to simplify the problem of *accessibility* of heap values. To this end, it describes a completely independent and novel *heap region system*. The region system is then used to present simplified well-formedness definitions for heaps.

Several results about the new model are then proven, including both existing properties found in older systems as well as new results from the region system. These proofs, utilising the features of the new model, are shown to be considerably simpler than older iterations.

Chapter 1

Introduction

An increasing portion of modern programming language research is being done with the goal of designing languages which only produce, to the furthest extent possible, *correct programs*. To this end, new languages are incorporating an ever-greater degree of *static analysis* features, which detect errors before the program is ever executed - and these features are themselves being backed by research in *formal models*, which mathematically describe and verify the correctness properties such static analysis claims to enforce.

A particularly active area of research is in the realm of *concurrent programs*, which are specially written to utilise multiple simulatenous computations to perform more work, more efficiently, and at a larger scale. The difficulty in writing correct concurrent programs has inspired new *concurrent languages*(Pony 2019; Rust 2020), which are specifically designed to tackle common correctness and performance mistakes that occur in such programs.

Concurrent languages with correctness features typically implement them as *zero-cost abstractions* - features which don't exist at runtime - so that they do not impact performance. However, this approach has an inherent risk: if the language does not provide the correctness guarantees it itself relies on, programs written in the language may end up incorrect. For this reason, formal models which check the correctness properties of concurrent languages are particularly valuable.

1.1 Pony

Pony (Chapter 3) is a concurrent programming language which claims to provide strong correctness guarantees for all of its programs. It does so through a combination of a restrictive concurrent paradigm (the *actor model*, Chapter 2) and compile-time type system features (*capabilities*, Section 3.2).

The main focus of Pony's concurrency guarantees is in preventing *shared mutability* of data, where some value can be read from and written to by multiple actors simultaneously. Pony's main claim is that its capability system completely forbids shared mutability of data, while avoiding unnecessary data duplication - normally a technical restriction of actor model implementations (Section 2.3).

1.2 Accessibility reasoning

Reasoning about the correctness of Pony's claims is a difficult problem which has been the subject of much recent work since its first published design in 2015. Efforts

to formalise the capability system have even found correctness bugs in the Pony compiler (Steed 2016). Previous attempts to verify its correctness claims have encountered complexity in the area of *accessibility* - specifically, in proving that access of a value with conflicting capabilities is impossible.

The approaches applied to reasoning about accessibility have so far centred around *paths* - sequences of connected values in the Pony *object graph* (Section 3.1.1). This approach has been limited by being *non-local* - changes to a single object could have far-reaching effects in the graph, adding or removing many paths at once. These path considerations have complicated reasoning about accessibility changes over the execution of a program.

1.3 Contributions

In this report, we present a new formal model of Pony, including a redesigned syntax and semantics, called *Pony with explicit aliasing*.

In the syntax (Chapter 4), we introduce *explicit temporaries* (Section 4.2) to eliminate the continuation-style of expressions from previous models. We also define a side-effecting typing judgement (Section 4.3.1), which allows expressions to modify what variables are in scope and how they are typed.

As part of the heap definitions (Chapter 5) for the semantics, we present a novel *heap region system* (Section 5.2), designed to describe Pony's capability guarantees. Using the region system, we improve on previous models in our definition of *well-formed heaps* (Section 5.3). In particular, we manage to simplify the problem of accessibility by, instead of considering paths in the object graph, using regions to control permitted capability values.

We make further modifications to previous models by allocating messages and frames in the heap, motivated by the new region system. In the operational semantics themselves (Chapter 6), we leverage our modifications to the syntax to give a new "big-step" presentation of the semantics of *Pony with explicit aliasing*, while arguing that the resulting model still has meaningful concurrency properties.

We go on to state and discuss several key results of the new Pony model (Chapter 7), including: soundness - that evaluated expressions yield their expected type (Section 7.1); progress - that programs can always be executed (Section 7.2); preservation of heap well-formedness - that programs cannot violate capability guarantees (Section 7.3); deadlock prevention - that programs cannot enter a non-complete stuck state (Section 7.4); and data race prevention - that programs do not incorrectly share memory (Section 7.5). We describe how we define these properties and what their consequences are.

Finally, we give a brief exploration into possible extensions of the model, including a partial implementation of *borrowing* in Pony (Chapter 9). We describe how the new model is better-suited to this extension than its predecessors, as well as the technical challenges involved.

Partial proofs of the results of this report are encoded in Coq. We describe the encoding and its features, as well as limitations of the proof system (Appendix A).

Chapter 2

Actor model

The actor model is a design paradigm for programming in concurrent systems, initially described in Hewitt, Bishop, and Steiger 1973.

In abstract terms, the actor model represents concurrent elements of computation as *actors*, which can only interact through (asynchronous) message passing. Actors may have private, mutable state, but data shared between actors cannot be mutated. In particular, naive data races - when there exist different possible interleavings of read and write operations on the same data, affecting execution outcomes - are impossible.

2.1 Data races

The actor model is intended for concurrent systems with elements of non-determinism. This is common in reality, particularly in the domain of modern computer programs. A single program may have multiple concurrent units in terms of threads, and aspects of how threads execute are non-deterministic - for example, which threads are executing at any one time, which order threads are started in, which order threads access some shared resource, etc.

In such a non-deterministic concurrent system with shared mutable resources, data races become possible.

Definition 2.1 (Data race).
A data race is any point in the program where more than one concurrent unit can simultaneously read from and write to the same resource.

Where a data race exists, the non-determinism of the concurrent system can affect the outcome of execution in different units - often in inconsistent ways.

For example, consider the program

```
x = 1
y = 2
{ x = y } || { y = x }
```

where || is the *parallelism* operator, which declares that the two instructions `x = y` and `y = x` should be run by different concurrent units in parallel. Since both concurrent units can read from and write to shared data in the form of the variables `x`, `y`, there is a data race. Indeed, there are two possible executions of the above program:

```
x = 1
y = 2
x = y
y = x
```

such that x = y = 2.

```
x = 1
y = 2
y = x
x = y
```

such that x = y = 1.

The two executions have different outcomes, and which execution occurs is non-deterministic.

2.2 Shared mutability

In order to prevent data races, the actor model forbids shared mutable resources entirely. If every unit that has access to some particular resource can only read from it (or equivalently, only write to it), the non-determinism of the concurrent system does not affect the execution of such reads and writes.

Due to this lack of shared mutability, the only instructions which affect multiple units are those which either send or receive messages - in this way, the actor model reduces the analysis of concurrent executions from all possible interleavings of instructions to just all possible interleavings of messages.

2.3 Limitations of safe resource sharing

This lack of shared mutability is a frustrating restriction when considering common applications of concurrent programming - naive (and safe) approaches, which copy all data sent between actors, would introduce massive inefficiencies parallelising computation over large datasets. In effect, this would completely eliminate from consideration the actor model as a concurrent abstraction in many parts of industry - which cannot, or do not want to, suffer this performance impact.

For this reason, concrete implementations in compiled actor programming languages often manage the separate classes of private and shared data through *static modifiers*. These modifiers allow for the compiler to reason about mutability as a compile-time concept, preventing the mutation of shared data as a compiler error. In the compiled low-level instructions, data can then be shared by reference, without being copied. An implementation is discussed in Chapter 3.2.

List of definitions in this chapter

Chapter 3

Pony - an actor-based language

Pony (Pony 2019) is a programming language which allows users to write programs using the actor model abstraction (Chapter 2). Pony was initially developed by Sylvan Clebsch in 2014 (Clebsch 2017), before being open-sourced in 2015.

Pony was created to combat issues with actor implementations in more general-purpose programming languages. Existing implementations were burdened by language constructs which were not compatible with actor-based concurrency, and the mismatch was causing performance and correctness issues. In particular, shared mutable data (from pointers) and deadlocks (from synchronisation primitives) were both common issues that limited productivity. Meanwhile, higher-level implementations such as Erlang were more correct but much less performant, due to their decision to copy data rather than share it (Section 2.3) (Clebsch 2017).

Pony is a statically-typed, object-oriented language. Actors are implemented as an extension to a typical object-oriented model. Pony manages the problem of shared mutability with *capabilities* (Section 3.2), type-level markers which define how shareable data is accessed by objects and actors. As a consequence of its static features, Pony claims to be *deadlock-free* (Section 3.3). An example of how Pony programs benefit from the capability system is given in Section 3.4.

Capabilities in particular, and Pony's correctness claims in general, have been the topic of multiple previous works (Section 3.5). These form the basis for a new model, *Pony with explicit aliasing*, which is described in Chapters 4, 5, and 6.

3.1 Objects and actors

Similar to many other popular object-oriented languages, Pony allows for declaration of *classes* (object types), which store instance information in fields and have methods which can be called on them. For example, a `Counter` class can be declared as:

```
class Counter
  var count : int ref

  new Counter() => None

  fun box value() : int ref => this.count

  fun ref increment() : None ref =>
    this.count = this.count + 1
```

Each object, which is an instance of a particular class, stores a copy of its own fields. Special methods, called *constructors*, allow for initialising new instances of a class. Other methods, called *functions*, can only be called on an existing instance; and doing so requires a reference to the instance with the correct *capability*, which is discussed later on.

Actors are defined similarly to objects: they also have fields, constructors, and functions. However, actors are distinguished because they also define *behaviours* (Pony 2020): a kind of asynchronous method. In high-level terms, each behaviour represents a possible message variant that the actor is able to receive, as well as the code that the actor will execute when that message is received. For example, we could change `Counter` to be an actor:

```
actor CounterActor
  var count : int ref

  new Counter() => None

  fun box value() : int => this.count

  fun ref increment() : None =>
    this.count = this.count + 1

  be setValue(value : int) =>
    this.count = value
```

In this definition, `Counter` now accepts messages of the form `setValue(int)`; and whenever it is able to process such a message, the value of `count` will be updated to the value given when the message was initially sent.

Instances of actors, like objects, store copies of their own fields. Additionally, actors store a message queue, which contains any messages they have yet to process; and a stack frame, which represents the current execution. The stack frame is discussed in more detail in Section 3.1.2.

The two main differences between functions and behaviours are: behaviours have a fixed receiver capability, and functions are synchronous while behaviours are asynchronous - these differences are described in Section 3.1.2 and 3.2.1.

3.1.1 The object graph

Definition 3.1 (Object graph).
The object graph for a particular heap in an object-oriented language is the directed abstract graph given by:

- *Taking objects in the heap as nodes, and*
- *Taking each field on an object as a directed arc starting from the object which stores the field to the object referred to by the field's value.*

In Pony, the object graph includes both objects and actor instances, since they both have fields. Paths in the object graph represent chains of fields which allow one object to access another. This makes the object graph a useful abstraction for describing accessibility.

3.1.2 Synchronous and asynchronous methods

In Pony's implementation of the actor model, all methods (and hence all code in a program) are executed by actor instances. Each actor behaves as a sequential program, and executes at most one method at any time. Actors execute in parallel - so there may be as many methods executing at one time as there are actor instances in the execution.

Actors are responsible for passing method arguments, running method bodies, and returning any values to the method caller - where the caller is that actor, or some object. How these responsibilities are handled depends on if the method is synchronous or asynchronous.

Definition 3.2 (Synchronous methods).
Synchronous methods, when called, are run immediately by the actor executing the code in which they were called, using a new frame in the stack frame *for that actor. The called method is run to completion (including any method calls in its body), and the value returned can be used in the caller code. The caller code does not continue to execute until the synchronous method is finished.*

Functions on both objects and actors are synchronous, as well as constructors on objects.

Definition 3.3 (Asynchronous methods).
Asynchronous methods are executed at some point after being called, possibly by some other actor than the one executing the caller code. The caller code continues executing immediately after the method call, and cannot *wait for the called method to finish executing: for this reason, asynchronous methods do not return values.*

Behaviours and constructors on actors are asynchronous.

When an asynchronous call is made, it is placed in the *message queue* for some actor - this allows actors to handle messages which arrive while they are executing code. An actor will begin executing the next message in its queue once its current execution has ended. Processing a message is possible only when the stack frame is empty - and the act of processing the next message causes the creation of a new frame. The actor which processes the method call depends on the type of method which is called: an actor will execute the constructor call which initialises it; actors will execute the bodies of messages sent to them.

3.2 Capabilities

It is beneficial for performance reasons for an actor-based programming language to be able to avoid copying shared data wherever possible (Section 2.3), in particular through the use of static analysis. To achieve this goal, Pony uses capabilities (Clebsch 2017).

In Pony, variables have a "capability" - a type-level marker which defines the permissions required by the variable and any other aliases of the data it contains. Capabilities decide if the data contained in a variable can be read from or written to by using

- the variable itself

- another variable which refers to the same data, contained in the same actor's code (a *local* alias)
- another variable which refers to the same data, contained in some other actor's code (a *global* alias)

Noting the restrictions of the actor model, it is immediately clear that no capability can allow for a variable to write to data so long as a global alias may read from the same data (or vice versa).

Definition 3.4 (Pony capabilities).
The capabilities available for variables in Pony (Clebsch et al. 2015) are

isolated `iso` - *which guarantees unique ownership of some data, so that it can then be mutated safely, as well as sent between actors without copying*

transition `trn` - *which guarantees unique mutable access to some data which may have other readable local aliases, and so cannot be sent between actors*

reference `ref` - *which represents a mutable handle on some data which may have other mutable local aliases, and so cannot be sent between actors*

value `val` - *which represents an immutable handle on some data and guarantees all handles to it are immutable, so that it can be shared by reference without naive data races*

box `box` - *which represents a readable handle on some data which may have other mutable local aliases, and so cannot be sent between actors*

tag `tag` - *which represents a handle on data which does not allow reading or writing, though other aliases may perform any operation on the data - because of this,* `tag` *can be sent between actors.*

When compiling a Pony program, the compiler is able to both check and use the capabilities of variables to verify that (im)mutability guarantees are not violated. For example, only variables with capabilities that have the same permissions for local and global aliases may safely be part of a message (Clebsch et al. 2015).

For an example of capabilities, consider a program with three variables: `x`, `y`, and `z`, such that `x` has capability `trn`, `y` has capability `val`, and `z` has capability `box`. Since `y` has the `val` capability, no mutable handle to the value of `y` can exist - but `trn` requires a mutable handle, so `x` and `y` cannot refer to the same value. However, since `val` and `box` both allow other handles to read from their value, it is possible for `y`, `z` to refer to the same value. Additionally, since `box` allows for other handles to write to its value, and `trn` allows other handles to read from its value, it is also possible for `x`, `z` to refer to the same value.

3.2.1 Methods and the receiver capability

Since methods can allow for both reading from and writing to data, they must play a role in the capability system. In particular, calling an object method which reads from that object's fields is equivalent to reading the fields directly - and similarly for writing. Therefore, from the perspective of data race prevention, the methods

which can be called on an object (or actor) depend on the capability with which the method call is made.

This is the role of the *receiver capability*.

Definition 3.5 (Receiver capability).
The capability required to call a particular method on an object or actor. The receiver capability decides, but may not be identical to, the capability of the receiver variable `this` *in a synchronous or asynchronous method.*

For functions, the receiver capability is specified by the function definition. Fields of the object on which the function is called are only readable (or writable) if the receiver capability allows it - for example, a function with receiver capability `tag` could not read nor write to any object fields. Calling a function with some receiver capability requires the object reference to have *at least* that capability. For object constructors, the receiver capability defaults to `ref`.

For actors, the receiver capability works differently. Since actors do not share state, each actor instance effectively has unique ownership over all its own data - so in actor constructors and behaviours, the actor is treated as an `iso`. However, this does not mean that calling a behaviour on an actor requires a `iso` reference to that actor - this would be unworkable. Instead, the receiver capability for calling behaviours is `tag` - it is not necessary to be able to read from, or write to, an actor's fields to call a behaviour on it.

3.3 Deadlocks

One significant property of Pony's asynchronous methods, and therefore its behaviours, is that Pony provides no mechanism to wait for an asynchronous call to complete. When an actor sends a message to another, it cannot in the same frame delay execution until that message has been received. Additionally, since actors automatically process the next message in the queue once the frame is empty, no frame can cause an actor to ignore certain messages.

As a consequence of these features, Pony claims to be *deadlock-free*.

Definition 3.6 (Deadlock).
A program state where execution is not complete, but progress is impossible.

To see an example of a deadlock, consider the use of *semaphores*, a typical synchronisation concept (Dijkstra 1963). If a program initialises two concurrent units with the code

```
down(a)
down(b)
...
up(b)
up(a)
```

```
down(b)
down(a)
...
up(a)
up(b)
```

then each unit will attempt to `down` the semaphores `a`, `b`. However, due to their different orderings, it is possible for both units to simultaneously `down` one semaphore and not the other. As a consequence, neither unit can progress - each is waiting for the other to `up` their respective semaphore. In this case, the system has deadlocked.

In `User` 1:

```
var list : list[String]
  = this.listOfItems;

list.push(item);

this.listOfItems = list
```

In `User` 2:

```
var list : list[String]
  = this.listOfItems;
list.push(item);

this.listOfItems = list
```

Figure 3.1: A possible interleaving of two calls to `addItem`. The local variables `list` are considered separate, rather than shared between the two concurrent units.

In Pony, such simple synchronisation concepts are impossible to implement - and also unnecessary. Since actors cannot share mutable memory, it is never necessary for multiple actors to need to synchronise access to a memory region.

3.4 An example in Pony: a shared shopping list

Consider an implementation of a concurrent shopping list, shared between multiple users of an app. Each user can add items to the list, and users can do so independently of each other.

```
class ShoppingList
  var listOfItems : List[String] ref

  fun ref addItem(item : String) : None =>
     var list : List[String] = this.listOfItems;
     list = list.push(item);
     this.listOfItems = list

actor User
  var shoppingList : ShoppingList ref // somehow shared

  be addItemToList(item : String) =>
    ...
    shoppingList.addItem(item)
    ...
```

In a traditional (i.e. not actor-based) concurrency model, two users, each a separate concurrent unit, might try to add items to the shopping list at the same time. Since the call to `addItem` is synchronous, both concurrent units would execute that code. From the perspective of the `ShoppingList` object, their executions of the function would be interleaved (an example is shown in Figure 3.1).

While some possible interleavings would result in each user's item being added to the shopping list, many would not. This is a consequence of a data race in the calls to `addItem`, since the `shoppingList` variable is both shared and mutable.

3.4.1 With actors

In fact, we know that the Pony capability system would forbid any version of `shoppingList` from being both shared and mutable. How then can it be edited by multiple users at the same time? By converting `ShoppingList` to be an actor, and making `addItem` a behaviour.

```
actor User
  var shoppingList : ShoppingList tag
  ...

actor ShoppingList
  var listOfItems : List[String] ref

  be addItem(item : String) =>
     var list : List[String] = this.listOfItems;
     list = list.push(item);
     this.listOfItems = list
```

In this version of the code, two separate `User`s can still call `addItem` at the same item. However, the `ShoppingList` actor will handle their messages sequentially, and the instructions will not be interleaved. This prevents the data race.

