CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Singly typed actors in Agda

An approach to distributed programming with dependent types

Master's thesis in Computer Science

PIERRE KRAFT

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Master's thesis 2018

Singly typed actors in Agda

An approach to distributed programming with dependent types

PIERRE KRAFT

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Singly typed actors in Agda
An approach to distributed programming with dependent types
PIERRE KRAFT

Supervisor: Andreas Abel, Computer Science and Engineering
Examiner: Mary Sheeran, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Singly typed actors in Agda
An approach to distributed programming with dependent types
PIERRE KRAFT
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

By requiring communication to take place using explicit message passing, the actor model has been shown to be an effective tool for building distributed systems. However, communication in the actor model has traditionally been untyped, i. e. any message can be sent to an actor, even though it most probably only handles specific ones. Singly typed actors have a single static type assigned to each actor, limiting messages to those that can be handled by the actor.

We present M_{act}, a formal model of singly typed actors, implemented as a shallow embedding in Agda. The shallow embedding allows for a minimal calculus that does not sacrifice usability, since programs are expressed using the full power of Agda—a deep embedding would not have allowed for the same experimental investigation without significantly more effort. We use this advantage to demonstrate how several common communication patterns can be expressed as abstractions inside the model.

We implement and compare implementation strategies for out of order communication—the foundation of the abstractions we build. With out of order communication we are able to emulate local channels, which we use to implement synchronous calls and active objects.

Like He (2014), we find that subtyping solves the problem of wide actor types when sending messages. However, the use of a single inbox per actor remains problematic when receiving messages, especially in the context of call-response patterns. Our emulation of local channels alleviates the problem of wide actor types, but we still propose extending the model with native support for multiple inboxes per actor as a better solution.

Keywords: Distributed programming, Actors, Dependent types, Agda, Selective receive

Acknowledgements

I would like to express my gratitude to my supervisor, Andreas Abel, for his support and advice—your expertise has been very valuable and I have cherished our meetings. I want to thank Mary Sheeran and Simon Fowler for your encouragement and feedback—your words have meant a lot. My earnest thanks also go to my friends and family for their help and care during the period of writing the thesis. I would finally like to thank everyone that has shown interest in the results of my work—it has motivated me tremendously.

Pierre Kraft, Gothenburg, June 2018

Contents

1

Introduction

Distributed systems are darn difficult to build and even more difficult to build correctly. Distributed systems use components that are spread across a network, which brings several challenges: communication latency, temporary and permanent failures, limited bandwidth, and the need for explicit synchronization. Nevertheless, distributed systems are ubiquitous: network applications, telecommunication networks, and aircraft control systems are all physically distributed.

The computational models used for sequential and parallel programming can not handle the challenges of distributed systems. Researchers must thus make the choice of either adapting their models, if possible, or develop new models that are designed for distribution. In the former camp we primarily find the adaptation of various process calculi and in the latter we find variations of the actor model.

The computational model considered in this thesis is a process-based actor model. Process-based actors are defined as computations which run from start to completion, where processes send messages to one another, and where every process has an inbox which contains messages that they have received. Communication in the actor model is done solely via asynchronous message passing, reflecting the fact that sending bits over a wire is the physical communication mode in a distributed setting.

The nature of actors makes it difficult to give them a static type. Actors may fail temporarily or completely, may require that intricate protocols are followed, and the asynchronous nature of distributed systems means that no actor can guarantee the current state of another actor. This has resulted in a tradition of dynamically typed actors and runtime supervision.

An important reason to type actors is to make communication play well with statically typed sequential code. This need can largely be captured by assigning a single static type to each actor, a model we call singly typed actors. The singly typed actor model has seen several adoptions by the industry (see section 3.1), but is not well studied in a formal setting. This thesis aims to put practice into theory by mechanizing singly typed actors as a formal model in Agda.

Fowler, Lindley and Wadler (2017) develop a formal account of singly typed actors as an extension to the simply typed λ-calculus. The simply typed λ-calculus (STLC) is a tiny core calculus embodying the key concepts of functional programming. STLC's simplicity has made it the canonical choice when studying extensions to functional programming, but it is not much of a programming language. This is reflected in the extensions they make to their core languages when translating between their λ-calculus for actors and their λ-calculus for channels—which they develop in the same paper.

The extensions made by Fowler, Lindley and Wadler (2017) show that a fully featured programming language is needed to write interesting actor programs. This sparked the idea that we could implement their model as an embedded domain-specific language (EDSL). An EDSL is a domain-specific language that is defined in terms of a more powerful host language. The advantage of an EDSL is that it inherits the infrastructure and abstractions of the language it is embedded in. This makes for a powerful programming model, without the need to build a completely new programming language.

We have chosen Agda as the host language for our actor DSL. The motivation behind this decision lies in our desire to both mechanize a formal model of the singly typed actor model and to make it easy to write interesting programs in it. Agda is both a functional programming language and a theorem prover, conveniently answering both desires.

Fowler, Lindley and Wadler (2017) develop an extension to their actor λ-calculus that enables out-of-order processing of messages via a *selective receive* construct. Selective receive is a construct that sees practical use in Erlang as a means to enable synchronization, for example when implementing synchronous call-response. Our intuition says that selective receive can be used to emulate many communication patterns, such as local channels, synchronous calls, active objects, and distributed state machines. This intuition is in fact what sparked the idea of this thesis and we dedicate a significant effort to investigate this idea.

1.1 Contributions

This thesis makes the following contributions:

- It presents M_{act}, a domain-specific language for the singly typed actor model, embedded in Agda.
- It presents a mechanized formalization of the singly typed actor model, implemented in Agda.
- It compares two methods of implementing selective receive, both as an extension to the language and as a library.

- It shows how selective receive can emulate local channels, which we use to implement synchronous calls and active objects.

1.2 Structure of the thesis

The rest of the thesis is organized as follows.

We begin chapter 2 with a short introduction to Agda, which leads up to an explanation of coinduction and its use in modelling infinite processes. We follow with an introduction to monads and parameterized monads, which will serve as the base of M_{act}. We provide a short introduction to subtyping, followed by an explanation of the actor model and a recital of standard actor-related vocabulary. We end the chapter with a note on the notational conventions we use in this thesis.

In chapter 3 we review related work in regards to typing distributed systems and formal models similar to the singly typed actor model.

Chapter 4 presents the syntax, typing rules, and semantics of M_{act}. This chapter also reviews the λ-calculus that M_{act} is based on and details several design decisions that we made.

The details of M_{act} lead up to chapter 5, where we compare two methods of implementing selective receive and we show how selective receive can be used to implement important abstractions.

Our findings are discussed in chapter 6, where we also make suggestions for further improvements and future work.

Lastly, the work is concluded in chapter 7.

2

Background

This thesis mechanizes a type-theory for the actor model in the Agda language. This chapter aims to explain important concepts when working in Agda, some type theory, and the actor model. We try to assume as little knowledge as possible and explain terminology as it appears, but we do expect prior experience in functional programming and some understanding of type theory.

2.1 Dependent types and Agda

Agda is a theorem prover and a functional programming language (Bove, Dybjer and Norell 2009). At the heart of Agda is its dependent type system, which is an extension of Martin-Löf's intuitionistic type theory (Bove, Dybjer and Norell 2009; Martin-Löf and Sambin 1984). The idea of dependent types is to let types be calculated from some other values, allowing a developer to make types as precise as required. Via the Curry–Howard correspondence, a type in Agda can also be seen as a proposition in constructive logic, where propositions are proved by writing a program inhabiting the corresponding type. This correspondence is what makes Agda both a theorem prover and a functional programming language.

Agda's main types are dependent function types, inductive data types, and coinductive data types. In Agda, types can be manipulated just like any other language construct. For example, they can be stored in variables, passed to functions, or constructed by functions. The type of types is called `Set`.

Inductive data types in Agda can be seen as dependent versions of the algebraic data types you might have seen in other functional programming languages. For example, the set of Peano numbers and its "less than or equal" relation can be defined like this:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {n m : ℕ} → n ≤ m → suc n ≤ suc m
```

The definition of `ℕ` should be read as there being two ways to construct a natural number. Zero is a natural number, and if `n` is a natural number, then `suc n` (the successor of `n`) is a natural number too. Similarly, the proposition `≤` also has two constructors. The first constructor, `z≤n`, states that `zero` is less than or equal to any natural number. The second constructor, `s≤s`, states that if `n ≤ m`, then `suc n ≤ suc m`.

The type of `≤` is `ℕ → ℕ → Set`. This means that `≤` is a family of types indexed by two natural numbers. So, for each natural number `n` and `m`, `n ≤ m` is a type. The constructors are free to construct elements in an arbitrary type of the family, so many or all of these types might be uninhabited. For example, `suc zero ≤ zero` is impossible to construct.

Perhaps simpler than inductive data types are dependent function types. In Agda we write `(x : A) → B` for the type of functions taking an argument `x` of type `A` and returning a result of type `B`, where `x` may appear in `B`. A special case of when `x` appears in `B` is when `x` itself is a type. For instance, we can define a polymorphic identity function, which takes a type argument `A` and an element of `A` and just returns the element:

```
identity : (A : Set) → A → A
identity A v = v
```

Surrounding an argument with `{}` makes it implicit. Implicit arguments can be omitted, provided they can be inferred by the type checker. Using this syntax we can define a new version of the `identity` function, where you do not have to supply the type argument. The functions `f` and `g` showcase how to guide the type checker, while `h` shows that we can apply the implicit argument manually.

```
identity' : {A : Set} → A → A
identity' v = v

f : ℕ → ℕ
f = identity'
g = identity' 2
h = identity' {Bool}
```

An important data type that uses most of the presented features is propositional equality `_≡_` (for `x, y : A`). By propositional equality we mean the standard equality type as introduced by Martin-Löf—see e. g. Nordström, Petersson and

Smith (1990, Sect. 8). ≡ has one constructor which says that ≡ is the least reflexive relation (modulo definitional equality):[1]

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x

trivial : 4 ≡ 4
trivial = refl
definitional : (2 + 2) ≡ 4
definitional = refl
```

The proposition `definitional` showcases Agda's built in definitional equality, which can be seen as 'equality after inlining all definitions and with normalization'. Definitional (trivial) equality only takes us so far, e. g. it can not prove `(n : ℕ) → (n + 0) ≡ (0 + n)`, which is why propositional equality exists as well. To prove this last equation we have to define a function that builds the propositional equality proof inductively:

```
cong : ∀ {a} {A B : Set a} {m n : A} → (f : A → B) → m ≡ n → f m ≡ f n
cong f refl = refl

id-0 : (n : ℕ) → (n + 0) ≡ (0 + n)
id-0 zero = refl
id-0 (suc n) = cong suc (id-0 n)
```

2.2 Coinduction and sized types

A basic principle of Curry-Howard is that the consistency property of a logic corresponds to checking that a program is total, i. e. it is not allowed to crash or non-terminate. We can showcase this by turning off Agda's termination checker to create a function that can prove any property. Via the function `anything`, which recurses on itself indefinitely, we are able to create a term of the clearly uninhabited type ⊥

```
{-# NON_TERMINATING #-}
anything : ∀ {a} {A : Set a} → A
anything = anything

data ⊥ : Set where

can-prove-⊥ : ⊥
can-prove-⊥ = anything
```

Since Agda enforces program termination, a natural conclusion would be that writing programs that run indefinitely, e. g. processes, servers, or infinite streams,

[1] The definition of ≡ uses type levels which we explain in section 2.7. You can safely think of `Set a` as `Set`.

is completely out of the picture. Using just the concepts we have seen so far, inductive data and recursive functions, that conclusion would be correct. Even though induction can model infinite structures, such as natural numbers, recursive definitions are only well-founded if the result is built from smaller building blocks.