3.4.2 With messages

Alternatively, we could leave `ShoppingList` as an object, sending it *between* the `User`s when necessary.

```
actor User
  var shoppingList : (ShoppingList | None) iso // possibly null

  var friend : User tag
  var backlog : List[String] ref

  be addItemToList(item : String) =>
    ...
    if shoppingList == None
      then
        backlog.push(item)
        friend.requestList()
      else
        shoppingList.addItem(item)
    end
    ...

  be requestList()
    if shoppingList != None
      then
        var list = (shoppingList = None)
        friend.receiveList(consume list)
    end
```

```
be receiveList(list : ShoppingList iso) =>
  shoppingList = consume list

  for item in backlog
    shoppingList.addItem(item)
  end
```

In this version of the code, each `User` may or may not be holding a *unique reference* to the shopping list (`(ShoppingList | None) iso`). Whenever a user wants to add an item, they first check if they have access to the list - if they don't, they request access from a friend using `requestList`, and put the item in a backlog. When a friend requests the list, the user holding it *gives up* their unique reference (`shoppingList = None`) and sends the list to the friend. The user who originally requested the list then has a unique reference to it (`shoppingList = consume list`) and can add all the items in their backlog.

This version also demonstrates a consequence of Pony's deadlock prevention - because a `User` cannot simply request the list and stop all progress until they receive it, they must put items in the backlog instead.

3.5 Formal models

The formalisation of Pony, and the verification of its various properties, has the been the topic of several previous works: including Clebsch et al. 2015, Steed 2016.

3.5.1 Clebsch et al. 2015

The first paper outlining a formal definition of Pony's capability system, Clebsch et al. 2015 was the first paper to introduce several key features shared by future models:

- The simplified Pony syntax subset - and in particular, a syntax with support for continuations, so that meaningful analysis of concurrent execution was possible.
- Viewpoint adaptation (Section 4.3.4) and safe-to-write (Section 4.3.5), which describe how Pony capabilities interact with each other and the basic operations of the language, specifically reading from and assigning to fields.
- The ephemeral capabilities (Section 4.3.2), which categorise values which have been *unaliased*.

Clebsch et al. 2015 was based on earlier work in Gordon et al. 2012, which described a formal system of *variable permissions* for safe parallelism, similar to Pony capabilities. Gordon et al. also introduced the concept of *recovery of isolated references* - implemented in Pony as the `recover ... end` statement.

3.5.2 Steed 2016

An Imperial College London final-year project, Steed 2016 largely improved on the foundation of Clebsch et al. 2015. It used a similar syntax and semantics, but produced a more detailed version of the capabilities system. Included in these improvements were:

- A improved version of the subcapability relation (Section 4.3.3) which also accounted for ephemeral capabilities. Steed also identified and corrected an error in the original definition - specifically, that `iso` <: `trn` allowed for incorrect aliasing.
- Better definitions of viewpoint adaptation, and the distinction between the extracting and non-extracting adaptation versions.
- Extensions of the syntax subset to include more advanced type features - such as inheritance, union types, and tuples. These types were included in the full Pony language, but had never been formalised.

Building on these improvements, Steed 2016 also defined the concept of *well-formed visibility* - a property of heaps in the semantics - which expressed that values in the heap had their expected capability. It went on to prove the major result of *preservation of well-formed visibility*: that the semantic rules preserved this property.

List of definitions in this chapter

Chapter 4

Pony with explicit aliasing - Syntax

We present a modified version of the language as considered in Clebsch et al. 2015, which is itself a simplified subset of the full Pony language.

We make several major changes to this subset:

- We make all aliasing explicit in the syntax, through the use of the $\alpha(\texttt{e})$ construction (Section 4.1).
- We consider expressions only in a type of *three-address notation* - this eliminates the need for expression continuations as used in the original paper. We model multiple successive field accesses through the use of *temporaries* (Section 4.2).
- We do not use **null** to `consume` variables. Instead, variable declarations are explicit at the expression level, and both declarations and `consume` modify the scope. This gives a *side-effecting* typing judgement (Section 4.3.1).
- We update the terminology of special capabilities `iso`$\circ$, `trn`$\circ$, and `ref`$\circ$ with Pony's own *ephemeral* (`iso^`, `trn^`, ...) capability syntax, which fulfil the same role (Section 4.3.2).
- We redefine *well-formed programs* (Section 4.4) to introduce additional restrictions on the syntax - such as the requirement that temporaries be used after being declared.

This modified version is shown in Figures 4.1, 4.2, 4.3. These changes are mainly done for simplicity and power in the resulting type system and semantics.

`A`	$\in$	Actor ID	`C`	$\in$	Class ID
t	$\in$	Temporary	`x`	$\in$	Variable
`m`	$\in$	Function ID	`b`	$\in$	Behaviour ID
`k`	$\in$	Constructor ID	`f`	$\in$	Field ID

Figure 4.1: Definition of identifiers

β	$\in$	Base Capability	:=	`iso` | `trn` | `ref` | `val` | `box` | `tag`
κ	$\in$	Capability	:=	β | β`^`
`S`	$\in$	Type ID	:=	`A` | `C`
`AT`	$\in$	Alias Type	:=	`S` β
`T`	$\in$	Type	:=	`S` κ
`p`	$\in$	Path	:=	`x` | (`consume x`) | t
`MC`	$\in$	Method Call	:=	α(`p`)`.m(`$\overline{\alpha(\texttt{p})}$`)` | α(`p`)`.b(`$\overline{\alpha(\texttt{p})}$`)` | `S.k(`$\overline{\alpha(\texttt{p})}$`)`
`RHS`	$\in$	Assignment RHS	:=	`p` | `p.f` = α(`p`) | `recover RHS end` | `MC`
`e`	$\in$	Expression	:=	`var x` | `x` = α(`RHS`) | t `<- p.f`
`E`	$\in$	Expression Sequence	:=	`p` | `e;E`

Figure 4.2: Definition of types and expressions

`F`	$\in$	Field Def	:=	`var f : AT`
`K`	$\in$	Constructor Def	:=	`new k(`$\overline{\texttt{x : AT}}$`)` $\Rightarrow$ `E`
`M`	$\in$	Function Def	:=	`fun` β `m(`$\overline{\texttt{x : AT}}$`) : T` $\Rightarrow$ `E`
`B`	$\in$	Behaviour Def	:=	`be b(`$\overline{\texttt{x : AT}}$`)` $\Rightarrow$ `E`
`CD`	$\in$	Class Def	:=	`class C` $\overline{\texttt{F}}$ $\overline{\texttt{K}}$ $\overline{\texttt{M}}$
`AD`	$\in$	Actor Def	:=	`actor A` $\overline{\texttt{F}}$ $\overline{\texttt{K}}$ $\overline{\texttt{M}}$ $\overline{\texttt{B}}$
`P`	$\in$	Program	:=	$\overline{\texttt{CD}}$ $\overline{\texttt{AD}}$

Figure 4.3: Definition of actors, classes, and programs

4.1 Explicit aliasing

In Pony, many operations result in the creation of an alias. In the code

```
var x;
var y;
var z;
x = y.m(z);
```

three aliases are created. The most obvious one is the alias created by explicitly assigning the result of the function `m` to the variable `x`. However, the two others are implicit: `z` is implicitly aliased as an argument to the function `m`, and `y` is implicitly aliased as the receiver of that function.

These aliases, which are less obvious, come from the fact that the call to `m` will result in a new frame on the stack. In that new frame, the first argument will refer to the value of `z`: so it will be an alias. However, the new frame will *also* have an alias to the value of `y`, through the variable `this`, which is defined in every frame as the receiver.

In *Pony with explicit aliasing*, all of these aliases are marked by the alias construction α.

Definition 4.1 (Explicit alias construction (α)).
The explicit alias construction α marks the points in a Pony with explicit aliasing *program where values are aliased.*

This allows for a clearer tracking of when aliases are created, and simplifies reasoning about them. The above code would then become:

```
var x;
```

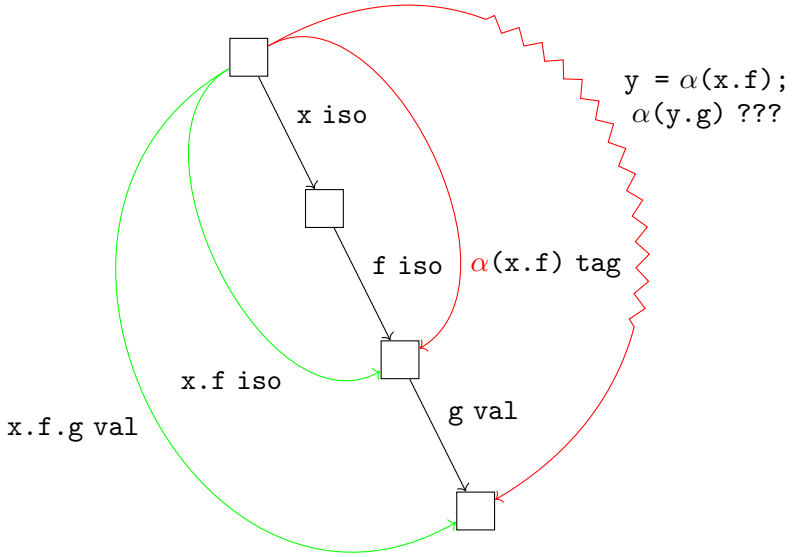


Figure 4.4: An example of multiple field accesses. In Pony, "field chains" are possible, so the green arrows are permitted. In *Pony with explicit aliasing*, "field chains" are forbidden, so there is no equivalent red arrow. Instead, *temporaries* are used.

```
var y;
var z;
x = α(α(y).m(α(z)));
```

4.2 Temporaries

4.2.1 Partial walks

In order to avoid the continuation-style model as presented in Steed 2016, we choose to only consider expressions in a simpler form, which access the heap at most once. However, just local variables would not be sufficient to encode all Pony programs in this way.

By forbidding "field chains", like `x.f1.f2`, some operations which should be permitted are now impossible. For example, in Figure 4.4, the local variable `x` with capability `iso` can be used to access an object with field `f` with capability `iso`. That object in turn has a field `g` with capability `val`.

If we wanted to access the value of `g`, starting from `x`, we could naively try to access each field one at a time:

```
var y;
var z;
```

```
y = α(x.f)
z = α(y.g) // impossible
```

However, this approach would fail. `x.f`, being an `iso` field on an `iso` variable, can be accessed with the `iso` capability. When aliased, `α(x.f)` has capability `tag` - and so does the variable `y`. Since it is not even possible to read from a `tag` variable, it is not possible to access the field `g` on variable `y`.

To overcome this, we introduce an explicit version of *temporaries*.

Definition 4.2 (Temporary).
A temporary is a non-aliasing accessor into the Pony object graph. Temporaries represent "partial walks" of the graph - they must start from some existing path, and each temporary can be used at most once to either navigate further, producing a new temporary, or to produce an alias.

Temporaries are *not* aliases: the capability of a temporary is identical to the path with which it is initialised.

In this temporaries syntax, the above example program becomes:

```
var z;
t <- x.f
u <- t.g
z = α(u)
```

A visual representation is given in Figure 4.5.

This program is now correct. The temporary t is initialised with the path `x.f`, so it has capability `iso` - but it is *not* an alias of `x.f`. The temporary u is initialised with the path `t.g`, which as a `val` field on a `iso` temporary can be accessed with the `val` capability. Finally, the variable `z` is assigned to the *alias* of u - and as aliasing a `val` value gives a `val` value, `z` has the `val` capability, as expected.

4.2.2 Restrictions on temporaries

The non-alias property of temporaries risks violating capability guarantees. A temporary which has partially-walked the interior of an `iso` object can be used to modify the fields of that object - even if the object itself were sent to another actor. To ensure correctness, actors must not be allowed to hold a temporary which has performed a partial walk after that walk becomes impossible. For this reason, temporaries must be used by an actor for navigation or aliasing immediately after being declared. This condition is enforced on well-formed expressions (Section 4.4.1).

Furthermore, actors which hold multiple temporaries referring to different parts of the same object also risk violating capability guarantees - much for the same reasons. In fact, we can strengthen this restriction to say that *not more than one temporary can exist per actor* at any point in execution for an actor, without loss of expressivity of programs. This condition in enforced on well-formed actors (Section 5.3).

An example of this is shown in Figure 4.6. The local `iso` variable `x` has a `iso` field `f`, the value of which in turn has another `iso` field `g`; but just one temporary at a time is sufficient to extract the value of *both* fields and send them in the same message. In general, there is an interpretation of any Pony expression in this reduced syntax that only requires one temporary at a time.

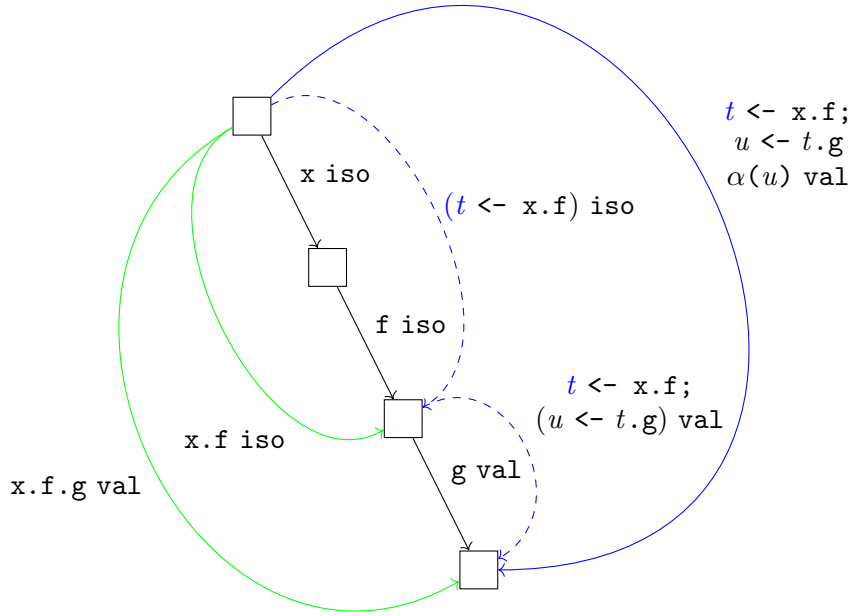


Figure 4.5: An example of temporaries in *Pony with explicit aliasing*. The first temporary t is used to partially walk the object graph. The second temporary u is then a continuation of the walk of the first temporary, and can reach the final value.

```
var a;
var b;

var y;
t <- x.f;
a = α(t.g = α(consume y));

var z;
b = α(x.f = α(consume z));

var r;
r = α(α(x).m(α(consume a), α(consume b)));
```

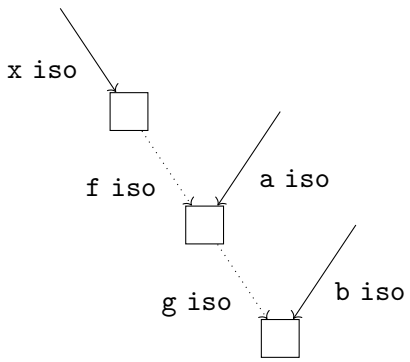


Figure 4.6: An example of how one temporary is sufficient for all possible operations.

The two conditions on temporaries are related: code that requires that a temporary be used immediately after being declared cannot allow actors to declare more than one temporary at a time. However, the second condition is still necessary when considering what heaps are valid at any point in a program's execution - including heaps where some local variables and temporaries have already being initialised. To allow for correctness statements about heaps which are "in progress", it is necessary to restrict temporaries in the semantics as well as the code.

4.2.3 Comparison to previous work

The syntactic construct of temporaries as presented here is closely related to the semantic concept of the same name in Steed 2016. In that work, a temporary is a value obtained by partially evaluating an expression, which is necessary to describe the continuation-style syntax. The distinction is made between *active temporaries*, which have yet to be aliased, and *passive temporaries*, which have been aliased.

Temporaries in this report have a similar role to active temporaries in Steed 2016. They are both a representation of non-aliased *partial walks* of the object graph; furthermore, the restriction exists on active temporaries that *at most active temporary exists per actor* at any time. The modifications made to the syntax in this report make *explicit* the use of active temporaries, by lifting them into the syntax.

4.3 Typing

4.3.1 Typing contexts

Our typing contexts Γ are partial maps of variables and temporaries to types. However, since variables are themselves aliases, we restrict the codomain of variables to be *alias types*. Temporaries, which are not aliases, may be mapped to any Pony type in a typing context.

In order to be able to type `consume` and temporary usage, our typing rules need to be able to affect the typing environment. This gives a *side-effecting* typing judgement.

Definition 4.3 (Typing judgement ($\vdash$)).
The typing judgement for Pony syntax is of the form

$$\Gamma \vdash \mathtt{e} : \mathtt{S}\ \kappa \dashv \Gamma'$$

where Γ is the typing environment before typing `e`*, and Γ' is the resulting environment.*

4.3.2 Ephemeral capabilities

The six Pony capabilities discussed so far (the *base* capabilities) are not sufficient to describe the capabilities of all values in a Pony program.For example, `consume`ing an `iso` variable yields a value which previously had a unique readable and writable reference, but now has no references. Such a value could not only be treated as an `iso`, but also as a `val` (since it is guaranteed to be immutable). In fact, an extension to the capability system is necessary to describe the capability of such a value.

Definition 4.4 (Ephemeral capability (β^)).
An ephemeral capability β^ is the capability produced by somehow removing the alias of an existing value with capability β, either through a consume (`consume x`*) or a destructive read (*`p.f =` α`(p)`*).*

Since an ephemeral value is one without an alias, aliasing it allows for the recovery of the original capability.

As in Clebsch et al. 2015, we define the aliasing map $\mathcal{A}$.

Definition 4.5 (Aliasing map ($\mathcal{A}$)).
The aliasing map $\mathcal{A}$: Capability $\rightarrow$ Capability is defined as

$$\mathcal{A}(\kappa) = \begin{cases} \beta & \kappa = \beta\texttt{\^{}} \\ \texttt{tag} & \kappa = \texttt{iso} \\ \texttt{box} & \kappa = \texttt{trn} \\ \beta & \kappa = \beta, \kappa \neq \texttt{iso}, \kappa \neq \texttt{trn} \end{cases}$$

As `trn` and `iso` are the only capabilities which give a kind of alias uniqueness (for example, `trn` must be the only write alias), they are the only capabilities for which the ephemeral version behaves differently to the non-ephemeral version.

For the other capabilities `ref`, `val`, `box`, and `tag`, we can identify each capability with its ephemeral version, as they are indistinguishable - that is, we can treat `ref` and `ref^` equally. In practice, this means that we should consider whether operations are well-defined with respect to this identity. The ephemeral version is sometimes written here for consistency.

Pony as a full language also has support for *aliased* capabilities, written β`!`. These are equivalent to $\mathcal{A}(\beta)$, and mainly exist for Pony's generic type polymorphism. Since our subset only has monomorphic types, we exclude these capabilities.

4.3.3 Subtyping relation

The subtyping relation in Pony is written <:. The symbol describes both the type ordering on types, and the type ordering on capabilities.

Definition 4.6 (Subtyping relation (<:)).
For types, `S` κ <: `S` κ*' if* $\kappa <: \kappa'$.

Definition 4.7 (Subcapability relation (<:)).
For capabilities, $\kappa <: \kappa'$ is given by the reflexive, transitive closure of:

- `iso^` <: `iso`, `trn^`
- `trn^` <: `trn`, `ref`, `val` <: `box`
- `iso`, `box` <: `tag`

These rules are given graphically in Figure 4.7 (reproduced from Steed 2016.)

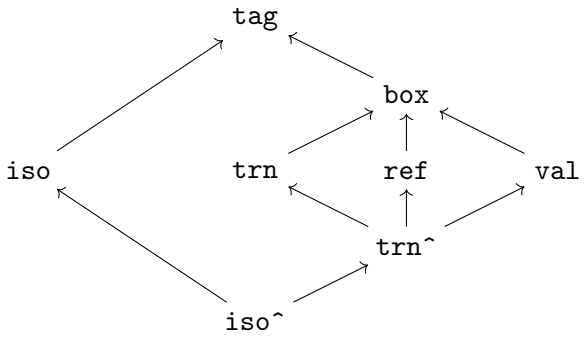


Figure 4.7: The subtyping relation on Pony capabilities

κ \ κ'	iso	trn	ref	val	box	tag
iso	iso	iso	iso	val	tag	tag
trn	iso	trn	trn	val	box	tag
ref	iso	trn	ref	val	box	tag
val	val	val	val	val	val	tag
box	tag	box	box	val	box	tag
tag	-	-	-	-	-	-
iso^	iso^	iso^	iso^	val	val	tag
trn^	iso^	trn^	trn^	val	val	tag

Figure 4.8: Viewpoint adaptation for $\kappa \triangleright \kappa'$

4.3.4 Viewpoint adaptation

As an object's fields have capabilities independent of the capability of the path used to access that object, the capability of a field access depends on both the field's capability and the path's capability.

This dependency is described through *viewpoint adaptation*, a function of the form (Capability × Capability) → Capability, which as in Steed 2016 has two versions:

Definition 4.8 (Non-extracting viewpoint adaptation ($\triangleright$)).
Non-extracting, or field read, viewpoint adaptation is expressed $\triangleright$. This is the capability of the field when it is being read for the purposes of aliasing or further field reading. The table is given in Figure 4.8.

Definition 4.9 (Extracting viewpoint adaptation ($\triangleright$)).
Extracting, or field write, viewpoint adaptation is expressed $\triangleright$. This is the capability of the value returned from reassigning the field to some new value. The table is given in Figure 4.9.

Note that, since fields cannot be declared with ephemeral capabilities, we have more rows than columns in both tables.

4.3.5 Safe-to-write

Similarly, the values which can be written into the field of an object depend on both the capability of the value being written and the capability used to access the object.

κ \ κ'	iso	trn	ref	val	box	tag
iso	iso^	val	tag	val	tag	tag
trn	iso^	val	box	val	box	tag
ref	iso^	trn^	ref	val	box	tag
val	-	-	-	-	-	-
box	-	-	-	-	-	-
tag	-	-	-	-	-	-
iso^	iso^	iso^	iso^	val	val	tag
trn^	iso^	trn^	trn^	val	box	tag

Figure 4.9: Viewpoint adaptation for $\kappa \triangleright \kappa'$

κ \ κ'	iso	trn	ref	val	box	tag
iso	✓			✓		✓
trn	✓	✓		✓		✓
ref	✓	✓	✓	✓	✓	✓
val						
box						
tag						
iso^	✓	✓	✓	✓	✓	✓
trn^	✓	✓	✓	✓	✓	✓

Figure 4.10: The safe-to-write relation $\kappa \triangleleft \kappa'$

This dependency is described through the *safe-to-write* relation.

Definition 4.10 (Safe-to-write ($\triangleleft$)).
The safe-to-write relation $\kappa \triangleleft \kappa'$ is defined to hold if a value of capability κ' can be safely written into a value of capability κ. The safe-to-write table is given in Figure 4.10 as a relation on capabilities.

Naturally, it should never be safe to write to a read-only capability.

There is no need to consider ephemeral columns, since aliasing is done before assignment - as can be seen in the syntax, where the right-hand-side of all assignments is aliased. Additionally, it is safe to write any value to any object of ephemeral capability, because such an object will have no subsequent alias, and so cannot be used to violate any of Pony's guarantees.

4.3.6 Field and method lookups

Definition 4.11 (Field lookup ($\mathcal{F}$)).
The function

$$\mathcal{F} : (\textit{Type ID}, \textit{Field ID}) \rightarrow \textit{Type}$$

maps a type ID and a field ID to the field type corresponding to that field on that type.

Definition 4.12 (Method lookup ($\mathcal{M}$)).
The function

$$\mathcal{M} : (\textit{Type ID}, \textit{Method ID}) \to \textit{Function Def} \cup \textit{Behaviour Def} \cup \textit{Constructor Def}$$

maps a type ID and a method ID in the program to the corresponding definition of that method on that type.

4.3.7 Sendable capabilities

Definition 4.13 (Sendable capability).
A capability is said to be sendable *if its local and global alias permissions are the same.*

Only values with sendable capabilities may be sent between actors. The sendable capabilities are `iso`, `val`, and `tag`, as well as their ephemeral versions.

Definition 4.14 (Sendable subset (Send)).
The sendable subset of a typing context Γ, written Send(Γ), is the partial function taken from the variable mappings in Γ that map to a sendable capability i.e.

$$\textit{Send}(\Gamma) = \{\mathtt{x} \mapsto \mathtt{S}\ \kappa \mid \Gamma(\mathtt{x}) = \mathtt{S}\ \kappa, \kappa\ \textit{is sendable}\}$$

A value which is the product of only sendable values must itself be sendable - providing no alias can escape the scope of its creation. This is the role of the `recover` command, which in the full Pony language can wrap a scoped sequence of expressions instead of just a single expression. The contents of the `recover ... end` only has access to the sendable values of the outer scope, so its result is guaranteed to be sendable.

Definition 4.15 (Sendability recovery map ($\mathcal{S}$)).
We define the sendability recovery map $\mathcal{S}$: Capability $\to$ Capability by

$$\mathcal{S}(\kappa) = \begin{cases} \mathcal{S}(\beta)\hat{} & \kappa = \beta\hat{} \\ \mathtt{iso} & \kappa = \mathtt{iso}\ \textit{or}\ \kappa = \mathtt{trn}\ \textit{or}\ \kappa = \mathtt{ref} \\ \mathtt{val} & \kappa = \mathtt{val}\ \textit{or}\ \kappa = \mathtt{box} \\ \mathtt{tag} & \kappa = \mathtt{tag} \end{cases}$$

4.3.8 Typing rules

The typing rules are given in Figures 4.11, 4.12, 4.13.

In most of the rules, the explicit aliasing construction α is responsible for subsumption through Expr-Alias. The exception is Expr-FieldAssign: since a capability may be safe-to-write even if some supercapability isn't, it is necessary to perform subsumption manually.

4.3.9 Comparison to previous work

Pony with explicit aliasing builds on large areas of the existing Pony models. It reproduces the improvements developed in Steed 2016 in the areas of the safe-to-write relation; the two viewpoint adaptation relations; and the subtyping relation.

$$\frac{\mathtt{x} \in \Gamma}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \dashv \Gamma} \textsc{Path-Var}$$

$$\frac{t \in \Gamma}{\Gamma \vdash t : \Gamma(t) \dashv \Gamma - t} \textsc{Path-Temp}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}\ \beta}{\Gamma \vdash (\mathtt{consume\ x}) : \mathtt{S}\ \beta\hat{} \dashv \Gamma - \mathtt{x}} \textsc{Path-Consume}$$

$$\frac{\Gamma \vdash \mathtt{p} : \mathtt{S}\ \kappa \dashv \Gamma' \qquad \mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S'}\ \kappa\text{'}}{\Gamma \vdash \mathtt{p.f} : \mathtt{S'}\ (\kappa \triangleright \kappa\text{'}) \dashv \Gamma'} \textsc{Path-Field}$$

Figure 4.11: Typing rules for paths

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\ \kappa \dashv \Gamma' \qquad \mathcal{A}(\kappa) <: \beta}{\Gamma \vdash \alpha(\mathtt{e}) : \mathtt{S}\ \beta \dashv \Gamma'} \textsc{Expr-Alias}$$

Figure 4.12: The alias rule

Compared to the existing Pony models, it achieves a simplification through the elimination of the aliasing judgement $\vdash_{\mathcal{A}}$ and subsumption judgement $\vdash_{\mathcal{S}}$. In that paper, the judgements are used to control when aliasing and subsumption are permitted in the typing rules, since they are not safe operations in general. In this version, the role of both judgements is fulfilled by the *explicit aliasing* construct and its corresponding rule, Expr-Alias: the Expr-Alias rule performs both aliasing of a type, and subsumption on the alias.

The novel features of the syntax and typing model also lay some groundwork for future extensions in the direction of *borrowing in Pony* (Chapter 9). The introduction of side-effects to the typing judgement is meant to allow for borrowing operations which change the typestate (Section 9.5) (Weiss et al. 2019). The explicit aliasing construct is intended to simplify the semantics around creating and extending loans - shareable loans, for example, are normally extended when aliased.

4.4 Well-formed programs

Given the typing rules, we can begin to define well-formed programs. These definitions are based on the requirement of being well-typed, but also introduce additional constraints which ensure the correctness of the program being executed.

4.4.1 Well-formed expressions

The well-formedness of expressions is motivated by the correctness of functions and behaviours - for this reason, it is defined from the final expression, which is the return value of a Pony function, backwards.

$$\frac{\mathtt{x} \notin \Gamma}{\Gamma \vdash \mathtt{var\ x : S}\ \beta \dashv \Gamma[x \mapsto \mathtt{S}\ \beta]}\ \textsc{Expr-VarDecl}$$

$$\frac{\Gamma \vdash \mathtt{e : S}\ \beta \dashv \Gamma' \qquad \Gamma'(\mathtt{x}) = \mathtt{S}\ \beta}{\Gamma \vdash \mathtt{x = e : S}\ \beta\mathtt{\hat{}} \dashv \Gamma'}\ \textsc{Expr-LocalAssign}$$

$$\frac{\Gamma \vdash \mathtt{p.f : S}\ \kappa \dashv \Gamma'}{\Gamma \vdash t\ \mathtt{<-\ p.f : S}\ \kappa \dashv \Gamma'[t \mapsto \mathtt{S}\ \kappa]}\ \textsc{Expr-TempAssign}$$

$$\frac{\Gamma \vdash \mathtt{e : S'}\ \beta \dashv \Gamma' \qquad \Gamma' \vdash \mathtt{p : S}\ \kappa \dashv \Gamma'' \qquad \begin{array}{l}\mathcal{F}(\mathtt{S}\ \kappa) = \mathtt{S'}\ \kappa\mathtt{'} \\ \kappa \triangleleft \beta \quad \beta <: \kappa'\end{array}}{\Gamma \vdash \mathtt{p.f = e : S'}\ (\kappa \triangleright \kappa\mathtt{'}) \dashv \Gamma''}\ \textsc{Expr-FieldAssign}$$

$$\frac{\mathrm{Send}(\Gamma) \vdash \mathtt{e : S}\ \kappa \dashv \mathrm{Send}(\Gamma) - \Delta}{\Gamma \vdash \mathtt{recover\ e\ end : S}\ \mathcal{S}(\kappa) \dashv \Gamma - \Delta}\ \textsc{Expr-Recover}$$

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{S}, \mathtt{m}) = \mathtt{fun}\ \beta\ \mathtt{m}(\overline{\mathtt{x : AT}})\ \mathtt{:\ T} \Rightarrow \mathtt{E} \\ \Gamma_i \vdash \alpha(\mathtt{p})_i : \mathtt{AT}_i \dashv \Gamma_{i+1} \qquad \Gamma_{|\overline{\mathtt{AT}}|+1} \vdash \alpha(\mathtt{p}) : \mathtt{S}\ \beta \dashv \Gamma'\end{array}}{\Gamma_1 \vdash \alpha(\mathtt{p}).\mathtt{m}(\overline{\alpha(\mathtt{p})}) : \mathtt{T} \dashv \Gamma'}\ \textsc{Expr-FunCall}$$

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{S}, \mathtt{b}) = \mathtt{be\ b}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \\ \Gamma_i \vdash \alpha(\mathtt{p})_i : \mathtt{AT}_i \dashv \Gamma_{i+1} \qquad \Gamma_{|\overline{\mathtt{AT}}|+1} \vdash \alpha(\mathtt{p}) : \mathtt{S\ tag} \dashv \Gamma'\end{array}}{\Gamma_1 \vdash \alpha(\mathtt{p}).\mathtt{b}(\overline{\alpha(\mathtt{p})}) : \mathtt{S\ tag} \dashv \Gamma'}\ \textsc{Expr-BeCall}$$

$$\frac{\mathcal{M}(\mathtt{C}, \mathtt{k}) = \mathtt{new\ k}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \qquad \Gamma_i \vdash \alpha(\mathtt{p})_i : \mathtt{AT}_i \dashv \Gamma_{i+1}}{\Gamma_1 \vdash \mathtt{C.k}(\overline{\alpha(\mathtt{p})}) : \mathtt{C\ ref\hat{}} \dashv \Gamma_{|\overline{\mathtt{AT}}|+1}}\ \textsc{Expr-ClassCon}$$

$$\frac{\mathcal{M}(\mathtt{A}, \mathtt{k}) = \mathtt{new\ k}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \qquad \Gamma_i \vdash \alpha(\mathtt{p})_i : \mathtt{AT}_i \dashv \Gamma_{i+1}}{\Gamma_1 \vdash \mathtt{A.k}(\overline{\alpha(\mathtt{p})}) : \mathtt{A\ tag\hat{}} \dashv \Gamma_{|\overline{\mathtt{AT}}|+1}}\ \textsc{Expr-ActorCon}$$

Figure 4.13: Typing rules for expressions

$$\frac{\Gamma \vdash \texttt{p} : \texttt{S}\ \kappa \dashv \Gamma'}{\Gamma \vdash_{\text{expr}} \texttt{p} : \texttt{S}\ \kappa}\ \text{WF-Return}$$

$$\frac{\Gamma \vdash \texttt{var x} : \texttt{S}\ \kappa \dashv \Gamma' \qquad \Gamma' \vdash_{\text{expr}} \texttt{E} : \texttt{T}}{\Gamma \vdash_{\text{expr}} \texttt{var x;E} : \texttt{T}}\ \text{WF-VarDecl}$$

$$\frac{\Gamma \vdash \texttt{x = } \alpha(\texttt{RHS}) : \texttt{S}\ \kappa \dashv \Gamma' \qquad \Gamma' \vdash_{\text{expr}} \texttt{E} : \texttt{T}}{\Gamma \vdash_{\text{expr}} \texttt{x = } \alpha(\texttt{RHS})\texttt{;E} : \texttt{T}}\ \text{WF-LocalAssign}$$

$$\frac{\Gamma \vdash t\ \texttt{<- p.f} : \texttt{S}\ \kappa \dashv \Gamma' \qquad \Gamma' \vdash_{\text{expr}} \texttt{p'} : \texttt{T}}{\Gamma \vdash_{\text{expr}} t\ \texttt{<- p.f;p'} : \texttt{T}}\ \text{WF-TempAssign-Final}$$

$$\frac{\Gamma \vdash t\ \texttt{<- p.f} : \texttt{S}\ \kappa \dashv \Gamma' \qquad t \in \text{Consume}(\texttt{e}) \qquad \Gamma' \vdash_{\text{expr}} \texttt{e;E} : \texttt{T}}{\Gamma \vdash_{\text{expr}} t\ \texttt{<- p.f;e;E} : \texttt{T}}\ \text{WF-TempAssign}$$

Figure 4.14: Well-formedness rules for expressions

Definition 4.16 (Well-formed expression judgement ($\vdash_{\text{expr}}$)).
The well-formed expression judgement $\vdash_{expr}$ is of the form

$$\Gamma \vdash_{expr} \texttt{E} : \texttt{T}$$

for Γ a typing context. The judgement states that with a local variable mapping that agrees in type with Γ, then the expression sequence `E` *could be evaluated, and the result would have type* `T`*.*

To assist with these definitions, we can define the Consume map.

Definition 4.17 (Consume map (Consume**)).**
Consume : *Expression $\rightarrow$ Temporary maps an expression to the temporaries that it consumes: an expression consumes a temporary when it appears in some path in the expression.*

While expression well-formedness largely asserts that expressions are well-typed, it has one important additional feature: enforcing the use of temporaries. Since a temporary is not an alias, allowing temporaries to be stored for later use might violate Pony's race condition guarantees. For this reason, temporaries must be used in the expression following the one in which they are initialised - this is the purpose of WF-TempAssign.

4.4.2 Well-formed actors and classes

The well-formedness of an actor or class is dependant on the well-formedness of its constructors and methods.

Definition 4.18 (Well-formed type judgement ($\vdash_{\text{type}}$)).
The judgement

$$\vdash_{type} \texttt{S} \diamond$$

states that the type `S` *is well-formed.*

The rules for the well-formedness of actors and classes are given in Figure 4.15.

$$\frac{\begin{array}{c}\forall \mathtt{k}, \mathcal{M}(\mathtt{C},\mathtt{k}) = \mathtt{new\ k}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \implies [\mathtt{this} \mapsto \mathtt{C\ ref}, \overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \\ \forall \mathtt{m}, \mathcal{M}(\mathtt{C},\mathtt{m}) = \mathtt{fun}\ \beta\ \mathtt{m}(\overline{\mathtt{x : AT}})\ \mathtt{:\ T} \Rightarrow \mathtt{E} \implies [\mathtt{this} \mapsto \mathtt{C}\ \beta, \overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T}\end{array}}{\vdash_{\text{type}} \mathtt{C} \diamond}\ \text{WF-Class-Type}$$

$$\frac{\begin{array}{c}\forall \mathtt{k}, \mathcal{M}(\mathtt{A},\mathtt{k}) = \mathtt{new\ k}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \implies [\mathtt{this} \mapsto \mathtt{A\ iso}, \overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \\ \forall \mathtt{m}, \mathcal{M}(\mathtt{A},\mathtt{m}) = \mathtt{fun}\ \beta\ \mathtt{m}(\overline{\mathtt{x : AT}})\ \mathtt{:\ T} \Rightarrow \mathtt{E} \implies [\mathtt{this} \mapsto \mathtt{A}\ \beta, \overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \\ \forall \mathtt{b}, \mathcal{M}(\mathtt{A},\mathtt{b}) = \mathtt{be\ b}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \implies \left\{\begin{array}{c}[\mathtt{this} \mapsto \mathtt{A\ iso}, \overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \\ \text{Send}([\overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}]) = [\overline{\mathtt{x}_i \mapsto \mathtt{AT}_i}]\end{array}\right\}\end{array}}{\vdash_{\text{type}} \mathtt{A} \diamond}\ \text{WF-Actor-Type}$$

Figure 4.15: Well-formedness rules for actors and classes

$$\frac{\begin{array}{c}\forall(\mathtt{class\ C\ \_\ \_\ \_}) \in \overline{\mathtt{CD}}, \vdash_{\text{type}} \mathtt{C} \diamond \\ \forall(\mathtt{actor\ A\ \_\ \_\ \_\ \_}) \in \overline{\mathtt{AD}}, \vdash_{\text{type}} \mathtt{A} \diamond\end{array}}{\vdash_{\text{prog}} (\overline{\mathtt{CD}}\ \overline{\mathtt{AD}}) \diamond}\ \text{WF-Program}$$

Figure 4.16: Well-formedness rules for programs

4.4.3 Well-formed programs

Definition 4.19 (Well-formed program judgement ($\vdash_{\mathbf{prog}}$)).
The judgement

$$\vdash_{prog} \mathtt{P} \diamond$$

states that the program `P` *is well-formed.*

The sole rule for the well-formedness of programs is shown in Figure 4.16. Unsurprisingly, it simply requires that all declared classes and actors in the program are well-formed.

4.4.4 Comparison to previous work

Explicit temporaries were responsible for large changes to the definitions of well-formed programs compared to Clebsch et al. 2015 and Steed 2016.

While the previous papers largely relied on well-typedness to describe well-formed expressions, our need to restrict temporary use - as being used in the next statement, and not more than once - meant that the well-formed expression judgement was necessary. This is an unfortunate piece of additional complexity; however, it is offset by the simplicity that the new syntax and semantics provide.