To model potentially infinite structures we have to turn to *coinduction*. Coinduction describes how an object may be 'observed', or 'destructed' into simpler objects, compared to induction which describes how an object may be 'built up' from simpler objects. Similarly, just as recursive functions are well-formed if the result is built from smaller objects, corecursive functions are well-formed if the result builds larger objects.

Agda supports two flavours of coinduction; we will only use the most modern flavour, which is defined using coinductive records and copatterns.[2] Let us illustrate how to program with these concepts using an example of generating an infinite stream of all natural numbers. We define `Stream` to be a coinductive record with two observations, `head` and `tail`:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

Similar to how we would define a recursive function using pattern matching, we will define the stream of natural numbers using copatterns. The copatterns `.head` and `.tail` let us define the projections from the record `nats n` as separate cases. `nats` does not reduce by itself; it only reduces if applied to one of the projections `.head` or `.tail`. In a sense, this simulates what is achieved by lazy evaluation in Haskell or Coq.

```
nats : ℕ → Stream ℕ
nats n .head = n
nats n .tail = nats (suc n)

all-nats = nats 0

third-tail : (all-nats .tail .tail .tail .head) ≡ 3
third-tail = refl
```

When reasoning about termination for coinductive data, one speaks of a definition as being productive, rather than terminating (Coquand 1993). Productivity essentially states that every finite approximation of a conceptually infinite value should be computable in a finite number of steps. The approximation of productivity developed by Coquand (1993) is called the principle of guarded recursion. The

[2]The old flavour of coinduction was not considered stable (Bove, Dybjer and Norell 2009) and could be used to prove absurdity. The modern flavour can not be used to prove absurdity (Abel and Pientka 2013) and is considered stable.

term guarded comes from the fact that self-referential calls have to be hidden under a coinductive constructor, e. g. the constructor for `Stream`. Intuitively, guarded recursion gives a guarantee that we can stop the recursion by choosing to not further evaluate the recursive constructor argument.

Copatterns are an alternative way of defining record constructors and definitions—they emulate guarded recursion. In the definition of `nats` it is the usage of copatterns, combined with the coinductive nature of `Stream`, that enables `nats` to recursively refer to itself while maintaining productivity.

Guarded recursion is a form of syntactic productivity check, meaning that if we restructure the code, e. g. by extracting an auxiliary function, the compiler might not be able to see that the definition is productive. In order to restore function compositionality and abstraction, Abel and Pientka (2013) adapt type-based termination to prove productivity of definitions by copatterns. The idea of type-based termination is to annotate types with a size index, indicating recursion depth. For recursive calls it is checked that the sizes decrease, which by well-foundedness entails termination (Abel 2010).

In this paper we will use Agda's built-in type `Size` which denotes a set of ordinals. Size `∞` stands for the highest ordinal needed for recursion in Agda. From `Size` we can access the size `∞`, the successor operator `↑`, and the type of smaller sizes `Size<`.

Abel and Pientka (2013) observe that when showing that a stream can safely handle a observations, it is safe to assume that we can make b observations for any $b < a$. E. g. if we are defining the third element of a stream, it is safe to observe the first or second element of the stream. For our new definition of `Stream` below, the `Size` can be understood as how many times the observation `tail` is at least defined for. More precisely, observing a `Stream i` will result in a `Stream j` of strictly smaller observation depth `i < j`.

```
record Stream (i : Size) (A : Set) : Set where
  coinductive
  field
    head : A
    tail : (j : Size< i) → Stream j A
```

Let us look at how type-based termination helps us define the stream of Fibonacci numbers. Intuitively, what we want to define is a stream starting with the elements 0 and 1, followed by the sum of the two previous stream elements. Fibonacci can be implemented according to this specification via `zipWith`, which pointwise applies a binary function to the elements of two streams. The `Size` parameter in `zipWith` guarantees that the function will produce an output stream of depth i if called with input streams of at least that depth.

```
zipWith : (i : Size) {A B C : Set} → (A → B → C) →
          Stream i A → Stream i B → Stream i C
(zipWith i f xs ys) .head = f (xs .head) (ys .head)
(zipWith i f xs ys) .tail j = zipWith j f (xs .tail j) (ys .tail j)

fib : (i : Size) → Stream i ℕ
fib i .head = 0
fib i .tail j .head = 1
fib i .tail j .tail k = zipWith k _+_ (fib k) (fib j .tail k)
```

2.3 Monads

A monad, or computation builder, is a design pattern used in functional programming to chain things together. Moggi (1991) was the first to apply monads to computer science, but in the context of functional programming it was Wadler (1992) that lead the way.

In essence, a monad lets you write execution steps and link them together using a function called 'bind', written >>=. The bind function's job is to take the output from the previous step and feed it into the next step. The other function that every monad has is called 'return'. The return function's job is to take a plain value and put it in the monad.

The simplest monad is the identity monad[3], which simply applies the bound function to its input without modification:

```
data MonadID (B : Set) : Set₁ where
  return : B → MonadID B
  _>>=_ : ∀ {A} → MonadID A →
            (A → MonadID B) →
            MonadID B
```

It is not an explicit or required part of the monad pattern, but it's often useful to provide a function that translates the value wrapped in the monad into something else. For the identity monad, we can define this translation function as simply applying the value to the bound function:

```
run-monadID : ∀ {A} → MonadID A → A
run-monadID (return x) = x
run-monadID (m >>= f) = run-monadID (f (run-monadID m))
```

[3] We have defined the identity monad as an explicit data type, which is not the usual choice in Agda. This is a choice that we have made to make the transition to the next section smoother. A monad in Agda should preferably implement the `RawIMonad` interface as well, not just create two free functions `return` and `_>>=_`.

If we change the structure of our monad pattern we can define many different kinds of computation chains. Common modifications of the pattern are to add more constructors and to perform some more advanced plumbing code in the bind function. E. g. in the Maybe monad, we add the constructor `nothing` and make `bind` only apply a value to f if there exists a value to apply:

```
data Maybe (A : Set) : Set₁ where
  nothing : Maybe A
  just : A → Maybe A

return : ∀ {A} → A → Maybe A
return = just

_>>=_ : ∀ {A B} → Maybe A → (A → Maybe B) → Maybe B
nothing >>= f = nothing
just x >>= f = f x
```

The identity monad and Maybe monad can be seen in the context of two examples in appendix A.

All instances of the monad pattern should obey the three monad laws (Wadler 1992). These laws are:

```
left-identity = ∀ {A B} → (x : A) → (f : A → Monad B) →
                  (return x >>= f) ≡ f x
right-identity = ∀ {A} → (m : Monad A) →
                   (m >>= return) ≡ m
associativity = ∀ {A B C} → (m : Monad A) →
                    (f : A → Monad B) → (g : B → Monad C) →
                ((m >>= f) >>= g) ≡ (m >>= (λ x → f x >>= g))
```

2.4 Parameterized monads

The monad pattern gives an elegant way to sequence computations and combine compatible sub-computations. In the pattern that we have presented, sub-computations are compatible if the first computations output is the input to the second computation. Conditions on the internal state (or on the implicit external state) are not captured in the type system, so sub-computations with incompatible conditions can still be composed and compiled.

Parameterized monads is a generalization of the monad pattern that lets us verify that all conditions are compatible, by making the conditions explicit in the type. Compared to externally verifying the monadic computation, parameterized monads can be made correct by construction. Parameterized monads should preferably have

been called (type) indexed monads, but that name is already used for a slightly different concept[4].

The idea of a parameterized monad is to encode the invariants (preconditions and postconditions) of the monad in its type. An easy way to do so is the parameterized monad à la Atkey (2009). This pattern (which can be implemented without dependent types) adds two indices to the monad type—one for the precondition, and one for the postcondition:

```
data MonadParam' (B : Set) : {p q : Set} → p → q → Set₁ where
  return : ∀ {p} → {pre : p} → B → MonadParam' B pre pre
  _>>=_ : ∀ {A X Y Z} → {pre : X} → {mid : Y} → {post : Z} →
          MonadParam' A pre mid →
          (A → MonadParam' B mid post) →
          MonadParam' B pre post
```

We are guided by intuition and the monad laws when deciding how return and bind should affect the invariants in `MonadParam'`. Bind connects the preconditions and postconditions of two sub-computations via a midcondition, similar to composition in Hoare logic (Hoare 1969). The result of binding two sub-computations is naturally a joined computation with the precondition from the first and postcondition from the second, that is:

$$\frac{\{P\}\ S\ \{Q\} \quad , \quad \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Return on the other hand does not have any clear precondition, but we can see that in order for return to be an identity element, the precondition and postcondition have to be the same. Return can thus be likened to the empty statement rule of Hoare logic, that is:

$$\frac{}{\{P\}\ \mathit{skip}\ \{P\}}$$

The `MonadParam'` pattern can be used to model invariants that do not depend on what happens during run-time. For example, we can create a monad carrying implicit state for which the type of the state can change over time. Another typical example would be a monad for file actions that can only write to a file that has been opened and not yet closed. If there is a need to model invariants that depend on run-time interactions, then we need to generalize the monad pattern a bit more.

In `MonadParam` below, we take advantage of the dependent type system by changing the postcondition from a static value to a function on the answer of the monad. This is an idea used by Nanevski et al. (2008) to model separation logic, and by Brady (2015) to model effectful computations in Idris. What dependent types

[4]The slightly different concept is the indexed monads captured by Agda's `RawIMonad`, which is essentially the same as the `MonadParam'` we present below.

brings is the ability for different branches (fail/success) to have different types. This is possible since Agda allow types to depend on the value of a term.

```
data MonadParam (B : Set) : {p q : Set} → p → (B → q) → Set₁ where
  return : {S : Set} → {post : B → S} →
           (val : B) →
           MonadParam B (post val) post
  _>>=_ : ∀ {A X Y Z} → {pre : X} →
           {mid : A → Y} →
           {post : B → Z} →
           MonadParam A pre mid →
           ((x : A) → MonadParam B (mid x) post) →
           MonadParam B pre post
```

The relation between the precondition and postcondition for return in `MonadParam` might seem surprising. The precondition is (*post val*), so, somehow, the precondition depends on the postcondition. The key thing to realize is that we must have that *pre* = (*post val*) in order for return to be an identity element. Rather than putting a constraint on the postcondition, return makes the relation hold trivially by setting *pre* = *post val*. This style of reasoning is similar to how one can reason about Hoare logic using Dijkstra's (1975) weakest preconditions and is inspired by the STsep monad of Nanevski et al. (2008).

We can use the same style of reasoning for the invariants of bind in `MonadParam`. The midcondition should make the postcondition of the first sub-computation and the precondition of the second sub-computation meet, which we trivially get by setting $mid_{post}\ x = mid_{pre}$.

The parameterized monad with dependent parameters is a powerful pattern that captures run-time invariants at compile time. Below we showcase a simplified monad for file handling that captures the notion that opening a file might fail. The invariance tracking and dependent types makes sure that every code branch performs the necessary run-time checks to ensure that the invariants hold. The relationship between an operation's invariant and return value might seem arbitrary, but they are all constructed so that the invariants can be completely inferred from the return value during run-time.

```
data FileState : Set where
  Open : String → FileState
  Closed : FileState
```

```
data FileMonad : (A : Set) → FileState → (A → FileState) → Set₁ where
  return : ∀ {A post} → (val : A) → FileMonad A (post val) post
  _>>=_ : ∀ {A B pre mid post} →
            FileMonad A pre mid →
            ((x : A) → FileMonad B (mid x) post) →
            FileMonad B pre post
  OpenFile : String → FileMonad FileState Closed (λ x → x)
  CloseFile : ∀ { s } → FileMonad ⊤ (Open s) (λ _ → Closed)
  WriteFile : ∀ { s } → String → FileMonad ⊤ (Open s) (λ _ → Open s)

writeHello : FileMonad ⊤ Closed (λ _ → Closed)
writeHello = OpenFile "world.txt" >>= λ {
  Closed → return _ ;
  (Open _) → WriteFile "hello" >>= λ _ → CloseFile
  }
```

2.5 Subtyping

Subtyping is a relation between two types in a type system. If S is a subtype of T, written $S <: T$, then any term of type S can be safely used in a context where a term of type T is expected. If we view types as denoting sets of values, then we can view subtyping as a relation between types induced by the subset relation between value sets. E. g. if $S = \{a, b\}$ and $T = \{a, b, c\}$, then $S <: T$ since $S \subseteq T$.

The semantics of subtyping depends on what 'safely used in a context where' means in a given language, but Pierce (2002) stipulates that subtyping should be reflexive and transitive:

$$S <: S \qquad \frac{S <: T \quad T <: U}{S <: U}$$

The subtyping relation does not need to be anti-symmetric, even though it often is. Consider, for example, permutation subtyping; in a type system with records, permutation subtyping is a relation between records with the same fields, but in a different order. Pierce (2002) notes that such relation is not anti-symmetric, e. g.

$$\begin{gathered} A : \{x : Bool, y : Int\} \\ B : \{y : Int, x : Bool\} \\ A <: B \\ B <: A \\ A \neq B \end{gathered}$$

The subtyping relation must also be defined for function types, that is, we should specify under what circumstances it is safe to use a function of one type in a context where a different function type is expected (Pierce 2002).

$$\frac{S <: A \quad B <: T}{A \to B <: S \to T}$$

Here, the subtype relation is reversed (contravariant) for the function arguments, while it runs in the same direction (covariant) for the result type. Intuitively, it is safe to allow a function of type $A \to B$ to be used in context of another function $S \to T$, as long as none of the arguments that may be passed in this context will surprise it ($S <: A$) and none of the results that it returns will surprise the context ($B <: T$) (Pierce 2002).

2.6 The actor model

The actor model (Hewitt, Bishop and Steiger 1973) is a concurrency model based on entities communicating via message passing. These entities, called actors, can send messages, make local decisions, spawn new actors, and determine how to respond to the next message they process. Communication in the actor model is one-way, asynchronous, and the arrival order of messages is both arbitrary and entirely unknown. These properties are required for the model to be realistically implementable in a distributed environment (Agha 1990).

A big difference between the actor model and the more heavily studied process algebra school is the choice of communication medium. CSP (Hoare 1978) and the π-calculus (Milner, Parrow and Walker 1992) are based on message passing over channels, in contrast to actors which communicate by sending messages to the addresses of other actors. The difference might seem small, but we argue that implementing distributed models based on channels pose difficult engineering problems.

The main obstacle in implementing channels in distributed programming is that a channel exists as an abstract transferable and shareable entity, and thus does not have a physical location. In order to send a message on a channel that can have multiple readers one has to solve a distributed consensus, since the readers on the channel have to agree in order for one reader to take the message and prevent the others from getting the same message (Vyšniauskas 2015). Vyšniauskas (2015) instead argues for a condition they call *full ownership*, where channels are always owned (only readable) by a particular process. This channel ownership model has striking similarities with the addresses of actors, with the difference being that their model allows a single actor to have multiple addresses.

Several variations of the actor model have been created over the years, each employing its own terms to describe their concepts. Thankfully, De Koster, Van

Cutsem and De Meuter (2016) have made an overview of the most prominent model variations: Classic Actors, Active Objects, Processes and Communicating Event-Loop. The variation considered in this thesis is actors based on processes, but we show in section 5.8 how our model can emulate active objects.

De Koster, Van Cutsem and De Meuter (2016) also provide a unified vocabulary which we will reiterate here and use throughout this paper.

message A message is the unit of communication between different actors. A message is a combination of an identifier that defines the type of message and a payload that contains additional information sent with that message. If one actor sends a message to another actor, that message is stored in the latter actor's inbox, independent of the recipient actor's current processing state.

inbox The inbox of an actor stores an ordered set of messages received by that actor. While the inbox defines the order in which the messages were received, that does not necessarily imply that those messages are processed by that actor in that order.

interface At any given point in time, an actor's interface defines the list and types of messages it understands. An actor can only process incoming messages that fit this interface. For some actor systems this interface is fixed while other actor systems allow an actor to change its interface, thus allowing it to process different types of messages at different points in time.

state At any given point in time, we define an actor's state as all the state that is synchronously accessible by that actor (i. e. state that can be read or written without blocking its thread of control). Depending on the implementation, that state can be mutable or immutable, and isolated or shared between actors.

actor An actor can be defined as a four-tuple: an execution context, an inbox, an interface and its state. An actor perpetually takes messages from its inbox and processes them in a new execution context with respect to that actor's interface and state. This continues until the inbox is empty after which the actor goes to an idle state until a new message arrives in its inbox.

2.7 Our conventions

The remainder of this thesis uses notation that might be unfamiliar to some. These notations are syntax for singleton lists, lifting of types, and do-notation.

Singleton lists

When building lists we often come to a point where we want to add a single element to the empty list, i. e. a singleton list. Agda does not come with syntax sugar for building list literals, but we can create a special function for adding an element to the empty list:

```
[ a ] = a :: []
```

We use three different list-like data structures in this thesis, meaning that we need three different singleton functions. We have marked the functions with different superscripts to indicate what type the list-like data structure has. For lists we use ^{l}, for subsets (membership relations) we use ^{m}, and for lists where all elements satisfy a given property (`All`) we use ^{a}.

Lifting of types

Not every Agda type is a `Set`. For example, we have `Bool : Set` and `Nat : Set`, but not `Set : Set`. However, we often have to work with these 'types of types'. Agda lets us do that via the type that contains types, $\texttt{Set}_1$, where `Set` is an element: `Set : Set₁`. To work with $\texttt{Set}_1$ we have that `Set₁ : Set₂`, and this tower of types continues indefinitely.

When we have an element of type `Set` but want an element of type $\texttt{Set}_1$ we can `Lift` the element to the right type level. The record `Lift` takes an element in some set and promotes it to the least upper bound of that set and another. For example, `Bool` can be lifted to $\texttt{Set}_1$ like so:

```
record Lift {a ℓ} (A : Set a) : Set (a ⊔ ℓ) where
  constructor lift
  field lower : A


Bool₁ = Lift {lzero} {lsuc lzero} Bool
```

We will use the notation that a type has been lifted to a higher level is marked with a subscript number, imitating the notation for `Set`, $\texttt{Set}_1$, … . For example `Bool` lifted to $\texttt{Set}_1$ will be called $\texttt{Bool}_1$

do-notation

Agda provides a syntactic sugar for using monads, called *do-notation*. Monads are useful for imitating imperative code, and do-notation makes it look like it actually is imperative. A do-block consists of the keyword `do`, followed by a sequence of do-statements, which gets translated into calls to `>>=` and `>>`. The syntactic sugars and what they translate to is provided in table 2.1

Statement	Sugar	Desugars to
Simple bind	`do x ← m` `  m'`	`m >>= λ x →` `m'`

Pattern bind	`do p ← m`
	`  where` $p_i \to m_i$
	`  m'`
		`m >>= λ where`
		`  p → m'`
		`  `$p_i \to m_i$

Non-binding statement	`do m` `  m'`	`m >>` `m'`
Let	`do let ds` `  m'`	`let ds in` `m'`

Table 2.1: Rules for desugaring do-notation

3

Related work

In this chapter, we review work relevant for adding types to distributed systems, formally verified actor models, and ongoing industry developments. Much of the terminology in distributed systems is overloaded, e. g. channels and actors are often conflated (Fowler, Lindley and Wadler 2017), so organizing our thoughts before getting into the meat of the thesis is important.

3.1 Singly typed actors

The main inspiration for the topic of this thesis is the recent developments, primarily in industry, of actor models that have static non-behavioural type systems. These are models where processes interact directly with their inboxes—in contrast to active object based actor systems where message passing and reading is hidden—and where inboxes have a type that does not evolve in relation to a protocol. The capability to read from an inbox must furthermore be tied to a single actor in order to enable easy distribution; this rules out Srinivasan and Mycroft's (2008) otherwise interesting Kilim framework.

The singly typed actor model can not automatically prove the absence of deadlocks or that protocols are followed, but it has gained many followers by being relatively simple. The common way to implement singly typed actors is as a library, which we find many examples of, such as He's (2014) TAkka, Akio[1], Actix[2], Charousset, Hiesgen and Schmidt's (2016) C++ Actor Framework, F#'s MailboxProcessor[3], Nact[4], Theron[5], and typed-actors[6]. Implementations where the singly typed actor model is added as constructs in the base of a programming language are fewer, with Alpaca[7] and Fowler, Lindley and Wadler's (2017) λ_{act} as the sole implementations we know of.

[1] https://github.com/kphelps/akio
[2] https://github.com/actix/actix
[3] https://en.wikibooks.org/wiki/F_Sharp_Programming/MailboxProcessor
[4] https://nact.io/
[5] http://www.theron-library.com/
[6] https://github.com/knutwalker/typed-actors
[7] https://github.com/alpaca-lang/alpaca

3.2 Active object based actor systems

The active object model is an actor model where explicit message passing is replaced with asynchronous method invocation. The model has its roots in Yonezawa, Briot and Shibayama's (1986) object-oriented programming language ABCL/1. Active objects have a fixed interface of messages that are understood (De Koster, Van Cutsem and De Meuter 2016), similar to the static interfaces of many conventional object-oriented programming languages.

Whereas messages sent to an actor are generally selected by pattern matching over the message queue, asynchronous method invocations restrict the communication between active objects to messages that trigger method activations (de Boer et al. 2017). This restriction adds important structure to the actor model, making it easy to add static types.

With method invocations being asynchronous we mean that invoking a method immediately gives back control to the caller. This means that method invocation has to return a future value, rather than a value that is available now. The future starts unresolved (without a value) but it will be resolved (have a value) when the asynchronous method completes. If the caller attempts to retrieve the content of a future that is unresolved, the caller suspends until the asynchronous method completes and the future is resolved. In essence, futures are a specialization of local channels, where a future corresponds to a channel that will only be used to deliver a single message.

Active object models are typically language based, since method invocation is usually intertwined with the core semantics of a language. We refer to de Boer et al. (2017) for a detailed review of the differences between the different languages, but notable examples include Rebeca (Sirjani et al. 2004), ABS (Johnsen, Hähnle et al. 2010), Creol (Johnsen, Owe and Arnestad 2003), and Pony (Clebsch et al. 2015). It is interesting to note that most active object languages have formal semantics, whereas the singly typed actor models generally do not.

The method based communication model is both the key advantage and key disadvantage of the active object model. The added structure of methods corresponds to what is probably the most common patterns used in actor programming, but inherently adds coupling between messaging and execution. A potential problem with coupling the processing of a type of message to a particular body of code is that the behaviour of an actor is often dynamic, and that the order that messages are sent does not always correspond to the order in which they should be processed.