Beyond expressions, the definitions for well-formed types and programs are largely the same. This is unsurprising, since these definitions are very natural for any object-oriented language. Additionally, since no changes were made in these areas in the syntax, it is to be expected that their well-formedness is also mostly unchanged.

List of definitions in this chapter

Chapter 5

Pony with explicit aliasing - Heaps

The heap system makes a large departure from the versions presented in Clebsch et al. 2015, Steed 2016. Messages and frames, rather than forming part of the actor definition, are instead allocated on the heap directly.

These changes are motivated by the new *region system* (Section 5.2). Regions are a new extension to the heap system which assign ownership and mutability information to sets of addresses. The region system is used to define a *perspective judgement* (Section 5.2.2) which describes valid possible accesses between heap addresses. This judgement then allows for a description of *well-formed heaps* (Section 5.3) which is simpler than that found in previous models.

The region system is intended to tackle the complexity of *heap accessibility*, typically reasoned about in terms of paths in the heap. Path reasoning had large parts of previous works dedicated to it (particularly Steed 2016), due to the difficulties involved in defining path compatibility and proving it was preserved by the semantics. By side-stepping path reasoning entirely, the region system is able to reduce proof complexity.

5.1 Heap structures

5.1.1 Notation - possibly-null sets

Definition 5.1 (Possibly-null sets ($S?$)).
We use the notation $S?$ to denote the set $S \cup \{\textbf{null}\}$, defined only if $\textbf{null} \notin S$.

In words, $S?$ is the set of elements of S, as well as the special element **null** which represents "no" element. This is helpful in definitions where an element of S is optional.

5.1.2 Message queues

In order to allow actors to handle sent messages asynchronously from the point of their sending, actors maintain a message queue. Message queues are stored as linked lists of messages in the heap, with the front of the queue being the start of the list.

Messages themselves are tuples of a behaviour identifier, and an argument mapping corresponding to the arguments for that behaviour.

$$
\begin{array}{rcrcl}
\chi & \in & \text{Heap} & = & \text{Address} \to (\text{Actor} \cup \text{Object} \cup \text{Message} \cup \text{Frame}) \\
\iota & \in & \text{Address} & = & \text{Actor Address} \cup \text{Object Address} \cup \text{Message Address} \cup \\
 & & & & \text{Frame Address} \\
v & \in & \text{Value} & = & \text{Address?} \\
 & & \text{Actor} & = & \text{Actor ID} \times (\text{Field ID} \to \text{Value}) \times \\
 & & & & \text{Message Address?} \times \text{Frame Address?} \\
 & & \text{Object} & = & \text{Class ID} \times (\text{Field ID} \to \text{Value}) \\
 & & \text{Message} & = & \text{Method ID} \times (\text{Variable} \to \text{Value}) \times \text{Message Address?} \\
 & & \text{Frame} & = & (\text{Variable} \to \text{Value}) \times \text{Expression Sequence} \times \\
 & & & & \text{Variable?} \times \text{Frame Address?}
\end{array}
$$

Figure 5.1: Heap definitions used in the operational semantics

Since queues are manipulated from both ends, we define an operation for appending messages to the end of an actor's queue.

Definition 5.2 (Appending operation (Append)).
The appending operation

$$\mathit{Append} : \mathit{Actor\ Address?} \times \mathit{Behaviour\ ID} \times (\mathit{Variable} \to \mathit{Value}) \times \mathit{Heap} \to \mathit{Heap}$$

appends a message to the end of the message queue for a given actor. If the queue is empty i.e. the message queue address for that actor is ***null****, then the given message becomes the first one in the queue.*

If the actor address is itself ***null****, then Append does nothing.*

5.1.3 Stack of frames

As actors are also execution units, they contain the current frame that they are executing expressions in. In order to model function calls, actors store a stack of frame addresses as a linked list, with the top of the stack being the frame currently being executed.

A frame is a tuple of a local variable mapping, some code to execute, and an optional callee-return variable. For frames where the return variable is not **null**, the code in the frame should not be executed further until a value is returned to the frame - at which point, the mapping of the return variable to the returned value is added to the local variable mapping, and the return variable becomes **null**.

5.2 Explicit regions

For the purposes of simplifying well-formedness properties of Pony heaps, we extend the existing models of heaps found in Clebsch et al. 2015; Steed 2016 with *explicit regions*. These regions represent collections of addresses which should have restricted access, such as the contents of an `iso` - which should not be readably or writably aliasable.

$$
\begin{array}{rcrcl}
\alpha & \in & \text{Accessor} & := & \texttt{f} \mid \texttt{x} \mid \texttt{next} \mid \texttt{caller} \\
 & & \text{Region} & := & (\texttt{iso}, \iota, \alpha) \mid (\texttt{trn}, \iota, \alpha) \mid \texttt{val} \\
S & \in & \text{Sentinel Map} & = & \text{Address} \to \text{Address} \\
R & \in & \text{Region Map} & = & \text{Address} \to \text{Region} \\
\mathbf{R} & \in & \text{Region Partition} & = & \text{Sentinel Map} \times \text{Region Map}
\end{array}
$$

Figure 5.2: Region definitions.

Regions aim to introduce *local* and *non-quantified* reasoning for the correctness of heap regions. In particular, the goal is to avoid assertions on all possible paths in the heap: such assertions have been points of complexity for previous well-formedness definitions.

5.2.1 Heap regioning

The definitions of regions structures are shown in Figure 5.2. We also give an explanation for the definitions.

Every defined address in the heap is a member of some region.

Definition 5.3 (Region).
Regions may be

- *Isolated regions* $(\texttt{iso}, \iota, \alpha)$ *for* α *an accessor.*
- *Transition regions* $(\texttt{trn}, \iota, \alpha)$.
- *The unique value region* `val`.

Definition 5.4 (Region owner).
For isolated and transition regions, the address ι *is called the* owner *of the region.*

Definition 5.5 (Accessor).
An accessor is a record of the method by which the owner can access the region. The accessor is one of

- `f` - *in this case, access is through the field* `f` *in the object or actor owner.*
- `x` - *in this case, access is through the local variable* `x` *in the frame or message owner.*
- `next` - *in this case, access is through a message queue address, either in an actor or another message.*
- `caller` - *in this case, access is through a stack frame address, either in an actor or another frame.*

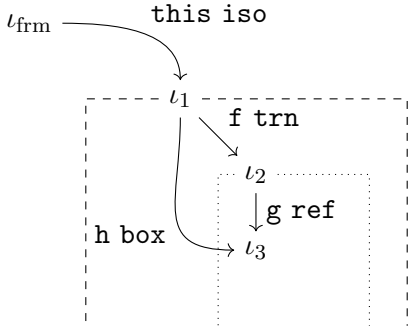


Figure 5.3: An example of a heap with explicit region information.

Definition 5.6 (Region partition (R)).
For a heap χ, a region partition $\mathbf{R} = (S, R)$ is a tuple of a sentinel map $S : \text{dom}(\chi) \to \text{dom}(\chi)$, and a region map $R : \text{im}(S) \to$ Region. S maps addresses to their "sentinel" - the address which creates the region they are a part of. R maps sentinel addresses to their corresponding region information. A region partition has the additional property that S is idempotent i.e. $S(S(\iota)) = S(\iota)$ - in other words, every sentinel address is its own sentinel.

An example of regions is shown in Figure 5.3. Here, ι_1 creates a `iso` region owned by a frame address ι_{frm} - meaning $S(\iota_1) = \iota_1$ and $R(\iota_1) = (\texttt{iso}, \iota_{\text{frm}}, \texttt{this})$. ι_2 also creates a region, but that region is owned by ι_1, since it has a `trn` reference to ι_2 - meaning $S(\iota_2) = \iota_2$, and $R(\iota_2) = (\texttt{trn}, \iota_1, \texttt{f})$. Finally ι_3 creates no region, but lies inside the region created by ι_2 - so $S(\iota_3) = \iota_2$.

5.2.2 Perspective judgement

With regions, it becomes possible to define a perspective judgement for objects in the heap.

Definition 5.7 (Perspective judgement ($\Vdash$)).
The perspective judgement

$$\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \kappa$$

states that in the heap χ with region partition $\mathbf{R}$, the object at address ι may safely hold the alias represented by the accessor α to the object at address ι' with capability κ.

This is intended to be a *local* judgement, avoiding accessing more than the two given addresses. The rules for the perspective judgement are given in Figure 5.4.

The two most significant rules for the region system are:

- PERSP-`box`-TRANS - the transitive `box` rule. As is demonstrated in Figure 5.3, `box` aliases may sometimes cross region boundaries - specifically, `trn` regions. Therefore, the PERSP-`box` rule, which only allows `box` aliases between objects in the same region, is insufficient.
- PERSP-`val`-REGION - the `val` region rule. This rule is discussed in Section 5.2.3.

$$\frac{R(\iota') = (\texttt{iso}, \iota, \alpha)}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \texttt{iso}} \textsc{Persp-iso}$$

$$\frac{R(\iota') = (\texttt{trn}, \iota, \alpha)}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \texttt{trn}} \textsc{Persp-trn}$$

$$\frac{S(\iota) = S(\iota')}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \texttt{ref}} \textsc{Persp-ref}$$

$$\frac{R(S(\iota')) = \texttt{val}}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \texttt{val}} \textsc{Persp-val}$$

$$\frac{R(S(\iota)) = \texttt{val} \qquad R(S(\iota')) = \texttt{val}}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \kappa} \textsc{Persp-val-region}$$

$$\frac{S(\iota) = S(\iota')}{(S, R), \chi, \iota, \alpha \Vdash \iota' : \texttt{box}} \textsc{Persp-box}$$

$$\frac{\mathbf{R}, \chi, \iota, \beta \Vdash \iota'' : \texttt{box} \qquad \mathbf{R}, \chi, \iota'', \gamma \Vdash \iota' : \texttt{box}}{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \texttt{box}} \textsc{Persp-box-trans}$$

$$\frac{}{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \texttt{tag}} \textsc{Persp-tag}$$

$$\frac{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \kappa \qquad \kappa <: \kappa'}{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \kappa'} \textsc{Persp-Subsume}$$

$$\frac{}{\mathbf{R}, \chi, \iota, \alpha \Vdash \mathbf{null} : \kappa} \textsc{Persp-null}$$

Figure 5.4: Perspective judgement rules

$$\dfrac{\dfrac{\dfrac{S(\iota_2) = \iota_2 \qquad R(\iota_2) = (\texttt{trn}, \iota_1, \texttt{f})}{(S,R), \chi, \iota_1, \texttt{f} \Vdash \iota_2 : \texttt{trn}}}{(S,R), \chi, \iota_1, \texttt{f} \Vdash \iota_2 : \texttt{box}} \qquad \dfrac{\dfrac{S(\iota_2) = S(\iota_3)}{(S,R), \chi, \iota_2, \texttt{g} \Vdash \iota_3 : \texttt{ref}}}{(S,R), \chi, \iota_2, \texttt{g} \Vdash \iota_3 : \texttt{box}}}{(S,R), \chi, \iota_1, \texttt{h} \Vdash \iota_3 : \texttt{box}}$$

Figure 5.5: The derivation that the alias `h` in Figure 5.3 is well-formed.

Using the perspective judgement, we can give the derivation of the well-formedness of the alias `h` in the example in Figure 5.3. This derivation is shown in Figure 5.5 - it uses the facts $\texttt{trn}, \texttt{ref} <: \texttt{box}$.

Critically, if `f` had been a `iso` alias instead, the condition that $\texttt{iso} <: \texttt{box}$ would have failed to hold, and this derivation would not have justified the well-formedness of `h` - which is good, because `h` would no longer be a well-formed alias in such a scenario.

5.2.3 The `val` region

`val` requires special rulings because it allows for the violation of other regions - for example, a `iso` value which appears in a `val` value may be aliased readably with `val`, though this would normally violate the condition that no part of a `iso` value may be readably aliased.

Our solution to this is to place all immutable values in the same `val` region, even if those values have some region-creating capability such as `iso`. Since capabilities do not create new regions in the `val` region, the Persp-`val`-region rule allows for aliases of any capability between immutable values - these aliases cannot create race conditions, so they are always safe.

An example is shown in Figure 5.6 - on the left, the alias `h` would not be well-formed, because it attempts to enter the `iso` region created by ι_3. However, this alias *should* be permitted, because ι_3 and its contents are actually immutable. This can be solved by making the modification shown on the right, where the region created by ι_3 is removed. `h` is now well-formed as ι_4 lies directly in the `val` region - but `g1` is *also* well-formed because both ι_2 and ι_3 lie in the `val` region, so there can be any alias from one to the other.

5.2.4 The subsentinel relation

Preventing regions from being created in the `val` region can be done through a *subsentinel* relation.

Definition 5.8 (Subsentinel relation ($\preceq_S$)).
For a region partition (S, R), the relation $(\preceq_S) \subset \mathrm{im}(S) \times \mathrm{im}(S)$ is defined as (the transitive closure of)

$$\iota \preceq_S \iota' \iff R(\iota) = (\_, \iota'', \_) \wedge S(\iota'') = \iota'$$

In other words, a sentinel is a subsentinel of another if its owner falls in the region of that sentinel.

Using the subsentinel relation, we can define an additional constraint on region partitions: that there are no subsentinels of a `val` sentinel.

$$\forall \iota, \iota' \in \mathrm{im}(S), R(\iota') = \texttt{val} \implies \iota \not\preceq_S \iota'$$

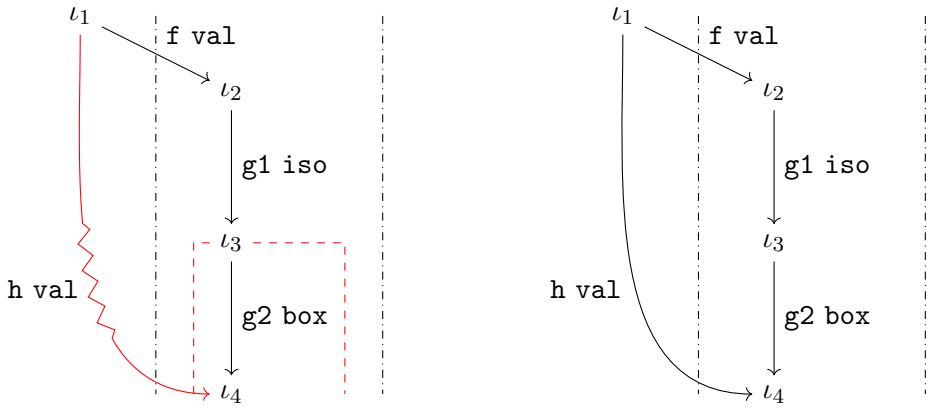


Figure 5.6: An example of how the val region works. On the left, the alias `h` is not well-formed, despite ι_4 being immutable. On the right, `h` is well-formed.

This prevents regions being created inside the `val` region - since any such region would have its sentinel be the subsentinel of a `val` sentinel.

Returning to Figure 5.6, we can see how this constraint prevents the region created by ι_3. When the region is present, then $R(\iota_3) = (\texttt{iso}, \iota_2, \texttt{g1})$ and $S(\iota_2) = \iota_2$ - so $\iota_3 \preceq_S \iota_2$. However, we can also see that $R(\iota_2) = \texttt{val}$ - so by the constraint we have added, we *should* have that $\iota_3 \not\preceq_S \iota_2$. Therefore, ι_3 should not create a subregion of the `val` region.

5.2.5 Temporaries

Up until now, the region system has not included definitions for valid temporaries. However, it is necessary to consider temporaries when discussing heap validity, since it is easy to formulate temporaries which *violate* capability guarantees. An example is shown in Figure 5.7.

When considering temporaries, we take the approach of extending the perspective judgement with special *temporary rules*, given in Figure 5.8. Significantly, *temporaries are not considered accessors* in these rules: similarly, it is *not possible* to have regions owned by a temporary.

The perspective judgement rules themselves reflect the partial-walk nature of temporaries. The rules are:

- With Persp-Temporary, a temporary can be created out of an existing accessor.
- With Persp-Temporary-Ephem, a temporary can be created out of an existing accessor *which cannot be used* to access the desired value in the heap. This is precisely the meaning of the ephemeral version of a capability.
- With Persp-Temporary-Trans, a temporary can be created by combining accesses between an existing temporary and an accessor. This second rule makes use of viewpoint adaptation (Section 4.3.4).
- With Persp-Temporary-**null**, a temporary which points to **null** can be treated as having any capability. This is justified because **null** is effectively

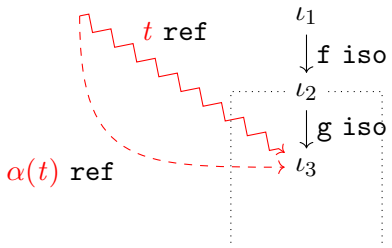


Figure 5.7: An example of an invalid temporary. t would allow for the violation of ι_2's `iso` region since, when aliased, it can produce a region-crossing `ref` variable.

$$\frac{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \kappa}{\mathbf{R}, \chi, \iota, t \Vdash \iota' : \kappa}\ \textsc{Persp-Temporary}$$

$$\frac{\mathbf{R}, \chi, \iota, \alpha \Vdash \iota' : \beta \qquad \chi(\iota, \alpha) \neq \iota'}{\mathbf{R}, \chi, \iota, t \Vdash \iota' : \beta\hat{}}\ \textsc{Persp-Temporary-Ephem}$$

$$\frac{\mathbf{R}, \chi, \iota, t \Vdash \iota'' : \kappa \qquad \mathbf{R}, \chi, \iota'', \alpha \Vdash \iota' : \kappa'}{\mathbf{R}, \chi, \iota, t \Vdash \iota' : (\kappa \triangleright \kappa')}\ \textsc{Persp-Temporary-Trans}$$

$$\frac{}{\mathbf{R}, \chi, \iota, t \Vdash \mathbf{null} : \kappa}\ \textsc{Persp-Temporary-}\mathbf{null}$$

Figure 5.8: Perspective judgement rules for temporaries

"outside" the object graph - so it cannot be used to invalidly access some part of the graph.

5.2.6 Non-determinism of regions

This definition of regions displays some features not determined by the heap being regioned. We have already seen in Section 5.2.3 that a `iso` or `trn` alias is not a sufficient condition for the creation of a region. However, such an alias is also not a necessary condition for region creation. Specifically, we can define a well-formed heap where some object creates a region which is unnecessary, such as shown in Figure 5.9.

In practice, these unnecessary regions can simply be ignored. However, they represent a trade-off in complexity - in order to make a `iso` or `trn` alias a necessary condition for region creation, the definition of a region partition would itself have to be made more complicated: for example, with the condition that some such alias exists between the owner and sentinel.

5.2.7 Comparison to existing work

Building on the model in Steed 2016, the explicit regions model makes many changes with the aim of simpler reasoning. The definition and results for "well-formed visibility" (Steed 2016, p.50-54, 58–87) are particularly extensive and complex, since they have to take into account the many possible cases in which various paths in the heap may conflict.

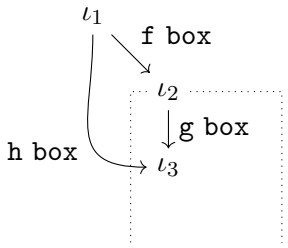


Figure 5.9: An example of a heap with an unnecessary region (the one created by ι_2).

By comparison, explicit regions aims to eliminate this case reasoning altogether. It uses a model with additional information to simplify the definition of its judgements; and also to completely forbid certain types of violations immediately. For example, since each actor in the heap defines its own `iso` region, it is no longer necessary to say that objects reachable by different actors must have globally compatible capabilities - instead, this is enforced by any region partition under which such an object is reachable.

Similarly, explicit regions allows for the assessment of whether a certain alias is valid *without needing to consider other aliases in the heap.* This is particularly powerful, because it means showing the preservation of validity of some alias under some heap operation depends only on how the operation and the alias interact.

5.3 Well-formed heaps

5.3.1 Types in the heap

As both objects and actors in the heap store some type information, it is possible to define a heap typing judgement.

Definition 5.9 (Heap typing judgement ($\vdash_{\mathbf{heap}}$)).
The judgement

$$\chi \vdash_{heap} \iota : \mathtt{S}$$

states that, in the heap χ, the address ι maps to a heap entity that can be interpreted as having type id `S`.

The heap typing judgement has two immediate rules, as well as a convenience rule for **null**. These are shown in Figure 5.10.

5.3.2 Well-typed mappings

As part of the well-formedness definitions, it is often necessary to state that there is an agreement between some typing context Γ and a variable (and temporary) mapping L. In particular, it is desirable to express that all the variables (and temporaries) in the typing context map to some value of their corresponding expected type: both in terms of heap type and region capability.

For this reason, we introduce notation to express when certain mappings are well-formed

$$\frac{\chi(\iota) = (\mathtt{C}, \_)}{\chi \vdash_{\text{heap}} \iota : \mathtt{C}}\ \textsc{HeapTyp-Object}$$

$$\frac{\chi(\iota) = (\mathtt{A}, \_, \_, \_)}{\chi \vdash_{\text{heap}} \iota : \mathtt{A}}\ \textsc{HeapTyp-Actor}$$

$$\frac{}{\chi \vdash_{\text{heap}} \textbf{null} : \mathtt{S}}\ \textsc{HeapTyp-}\textbf{null}$$

Figure 5.10: The heap typing rules

Definition 5.10 (Well-typed local variables).
The notation $\mathbf{R}, \chi, \iota \vdash_{heap} L \diamond : \Gamma$ *expresses the condition:*

$$\forall k, \Gamma(k) = \mathtt{S}\ \kappa \implies \exists v, L(k) = v \wedge \chi \vdash_{heap} v : \mathtt{S} \wedge \mathbf{R}, \chi, \iota, k \Vdash v : \kappa$$

Here, k *can be either a variable or a temporary.*

A similar expression is also desirable for field mappings, and is expressed in a similar way.

Definition 5.11 (Well-typed fields).
The notation $\mathbf{R}, \chi, \iota \vdash_{heap} F \diamond : \mathtt{S}$ *expresses the condition:*

$$\forall \mathtt{f}, \mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\ \kappa \implies \exists v, F(\mathtt{f}) = v \wedge \chi \vdash_{heap} v : \mathtt{S}' \wedge \mathbf{R}, \chi, \iota, \mathtt{f} \Vdash v : \kappa$$

Both conditions, in words, require that every key with a type (found in the context or the field types) has a value which agrees with that type, in terms of heap type and region capability.

5.3.3 Well-formed heap entries

Using the heap typing and perspective judgements, we can define a well-formedness property for individual addresses in the heap.

Definition 5.12 (Well-formed heap entry judgement ($\Vdash \ldots \diamond$)).
The well-formedness judgement

$$\mathbf{R}, \chi \Vdash \iota \diamond$$

states that, in the heap χ *with region partition* $\mathbf{R}$*, the entry at address* ι *is well-formed.*

The qualification for an entry to be well-formed depends on what the entry is - but roughly, an entry must be able to safely hold all aliases it contains, and they must be of the expected type.

The rules for the well-formedness judgement for object and actors are given in Figure 5.11. For objects, it is sufficient that the field mapping be well-typed. For actors, the field mapping must be well-typed, the actor must own itself uniquely, and the actor's message queue and stack frame must be well-formed.

$$\frac{\chi(\iota) = (\mathtt{C}, F) \qquad \mathbf{R}, \chi, \iota \vdash_{\text{heap}} F \diamond : \mathtt{C}}{\mathbf{R}, \chi \Vdash \iota \diamond} \text{ WF-Object}$$

$$\frac{\chi(\iota) = (\mathtt{A}, F, v_\mu, v_\phi) \quad \mathbf{R}, \chi, \iota \Vdash \iota : \mathtt{iso} \quad \mathbf{R}, \chi, \iota \vdash_{\text{heap}} F \diamond : \mathtt{C} \quad \mathbf{R}, \chi \Vdash_{\text{mes}} v_\mu, \mathtt{A} \diamond \quad \mathbf{R}, \chi \Vdash_{\text{frm}} v_\phi \diamond \quad \chi \vdash_{\text{active}} v_\phi \diamond}{\mathbf{R}, \chi \Vdash \iota \diamond} \text{ WF-Actor}$$

Figure 5.11: Well-formedness of objects and actors

5.3.4 Well-formed stacks and queues

The well-formedness of stacks and queues is more complicated - since they are both recursive structures where the individual structural components are related in some way. These rules are shown in Figure 5.12.

For queues, it is only necessary to check that, as well as aliases being correct and safe, the messages in the queue are processable by the actor which will receive them. This gives a particular well-formedness judgement for queues:

Definition 5.13 (Well-formed message queue judgement ($\Vdash_{\mathbf{mes}}$)).
The well-formed message queue judgement is of the form

$$\mathbf{R}, \chi \Vdash_{mes} \iota, \mathtt{A} \diamond$$

and states that the queue at ι is well-formed, provided the messages are only eventually processed by an actor of type `A`*.*

There are two rules for messages:

- By WF-Message-**null**, the **null** end of a queue is always well-formed.
- By WF-Message, a non-**null** queue is well-formed if the message at the front of the queue is a valid message for the actor of type `A` - that is, the behaviour identifier corresponds to a defined behaviour on `A`, and the message arguments agree in type and capability with the expected arguments of that behaviour. Additionally, the tail of the queue must also be well-formed with the same receiver - and be owned uniquely by the front of the queue.

For stacks, the rules are more complex. A frame can be well-formed without being executable - in particular, if it is waiting for the return value of a synchronous method call. Return values also mean that type information must propagate up the stack - a frame must not return a value to a well-formed superframe if the value returned is not of the type expected.

All this together gives a well-formedness judgement for stack frames:

Definition 5.14 (Well-formed stack frame judgement ($\Vdash_{\mathbf{frm}}$)).
The well-formedness judgement for stack frames is of the form

$$\mathbf{R}, \chi \Vdash_{frm} \iota, \mathtt{T} \diamond$$

and states the frame at ι is well-formed, provided the value returned into it can be safely typed with `T`*. We also allow the judgement*

$$\mathbf{R}, \chi \Vdash_{frm} \iota \diamond$$

for frames, if they are not waiting for a return value.

There are four rules for frames:

- By WF-Frame-**null**, the **null** bottom of a stack is always well-formed, and doesn't need to be returned a value.
- By WF-Frame-Top, a non-**null** stack is well-formed if the top frame is valid - that is, the remaining code in the frame is well-formed with respect to some typing context such that the frame's local variables agree with that context. Additionally, the top frame must be able to run without awaiting any return value. Finally, the remainder of the stack must be well-formed if it would be returned a value of the type of the code in the top frame - and that remainder must be owned uniquely by the top frame.
- By WF-Frame, a non-**null** stack may also well-formed if the top frame is valid, but expects some return value. In this case, the remaining code in the frame must well-formed with respect to some typing context such that the frame's local variables *and the returned value* agree with that context. The other conditions are identical to those for WF-Frame-Top.
- Finally, by WF-Frame-NoReturn, any stack which is well-formed without expecting a return value, can be made to expect a return value of any type - by simply discarding the value returned. This allows for stacks where the return value of a frame has no effect on its parent frame: such as in constructor calls.

Moreover, well-formed stacks have some restrictions on their number of temporaries. The top frame may have at most one temporary, and all frames below it must have none.

Definition 5.15 (Active stack frame judgement ($\vdash_{\mathbf{active}}$)).
The active-ness judgement for stack frames is of the form

$$\chi \vdash_{active} \iota \diamond$$

and states that the frame at ι contains the unique temporary (if any) of its stack.

The active judgement is so-called because it asserts that the frame holds the unique *active temporary* (in the terminology of Steed 2016) for some actor instance.

Definition 5.16 (Passive stack frame judgement ($\vdash_{\mathbf{passive}}$)).
The passive-ness judgement for stack frames is of the form

$$\chi \vdash_{passive} \iota \diamond$$

and states that the stack at ι has no temporaries.

$$\frac{\chi(\iota) = (\mathtt{b}, A, v_{\mathtt{next}}) \quad \mathcal{M}(\mathtt{A}, \mathtt{b}) = \mathtt{be\ b}(\overline{\mathtt{x : AT}}) \Rightarrow \mathtt{E} \quad \mathbf{R}, \chi, \iota \vdash_{\text{heap}} A \diamond : [\mathtt{x}_i \mapsto \mathtt{AT}_i] \quad \mathbf{R}, \chi \Vdash_{\text{mes}} v_{\mathtt{next}}, \mathtt{A} \diamond \quad \mathbf{R}, \chi, \iota, \mathtt{next} \Vdash v_{\mathtt{next}} : \mathtt{iso}}{\mathbf{R}, \chi \Vdash_{\text{mes}} \iota, \mathtt{A} \diamond}\ \text{WF-Message}$$

$$\frac{}{\mathbf{R}, \chi \Vdash_{\text{mes}} \mathbf{null}, \mathtt{A} \diamond}\ \text{WF-Message-}\mathbf{null}$$

$$\frac{\chi(\iota) = (L, \mathtt{E}, \mathbf{null}, v_{\mathtt{caller}}) \quad \Gamma \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \quad \mathbf{R}, \chi, \iota \vdash_{\text{heap}} L \diamond : \Gamma \quad \mathbf{R}, \chi \Vdash_{\text{frm}} v_{\mathtt{caller}}, \mathtt{T} \diamond \quad \mathbf{R}, \chi, \iota, \mathtt{caller} \Vdash v_{\mathtt{caller}} : \mathtt{iso}}{\mathbf{R}, \chi \Vdash_{\text{frm}} \iota \diamond}\ \text{WF-Frame-Top}$$

$$\frac{\chi(\iota) = (L, \mathtt{E}, \mathtt{y}, v_{\mathtt{caller}}) \quad \Gamma[\mathtt{y} \mapsto \mathtt{T}'] \vdash_{\text{expr}} \mathtt{E} : \mathtt{T} \quad \mathbf{R}, \chi, \iota \vdash_{\text{heap}} L \diamond : \Gamma \quad \mathbf{R}, \chi \Vdash_{\text{frm}} v_{\mathtt{caller}}, \mathtt{T} \diamond \quad \mathbf{R}, \chi, \iota, \mathtt{caller} \Vdash v_{\mathtt{caller}} : \mathtt{iso}}{\mathbf{R}, \chi \Vdash_{\text{frm}} \iota, \mathtt{T}' \diamond}\ \text{WF-Frame}$$

$$\frac{\mathbf{R}, \chi \Vdash_{\text{frm}} v \diamond}{\mathbf{R}, \chi \Vdash_{\text{frm}} v, \mathtt{T} \diamond}\ \text{WF-Frame-NoReturn}$$

$$\frac{}{\mathbf{R}, \chi \Vdash_{\text{frm}} \mathbf{null} \diamond}\ \text{WF-Frame-}\mathbf{null}$$

Figure 5.12: Well-formedness of frame stacks and message queues

The passive judgement is so-called because it asserts that there are only *passive temporaries* (i.e. no active ones) in any of the frames in the stack.

The rules for both the active and passive judgements are given in Figure 5.13.

5.3.5 Well-formed heaps

It is finally possible to define heap well-formedness.

$$\frac{\chi(\iota) = (L, \_, \_, v_{\mathtt{caller}}) \quad \forall t, t' \in \text{dom}(L), t = t' \quad \chi \vdash_{\text{passive}} v_{\mathtt{caller}} \diamond}{\chi \vdash_{\text{active}} \iota \diamond}\ \text{Active-Frame}$$

$$\frac{}{\chi \vdash_{\text{active}} \mathbf{null} \diamond}\ \text{Active-Null}$$

$$\frac{\chi(\iota) = (L, \_, \_, v_{\mathtt{caller}}) \quad \forall t, t \notin \text{dom}(L) \quad \chi \vdash_{\text{passive}} v_{\mathtt{caller}} \diamond}{\chi \vdash_{\text{passive}} \iota \diamond}\ \text{Passive-Frame}$$

$$\frac{}{\chi \vdash_{\text{passive}} \mathbf{null} \diamond}\ \text{Passive-Null}$$

Figure 5.13: Active and passive rules for stack frames

$$\frac{\forall \iota \in \mathrm{dom}(\chi), (\chi(\iota) = (\mathtt{S}, F) \vee \chi(\iota) = (\mathtt{S}, F, v_\mu, v_\phi)) \implies \mathbf{R}, \chi \Vdash \iota \diamond}{\mathbf{R} \Vdash \chi \diamond}\ \text{WF-HEAP}$$

Figure 5.14: Well-formedness of heaps

Definition 5.17 (Well-formed heap judgement ($\Vdash \ldots \diamond$)).
The well-formedness judgement

$$\mathbf{R} \Vdash \chi \diamond$$

states that the heap χ is well-formed.

The sole rule for the heap well-formedness judgement is given in Figure 5.14. It states that a heap is well-formed if all objects and actors in the heap are well-formed.

5.4 Comparison to existing work

The region system motivates a large departure from existing Pony models - specifically, in allocating messages and frames on the heap. In Clebsch et al. 2015; Steed 2016, the message queue and stack frame form part of the actor heap definition.

This approach was *not* taken, with the aim of simplifying accessor definitions: for example, if all frames for an actor were defined at the same address, a local variable accessor would have to declare not only the variable, but also the frame number, used to access a region. The particular complexity for this approach came from messages - if an accessor had to refer to a message's position in the queue, that accessor would need to be updated every time a message was processed (since the position of messages in the queue would change). Other solutions to this problem, including a global message index for each actor, were also considered and found too complex.

Instead, the region system uses the current approach, where the pair of an address and an accessor are sufficient to decide *how* the region is accessed. An unexpected simplicity of allocating messages and frames in terms of linked lists is that the region system itself already provides the mechanisms for asserting unique ownership of the tail of a list. Because messages are required to have an `iso` reference to the rest of the queue, no two queues can share any allocated messages - and similarly for frames. For the same reasons, these linked lists also cannot contain loops.

List of definitions in this chapter

Chapter 6

Pony with explicit aliasing - Operational Semantics

Our operational semantics for Pony with explicit aliasing is based on the existing models in Clebsch et al. 2015; Steed 2016. However, unlike in Clebsch et al. 2015; Steed 2016, our choice of "three-address" notation allows us to avoid the expression continuation mechanism, the semantic rules for expressions are given in "big-step" form. Our analysis of data-race prevention in this semantics is still meaningful, however, because each expression accesses the heap at most once - this simplifies the analysis of heap access interleavings to the analysis of expression interleavings.

We have two semantic judgements:

Definition 6.1 (Evaluating judgement ($\rightsquigarrow$)).
The evaluating judgement $\chi, \phi, \mathsf{e} \rightsquigarrow \chi', \phi', v$ *states that, in heap* χ *and frame* ϕ*, the expression* e *can be evaluated to have value* v*, with resulting heap* χ' *and frame* ϕ'*.*

Definition 6.2 (Advancing judgement ($\rightsquigarrow$)).
The advancing judgement $\chi, \iota \rightsquigarrow \chi'$ *states that in heap* χ*, the address* ι *can be advanced to heap* χ'*.*

This second judgement is necessary for e.g. function calls, which add a new frame to the stack of frames, but do not immediately evaluate to any value.

6.1 Expressions

Most expressions do not directly modify the stack frame, and can therefore be evaluated in terms of a single frame. The evaluation of these expressions depends on the current heap χ and current local variable (and temporary) mapping ϕ.

The operational semantics for such expressions are given in Figures 6.1, 6.2, 6.3, and 6.4.

6.2 Global execution

Since behaviour execution is asynchronous, the operational semantics must be able run invoked behaviours at appropriate points from outside expressions. This is the role of the Global rules (Figure 6.6):

$$\frac{\phi(\texttt{x}) = v}{\chi, \phi, \texttt{x} \rightsquigarrow \chi, \phi, v}\ \textsc{Sem-Local}$$

$$\frac{\phi(t) = v}{\chi, \phi, t \rightsquigarrow \chi, \phi - t, v}\ \textsc{Sem-Temp}$$

$$\frac{\phi(\texttt{x}) = v}{\chi, \phi, \texttt{consume x} \rightsquigarrow \chi, \phi - \texttt{x}, v}\ \textsc{Sem-Consume}$$

$$\frac{\chi, \phi, \texttt{p} \rightsquigarrow \chi', \phi', v \qquad \chi'(v, \texttt{f}) = v'}{\chi, \phi, \texttt{p.f} \rightsquigarrow \chi', \phi', v'}\ \textsc{Sem-Field}$$

$$\frac{\chi, \phi, \texttt{e} \rightsquigarrow \chi', \phi', v}{\chi, \phi, \alpha(\texttt{e}) \rightsquigarrow \chi', \phi', v}\ \textsc{Sem-Alias}$$

Figure 6.1: Operational semantics for paths and aliasing

$$\frac{}{\chi, \phi, \texttt{var x} \rightsquigarrow \chi, \phi[\texttt{x} \mapsto \textbf{null}], \textbf{null}}\ \textsc{Sem-VarDecl}$$

$$\frac{\chi, \phi, \texttt{e} \rightsquigarrow \chi', \phi', v \qquad \phi'(\texttt{x}) = v'}{\chi, \phi, \texttt{x = e} \rightsquigarrow \chi', \phi'[\texttt{x} \mapsto v], v'}\ \textsc{Sem-LocalAssign}$$

$$\frac{\chi, \phi, \texttt{p.f} \rightsquigarrow \chi', \phi', v}{\chi, \phi, t\ \texttt{<- p.f} \rightsquigarrow \chi', \phi'[t \mapsto v], \textbf{null}}\ \textsc{Sem-TempAssign}$$

$$\frac{\chi, \phi, \texttt{e} \rightsquigarrow \chi', \phi', v \qquad \chi', \phi', \texttt{p} \rightsquigarrow \chi'', \phi'', u \qquad \chi''(u, \texttt{f}) = v'}{\chi, \phi, \texttt{p.f = e} \rightsquigarrow \chi''[u, \texttt{f} \mapsto v], \phi'', v'}\ \textsc{Sem-FieldAssign}$$

$$\frac{\chi, \phi, \texttt{e} \rightsquigarrow \chi', \phi', v}{\chi, \phi, \texttt{recover e end} \rightsquigarrow \chi', \phi', v}\ \textsc{Sem-Recover}$$