Examples of dynamic behaviour can be seen in most protocols. Even actors as simple as a remote light switch inhibit dynamic behaviour, i. e. switching on a light that is already on should not turn on the electricity again. Active object based systems assign a static behaviour for message types and thus dynamic behaviour has to be implemented by manual branching on internal state. We believe that this

style of branching tends to be more difficult to understand, compared to the explicit message reception of other actor models.

Deeply related to this dynamic behaviour is that message processing order should be decidable by the actor. As an actor goes through different states, only some types of messages or messages from specific actors makes sense to process at that moment. We will see in chapter 5 that message reordering can simplify control flow, and remove most need for branching, by letting you delay messages until they become relevant.

We show in section 5.8 that the singly typed actor model can emulate active objects by assigning a single method to each message type and by using a single receive-send-loop per actor. If similar emulation can be used in the other direction is not clear, indicating that singly typed actors might be more expressive, or at least less strict, than active objects.

3.3 Session types

Session types (Honda 1993; Honda, Vasconcelos and Kubo 1998; Takeuchi, Honda and Kubo 1994) are a group of type systems based on the observation that 'a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or *session*' (Honda, Yoshida and Carbone 2008).

Session types are generally studied in specialized process calculi resembling the π-calculus; Dardha, Giachino and Sangiorgi (2017) show how session types can be interpreted as standard π types—more precisely, into linear types and variant types. Linear π types (Kobayashi, Pierce and Turner 1999) force a channel to be used exactly once, making for channels that are easy to distribute. Unrestricted channels, i. e. channels that are transferable between multiple readers, can also be used, but they require some means of delegating channels. Hu, Yoshida and Honda (2008) propose several methods for implementing channel delegation, but the concept ultimately requires building an implicit distributed algorithm not directly apparent in the protocol.

Session types can alternatively be added to the actor model. Mostrous and Vasconcelos (2011) enrich a communication-centered fragment of Erlang with sessions and session types. They link messages to sessions via unique references and Erlang's selective receive statement is used to receive messages from a specific session. We use this technique in section 5.5 to emulate local channels.

The Scribble language is a platform-independent description language for the specification of asynchronous, multiparty message passing protocols (Yoshida et al. 2013). The study of *multiparty* session types (Honda, Yoshida and Carbone 2008) underpins the design of Scribble, but Scribble manages to avoid the distribution issues of π-calculus by not supporting delegation.

Scribble does not prescribe a certain communication medium, and instead just states that communication somehow takes place between the roles of a session, where a role is an abstract description of a participant's behaviour within a session. This freedom has allowed for the application of multiparty session types to the actor model. Research on applying Scribble to actors has so far focused on dynamic monitoring of session types, e. g. Neykova and Yoshida (2017) add runtime protocol verification to the Python framework Celery, and Fowler (2016) continues this research with a tool for monitoring communication in Erlang applications. How, and to what extent, static verification of session types can be applied to the actor model is still an open question.

3.4 Similar formal models

Proofs that can be verified by a machine have to be written in a formal language that the machine can understand. The process of writing these proofs is difficult, because one usually has to add quite a lot of detail to a mathematical proof on paper to make the proof assistant accept it (Geuvers 2009). Even though the proofs are different, the techniques used when formalizing similar models should provide transferable insight. The models we consider similar can be put in two groups: process calculi and verified distributed models.

3.4.1 Process calculi

The family of process calculi is large and diverse, but shares a basic orientation of focusing on interaction via communication over channels, on using a small set of primitive operators, and on deriving useful algebraic laws for manipulating expressions written using these operators (Pierce 1997). As noted by Vyšniauskas (2015), the majority of process calculi build upon synchronous communication, which is fitting in a concurrent setting, but complicates distribution.

There exists many formal encodings of process calculi, both in Agda and in other proof assistants. Igried and Setzer's (2016) model of the CSP calculus in Agda uses several relevant techniques, such as a coinductive representation of processes and the use of monadic composition to integrate Agda functions in the model. Maksimovic and Schmitt (2015) formalizes higher-order process calculus in Coq and show that their calculus is Turing-complete. Perera and Cheney (2016) encode π-calculus in Agda and prove properties about the traces of concurrent transitions. Bengtson and Parrow (2009) present a formalization of π-calculus that uses nominal logic to reduce the need of proofs about bound names.

3.4.2 Formally verified distributed systems

Formally verified distributed systems is the application of formal proofs to the verification of distributed protocols. A program is written as normal, using a core

model of distribution, but additionally has to be proven correct and in accordance to the given protocol. Verified distributed systems provide strong static guarantees which are unparalleled by other techniques so far.

The disadvantages of using formal proofs as the verification technique are the same as the drawbacks of any formal method: accuracy depends on the accuracy of the model, proof writing is very time consuming, and developing specifications requires a familiarity with discrete mathematics and logic. To limit these disadvantages, work on formally verifying distributed systems concentrates on finding models which have a small core, reasonable performance, and a tolerable proof burden.

There are several different frameworks for formally verifying distributed systems. Each of them uses widely different distributed models. IronFleet (Hawblitzel et al. 2017), for example, proposes a methodology that relies on state machine refinement and Hoare-logic verification. It aims at proving the safety and liveness of a MultiPaxos server library, which the authors use to build both a replicated state machine library and a sharded key-value store.

Another framework is Verdi (Wilcox et al. 2015). In this framework, the developer writes and proves their system correct in a simplified environment (e. g. a system with a perfectly reliable network). The developer then selects the target network semantics that reflects their environment's fault model and applies a verified system transformer to transform their implementation into one that is correct in that fault model.

Some verification frameworks use actors as their model of distribution. Musser and Varela (2013), Yasutake and Watanabe (2015), and Summers and Müller (2016) all propose formalized calculi for the actor model, which allows a user of these models to verify desired properties. As with the other verified distributed systems it is up to the developer to devise their own strategy for making verification easy. An in this model does not have a static type; instead, formal proofs are used to guarantee correctness.

4

$\mathbf{M_{act}}$

In this chapter we introduce M_{act}, a language for the core of actor-based concurrency. Depending on your point of view, M_{act} can be seen as a calculus that embeds the meta-language Agda for convenience, or as a monadic domain-specific language (DSL) in Agda. We will use both notions throughout, seeing M_{act} primarily as a calculus when describing its syntax and semantics, while viewing it as a DSL when writing examples and abstractions.

4.1 Syntax

We start the description of M_{act} by looking at the syntax and typing rules of the calculus it extends, Fowler, Lindley and Wadler's (2017) λ_{act}.

By the taxonomy of De Koster, Van Cutsem and De Meuter (2016), λ_{act} should be classified as process-based, where actors are modelled as named processes associated with an inbox. Following Erlang, actors are provided a `receive` operation that optimistically retrieves a message from its inbox. The method `receive` takes no arguments, so to give the operation a type, every actor is tagged with the type of messages it can `receive`. This tag is a simple type-and-effect system à la D. K. Gifford and Lucassen (1986).

Figure 4.1 shows the syntax and typing rules for λ_{act}. Letter α ranges over variables x and *references* a. `ActorRef`(A) is the type of references to actors that accept messages of type A. As for communication and concurrency primitives, `spawn` M spawns a new actor to evaluate a computation M; `send` V W sends a value V to an actor referred to by reference W; `receive` receives a value from the actor's inbox, waiting indefinitely if the inbox is empty; and `self` returns an actor's own actor reference.

Function arrows $A \longrightarrow^C B$ are annotated with a type C which denotes the type of the inbox of the actor evaluating the term. This is captured in the typing rules, where there are two typing judgements: the standard judgement on values $\Gamma \vdash V : A$, and a judgement Γ | B M : A which states that a term M has type A under typing context Γ and can receive values of type B.

Syntax

$$
\begin{array}{lrl}
\text{Types} & A, B, C & ::= 1 \mid A \to^{C} B \mid \texttt{ActorRef}(A) \\
\text{Variables and names} & \alpha & ::= x \mid a \\
\text{Values} & V, W & ::= \alpha \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= V\ W \\
 & & \mid\ \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N \mid \texttt{return}\, V \\
 & & \mid\ \texttt{spawn}\, M \mid \texttt{send}\, V\ W \mid \texttt{receive} \mid \texttt{self}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash a : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^{C} B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1}$$

Computation typing rules $\boxed{\Gamma \mid B \vdash M : A}$

$$\textsc{App}\ \frac{\Gamma \vdash V : A \to^{C} B \quad \Gamma \vdash W : A}{\Gamma \mid C \vdash V\ W : B} \qquad \textsc{EffLeft}\ \frac{\Gamma \mid C \vdash M : A \quad \Gamma, x : A \mid C \vdash N : B}{\Gamma \mid C \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : B} \qquad \textsc{EffReturn}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \vdash \texttt{return}\, V : A}$$

$$\textsc{Send}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \texttt{ActorRef}(A)}{\Gamma \mid C \vdash \texttt{send}\, V\ W : 1} \qquad \textsc{Recv}\ \frac{}{\Gamma \mid A \vdash \texttt{receive} : A}$$

$$\textsc{Spawn}\ \frac{\Gamma \mid A \vdash M : 1}{\Gamma \mid C \vdash \texttt{spawn}\, M : \texttt{ActorRef}(A)} \qquad \textsc{Self}\ \frac{}{\Gamma \mid A \vdash \texttt{self} : \texttt{ActorRef}(A)}$$

Figure 4.1: Syntax and typing rules for λ_{act}

$$
\begin{array}{lrl}
\text{Agda term} & v, x & \\
\text{Reference variable} & \alpha & ::= a \\
\text{Message} & msg & \\
\text{Subset} & \varphi & \\
\text{Computations} & M & ::= \texttt{Return}\, v \mid M \texttt{>>=} f \mid \texttt{Spawn}\, M \mid \texttt{Send}\, \alpha\ msg \\
 & & \mid\ \texttt{Receive} \mid \texttt{Self} \mid \texttt{Strengthen}\, \varphi \\
\text{Continuation} & f & ::= \lambda x.M \mid \ldots
\end{array}
$$

Figure 4.2: Syntax of M_{act}

The λ_{act} calculus is based on fine-grain call-by-value (Levy, Power and Thielecke 2003): terms are partitioned into values and computations. Key to this formulation are two constructs: `return` V represents a computation that has completed, whereas `let` $x \Leftarrow M$ `in` N evaluates M to `return` V, effectively substituting V for x in N.

Levy, Power and Thielecke's (2003) call-by-value calculus is based on Moggi's (1991) monadic metalanguage. Turning λ_{act} into a monadic DSL can thus be done through a fairly simple conversion: `let` $x \Leftarrow M$ `in` N corresponds to M `>>=` $\lambda x.N$, `return` corresponds to the monad's `return`, and the communication primitives are added as operations in the monad.

The syntax after converting λ_{act} into M_{act} is seen in figure 4.2. We have added an operation that is not in λ_{act}: `Strengthen` φ; it is used to rearrange the actor's reference variable context (see section 4.3.3). Combined with `>>=`, `Strengthen` φ can be used to strengthen the precondition of a computation, where the precondition is a variable typing context as described in section 4.2. This is an operation that is quite administrative and it is only needed because we track variables in the type system.

4.2 A monad for actors

The goal when translating λ_{act} to an Agda embedding was to make the actors correct by construction, and to make it possible to type-check each actor separately. The pattern we found suitable is a monad parameterized by the type of the actor's inbox (`InboxShape`) and indexed by a reference variable context. Following Fowler, Lindley and Wadler (2017), the `InboxShape` of an actor is constant over its lifetime and should be likened with the type-and-effect system of λ_{act}. Each actor has their own reference variable context and the way it changes over time is kept track of in the style of the parameterized monad pattern, capturing preconditions and postconditions in the type.

Actors are potentially infinite processes, making it suitable to model M_{act} using coinduction. Following the lead of Abel and Chapman (2014), we represent the monad `ActorM` as a mutual definition of an inductive data type and a coinductive record. The record `∞ActorM` is a coalgebra that one interacts with by using its single observation (copattern) `force`. `force` gives us an element of the `ActorM` datatype on which we can pattern match to see which computation to perform next.

Both `ActorM` and `∞ActorM` are indexed by a size i. The size is a lower bound on the number of times we can iteratively `force` the computation, but should primarily be seen as a means to establish productivity of recursive definitions. When we actually simulate running actors, we only care for `ActorM ∞ A` of infinite depth.

The definition of `ActorM` is presented in figure 4.3. The next few sections will break this definition down and explain the concepts in detail. In section 4.3 we explain the typing rules of inboxes, references, and variables. Section 4.4 explains

```
data ActorM (i : Size) (IS : InboxShape) : (A : Set₁) →
             (pre : TypingContext) →
             (post : A → TypingContext) →
             Set₂

record ∞ActorM (i : Size) (IS : InboxShape) (A : Set₁)
               (pre : TypingContext)
               (post : A → TypingContext) :
               Set₂ where
  coinductive
  constructor delay_
  field force : ∀ {j : Size< i} → ActorM j IS A pre post

data ActorM (i : Size) (IS : InboxShape) where

  Return : ∀ {A post} →
    (val : A) →
    ActorM i IS A (post val) post

  _∞>>=_ : ∀ {A B pre mid post} →
    (m : ∞ActorM i IS A pre mid) →
    (f : (x : A) → ∞ActorM i IS B (mid x) (post)) →
    ActorM i IS B pre post

  Spawn : {NewIS : InboxShape} → {A : Set₁} → ∀ {pre postN} →
    ActorM i NewIS A [] postN →
    ActorM i IS T₁ pre λ _ → NewIS :: pre

  Send : ∀ {pre} → {ToIS : InboxShape} →
    (canSendTo : pre ⊢ ToIS) →
    (msg : SendMessage ToIS pre) →
    ActorM i IS T₁ pre (λ _ → pre)

  Receive : ∀ {pre} →
    ActorM i IS (Message IS) pre (add-references pre)

  Self : ∀ {pre} →
    ActorM i IS T₁ pre (λ _ → IS :: pre)

  Strengthen   : ∀ {ys xs} →
    (inc : ys ⊆ xs) →
    ActorM i IS T₁ xs (λ _ → ys)
```

Figure 4.3: Definition of `ActorM`

our representation of messages and we end the break down of `ActorM` by looking at its definition in detail in section 4.5.

4.3 Typing rules

Type systems can serve many different purposes, provide different guarantees, and have varying degrees of strictness. In M_{act}, types are used to ensure that all messages contained in an actor's inbox are well-typed with respect to the inbox's type, i. e. that an actor will only receive messages according to its interface.

We embed M_{act} into Agda by defining a data type of computations and an interpretation function. Decisions and control flow in M_{act} are performed by Agda functions, so the embedding leans towards the shallow side. To make M_{act} a deep embedding one would have to replace the use of Agda functions with constructs that can emulate their power. Deep embeddings typically lead to more code; however, they have the advantage that programs written in the embedding are represented as an AST and can therefore be compiled to other targets (Svenningsson and Axelsson 2012). Our focus is on creating a small and powerful calculus, making a shallow embedding suitable. Future work, such as support for serialization, might have to make other choices.

Being an embedding in Agda, M_{act} adheres to the same typing rules as any Agda program. The program sections without side-effects thus support all of the concepts we are used to in Agda, like dependent types, functions, and values in any universe level. Unfortunately, in the sections that do communicate, we cannot be as permissive.

4.3.1 Typing inboxes

The typing rules for communication in M_{act} are based on the idea that a very important property of a message is that it can be understood by the receiver. The type system will thus be used to limit what type a message sent to an inbox can have. By limiting the types of messages sent to an inbox, we can make sure that every message read from the inbox has a valid type. In terms of De Koster, Van Cutsem and De Meuter (2016), this is the interface of the actor.

```
mutual
  data MessageField : Set₁ where
    ValueType : Set → MessageField
    ReferenceType : InboxShape → MessageField

  MessageType = List MessageField

  InboxShape = List MessageType
```

We define the type of an actor's inbox as an algebraic data-type with sums (`InboxShape`) of products (`MessageType`). The `InboxShape` is a variant type that details all the messages that can be received from an inbox. Every message sent to an actor will have exactly one of the types that is listed, which we communicate as a tag attached to the message (see section 4.4). We can think of the `InboxShape` as a set of types, and every message coming paired with a proof that the message is a type from that set. To know what type a message has you have to inspect the proof, and the fields of the message will become accessible.

`MessageType` also uses a list as its underlying data structure, but for a rather different purpose. `MessageType` is the type of a single message, and every element in the list represents a single field in a record. `MessageType` should thus be seen as a product type, similar to Haskell's tuples.

The fields in `MessageType` are combinations of value types and reference types. A value type is any type from Agda's lowest set universe. Typical examples are `Bool`, `ℕ`, `String` `List` `ℕ`, `Bool` `×` `ℕ`, and many user-defined types that are non-polymorphic. We limit the types to the lowest set universe as a sort of approximation of serializable values. It would be possible to further constrain the types to only those that are serializable, but due to its insignificance to the calculus we have opted not to. A future improvement would be to ensure that all communication can be serialized, which most importantly involves developing a serialization solution for `Spawn`.

A reference provides the capability to send messages to an actor's inbox, where receiving a message containing a reference is one of the few ways to increase an actor's capabilities. The type of a reference specifies what type a message sent via the reference can have, and is used to uphold the guarantee that every message in the receiver's inbox is well-typed. By using typed actor references, the receiver does not need to worry about unexpected messages, while senders can be sure that messages will be understood. Typically, the reference type of a field should be the smallest set of types that will be sent using that reference.

Below we have created an instance of an `InboxShape`, showcasing the important concepts. `TestInbox` is an inbox that can receive two kinds of messages: messages containing a single boolean value, and messages containing a special kind of reference together with a natural number. The reference in the second kind of message can be sent two kinds of messages as well: messages containing a single boolean value, and messages containing a single string value.

```
BoolMessage = [ ValueType Bool ]¹
StringMessage = [ ValueType String ]¹

OtherInbox : InboxShape
OtherInbox = StringMessage :: [ BoolMessage ]¹

RefNatMessage : MessageType
RefNatMessage = ReferenceType OtherInbox :: [ ValueType ℕ ]¹
```

```
TestInbox : InboxShape
TestInbox = RefNatMessage :: [ BoolMessage ] 1
```

4.3.2 Inbox Subtyping

The Interface Segregation Principle (ISP) is a design principle that states that no client should be forced to depend on methods it does not use; when clients depend upon objects which contain methods used only by other clients, this can lead to fragile code (Martin 2002). In particular, if a client depends on an interface that contains methods that the client does not use but others do, that client will be affected by changes that those other clients force on the interface. This idea has also been addressed in the context of actor systems where He, Wadler and Trinder (2014) uses the term *type pollution problem* to describe the issue of actor interfaces being too fat.

Just as the advice of ISP is to break each class into granular interfaces, the advice of He, Wadler and Trinder (2014) is to break the type of inboxes into granular subtypes. Subtyping of inboxes means that given two inboxes A and B, if $A <: B$, then every message in A is also a valid message in B. Since we model `InboxShape` as a set, the subtype relation can be taken to just be the subset relation.

```
_<:_ = _⊆_ {A = MessageType}
```

Our representation of subsets uses two data structures: $\in$ and $\subseteq$. $\in$ is the list or set membership relation, which we model as a Peano number that tells at what position the element occurs in the list. An element that appears at the head of a list is at index `Z`, and the index of an element that appears somewhere in the tail of the list is the successor (`S`) of its index in the tail.

```
data _∈_ {a} {A : Set a} : A → (List A) → Set a where
  Z : ∀ {x xs} → x ∈ (x :: xs)
  S : ∀ {x y xs} → x ∈ xs → x ∈ (y :: xs)
```

The subset relation `A ⊆ B` holds if every member of `A` is also a member of `B`. This can be modelled as a function from indices of `A` to indices of `B`, but this turned out to be inconvenient for our purposes. An alternative approach is to build subsets as a *view* (McBride and McKinna 2004; Norell 2008) of the lists in question. A view is a data type that reveals something interesting about its indices, e. g. that a list is a subset of another. To define $\subseteq$ we state that the empty list (`[]`) is a subset of all lists, an if you are able to prove that an element is a member of a set, we state that you can add (`::`) that element to a subset of the set.

```
data _⊆_ {a} {A : Set a} : List A → List A → Set a where
  [] : ∀ {xs} → [] ⊆ xs
  _::_ : ∀ {x xs ys} → x ∈ ys → xs ⊆ ys → (x :: xs) ⊆ ys
```

The drawback of using subsets is that recursive subtypes are not captured, but we deem it good enough for our purposes and simple to work with. Proving that `A <: B` boils down to providing an index into `B` for every element in `A`. For example, `text-subtyping` proves that an inbox that only accepts `BoolMessage` is a subtype of the `TestInbox` from before.

```
TestInbox : InboxShape
TestInbox = RefNatMessage :: [ BoolMessage ]¹


Boolbox : InboxShape
Boolbox = [ BoolMessage ]¹

test-subtyping : Boolbox <: TestInbox
test-subtyping = [ S Z ]ᵐ
```

4.3.3 References

A reference to an actor is a name that is used to look up that actor's inbox in order to write or read from it. We assign types to references in order to statically guarantee that we will only send messages to an inbox which the receiving actor can understand. The property to maintain for this to hold is that the type of the reference must be a subtype of the inbox it references. That is, given that an inbox is globally referred to via *name* and has messages of type T, the type of a reference to *name* must have a type S, such that $S <: T$:

$$\frac{S, T : \texttt{InboxShape} \quad S <: T \quad inbox : \overrightarrow{\text{Message}(T)} \quad name \mapsto inbox}{\text{Reference}(name) : S}$$

The desire to statically check that references are well-typed has affected the design of M_{act} a lot. We first tried an encoding of references where a name is simply tagged with a type:

```
record ActorRef (InboxType : Set) : Set where
  constructor create-actor-ref
  field
    name : Name
```

This encoding is intuitive and simple, but it does not capture $S <: T$, nor does it capture $name \mapsto inbox$. Using this encoding thus requires finding a way to maintain additional proofs.

To maintain the necessary proofs of references being valid requires that we encode more things in the type system. The solution that seem to fit best in the setting of Agda is to make a bigger distinction between references and values. Values are kept as normal Agda terms, only restricted under Agda's usual typing rules. References, on the other hand, are treated as variables that are maintained explicitly in the

model, encoded in the type parameter of each actor. This technique makes it possible to maintain type correctness of references, as long as the effect every actor operation has on the reference variables is carefully designed.

We type check references in the standard way of maintaining a variable typing context. A typing context associates variables to types, where variables are commonly referred to by their name. Variable names makes expressions easy to understand for humans, but pose two annoying problems: α-equivalence and α-renaming.