Figure 6.2: Operational semantics for evaluated expressions

$$\frac{\begin{array}{c}\mathcal{M}(\texttt{A}, \texttt{b}) = \texttt{be b}(\overline{\texttt{x : AT}}) \Rightarrow \texttt{E} \\ \chi_i, \phi_i, \alpha(\texttt{p})_i \rightsquigarrow \chi_{i+1}, \phi_{i+1}, v_i \quad \chi_{|\overline{\alpha(\texttt{p})}|+1}, \phi_{|\overline{\alpha(\texttt{p})}|+1}, \alpha(\texttt{p}) \rightsquigarrow \chi', \phi', u \\ \chi' \vdash_{\text{heap}} u : \texttt{A} \quad \chi'' = \text{Append}(u, \texttt{b}, [\overline{\texttt{x}_i \mapsto v_i}], \chi')\end{array}}{\chi_1, \phi_1, \alpha(\texttt{p}).\texttt{b}(\overline{\alpha(\texttt{p})}) \rightsquigarrow \chi'', \phi', u}\ \textsc{Sem-BeCall}$$

$$\frac{\begin{array}{c}\mathcal{M}(\texttt{A}, \texttt{k}) = \texttt{new k}(\overline{\texttt{x : AT}}) \Rightarrow \texttt{E} \\ \chi_i, \phi_i, \alpha(\texttt{p})_i \rightsquigarrow \chi_{i+1}, \phi_{i+1}, v_i \quad \iota, \iota' \notin \chi_{|\overline{\alpha(\texttt{p})}|+1} \quad \iota \neq \iota' \\ \chi' = \chi_{|\overline{\alpha(\texttt{p})}|+1}[\iota \mapsto (\texttt{A}, \emptyset, \iota', \textbf{null}), \iota' \mapsto (\texttt{k}, [\overline{\texttt{x}_i \mapsto v_i}], \textbf{null})]\end{array}}{\chi_1, \phi_1, \texttt{A.k}(\overline{\alpha(\texttt{p})}) \rightsquigarrow \chi'', \phi_{|\overline{\alpha(\texttt{p})}|+1}, \iota}\ \textsc{Sem-ActorCon}$$

Figure 6.3: Asynchronous methods

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{C},\mathtt{m}) = \mathtt{fun}\ \beta\ \mathtt{m}(\overline{\mathtt{x}\ :\ \mathtt{AT}})\ \Rightarrow\ \mathtt{E} \\ \chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, \iota_{\mathtt{caller}}) \quad \chi(\iota_{\mathtt{caller}}) = (L, \mathtt{x} = \alpha(\mathtt{p}).\mathtt{m}(\overline{\alpha(\mathtt{p})})\mathtt{;E}', \mathbf{null}, v_{\mathtt{caller}}) \\ \chi_i, \phi_i, \alpha(\mathtt{p})_i \rightsquigarrow \chi_{i+1}, \phi_{i+1}, v_i \quad \chi_{|\overline{\alpha(\mathtt{p})}|+1}, \phi_{|\overline{\alpha(\mathtt{p})}|+1}, \alpha(\mathtt{p}) \rightsquigarrow \chi', \phi', u \quad \chi' \vdash_{\text{heap}} u : \mathtt{C} \\ \iota' \notin \mathrm{dom}(\chi') \quad \chi'' = \chi'[\iota' \mapsto ([\mathtt{this} \mapsto u, \overline{\mathtt{x}_i \mapsto v_i}], \mathtt{E}, \mathbf{null}, \iota_{\mathtt{caller}})]\end{array}}{\chi, \iota \rightsquigarrow \chi''[\iota \mapsto (\mathtt{A}, F, v_{\mathtt{next}}, \iota'), \iota_{\mathtt{caller}} \mapsto (L, \mathtt{E}', \mathtt{x}, v_{\mathtt{caller}})]}\ \textsc{Sem-FunCall}$$

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{C},\mathtt{k}) = \mathtt{new}\ \mathtt{k}(\overline{\mathtt{x}\ :\ \mathtt{AT}})\ \Rightarrow\ \mathtt{E} \\ \chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, \iota_{\mathtt{caller}}) \quad \chi(\iota_{\mathtt{caller}}) = (L, \mathtt{x} = \mathtt{C}.\mathtt{k}(\overline{\alpha(\mathtt{p})})\mathtt{;E}', \mathbf{null}, v_{\mathtt{caller}}) \\ \chi_i, \phi_i, \alpha(\mathtt{p})_i \rightsquigarrow \chi_{i+1}, \phi_{i+1}, v_i \quad \chi' = \chi_{|\overline{\alpha(\mathtt{p})}+1|} \quad \iota', \iota_{new} \notin \mathrm{dom}(\chi') \quad \iota' \neq \iota_{new} \\ \chi'' = \chi'[\iota' \mapsto ([\mathtt{this} \mapsto \iota_{new}, \overline{\mathtt{x}_i \mapsto v_i}], \mathtt{E}, \mathbf{null}, \iota_{\mathtt{caller}}), \iota_{new} \mapsto (\mathtt{C}, \emptyset)]\end{array}}{\chi, \iota \rightsquigarrow \chi''[\iota \mapsto (\mathtt{A}, F, v_{\mathtt{next}}, \iota'), \iota_{\mathtt{caller}} \mapsto (L, \mathtt{E}', \mathtt{x}, v_{\mathtt{caller}})]}\ \textsc{Sem-ClassCon}$$

Figure 6.4: Synchronous methods - a diagram of their effect is shown in Figure 6.5.

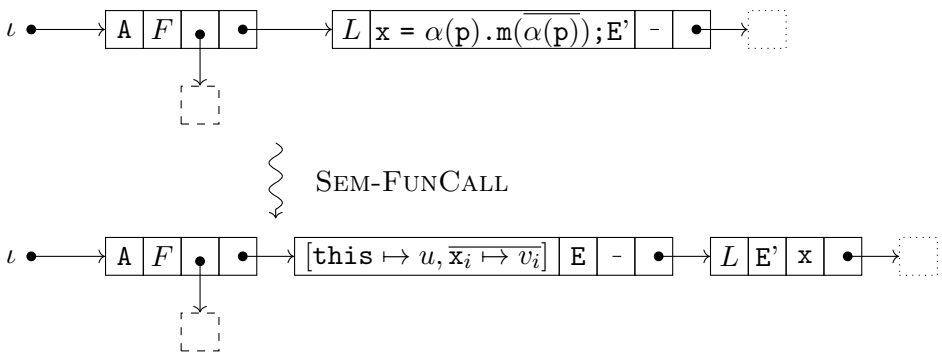


Figure 6.5: A diagram representation of Sem-FunCall - the top frame of the actor makes a synchronous function call, and waits for the return value to be assigned to the variable x.

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{A},\mathtt{b}) = \mathtt{be}\ \mathtt{b}(\overline{\mathtt{x}\ :\ \mathtt{AT}}) \Rightarrow \mathtt{E} \\ \chi(\iota) = (\mathtt{A}, F, \iota', \mathbf{null}) \quad \chi(\iota') = (\mathtt{b}, A, v_{\mathtt{next}}) \quad L = A[\mathtt{this} \mapsto \iota]\end{array}}{\chi, \iota \rightsquigarrow \chi[\iota \mapsto (\mathtt{A}, F, v_{\mathtt{next}}, \iota'), \iota' \mapsto (L, \mathtt{E}, \mathbf{null}, \mathbf{null})]}\ \textsc{Sem-Global-Receive}$$

$$\frac{\chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, \iota') \qquad \chi(\iota') = (\phi, \mathtt{e};\mathtt{E}, \mathbf{null}, v_{\mathtt{caller}}) \qquad \chi, \phi, \mathtt{e} \rightsquigarrow \chi', \phi', v}{\chi, \iota \rightsquigarrow \chi'[\iota' \mapsto (\phi', \mathtt{E}, \mathbf{null}, v_{\mathtt{caller}})]}\ \textsc{Sem-Global-Advance}$$

$$\frac{\begin{array}{c}\chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, \iota') \quad \chi(\iota') = (\phi, \mathtt{p}, \mathbf{null}, \iota_{\mathtt{caller}}) \\ \chi(\iota_{\mathtt{caller}}) = (\phi_{\mathtt{caller}}, \mathtt{E}, \mathtt{x}, v_{\mathtt{caller}}) \quad \chi, \phi, \mathtt{p} \rightsquigarrow \chi', \phi', v\end{array}}{\chi, \iota \rightsquigarrow \chi'[\iota \mapsto (\mathtt{A}, F, v_{\mathtt{next}}, \iota_{\mathtt{caller}}), \iota_{\mathtt{caller}} \mapsto (\phi_{\mathtt{caller}}[\mathtt{x} \mapsto v], \mathtt{E}, \mathbf{null}, v_{\mathtt{caller}})]}\ \textsc{Sem-Global-Return}$$

$$\frac{\begin{array}{c}\chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, \iota') \quad \chi(\iota') = (\phi, \mathtt{p}, \mathbf{null}, v_{\mathtt{caller}}) \\ v_{\mathtt{caller}} = \iota_{\mathtt{caller}} \implies \chi(\iota_{\mathtt{caller}}) = (\_, \_, \mathbf{null}, \_) \quad \chi, \phi, \mathtt{p} \rightsquigarrow \chi', \phi', v\end{array}}{\chi, \iota \rightsquigarrow \chi'[\iota \mapsto (\mathtt{A}, F, v_{\mathtt{next}}, v_{\mathtt{caller}})]}\ \textsc{Sem-Global-End}$$

Figure 6.6: Operational semantics for global execution

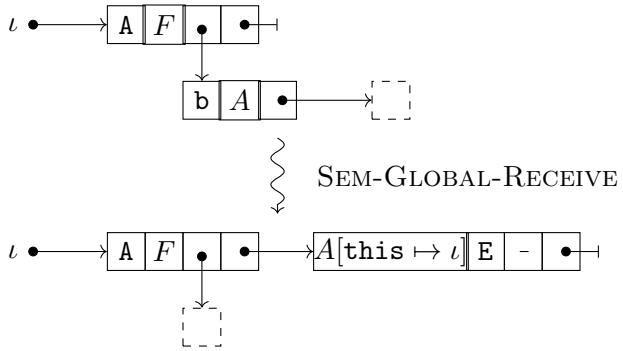


Figure 6.7: A diagram representation of Sem-Global-Receive - the actor at ι has an empty frame stack, so it processes the next message in the queue.

- Sem-Global-Receive, for some actor which is not currently executing a behaviour and has messages in its queue, begins the execution of the next message handler.
- Sem-Global-Advance executes the next expression in the current frame for some actor.
- Sem-Global-Return returns the result of the last expression in a frame to the parent frame for some actor.
- Sem-Global-End executes the last expression in a frame which either returns nothing or otherwise has no parent.

Diagrams showing the effect of these rules on the heap are shown in Figures 6.7, 6.8.

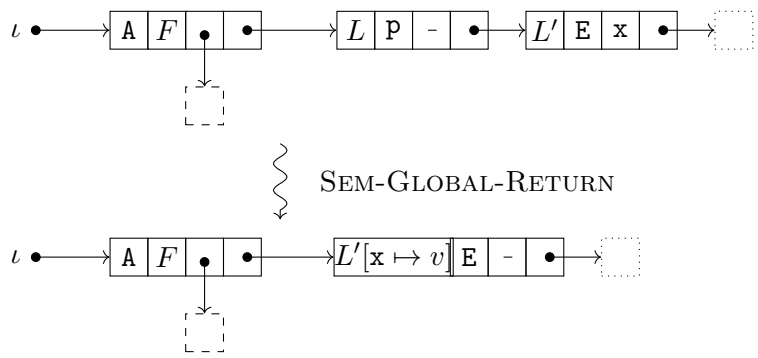


Figure 6.8: A diagram representation of Sem-Global-Return - the top frame of the actor at ι is finished executing, and it returns its value into the caller frame.

6.3 Comparison to previous work

The form of the semantics is very different from those described in Steed 2016. The main reason for this is the change to have frames be heap-allocated - it would be significantly more complex for the evaluation judgement to manipulate a stack of frames, since allocating and de-allocating frames for synchronous method calls requires also manipulating the heap. While this approach could have been taken, it was ultimately decided that it was not necessary. Instead, all stack manipulations are performed by the advancing judgement, which avoids the problem by not having frame information.

The definition of the semantic rules themselves are also very different. As remarked before, the fact that the syntax of *Pony with explicit aliasing* only allows expressions in a form that does not require continuations means that every expression can be evaluated all-at-once. This allows the rules themselves to be written in "big-step" form. For example, the Sem-FunCall rule evaluates all arguments in sequence - since each argument is a path with no side-effect on the heap, concurrency analysis is not affected by "batching" this argument evaluation.

List of definitions in this chapter

Chapter 7

Pony with explicit aliasing - Results

7.1 Soundness and preservation of expression well-formedness

Result 1 (Type soundness)**.** *If* $\Gamma \vdash \texttt{e} : \texttt{S}\ \kappa \dashv \Gamma'$*, and* $\Gamma \vdash_{expr} \texttt{e;E} : \texttt{T}$*, and* $\mathbf{R} \Vdash \chi \diamond$*, and* $\chi(\iota) = (\phi, \texttt{e;E}, \mathbf{null}, \_)$*, and* $\mathbf{R}, \chi, \iota \vdash_{heap} \phi \diamond : \Gamma$*, and* $\chi, \phi, \texttt{e} \rightsquigarrow \chi', \phi', v$*, then* $\chi' \vdash_{heap} v : \texttt{S}$*, and* $\Gamma' \vdash_{expr} \texttt{E} : \texttt{T}$*.*

Result 2 (Type soundness of returned values)**.** *If* $\Gamma \vdash_{expr} \texttt{p} : \texttt{S}\ \kappa$*, and* $\mathbf{R} \Vdash \chi \diamond$*, and* $\chi(\iota) = (\phi, \texttt{p}, \mathbf{null}, \_)$*, and* $\mathbf{R}, \chi, \iota \vdash_{heap} \phi \diamond : \Gamma$*, and* $\chi, \phi, \texttt{p} \rightsquigarrow \chi', \phi', v$*, then* $\chi' \vdash_{heap} v : \texttt{S}$*.*

Results 1 and 2 are two cases of the same general property of type soundness. They assert that evaluating well-formed expressions in well-formed heaps gives values of the expected type, and any remaining expressions are still well-formed with respect to the resulting typing context.

7.2 Progress of well-formed heaps

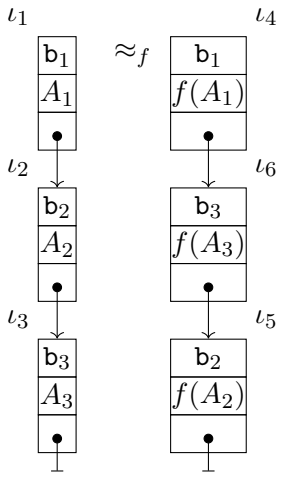
Result 3 (Progression of well-formed heaps)**.** *If* $\mathbf{R} \Vdash \chi \diamond$*, then either*

- $\forall \iota \in \mathrm{dom}(\chi), \chi(\iota) = (\texttt{S}, F, v_{\texttt{next}}, v_{\texttt{caller}}) \implies v_{\texttt{next}} = \mathbf{null} \wedge v_{\texttt{caller}} = \mathbf{null}$ *i.e. there is no code left to run, because all message queues and stacks of frames are empty, or*
- $\exists \chi', \iota$ *such that* $\chi, \iota \rightsquigarrow \chi'$ *i.e. some execution is possible.*

7.3 Preservation of heap well-formedness

Result 4. *If* $\mathbf{R} \Vdash \chi \diamond$ *and* $\chi, \iota \rightsquigarrow \chi'$*, then there exists a region partition* $\mathbf{R}'$ *such that* $\mathbf{R}' \Vdash \chi' \diamond$*.*

While this result may initially seem quite weak, since it only asserts that *some* region partition for the resulting heap exists, that is itself a sufficiently strong statement to be meaningful. Since every region partition enforces *at least* the well-formedness of the heap being partitioned, the existence of some partition is enough to assert heap well-formedness. Which particular partition can be applied to the heap is not as relevant.



Then $\iota_1 \approx_f \iota_4$, where

$$f = \left\{ \begin{array}{ccc} \iota_1 & \mapsto & \iota_4 \\ \iota_2 & \mapsto & \iota_5 \\ \iota_3 & \mapsto & \iota_6 \\ & \vdots & \end{array} \right\}$$

Figure 7.1: An example of the $\approx_f$ relation. The two addresses ι_1, ι_4 are related by the bijection f between them - except that the last two messages are allowed to be swapped.

7.4 Lack of deadlocks

Proving that this Pony model is deadlock-free is simply a question of applying Results 3 and 4. If well-formed heaps always progress while some code exists to execute, and progression preserves heap well-formedness, then every well-formed heap can be progressed either infinitely or until there is no longer any code left to execute.

7.5 Prevention of data races

7.5.1 Equivalent heaps

Since allocation is non-deterministic, it is valuable to have a notion of heap equivalence which is not dependent on specific heap addresses. However, Pony also permits other non-deterministic choices which should not impact local state - one of which is the reordering of messages up to the causal ordering.

Particularly, we can define a stronger relation for two executions - the $\approx_f$ relation. This relation is parameterised by an address bijection $f : \mathrm{dom}(\chi) \to \mathrm{dom}(\chi')$.

Definition 7.1 (Message queue permutation relation ($\approx_f$)).
The relation $\iota \approx_f \iota'$ holds for two message addresses ι, ι' if and only if they are identical under the bijection f except possibility with the last two messages reordered.

This relation implies a causal reordering of the related queues, if the last two messages in the queues are sent from different actors. An example is shown in Figure 7.1.

This allows for a heap equivalence relation, denoted $\chi \sim \chi'$, which states that the heaps χ, χ' do not differ in the local state of any actor.

Definition 7.2 (Equivalent heaps ($\sim$)).

$$\chi \sim \chi' \iff \exists f : \mathrm{dom}(\chi) \to \mathrm{dom}(\chi') \text{ a bijection s.t.}$$
$$\chi'(f(\iota)) = \begin{cases} (\mathtt{C}, f(F)) & \chi(\iota) = (\mathtt{C}, F) \\ (f(L), \mathtt{E}, v_{\mathtt{x}}, f(v_{\mathtt{caller}})) & \chi(\iota) = (L, \mathtt{E}, v_{\mathtt{x}}, v_{\mathtt{caller}}) \\ (\mathtt{b}', A', \_) & \chi(\iota) = (\mathtt{b}, A, \_) \\ (\mathtt{A}, f(F), v'_{\mathtt{next}}, f(v_{\mathtt{caller}})) & \\ \quad s.t.\ v'_{\mathtt{next}} \approx_f v_{\mathtt{next}} & \chi(\iota) = (\mathtt{A}, F, v_{\mathtt{next}}, v_{\mathtt{caller}}) \end{cases}$$

where $f(M) = \{\mathtt{x} \mapsto f(v) \mid M(\mathtt{x}) = v\}$.

Notably, while most of the addresses are related directly by the bijection f, messages are not, and the restriction on message queues is that of being related under $\approx_f$.

7.5.2 Data races

Result 5. *If* $\chi, \iota_a, \rightsquigarrow \chi_a$, *and* $\chi, \iota_b \rightsquigarrow \chi_b$, *and* $\iota_a \neq \iota_b$, *then* $\chi_a, \iota_b \rightsquigarrow \chi'$ *and* $\chi_b, \iota_a \rightsquigarrow \chi''$ *and* $\chi' \sim \chi''$.

In words - in a heap where it is possible to advance two distinct actors to new heaps, then in each new heap it is possible to advance the other actor as well, and any resulting heaps after these two steps are equivalent.

7.5.3 Logical race conditions

Significantly, Result 5 is *not* transitive, and cannot be repeatedly applied. This is expected - if the result was transitive, it would imply the much stronger property of a lack of *logical race conditions*: where message reordering affects some outcome of global execution.

For example, in the scenario shown in Figure 7.2, the final value of the field `x` in actor `C` depends on the order in which the messages `doubleX` and `add2toX` are received. This represents a logical race condition - but not a data race, since there is no use of shared mutable memory.

We expect the two resulting heaps from the scenario shown in Figure 7.2 to possibly diverge - for example, code in actor `C` may run two different ways, depending on if the final value of `x` is 6 or 8. If Result 5 *were* transitive, this divergence wouldn't be possible.