α-equivalence is a form of equivalence that captures the intuition that the particular choice of a bound variable name does not usually matter (Turbak, D. Gifford and Sheldon 2008). Renaming the variables of an expression in a way that preserves α-equivalence is called α-renaming (Turbak, D. Gifford and Sheldon 2008). α-renaming is a part of the general concept of substitution, the operation of replacing free occurrences of a variable with an expression. The role of α-renaming in substitution is to avoid accidental variable name captures by renaming variables so that substitution does not change the meaning of functions (Turbak, D. Gifford and Sheldon 2008).

In order to avoid the problem of α-equivalence and α-renaming, a common formalization technique is the use of de Bruijn indices to represent variable binders (Berghofer and Urban 2007). A de Bruijn index is a natural number that represents an occurrence of a variable in a λ-term, and denotes the number of binders that are in scope between that occurrence and its corresponding binder. Table 4.1 shows some examples comparing a λ-calculus with names to a λ-calculus using de Bruijn indices.

Named	de Bruijn
$\lambda\ x \longrightarrow x$	λ 1
$\lambda\ x \longrightarrow \lambda\ y \longrightarrow x$	λ λ 2
$\lambda\ x \longrightarrow \lambda\ y \longrightarrow \lambda\ z \longrightarrow x\ z\ (y\ z)$	λ λ λ 3 1 (2 1)

Table 4.1: Comparison between λ-calculus with names and λ-calculus using de Bruijn indices

What makes de Bruijn indices easy to work with in Agda is that it lets us manage the variable typing context as a list of types, with variables as (de Bruijn) indices into that list. We choose to represent the indices as the membership proposition in order to make the de Bruijn indices correct by construction. This lets us define the type judgement $\Gamma \vdash T$ as:

```
ReferenceTypes = List InboxShape
TypingContext = ReferenceTypes

_⊢_ : TypingContext → InboxShape → Set₁
Γ ⊢ T = T ∈ Γ
```

For inboxes, the subtype relation A `<:` B says that every message in A is also a valid message in B. It should therefore be possible to downcast a reference of type B to a reference of type A, since every message sent to a reference of type A will be a valid message in B. We can see that the subtype relation for references is reversed, and is thus contravariant in its argument (He, Wadler and Trinder 2014). We capture this property in a special version of the reference typing judgement, where $\Gamma \vdash\!> T$ says that T is a subtype of some type in Γ.

```
record _⊢>:_ (Γ : TypingContext)
             (requested : InboxShape) : Set₁ where
  constructor [_]>:_
  field
    {actual} : InboxShape
    actual-is-sendable : Γ ⊢ actual
    actual-handles-requested : requested <: actual
```

The notion of a subtype for references is important to implement the pattern of sending a command together with what reference to reply to, since different actors receiving the reply will have a different `InboxShape`. This pattern, together with a selective receive construct, can be used to implement synchronous calls, which we explore in chapter 5.

4.4 Messages

Messages in M_{act} are made up of a tag, indicating the type of the message, and instantiations of the fields in that message type. We have made the unusual decision to give outgoing and incoming messages slightly different shapes. This choice is purely for ease of implementation and does not affect the power of the model.

An outgoing message is made of two parts: the tag that indicates which type of message it is and instantiations of every field of the type selected by the tag. We index the type of a message by the type of inbox it is being sent to and incidentally by the variable typing context. Selecting which type variant this message has is done by indexing into the `InboxShape`, using the $\in$ property. The rest of the message is made up of instantiations of the fields from the selected `MessageType`. For values, the instantiation of a field is simply an Agda term of the type specified by the field. For references, the instantiation is not a simple Agda term, but rather a variable in the reference context. The type of the selected reference variable must be compatible with the type that the receiver expects, i. e. they must have the correct subtype relation.

```
send-field-content : TypingContext → MessageField → Set₁
send-field-content Γ (ValueType A) = Lift A
send-field-content Γ (ReferenceType requested) = Γ ⊢>: requested
```

```
record SendMessage (To : InboxShape)
                   (Γ : TypingContext) : Set₁ where
  constructor SendM
  field
    {MT} : MessageType
    selected-message-type : MT ∈ To
    send-fields : All (send-field-content Γ) MT
```

Incoming messages differ from outgoing messages in the instantiation of reference fields. Looking at `receive-field-content` below, one would expect the content of a reference field to be some representation of a reference, e. g. an index into the variable context. Instead we find the unit type `⊤`, which has no computational content at all. The answer to this puzzle lies in that receiving a message will have the side-effect of adding the references from every reference field to the variable context. We saw in figure 4.3 that the shape of the variable typing context is maintained in the type of the monad, making it easy to create indices into the variable context without making them a part of the message. It would be possible to make the reference field content be an index into the variable context, but it would make the model slightly more complicated without making it more powerful; the programmer can already create indices when they are needed and, since the typing context is constantly changing, a specific index is often not valid for long.

```
receive-field-content : MessageField → Set
receive-field-content (ValueType A) = A
receive-field-content (ReferenceType Fw) = ⊤
```

```
record Message (To : InboxShape) : Set₁ where
  constructor Msg
  field
    {MT} : MessageType
    received-message-type : MT ∈ To
    received-fields : All receive-field-content MT
```

```
extract-references : MessageType → ReferenceTypes
extract-references [] = []
extract-references (ValueType x :: mt) = extract-references mt
extract-references (ReferenceType T :: mt) = T :: extract-references mt
```

```
add-references : ∀ {To} → TypingContext → Message To → TypingContext
add-references Γ (Msg {MT} x x₁) = extract-references MT ++ Γ
```

4.5 ActorM revisited

We are now ready to break down the definition from figure 4.3. Starting with the type signature of `ActorM`, we see that actors have a static interface `IS`. We also see that a `TypingContext` is managed as an invariant of the parameterized monad.

```
data ActorM (i : Size) (IS : InboxShape) : (A : Set₁) →
             (pre : TypingContext) →
             (post : A → TypingContext) →
             Set₂

record ∞ActorM (i : Size) (IS : InboxShape) (A : Set₁)
               (pre : TypingContext)
               (post : A → TypingContext) :
               Set₂ where
  coinductive
  constructor delay_
  field force : ∀ {j : Size< i} → ActorM j IS A pre post

data ActorM (i : Size) (IS : InboxShape) where
```

Return and bind of the actor monad use the invariants that are expected from any parameterized monad, which is explained in section 2.4. Bind chains together potentially infinite sub-computations, and we can see that bind preserves the size (observation depth) i. Agda implements subtyping for sizes, so computations of different sizes can still be composed via bind.

We don't give the constructors `Return` and `∞>>=` the names **`return`** and `>>=` since we want **`return`** and `>>=` to refer to the type `∞ActorM`. Instead, **`return`** and `>>=` are defined as a separate functions that wrap `Return` and `∞>>=`. A similar choice is made for the other constructors, which can be seen in the auxiliary function definitions presented in appendix B.

```
Return : ∀ {A post} →
  (val : A) →
  ActorM i IS A (post val) post

_∞>>=_ : ∀ {A B pre mid post} →
  (m : ∞ActorM i IS A pre mid) →
  (f : (x : A) → ∞ActorM i IS B (mid x) (post)) →
  ActorM i IS B pre post
```

`Spawn` creates a new actor and adds a reference to the spawned actor to the spawning actor's variable context. The spawned actor starts with both an empty inbox and an empty variable context.

```
Spawn : {NewIS : InboxShape} → {A : Set₁} → ∀ {pre postN} →
  ActorM i NewIS A [] postN →
  ActorM i IS T₁ pre λ _ → NewIS :: pre
```

An actor can send messages to any actor it has a reference to. It is not explicit in the monad what happens to a sent message, but we see in section 4.8 that evaluating Send will append the message to the actor being referenced. The reference variable might be a subtype to the underlying actor it references, but this fact is completely opaque to the `Send` construct. The content of the message is detailed in section 4.4. Sending a message does of course not affect the variable context, which is captured in the postcondition being the same as the precondition.

```
Send : ∀ {pre} → {ToIS : InboxShape} →
  (canSendTo : pre ⊢ ToIS) →
  (msg : SendMessage ToIS pre) →
  ActorM i IS T₁ pre (λ _ → pre)
```

The `Receive` construct is an operation that optimistically tries to retrieve a message from the actor's inbox. Messages are retrieved in the order they are added to the inbox, and in the case of an empty inbox the actor is paused indefinitely. As discussed in section 4.4, every reference in the received message will be added to the actor's variable context. `Receive` is the only construct that has a postcondition that depends on run-time behaviour, so a model where references are handled differently could allow for actors to be implemented as non-parameterized monads.

```
Receive : ∀ {pre} →
  ActorM i IS (Message IS) pre (add-references pre)
```

A key concept in Erlang-style actors is that actors can easily get a reference to themselves. For example, in order to implement callbacks the initiating actor must include a reference to itself for the corresponding actor to reply to. In `ActorM`, this need is fulfilled by the `Self` construct, which adds a reference to the actor itself to the variable context.

```
Self : ∀ {pre} →
  ActorM i IS T₁ pre (λ _ → IS :: pre)
```

The postconditions and preconditions that we get from our Hoare-style reasoning are sometimes not quite what we want. The invariants might be logically equivalent but have a different syntactic form. For example, the variable context might be in the wrong order or knows about more references than is expected. We thus need a construct for reordering or forgetting about variables to make subsequent invariants compatible. What we chose to implement is similar to the strengthening rule in Hoare logic, which affects the precondition of a statement. The operation that `Strengthen` performs is a re-ordering of the variables in the variable context. It does so by relating the precondition to the postcondition via a subset relation. The subset relation supports re-ordering, duplication, and to forget variables, making it powerful without sacrificing correctness.

```
Strengthen : ∀ {ys xs} →
  (inc : ys ⊆ xs) →
  ActorM i IS T₁ xs (λ _ → ys)
```

4.6 A simple actor program

```
TickTock : MessageType
TickTock = [ ValueType Bool ]¹

TickTocker : InboxShape
TickTocker = [ TickTock ]¹

tick-tocker : ∀ {i} → ∞ActorM (↑ i) TickTocker T₁ [] (λ _ → [])
tick-tocker .force = (do
  self
  Msg Z (b :: []) ← receive
    where Msg (S ()) _
  let
    Γ = [ TickTocker ]¹
    to : Γ ⊢ TickTocker
    to = Z
    tag : TickTock ∈ TickTocker
    tag = Z
    fields : All (send-field-content Γ) TickTock
    fields = lift (not b) :: []
  to ![t: tag ] fields
  strengthen []) ∞>>=
  λ _ → tick-tocker

spawner : ∀ {i} → ∞ActorM i [] ℕ₁ [] (λ _ → [ TickTocker ]¹)
spawner = do
  spawn∞ tick-tocker
  let
    Γ = [ TickTocker ]¹
    to : Γ ⊢ TickTocker
    to = Z
    tag : TickTock ∈ TickTocker
    tag = Z
    fields : All (send-field-content Γ) TickTock
    fields = lift true :: []
  to ![t: tag ] fields
  return 42
```

Figure 4.4: A simple actor program

We can now write our first program in M_{act}! Figure 4.4 defines two actors.

The actor `spawner`

1. spawns the other actor
2. sends it a message containing the value `true`
3. returns 42

The actor `tick-tocker`

1. calls self, adding a reference to itself to its variable context
2. receives a boolean value
3. sends the inverse of that boolean to itself
4. empties its variable context
5. repeats this procedure forever

Unfortunately there is no simple way make the definitions of `tick-tocker` and `spawner` less noisy. Our definitions of tags and variables rely completely on elements' position in a list, so to tell Agda which tag or variable we want to refer to we must provide its index. Brady (2015) show how effects in Idris—which use a type of parameterized monads—can be given labels. Via the automation facilities of Idris, a programmer can simply provide the label they want to access and the compiler will automatically find its index. Implementing labels and a similar proof automation for $\in$ in Agda should be possible, but it is out of scope for this thesis.

4.7 Representing references

We described in section 4.3.3 how each actor is provided a variable context that contains the references that the actor has access to. A reference is in its turn also a kind of variable or address that leads to the actual inbox. It would make sense to represent references as de Bruijn indices, since they can be seen as variables, but the constraints imposed on us by modelling a distributed system make this a poor choice.

When a variable is added to to a variable context that uses de Bruijn indices, every previous index into the variable context is invalidated. The other indices are no longer valid since the variable that their index points to has been shifted by one. To avoid invalidating indices, each index has to be incremented by one, through a substitution process called *lifting*.

In the context of inboxes and references, lifting needs to occur whenever a new inbox is added, i. e. whenever an actor is spawned. References are prevalent throughout the whole model, so these lifting substitutions would have to be performed both for every reference available to an actor and for every message stored in any inbox. Substituting message terms whenever an actor is spawned works in the setting of

an abstract machine, but also suggests that the model requires synchronization whenever an actor is spawned. We wanted our model to have a simple translation to a distributed setting outside the context of an abstract machine, making de Bruijn subpar for the task.

References need to be given a representation that is persistent and that can uniquely identify an inbox in order to avoid unnecessary substitutions. In section 4.3.3 variable names were ruled out due to the problems of α-renaming and α-equivalence, but these are problems that mainly exist when variable names can be chosen freely. The representation of references is not visible to the programmer in M_{act}, so the problems with clashing names can be completely avoided as long as the system generates a unique name for every actor. This is a property that is very easy to maintain, e. g. in our model names are natural numbers and uniqueness is maintained by increasing a counter.

4.8 Semantics

Before describing the semantics of M_{act} we will once again look at Fowler, Lindley and Wadler's (2017) λ_{act}. Figure 4.5 shows evaluation contexts and configurations, as well as reductions on terms and configurations.

The concurrent behaviour of λ_{act} is given by a non-deterministic relation on configurations, consisting of parallel composition $\mathcal{C} \parallel \mathcal{D}$, name restrictions $(\nu a)\mathcal{C}$, and actor configurations $\langle a, M, \overrightarrow{V} \rangle$. An actor configuration $\langle a, M, \overrightarrow{V} \rangle$ models an actor with name a, evaluating the term M, with an inbox consisting of values $\overrightarrow{V}$.

Reduction is defined in terms of evaluation contexts E. The syntax of evaluation contexts is defined to make it simple to define reductions that follow the precise evaluation order specified by fine-grain call-by-value. Reduction on terms $(\longrightarrow_M)$ is defined to be deterministic, whereas configurations $(\longrightarrow)$ use a non-deterministic reduction relation.

The rule under 'Additional structural congruence' is not present in the original work by Fowler, Lindley and Wadler (2017); Fowler notes that it was left out by accident (personal communication, May 29, 2018). As with the other structural congruence rules, it is an equivalence relation and therefore reflexive, symmetric, and transitive. Reduction on configurations is done modulo these equivalences, i. e. the rule in figure 4.6 is implicit.

4.8.1 Environment of $\mathbf{M_{act}}$

The semantics of M_{act} is defined in terms of reductions of a global environment. Reduction on environments corresponds to λ_{act}'s reduction on configurations, but the definition of environments differs significantly from λ_{act}'s definition of configurations. The difference primarily stems from our goal of making it easy to create an

Syntax of evaluation contexts and configurations

$$\begin{array}{lll} \text{Evaluation contexts} & E & ::= [\,] \mid \texttt{let}\, x \Leftarrow E\, \texttt{in}\, N \\ \text{Configurations} & \mathcal{C}, \mathcal{D}, \mathcal{E} & ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid \langle a, M, \overrightarrow{V} \rangle \\ \text{Configuration contexts} & G & ::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a) G \end{array}$$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Pid}\ \frac{\Gamma, a : \texttt{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

$$\textsc{Actor}\ \frac{\Gamma, a : \texttt{ActorRef}(A) \mid M : 1 \quad (\Gamma, a : \texttt{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \texttt{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

Reduction on terms

$$(\lambda x.M)\ V \longrightarrow_M M\{V/x\} \qquad \texttt{let}\, x \Leftarrow \texttt{return}\, V\, \texttt{in}\, M \longrightarrow_M M\{V/x\}$$

$$E[M] \longrightarrow_M E[M'] \quad (\texttt{if}\ M \longrightarrow_M M')$$

Structural congruence

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D})\ \ \texttt{if}\ \ a \notin \texttt{fv}(\mathcal{C}) \qquad G[\mathcal{C}] \equiv G[\mathcal{D}]\ \ \texttt{if}\ \ \mathcal{C} \equiv \mathcal{D}$$

Additional structural congruence

$$(\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

Reduction on configurations

$$\begin{array}{rrcl} \textsc{Spawn} & \langle a, E[\texttt{spawn}\, M], \overrightarrow{V} \rangle & \longrightarrow & (\nu b)(\langle a, E[\texttt{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \epsilon \rangle) \quad (b \text{ is fresh}) \\ \textsc{Send} & \langle a, E[\texttt{send}\, V'\ b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle & \longrightarrow & \langle a, E[\texttt{return}()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle \\ \textsc{SendSelf} & \langle a, E[\texttt{send}\, V'\ a], \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\texttt{return}()], \overrightarrow{V} \cdot V' \rangle \\ \textsc{Self} & \langle a, E[\texttt{self}], \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\texttt{return}\, a], \overrightarrow{V} \rangle \\ \textsc{Receive} & \langle a, E[\texttt{receive}], W \cdot \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\texttt{return}\, W], \overrightarrow{V} \rangle \\ \textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_1] \quad (\texttt{if}\ \ \mathcal{C}_1 \longrightarrow \mathcal{C}_2) \\ \textsc{LiftM} & \langle a, M_1, \overrightarrow{V} \rangle & \longrightarrow & \langle a, M_2, \overrightarrow{V} \rangle \quad (\texttt{if}\ \ M_1 \longrightarrow_M M_2) \end{array}$$

Figure 4.5: λ_{act} evaluation contexts, configurations, and reductions

$$\frac{\mathcal{C} \equiv \mathcal{D} \quad \mathcal{D} \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{E}}{\mathcal{C} \longrightarrow \mathcal{E}}$$

Figure 4.6: Reduction on equivalences of configurations

interpreter for M_{act}. A mechanization that focuses on the mathematical properties of singly typed actors would likely end up with a representation more similar to Fowler, Lindley and Wadler's (2017) representation.

Environments in M_{act} are made up of four kinds of data: a typing context for inboxes, raw data, coherence proofs, and a proof of weak progress. The raw data constitutes the computational content in the environment, such as the actors and their inboxes; the coherence proofs are used to ensure that the raw data is well typed in the inbox typing context; and the proof of weak progress is used to constrain the situations where an actor can be considered blocked. Preservation is captured automatically by `ActorM` (together with reductions not changing the return type of computations), and we can therefore say that our system is well-typed, modulo system liveness, according to the standard definition of type safety as proposed by Wright and Felleisen (1994).

Inbox typing contexts.

The inbox typing context associates actor names to inbox types. Actor names are represented by natural numbers, as mentioned in section 4.7, and inbox types are the `InboxShape`'s of section 4.3.1. The typing context is represented as an association list. This is a representation that has the advantage of it being simple to prove that adding a new element with an unused key will not shadow any existing entries.

```
Name = ℕ

record NamedInbox : Set₁ where
  constructor inbox#_[_]
  field
    name : Name
    shape : InboxShape

Store = List NamedInbox
```

Raw data.

An inbox is a list of messages delivered to an actor that have not yet been processed. We use the convention of reading messages from the top and appending them at the back, creating a FIFO queue. Inboxes are parameterized by the inbox typing context to ensure that inboxes are well-typed in the environment.

```
Inbox : InboxShape → Set₁
Inbox is = List (NamedMessage is)

data Inboxes : (store : Store) → Set₁ where
  [] : Inboxes []
  _::_ : ∀ {store name inbox-shape} →
         Inbox inbox-shape →
         (inboxes : Inboxes store) →
         Inboxes (inbox# name [ inbox-shape ] :: store)
```

We saw in section 4.2 that messages have a different representation when sent and received. Messages in transit have yet a different representation: `NamedMessage`. The differentiating factor of `NamedMessage` is that reference fields store the actual name of the actor it references. This makes `NamedMessage` a representation that allows for immutability of messages in transit—an important property in distributed systems.

```
named-field-content : MessageField → Set
named-field-content (ValueType A) = A
named-field-content (ReferenceType Fw) = Name

record NamedMessage (To : InboxShape) : Set₁ where
  constructor NamedM
  field
    {MT} : MessageType
    named-message-type : MT ∈ To
    named-fields : All named-field-content MT
```

Actors in the environment are separated into actors that can be reduced and actors that can not. By making this separation we make it simpler to implement an interpreter and the proof of weak progress comes almost for free.

An actor consists of a name, the references it knows of, and a computation. The computation consists of a term `ActorM` and a stack of continuations. The stack of continuations serves the same purpose as λ_{act}'s evaluation context: it enables reducing actors one step at a time. To reduce a term $M >\!\!>\!\!= f$ we must first reduce the term M, which might require multiple reductions. We can postpone the reduction waiting for the value from M by adding the function f to the stack. Actors can thus be reduced one step at a time by repeating this procedure until we reach a term that is not of the form $M >\!\!>\!\!= f$—the only form that requires of multiple steps.

The environment also contains an infinite supply of fresh names. Freshness is ensured by using a monotonically increasing counter and by maintaining a proof stating that the next name is strictly larger than every name used so far.

Coherence proofs

The names of actors and references are not constrained in the raw data. To ensure that environments are well-typed we must thus maintain additional proofs of names being associated to the right type in the inbox typing context.

We maintain two types of coherence proofs for actors: a coherence proof for the actor's name and coherence proofs for the references known by the actor. The coherence proof for actor names states that the name of an actor should be associated to the actor's `InboxShape`, that is:

```
actor .name ↦ actor .inbox-shape ∈e store
```

The coherence proof for references states that every reference should be associated to a compatible `InboxShape`, that is:

```
(reference .name ↦ B ∈e store) × ( reference .shape <: B)
```

Coherence proofs for `NamedMessage`'s follow in a similar fashion: value-fields are always coherent, while reference-fields are coherent if the stored name is associated to a compatible `InboxShape`.

Weak progress

The calculus of λ_{act}, and in consequence M_{act}, allows for configurations that are deadlocked, i. e. where every running actor is waiting to receive a message. Nevertheless, Fowler, Lindley and Wadler (2017) develop a notion of *weak progress*, stating that the only situation in which a well-typed closed configuration cannot reduce further is when each actor is either of the form $\langle a, \texttt{return}\, W, \overrightarrow{V} \rangle$, for some value W, or of the form $\langle a, \texttt{receive}, \epsilon \rangle$.