List of definitions in this chapter

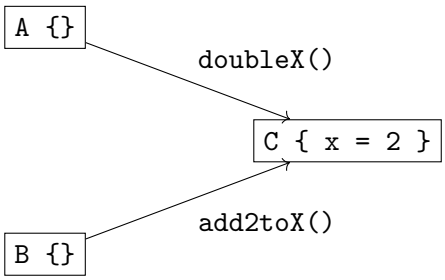


Figure 7.2: A logical race condition.

Chapter 8

Conclusions

8.1 Evaluating the formal model

Empirical comparisons between the various formal models of Pony (Section 3.5) is very difficult. Despite this, we hold that the region system has produced a significantly simpler model for accessibility, and in turn simplified the statement and proof of Pony's correctness properties.

The region approach fits more closely with user intuition about the nature of capabilities - in fact, regions were used in Clebsch et al. 2015 to describe the `iso` and `trn` capabilities. More generally, the region system has a *local* consideration of the heap - and local reasoning is more desirable (O'Hearn, J. Reynolds, and Yang 2000). Because the new approach does *not* require knowledge of the heap as a whole to decide the validity of individual accesses, it is scalable - and that scalability makes it valuable for both human and computer analysis.

8.2 The Coq model

The Coq model (Appendix A) was a major part of the workload for the project as a whole, despite not representing any novel contributions to the theory. The use of Coq was intended to simply replace the need for handwritten proofs of important properties, such as soundness (Result 1) and progress (Result 3). It was expected that Coq's strong support for inductive definitions would reduce the time spent in inductive proofs, which would comprise most of the proofs relevant to the project; while at the same time, a computer-checkable formalisation would prevent proof errors.

Instead, the scope of such a model, even describing just *some* of the results of the report, proved to be larger than the project period allowed for. The actual encoding of the syntax, semantics, and region system took much more time and effort than planned for - especially as multiple technical limitations of Coq were run up against (Appendix A.2).

Ultimately, the work required to encode and prove even just Result 1 would have proved to be too great for the time provided by a JMC project. The goal of simultaneously describing and verifying a novel region system was too ambitious, and had the downside of diverting effort way from handwritten proofs.

8.2.1 Lack of full proofs

Since the value of a formal model such as the one presented in this report is in its proven properties, the lack of full proofs is a significant limitation. While it is possible to intuit whether the results presented in Chapter 7 do truly hold, such an assessment is much less meaningful than a true proof.

It is even possible that one or more of the results, as stated, is technically incorrect. Without a proof, errors in the statement of these results - such as the omission of a necessary precondition, or a too-strong postcondition - are much harder to detect and correct.

While a correct Coq proof would totally eliminate this class of errors, it is likely that even a handwritten proof would have allowed for finding and correcting most or all such mistakes.

Chapter 9

Future work - Borrowing in Pony

9.1 Motivating example: the n-body problem

An n-body simulation is one of the programs which is part of the Savina actor benchmark (Imam and Sarkar 2014). Savina is a collection of abstract programs implemented in multiple actor-based languages - including Pony - to compare performance between those languages.

In the n-body program, a central actor maintains a copy of the state of the simulation, including the position, direction, and other characteristics of all the bodies. Each body has a corresponding actor. The simulation progresses in discrete intervals - the central actor shares the global state with each body's actor, and that actor responds with the changes that its body should undergo in the interval. Once all the body actors have responded, the central actor applies all the changes, and the process repeats.

Currently, Pony is unable to efficiently perform this simulation. The particular concern for performance is the need to create a copy of the global state in order to share it with the body actors: the original global state, being mutable, cannot be shared between multiple actors. The global state cannot be made immutable itself since this process is irreversible, and the state must be updated at each discrete interval.

From the perspective of memory safety, the global state copy is not strictly necessary - the central actor does not mutate the global state unless all body actors have given their update, at which point no body actor will read the old state copy again. In effect, if it were possible to flag to the compiler that the global state was `iso` between simulation intervals, when it is being updated, and `val` otherwise, when body actors are reading from the state, then the copy could be eliminated. This is the role of *borrowing*.

9.2 Borrowing

Borrowing is the terminology for a language feature in which a value, which is held with some capability, can be temporarily given a normally-conflicting capability. This operation is called a *loan*, or *borrow*.

The act of loaning a value also changes the capability with which it was originally accessed, to prevent violations of capability guarantees - for example, loaning a `iso`

as `val` requires that the original alias not still be mutable, as this would violate the immutability guarantee of `val`.

9.3 Glossary

Definition 9.1 (Loan).
A (temporary) path used to access a value without needing ownership of it. A loan is created by an actor from an owned value.

Definition 9.2 (Borrow).
Same as a loan. Existing literature uses "borrow" and "loan" interchangeably.

Definition 9.3 (Loaned value).
The value from which a loan is created. The term "loaned value" is used only to refer to the value from the perspective of its original owner, not the user of the loan.

Definition 9.4 (Borrowed value).
The value accessible through a loan. The term "borrowed value" is used only to refer to the value from the perspective of the loan user, not its original owner.

In practice, borrowed values are often themselves loanable.

Definition 9.5 (Loan extension).
A loan on an existing borrowed value, so-called because it "extends" the term of the original loan.

A loan extension will typically prevent the original loan from ending until the extension itself has ended. Similarly to other loans, a loan extension modifies the capability with which the borrowed value can be accessed from the original loan.

9.4 Loans in an asynchronous language

Loans with no determinable end are undesirable. A value which is loaned indefinitely could never again be safely used in its non-loaned form, since there would be no guarantee that the loan has ended. In this sense, there would be no difference between an indefinite loan and using the existing features of Pony to change the capability of some value. For this reason, we are only interested in loans which have some definite (if over-approximated) end.

In sequential languages, determining the *lifetime* of a loan can be done in terms of *variable scopes* (Weiss et al. 2019; Matsakis 2018): a loan is said to last for as long as some variable scope is still borrowing it. This allows for borrowed values to be passed as arguments to functions - which create new scopes - while still having a determinable loan end from the perspective of the loaned value's owner.

However, this approach is not as applicable to concurrent languages, particularly actor languages. Since Pony's message passing is asynchronous, control-flow-based approaches cannot be used between actors; message handlers execute largely non-deterministically with respect to each other. The relevant control flow property in Pony in these instances is the causal message-sending relation.

For this reason, we take the approach of having loan ends be marked by a specific message for that loan, which allows the loaning actor to *recover* the non-loaned capabilities for the value. The loan-end messages must leverage the causal message-sending relation to ensure the correctness of inter-actor loans.

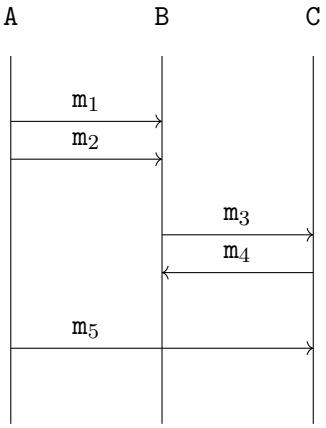


Figure 9.1: An example of some messages being sent. $\mathtt{m}_2$ and $\mathtt{m}_5$ are causally ordered because they have the same sender. $\mathtt{m}_2$ and $\mathtt{m}_3$ are causally related because the first is received before the second is sent. However, $\mathtt{m}_3$ and $\mathtt{m}_5$ have no causal relation.

9.4.1 Causal ordering

Messages in Pony are guaranteed to be delivered to the recipient actor in some *causal* order.

Definition 9.6 (Causal order).
The causal ordering of messages is given by the transitive closure of:

1. *If an actor sends a message, and then sends another message, the two messages are causally ordered.*
2. *If an actor receives a message, and then sends a message, the two messages are causally ordered.*

The above two rules are not sufficient to give a *global ordering* (i.e. one where all messages sent during execution are always delivered in the same order). Instead, the causal ordering of messages allows for many *actual* delivery orders, so long as they obey the causal rules.

An example is shown in Figure 9.1. The messages $\mathtt{m}_1$ and $\mathtt{m}_2$ must be delivered in the order they are sent, since by the first rule they are causally ordered. However, since $\mathtt{m}_3$ and $\mathtt{m}_5$ are *not* causally ordered, there are multiple possible delivery orders for them - the actor `C` might receive either of $\mathtt{m}_3, \mathtt{m}_5$ first. Both would be correct.

9.4.2 Loan messages

We can use the causal ordering to implement borrowing in terms of two messages.

- *Loan extension* - this message would inform the actor that the loan on a particular field has been extended.
- *Loan end* - this message would inform the actor that some loan (possibly a loan extension) on a particular field has ended.

When the loan is first created, the actor knows that exactly one loan exists on the loaned value. Whenever some loan on the field is extended, the actor will be sent a

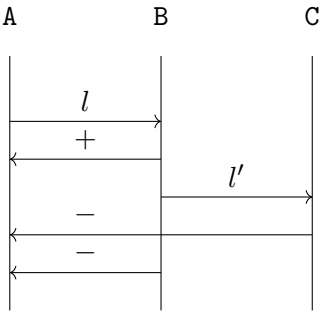


Figure 9.2: An example of loan messages. The messages l, l' represent loans being sent. The messages $+, -$ represent extension and end messages, respectively.

loan extension message; and whenever some loan on the field ends, the actor will be sent the corresponding *loan end*. So long as the loan extension messages are causally ordered before the loan end messages, the actor will always over-approximate the period of the loan.

An example is shown in Figure 9.2. The actor `A` loans some value to `B`. `B` can then extend the loan to `C` - but it must first inform `A` of the extension. Then `B`, `C` can both use their loans, and finally send their end messages to `A`. Because the extension message sent by `B` is causally ordered before either of the end messages, `A` can never mistakenly believe the loan has ended while some extension still exists.

Importantly, `B` *must* send the extension message before sharing the loan with `C`. Otherwise, the extension message sent by `B` and the end message sent by `C` would not be causally related - and if `A` received the end message first, it would believe that the only loan which it is aware of (the one held by `B`) had ended.

9.5 Loans at compile-time

A particular concern for actors which loan values is - what should happen in behaviours where a value which is loaned needs to be used in its original capability? More generally, how can actors work around the possibility that, at any time, some field may be loaned?

There are various possibilities. One is that each loanable value can be considered in actuality having a sum type of all its possible loaned forms - for example, a `iso` is in fact a value which can have capability `iso` (because it is not loaned), `val` (because it is readably loaned), or `tag` (because it is writably loaned). However, this approach could introduce problems for programmers - if the lack of a certain capability is a result of programmer error, the compiler could not help to prevent it.

Another approach involves partially or totally suspending (at runtime) the actor's behaviours which could require a value in its non-loaned form until the loan has ended. However, this would break Pony's promise of deadlock-free code - if a behaviour would be prevented from running while a loan is in progress, the actor using the borrowed value could simply invoke that behaviour and never end the loan. The result would be that some actor would have a message it could never process: hence, a deadlock.

The third approach is to attempt to prevent at compile-time any possible violations of a loan. However, this requires a static approach to detecting what loans are

active at what points in the code. We do this through *typestates*.

9.5.1 Typestates

The term *typestates* refers to a concept of type-level state. This state can decide which subsets of a type's functionality can be used at any point in the program (Flint 2020), or even allow for a complete change of type for a value (S. Drossopoulou et al. 1999; Sophia Drossopoulou et al. 2002). Typically, these states exist only at compile-time to describe and enforce some additional correctness checks - for example, in Flint 2020, typestates are used to control the various states a smart contract can be in, and prevent invalid state transitions.

Borrowing in Pony, and specifically tracking which fields are borrowed on an actor, could be done through typestates. The state of an actor type could be a aggregate of all of the borrowed states of its fields - for example, in the actor type:

```
actor A
  var f : int iso
  var g : string trn
```

the possible typestates for `A` include

- Neither `f` nor `g` are borrowed - this is the default typestate that existing Pony programs use for all types.
- `f` is borrowed writably, and `g` is not borrowed.
- `f` is borrowed readably, and `g` is borrowed readably.
- `f` is not borrowed, and `g` is borrowed readably.
- ...

In fact, there are as many typestates for `A` as there are combinations of possible loans on the fields `f`, `g`.

9.5.2 Mutable and immutable behaviours

As the action of loaning a value would alter the typestate of an actor, it is necessary to know which behaviours of an actor result in a loan on which fields. Additionally, since it should not be possible (in general) to loan an already-loaned value, it is also necessary to know for which typestates a behaviour which results in a loan may be called - specifically, those typestates where the value is not already loaned.

A system for describing how methods rely on and alter typestate is described in S. Drossopoulou et al. 1999. In that report, types which may change typestate are described as *mutable*, while all other types are considered *immutable*. A similar distinction is made for methods: methods which may change the typestate of their receiver are called *mutable*, and methods which cannot change the type of their receiver are called *immutable*. A mutable method can only be defined with a mutable receiver type, for obvious reasons.

We reproduce these concepts in an appropriate form for Pony. Since we are only interested in loaning actor fields, it is sufficient to treat only actor types (and to treat all actor types) as mutable types. All other types in the program are immutable. We also define mutable and immutable behaviours.

Definition 9.7 (Mutable behaviour).
A behaviour is mutable *if it may loan a value.*

Definition 9.8 (Immutable behaviour).
A behaviour is immutable *if it does not loan any values, but may require some value to not be loaned.*

These definitions are similar to those for mutable and immutable methods in S. Drossopoulou et al. 1999, simply redefined in terms of our typestates. Since a loan changes the actor's typestate, it follows that behaviours which create loans must be mutable.

The receiver type for both mutable and immutable behaviours must then take into account the typestates permitted to *enter* the behaviour - and for mutable behaviours, the typestates that may *exit* the behaviour. For a mutable behaviour which creates a loan on a field, the entering typestate must have that field be unloaned; and the exiting typestate must have the field loaned. Immutable behaviours do not create loans, but may still require some value to be unloaned - so they also place restrictions on the entering typestate.

9.5.3 Behaviours and `tag`

Since the intention of typestates is to prevent the invocation of behaviours which should not be executed - because the receiving actor cannot use some loaned value - there must be some change to how behaviours are called. In *Pony with explicit aliasing*, this was done exclusively through a `tag` reference.

`tag` references to an actor may be held by multiple other actors - so they must forbid invocations to mutable behaviours. The reason for this is simple: if some behaviour creates a loan of a field, it requires the field to be unloaned; if two actors could invoke that behaviour, the first call would succeed and the second would fail.

Immutable behaviours do not have the same restriction - they could be invoked by multiple actors at a time. However, since immutable behaviours can restrict the set of possible entering typestates, it must be clear to the type system at the point the behaviour is called that the receiving actor is in a valid typestate. Furthermore, for as long as some `tag` reference to the actor exists, *the actor cannot change typestate* - since this would invalidate the ability to call the immutable behaviour.

In practice, this means that we have good reason to forbid the invocation of both mutable and immutable behaviours on `tag` references. The only behaviours which are safe to call through a `tag` are those that have no typestate interaction - they do not use possibly-loaned values, and they do not loan values.

9.5.4 Borrowing as a contract

If a `tag` reference to an actor is not enough to invoke its mutable and immutable behaviours, how then can those behaviours be invoked? We take the approach of *loaning the actor references themselves.*

There are many parallels between the mutability of data and the mutability of behaviours: it should not be possible to allow multiple typestate-mutable references to a single actor, similarly to how multiple mutable references to data cannot be shared between actors. Equivalently, it would be permissible to allow multiple typestate-immutable references to a single actor (since none could invalidate any other), just as

it is permitted to have multiple immutable references (through `val`) shared between actors.

These similarities motivate two new kind of actor reference:

- A *unique mutable* reference - effectively a typestate-`iso`. Such a reference could be used to invoke mutable behaviours, and could be sent between actors.
- A *non-unique immutable* reference - effectively a typestate-`val`. Such a reference could be used to invoke immutable behaviours. An unlimited number of this kind of reference could exist, since all the behaviours invokable with it would not change the typestate.

Moreover, both kinds of references are motivated to be *temporary*. Since either kind of reference precludes the existence of the other, allowing for the creation of one kind of reference would mean forbidding the other kind: much in the same way that, if a `val` reference exists for some value, no `iso` reference to it could exist. Programs which want to invoke both mutable and immutable behaviours will therefore want that different kinds of actor references can exist at different points in the execution. In effect, *actor references themselves* will need to be *loaned*.

For this reason, we can use the loan of an actor reference by that same actor to model a commitment to a temporary contract. For example, if an actor loans a unique mutable reference of itself, it gives a commitment to the holder of that reference that the actor will be in a certain typestate, and able to transition between typestates as necessary, for as long as the borrowed reference exists. The actor which holds the reference can safely call mutable and immutable behaviours (which have the correct entry typestate) at will, by using the borrowed reference.

For the case of non-unique immutable actor references, the model is similar. An actor which loans a non-unique immutable reference of itself commits that its typestate will not change for as long as the borrowed reference exists. The actors which hold any copy of the borrowed reference can safely call the immutable behaviours (which have the correct entry typestate) by using that reference, since the receiving actor cannot transition away from it.

This requires an addition to the typestate for an actor - as well as the loan status of the actor's fields, the actor's typestate must also track how the actor's reference of itself is loaned, and with what contract. An actor which wishes to commit to some contract for invokable mutable and immutable behaviours must not have already loaned its reference in a different way. Similarly, the actor should not be able to violate the commitment of the contract it has itself issued, by loaning its reference to another actor.

9.5.5 Compatibility with existing programs

This typestate approach does not invalidate any existing *Pony with explicit aliasing* programs - specifically, since a Pony program written without borrowing uses no typestates other than the default one, and never transitions between different typestates, it has no mutable or immutable behaviours. Therefore, the use of `tag` references is sufficient to invoke all behaviours on an actor.

This is to be expected - the addition of borrowing to the language does not invalidate or conflict with any existing operations.

λ	$\in$	Loan Capability	$:=$	`unique` | `borrow` | `handle` | `window`
κ	$\in$	Capability	$:=$	β | β`^` | λ | λ`^`
`ST`	$\in$	Scope Type	$:=$	`S` β | `S` λ
`l`	$\in$	Loanable	$:=$	`this` | `f`
`RHS`	$\in$	Assignment RHS	$:=$	`p` | `p.f = p` | `recover RHS end` | `MC` | `loan[l]`
`e`	$\in$	Expression	$:=$	`var x` | `x =` α`(RHS)` | t `<- p.f` | `l.andThen(E)`

Figure 9.3: Definition of types and expressions with borrowing

`K`	$\in$	Constructor Def	$:=$	`new k(`$\overline{\texttt{x : ST}}$`) ⇒ E`
`M`	$\in$	Function Def	$:=$	`fun` β `m(`$\overline{\texttt{x : ST}}$`) : T ⇒ E`
`B`	$\in$	Behaviour Def	$:=$	`be b(`$\overline{\texttt{x : ST}}$`) ⇒ E`

Figure 9.4: Definition of actors, classes, and programs with borrowing

9.6 Pony with explicit aliasing and borrowing - Syntax

In extending the model of *Pony with explicit aliasing* with borrowing, we find that many of the syntax structures remain identical. The major changes due to the extension are:

- We add the new *loan capabilities*, which are restricted in their usage and cannot appear arbitrarily in a program (similar to the ephemeral capabilities).
- We add the syntax category of *loanables* - values which can be loaned. The two kinds of loanable are
  - A field of the current actor, referred to by its field ID `f`.
  - The actor's own reference, referred to by the self-reference `this`.
- We define the `loan` RHS, which takes a loanable as an argument and produces the appropriate borrowed value.
- We define the `andThen` expression, which declares what code an actor will execute once it knows that the loan has ended. This allows the actor to perform some action after it has recovered the loaned value's original capability - such as in the n-body simulation, where the central actor advances the simulation once it knows that the loan of the global state has ended.

9.6.1 Loan capabilities

Definition 9.9 (Loan capabilities).
We define two new Pony capabilities, each of which represents a kind of loan.

- *A* unique loan*, written* `unique`*. The borrowed value can be read from and written to.*
- *A* shareably loan*, written* `borrow`*. The borrowed value can be read from.*

In addition, we define two Pony capabilities that are only meaningful for actor references.

- *A* unique mutable actor reference loan*, written* `handle`*. The borrowed value can be used similarly to a* `tag`*, except that it may invoke certain mutable and immutable behaviours.*
- *A* non-unique immutable actor reference loan*, written* `window`*. The borrowed value can be used similarly to a* `tag`*, except that it can invoke certain immutable behaviours.*

How a value can be loaned is based on its capability: every operation permitted on the borrowed value must have been permitted on the loaned value before the loan began. Additionally, since loans are intended for inter-actor messages, they must be sendable: the local and global alias rules for the loan capabilities must agree.

In the style of Clebsch et al. 2015, we can describe the rules for loanability in terms of *denying* operations for local and global aliases.

- A value with a capability which allows for reading and writing, and denies local read/write aliases, can be loaned uniquely. If local read/write aliases were permitted, they would also be aliases for the borrowed value - so they would need to be permitted as global aliases as well.
- A value with a capability which allows reading, and denies local write aliases, can be loaned shareably. If local write aliases were permitted, then global write aliases would need to be permitted for the borrowed value.

Since a loan is a transfer of operation permissions, the loaned value is not considered an alias in terms of its original capabilities for the purpose of these rules. Instead, for the duration of the loan the loaned value loses all permissions not permitted on its own local aliases - this allows the borrowed value to be sendable.

This has the following consequences for loans:

- Only `iso` values can be loaned uniquely. The loaned value can only be used as a tag for the duration of the loan.
- Any of `iso`, `trn`, `val` can be loaned shareably. The loaned value can be read from for the duration of the loan.

For the actor reference capabilities, deny definitions are not as relevant. The only rule required is that in order to loan its reference as either `handle` or `window`, an actor must not be already loaning its reference.

λ \ κ	iso	trn	ref	val	box	tag
unique	unique	unique	unique	borrow	tag	tag
borrow	borrow	borrow	borrow	borrow	borrow	tag
handle	-	-	-	-	-	-
window	-	-	-	-	-	-
unique^	unique^	unique^	unique^	borrow	borrow	tag

Figure 9.5: Viewpoint adaptation for $\lambda \triangleright \kappa$

9.6.2 Loans of borrowed values

The loan capabilities also satisfy the above rules, and can themselves be loaned onward. That is

- A value with capability `unique` can be loaned uniquely or shareably.
- A value with capability `borrow` can be loaned shareably.

9.6.3 Aliasing borrowed values

Since the loan capabilities are first-class capabilities, it is necessary to define what happens when they are aliased. For the shareable capabilities (`borrow` and `window`), it is desirable that aliasing *extend* the loan by creating a new borrowed reference - this allows for some code to duplicate and share values with such shareable capabilities. For unique capabilities (`unique` and `handle`), it is necessary that aliasing give no access to the value i.e. yields `tag`.

This motivates the following modification to the aliasing map.

Definition 9.10 (Aliasing map ($\mathcal{A}$)).
The aliasing map $\mathcal{A}$: Capability $\rightarrow$ Capability is defined as

$$\mathcal{A}(\kappa) = \begin{cases} \beta & \kappa = \beta\texttt{\^{}} \\ \lambda & \kappa = \lambda\texttt{\^{}} \\ \texttt{tag} & \kappa = \texttt{iso} \vee \kappa = \texttt{unique} \vee \kappa = \texttt{handle} \\ \texttt{box} & \kappa = \texttt{trn} \\ \beta & \kappa = \beta, \kappa \neq \texttt{iso}, \kappa \neq \texttt{trn} \\ \lambda & \kappa = \lambda, \kappa \neq \texttt{unique}, \kappa \neq \texttt{handle} \end{cases}$$

9.6.4 Viewpoint adaptation

We can also extend the definition of the viewpoint adaptation operation to loan capabilities. These extended definitions are shown in Figures 9.5, 9.6. They are very similar to those for the base capabilities - in general, there is a correspondence between the behaviour of `unique` and `iso`, `borrow` and `val`, and `handle`/`window` and `tag`.

Note that it is not necessary to extend the columns of these definitions, because programs have *not* been changed to allow loan capabilities on fields. It is therefore not necessary to consider accesses or assignments of fields with loan capabilities.

λ \ κ	iso	trn	ref	val	box	tag
unique	iso^	val	tag	val	tag	tag
borrow	-	-	-	-	-	-
handle	-	-	-	-	-	-
window	-	-	-	-	-	-
borrow^	iso^	iso^	iso^	val	val	tag

Figure 9.6: Viewpoint adaptation for $\lambda \triangleright \kappa$

λ \ κ	iso	trn	ref	val	box	tag
unique	✓			✓		✓
borrow						
handle						
window						
unique^	✓			✓		✓

Figure 9.7: The safe-to-write relation $\lambda \triangleleft \kappa$

9.6.5 Safe-to-write

The definition of safe-to-write can also be extended to loan capabilities. Similarly to the case of viewpoint adaptation, there is a correspondence between the loan capabilities and their sendable base counterparts - with one exception. *Unlike* `iso^`, it is not safe to write a value with capability `trn` (or `ref`, `box`) to a value with capability `unique^`.

This is due to the nature of borrowing itself - while an unaliased `iso` is inaccessible from that point on in the execution of the program, an unaliased `unique` should be recoverable by the original owner. Therefore, if we were to allow writing of a `trn` to a `unique^`, the original owning actor would be able to recover the loaned value as `iso` - and that `iso` would contain a `trn` which was originally contained in some other actor's code. In other words, the `trn` would have been *sent* between the two actors.

Therefore, to preserve the invariants of the actor model, such operations must be forbidden.

List of definitions in this chapter

Bibliography

Ahrendt, Wolfgang and Maximilian Dylla (2012). "A system for compositional verification of asynchronous objects". In: *Science of Computer Programming* 77.12. International Conference on Formal Engineering Methods—ICFEM 2009, pp. 1289–1309. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2010.08.003. URL: http://www.sciencedirect.com/science/article/pii/S0167642310001553.

Clebsch, Sylvan (2017). *An Early History of Pony - Pony*. Accessed: 06-01-2020. URL: https://www.ponylang.io/blog/2017/05/an-early-history-of-pony/.

Clebsch, Sylvan et al. (2015). "Deny Capabilities for Safe, Fast Actors". In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE! 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 1–12. ISBN: 9781450339018. DOI: 10.1145/2824815.2824816. URL: https://doi.org/10.1145/2824815.2824816.

Dijkstra, Edsger W. (1963). *Over de sequentialiteit van procesbeschrijvingen (EWD-35)*. Tech. rep. University of Texas at Austin.

Drossopoulou, S. et al. (1999). *Objects dynamically changing class*. Tech. rep. Imperial College.

Drossopoulou, Sophia et al. (2002). "More dynamic object re-classification: FickleII". In: *ACM Transactions On Programming Languages and Systems* 24, pp. 153–191.

Flint (2020). *Type States*. Accessed: 14-06-2020. URL: https://github.com/flintlang/flint#type-states.

Gordon, Colin S. (2019). "Modal Assertions for Actor Correctness". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2019. Athens, Greece: ACM, pp. 11–20. ISBN: 978-1-4503-6982-4. DOI: 10.1145/3358499.3361221. URL: http://doi.acm.org/10.1145/3358499.3361221.

Gordon, Colin S. et al. (Oct. 2012). *Uniqueness and Reference Immutability for Safe Parallelism*. Tech. rep. MSR-TR-2012-79. Microsoft Research. URL: https://www.microsoft.com/en-us/research/publication/uniqueness-and-reference-immutability-for-safe-parallelism/.

Hewitt, Carl, Peter Bishop, and Richard Steiger (1973). "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

Hoare, C. A. R. (Oct. 1969). "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

Howard, William A. (1969). "The formulae-as-types notion of construction". In:

Imam, Shams M. and Vivek Sarkar (2014). "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries". In: *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. AGERE! '14. Portland, Oregon, USA: Association for Computing Machinery, pp. 67–80. ISBN: 9781450321891. DOI: 10.1145/2687357.2687368. URL: https://doi.org/10.1145/2687357.2687368.

Jones, C. B. (June 1981). "Development Methods for Computer Programs including a Notion of Interference". Printed as: Programming Research Group, Technical Monograph 25. PhD thesis. Oxford University. URL: http://www.cs.ox.ac.uk/files/9025/PRG-25.pdf.

Matsakis, Nico (2018). *An alias-based formulation of the borrow checker*. Tech. rep. Accessed: 14-06-2020. URL: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

Microsoft (2016). *Asynchronous Programming - C#*. Accessed: 15-01-2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/async.

Mozilla (2020). *Making asynchronous programming easier with async and await - Learn web development — MDN*. Accessed: 15-01-2020. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async%5C_await.

Müller, Peter (2002). "Modular Specification and Verification of Object-Oriented Programs". In: *Lecture Notes in Computer Science*.

O'Hearn, Peter, John Reynolds, and Hongseok Yang (2000). "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic*. Ed. by Laurent Fribourg. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–19. ISBN: 977-3-540-44802-0.

Pony (2019). *Pony*. Accessed: 20-12-2019. URL: https://www.ponylang.io/.

– (2020). *Pony Tutorial — Actors*. Accessed: 15-01-2020. URL: https://tutorial.ponylang.io/types/actors.html.

Pratt, Vaughan R. (1976). "Semantical Consideration on Floyd-hoare Logic". In: *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*. SFCS '76. Washington, DC, USA: IEEE Computer Society, pp. 109–121. DOI: 10.1109/SFCS.1976.27. URL: http://dx.doi.org/10.1109/SFCS.1976.27.

Reynolds, John C. (2002). "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74.

Rust (2020). *Getting Started - Asynchronous Programming in Rust*. Accessed: 15-01-2020. URL: https://rust-lang.github.io/async-book/.

Steed, George (June 2016). "A Principled Design of Capabilities in Pony". MA thesis. Imperial College London.

Summers, Alexander J. and Peter Müller (2016). "Actor Services". In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 699–726. ISBN: 978-3-662-49498-1.

van Bakel, Isaac (2020). *A formalisation of (a minimal subset of) Pony in Coq*. URL: https://github.com/ivanbakel/minimal-pony-coq.

Weiss, Aaron et al. (2019). *Oxide: The Essence of Rust*. arXiv: 1903.00982 [cs.PL].

Zürich, Eidgenössische Technische Hochschule (2020). *Viper - Programming Methodolgy Group — ETH Zurich*. Accessed: 17-01-2020. URL: https://www.pm.inf.ethz.ch/research/viper.html.

Appendix A

The Coq model

As part of the development of the new formalisation of Pony, I also developed a formal encoding of the model in Coq (van Bakel 2020).This model includes:

- a formal encoding of the full language syntax, the typing system and its judgements, and the program well-formedness judgements.
- a formal encoding of all heap structures, the region system and its judgements, and the heap well-formedness judgements.
- a formal encoding of the operational semantics and its judgements.
- the description and partial proof of Result 1.
- the description and partial proof of Result 2.
- the description and partial proof of Result 3.
- the description and partial proof of Result 4.

A.1 The encoding

A.1.1 Structures and judgements

Structures, particularly the syntax definitions, were implemented using Coq's `Inductive` definitions, which are a kind of algebraic data type with automatically-derived properties such as an inductive principle and constructor injectivity.

Such definitions were also used to implement the various inductive judgements presented throughout the report. For example, the typing judgement has a recursive structure, and was implemented as an inductive data type. Since Coq allows for the definition of propositions as a first-class language construct, then these inductive definitions could be made propositions: in other words, these judgements could be asserted, proven, and disproven using their inductive structure.

A.1.2 Polymorphic judgements

As presented in this report, the Pony model has several judgements over multiple kinds of structures. For example, the typing judgement is defined over paths (Path-Var) and expressions (Expr-VarDecl). Additionally, the *explicit alias* construction $\alpha(\dots)$ applies to both paths and assignment right-hand-sides, so Expr-Alias itself applies to multiple possible forms of typing judgements.

While the encoding of these judgements could have been expressed as one-per-type, this would have required introducing multiple nearly-identical judgements. Obvious properties for the judgements, such as determinism of type for the typing judgement, would have had to be named and proven multiple times. In addition, for those judgements with a recursive structure, induction would have to be expressed over n judgement types, which would significantly complicate the proof structure.

Instead, the approach was taken of making each of these judgements polymorphic - therefore, there would only be one judgement of each form, no matter the type of value being judged.

Closed-world encoding

As in many polymorphic languages, Coq's polymorphism operates using an *open-world* assumption: that is, it is assumed the types which may inhabit a type variable are not limited to those defined at the point the variable is declared. While this is normally desirable, the Pony model circumvented it (for reasons described in Section A.2.3).

Instead, the model uses a *closed-world* encoding for every polymorphic judgement. Unlike in the open-world case, the closed-world encoding gives an explicit declaration of which types may inhabit a type variable. This limits the possible values of the variable; in particular, additional type definitions cannot affect the set of values taken by that variable.

A.2 Limitations

While encoding the model, I ran up against some limitations of Coq as a proof system.

A.2.1 Finite maps

The Coq standard library provides support for finite maps in terms of an abstract interface and several possible implementations. Definitions and proofs which require a finite map are typically *parameterised* by a type-level variable which implements this interface; this allows for implementation-agnostic proofs that can still rely on finite map properties.

However, these variables must actually be defined at the *module level*. The finite map interface is in fact a *module signature*, and each implementation is a module implementing that signature. In order to parameterise by a finite map implementation, it is necessary to define a *module functor* - a module which takes one or more modules as an argument.

For example, in the code:

```
Module Type S.
End S.

Module T <: S.
End T.

Module M (s : S).
End M.
```

The module type `S` is a module signature; the module `T` implements the signature `S`; and the module `M` is a module functor, taking a module argument `s` which implements the signature of `S`. In order to use `M`, it is necessary to prove the argument `s` - for example, `M T` would be a module, created by applying the functor `M` to argument `T`.

This system is less-than-perfect. Every time the contents of a particular module functor are needed, the functor needs to be *fully-qualified*: all of its module arguments need to be provided. In practise, the user of the functor *also* wants to be implementation-agnostic, and so must introduce ts own module argument. This means that the use of finite maps in definitions and proofs is viral, propagating out from the original use site.

Reusing the above example, if we wish to define a module which uses `M`, we would have to write.

```
Module N (s : S).

Module M' := M s. (* Create M' by applying M to s *)
Import M'. (* Since M' is a module, it can be imported *)

End N.
```

The module `N` is forced itself to become a module functor if it does not want to choose a concrete value for the argument `s` of `M`.

A.2.2 The module system

The use of maps from Coq's standard library also introduced another problem - since the "modules" in the Pony module were now in fact module functors, every use of one of those modules instantiated a new copy of the module. This meant that, if the same module was used in two different locations, Coq actually treated those as two *separate* declarations of the module contents. Therefore, they would not be unifiable.

For example, in the module structure:

```
Module Type S.
End S.

Module F (s : S).
Inductive foo : Type := a.
End F.

Module G (s : S).
Export (F s).
End G.

Module H (s : S).
Export (F s).
End H.

Module I (s : S).

Module G := G s.
Module H := H s.
```

```
(* This is a type error - but the foos are the same! *)
Axiom bar : forall g : G.foo, forall h : H.foo, g = h.

End I.
```

The module `F` is a module functor, taking an argument of module type `S`. When `F s` is used in `G`, Coq considers it a new copy of `F` with argument `s` - and similarly in `H`. Therefore, in `I`, the two copies of `F` are *not* considered the same: and in particular, Coq cannot unify the declaration of `foo` in `G` with that in `H`. This occurs even though the module arguments are the same.

The effect of this limitation in the Pony model was purely organisational. Because of the use of module functors, it became necessary to have only one canonical instantiation of each module, so that no types, functions, or theorems were declared twice. This solved the problem of non-unification.

A.2.3 Coq and polymorphism

The use of the closed-world-encoding was not the first approach taken for building the model of Pony in Coq. As a dependently-typed language, Coq had the features necessary to produce a truly polymorphic typing judgement (meaning, in terms of Coq types).

However, using true polymorphism in Coq introduced several issues. These arose from limitations of the logic system on which Coq is based - and those limitations complicated the resulting proofs. Ultimately, the benefits of the polymorphic approach were outweighed by its large downsides, and it was abandoned.

Type equality

The underlying mathematical model used by the Coq proof system is called the *Calculus of Inductive Constructions*. One feature of the CIC is its compatibility with the *axiom of univalence* - that all isomorphic types are equal. This means that, in the logical system obtained by taking the CIC and assuming the axiom of univalence, it is impossible to arrive at a contradiction.

The consequence of this fact is that it is impossible to prove that isomorphic types are *not* equal in the CIC. For example, Coq has support for both the type of natural numbers `nat`, as well as the type of strings `string`. However, it is *not* possible to prove the seemingly-obvious fact that `nat <> string` - that the types `nat`, `string` are not equal. This is because both types are countably infinite, and there exists an isomorphism between them.

If it were possible to prove `nat <> string` in the CIC, it would be possible to also prove it with the addition of the axiom of univalence. But the axiom itself would allow for the proof of `nat = string` (because the types are isomorphic). Consequently, the addition of the axiom of univalence would no longer be consistent.

This limitation became relevant due to the polymorphism used in the encoding of the Pony model. In order to prove that, for example, all evaluation judgements for paths did not alter the heap, it was necessary to argue that all the other evaluation judgement forms resulted in a contradictory type equality - e.g. that Sem-VarDecl could not be used for paths, because it clearly only applies to variable declarations, which are a form of expression. In the model itself, this would lead to the contradictory hypothesis `expression = path`. But since expressions and paths are both

countable, the types were actually isomorphic, and proving this to be a contradiction was impossible.

This required adding in additional axioms to declare that the necessary types were unequal - in the above example, this resulted in an axiom that `expression <> path`. These axioms remained consistent for as long as no contradictory extension was made to the model.

Decidable type equality

A secondary limitation of the CIC is its lack of decidable type equality. The equality of an abstract collection is said to be decidable if, for any two objects in the collection, it is possible to compute whether or not they are equal. While most mathematics treats equality as always being decidable, in Coq it is *not* considered to hold in general. Types are one such case: it is not possible to decide if two Coq types are equal or not.

However, again due to the polymorphic features used by the model's encoding, it was sometimes necessary to decide on the equality of two types. For this reason, it was necessary to introduce the additional axiom of *decidable type equality*:

```
Axiom decidable_type_equality :
  forall X Y : Type, { X = Y } + { X <> Y }.
```

The axiom states that, for every choice of two types, their equality is decidable.

This axiom also had an interesting technical side-effect. Typical Coq proofs, written using the CIC, use only intuitionistic logic, and so respect the Curry-Howard isomorphism (Howard 1969).Coq proofs are themselves programs, and can as such be executed.

The axiom of decidable type equality actually states that, for every pair of types, it is possible to *compute* whether or not they are equal - and in fact, Coq treats the axiom as the function which computes that equality status. However, the axiom itself does not declare how the computation is done. As a consequence, proofs which use the axiom are *no longer computable*.