The environments in M_{act} store the actors that can reduce separately from those that can not. To show a that M_{act} has weak progress we must therefore show that we have a reduction for every actor in the first list, which we will do in section 4.8.2. To show that every actor in the second list is of the form $\langle a, \texttt{return}\, W, \overrightarrow{V} \rangle$ or $\langle a, \texttt{receive}, \epsilon \rangle$ we require that those actors inhabit the view `IsBlocked`. The constructors of `IsBlocked` only match actors of the mentioned forms, thereby encoding the desired notion of weak progress.

```
data IsBlocked (store : Store)
               (inboxes : Inboxes store)
               (actor : Actor) : Set₂ where
  BlockedReturn :
    ActorAtConstructor Return actor →
    ActorHasNoContinuation actor →
    IsBlocked store inboxes actor
  BlockedReceive :
    ActorAtConstructor Receive actor →
    (p : has-inbox store actor) →
    InboxForPointer [] store inboxes p →
    IsBlocked store inboxes actor
```

4.8.2 Reductions

Reductions on environments are in M_{act} defined in terms of a `reduce` function. The function is used by the interpreter, which calls it repeatedly until there are no more actors that can reduce. The interpreter selects an actor in the environment to focus on and `reduce` performs the appropriate reduction.

```
reduce : {actor : Actor} → (env : Env) → Focus actor env → Env
```

We define reductions as separate functions that pose different constraints on the focused actor. The constraint is in general that the computation of the focused actor is in the form of a specific constructor, but the functions for `Return` and `Receive` use additional constraints to branch on whether the actor can reduce or not.

Bind and return

The reduction functions for bind and return can be seen as antagonists. Reduction of bind matches a focused computation of the form

```
m ∞>>= f → continuation
```

which it reduces to a computation where `f` is added to the continuation stack:

```
m .force → f :: continuation
```

Reduction of return matches a focused computation of the form

```
Return v → f :: continuation
```

which it reduces to a computation where `v` is applied to `f`:

```
f v .force → continuation
```

The focused actor can also be in a form that can not reduce

```
Return v → []
```

However, this matches the constructor `BlockedReturn` from `IsBlocked`. We can thus move the focused actor to the list of actors that can not reduce.

Spawn

Reduction of spawn matches a focused computation of the form

```
Spawn actor → continuation
```

To reduce this computation we must create a fresh actor name, add a new entry to the inbox typing context, create a matching empty inbox, add a reference of the new inbox to the focused actor, update the coherence proofs, and add the spawned actor to the list of running actors. That is a mouthful, but they are each just a simple application of `::` in the case of adding to a list and applications of `suc` in the case of updating a coherence proof. The focused computation reduces to return of unit

```
Return (lift tt) → continuation
```

and the spawned `actor` is given an empty continuation stack

```
actor → []
```

Send

Reduction of send matches a focused computation of the form

```
Send Γ⊢ToIS (SendM tag fields) → continuation
```

which reduces to return of unit

```
Return (lift tt) → continuation
```

Converting the message `(SendM tag fields)` to a valid `NamedMessage` requires more finesse. Via the reference pointed to by `Γ⊢ToIS` and the reference's coherence proof we get a name associated to an inbox of type `B` and a subtype relation between `ToIS` and `B`:

```
(name ↦ B ∈e store) × ( ToIS <: B)
```

The subtype relation is used to make `tag` a tag in `B`, via

```
ToIS <: B = ToIS ⊆ B
tag = MT ∈ ToIS
ToIS ⊆ B → MT ∈ ToIS → MT ∈ B
```

Reference fields in `SendMessage` have a subtype relation between the type of the forwarded reference and the type of the reference variable. Reference variables in turn have a subtype relation to their associated inboxes. To create the coherence proof for the new `NamedMessage` we must thus apply transitivity of subtyping:

```
actual-is-sendable = Γ ⊢ actual
actual-handles-requested = requested <: actual
actual <: B
requested ⊆ actual → actual ⊆ B → requested ⊆ B
```

Finally, sending a message will unblock the actor that receives the message. We have structured the function that updates an inbox in a way that enables us to prove that other inboxes are unaffected. This means that the only actor that has its proof of `IsBlocked` invalidated, and consequently is moved to the actors that can reduce, is the affected actor.

Receive

To reduce receive when there exists a message to be received involves removing the next message from its inbox, converting it into a `Message`, and adding the received references to the actor. These operations all follow from the raw data of the received message and its coherence proof. For example, to extract the references from the `NamedMessage` we traverse its fields and extract the stored names

```
extract-inboxes : ∀ {MT} → All named-field-content MT → List NamedInbox
extract-inboxes [] = []
extract-inboxes (_::_ {ValueType _} _ ps) =
  let rec = extract-inboxes ps
  in rec
extract-inboxes (_::_ {ReferenceType x} name ps) =
  let rec = extract-inboxes ps
  in inbox# name [ x ] :: rec

named-inboxes : ∀ {S} → (nm : NamedMessage S) → List NamedInbox
named-inboxes (NamedM tag fields) = extract-inboxes fields
```

and in the case of converting the `NamedMessage` to a `Message`, via the function `unname-message`, we replace the stored names with instances of T_1. The actor is then reduced to a computation that returns the converted message:

```
Return (unname-message named-message) → continuation
```

We also have to handle the case when there is no message to receive. This is the second and final case of when an actor can be considered blocked, and it matches the constructor `BlockedReceive` from `IsBlocked`. We can thus move the focused actor to the list of actors that can not reduce.

Self

The raw data of an actor contains both its name and `InboxShape`. Creating the reference needed when reducing self is thus easy:

```
inbox# (actor .name) [ actor .inbox-shape ]
```

To produce a coherence proof for the added reference we use the coherence proof for actor names and reflexivity of subtyping:

```
(actor .name ↦ actor .inbox-shape ∈e store)
                          ×
( actor .inbox-shape <: actor .inbox-shape)
```

The computation reduces to return of unit

```
Return (lift tt) → continuation
```

Strengthen

Reduction of strengthen matches a focused computation of the form

```
Strengthen Δ⊆Γ → continuation
```

The computation reduces to return of unit and the actor's known reference variables are changed from Γ to Δ. We get the references and coherence proofs we need by following each $\in$-relation of the subset $\Delta \subseteq \Gamma$.

```
Return (lift tt) → continuation
```

5

Selective receive

In the previous chapter we defined the syntax, typing rules, and semantics of M_{act}. With it, we can define actors and see them interact by running them in our interpretation function. As defined, however, it has one particularly annoying property: messages have to be processed in the order they are received.

One of the key properties of the actor model is how actors are encapsulated from each other. In contrast to method-based object oriented programming, actors can decide to handle messages at a later time or even discard messages. By allowing for out-of-order processing, actors are given control over their own state and behaviour. We can say that we gain increased encapsulation by decoupling execution from signaling.

The `Receive` construct of `ActorM` does not provide actors with the means to process messages out-of-order; actors can only read messages in the order they were received. An actor that is waiting for a specific message, e. g. the response from a method invocation or the next sequence of a protocol, can not simply wait for that message—it is forced to first process any message that happens to come in-between. This risks leading to actors having control flow where multiple separate tasks are performed at the same time, making for confusing code.

The inboxes of Erlang are also first-in, first-out. However, Erlang additionally provides a *selective receive* construct, where messages are matched against multiple patterns, allowing for out of order processing of messages. Messages that do not match the patterns are not lost, they are instead put aside for later processing. We can understand how selective receive is used in practice by looking at how Erlang/OTP[1] implements synchronous calls. The code below is a modification of their code, simplified to extract its essence.

```
Identifier = erlang:make_ref(),
Process ! {{self(), Identifier}, Request},
receive
    {Identifier, Reply} -> {ok, Reply}
end
```

[1] https://github.com/erlang/otp/blob/f59026a57ccdfe4e462cbab9b97e9cd377dd5bdb/lib/stdlib/src/gen.erl#L172-L186

The call starts by sending a message that includes a unique identifier. Selective receive is then used to wait for a message containing that identifier—the reply to the call. Erlang uses a powerful type of pattern matching which is used in the pattern of the receive clause. First is the pattern on Identifier, which is a pattern bound to a value. This case will only match if the expression evaluates to the same value. Second is the pattern on Reply, which is an unbound variable. When the pattern match succeeds, the variable is bound to the value of the expression, which can then be used in the following statement. We see that the code that implements synchronous calls is able to use selective receive to ignore messages that are currently irrelevant, and thus allows the control-flow to be straight forward and without interleaved processing of other messages.

5.1 Theory of selective receive

There are primarily two possible alternatives for implementing selective receive: by extending `ActorM` with an additional constructor for selective receive, or as a user-space library. Both options are intriguing in their own way, so we have investigated them both in order to make a fair comparison.

By studying Erlang's selective receive we can see that it serves two main purposes: to select what kind of messages can be received and to provide alternative code paths based on which pattern it was that actually matched. Agda does not provide the facilities to implement the same kind of pattern matching that Erlang uses, but we can devise an alternative mechanism to serve its purpose.

The first feature needed in the replacement mechanism is the ability to filter out what messages to accept. We need to know for each message: does this message match the constraints we have set up? In its essence, this is just a function from `Message` to `Bool`.

```
MessageFilter : (IS : InboxShape) → Set₁
MessageFilter IS = Message IS → Bool
```

In the example from Erlang/OTP there is a single pattern and thus a single code path. It is important to note that there are no code paths for the messages that do not match a pattern, those messages are implicitly stored for later processing. Agda does not allow that we omit cases in a pattern—incomplete patterns are rejected by the coverage checker. Selective receive would be fairly useless if the code still handles every case, so working around this problem is crucial.

When we omit cases in a pattern we create a function that is only partially defined—a partial function. This is a function that is not defined for some part of its domain. We might also think of it as being a total function for the parts that we *did* define, but that the domain is stretched. We would be left with the core that is total if we shrink the domain to the parts that are defined—a function we can implement in Agda.

Shrinking the domain of a function in Agda is surprisingly easy. By adding propositions on the arguments of the function it is possible to limit the values of a domain to just the desired subset. Below we have defined a function over natural numbers, which we, by adding the constraint that n = 2, only have to define for exactly that case.

```
partial : (n : ℕ) → (n ≡ 2) → Bool
partial 0 ()
partial 1 ()
partial 2 p = false
partial (suc (suc (suc n))) ()
```

Using partial functions to implement selective receive is not new. Haller and Odersky (2009) use Scala's `PartialFunction` to empower the Scala Actors library with a receive statement similar to Erlang's. Scala is not a total language, so their partial functions can be implemented as a function `apply`(x : a) : b paired with an independent function `isDefinedAt`(x : a) : `Boolean`. This definition is very similar to ours, with the only difference being that our partial functions need to supply an additional proof that the element is in the domain.

5.2 Selective receive as a primitive operation

We apply the knowledge that shrinking the domain lets us define 'partial functions' and define a data-type where messages are limited to those that match the filter.

```
record SelectedMessage {IS : InboxShape}
         (f : MessageFilter IS) : Set₁ where
  constructor sm:_[_]
  field
    msg : Message IS
    msg-ok : f msg ≡ true
```

`MessageFilter` and `SelectedMessage` together is enough to implement our first version of selective receive. It is implemented by adding a new constructor to `ActorM`, so we will call this version SR_{prim}, from *prim*itive operation. The constructor that we add is `SelectiveReceive`, which behaves exactly the same as the `Receive` construct, except that messages have to pass the filter to be accepted.

```
selected-type : ∀ {IS} {filter : MessageFilter IS} →
                SelectedMessage filter →
                MessageType
selected-type sm: msg [ msg-ok ] = msg .MT

add-selected-references : TypingContext → ∀ {IS}
                            {filter : MessageFilter IS} →
                            SelectedMessage filter →
                            TypingContext
add-selected-references Γ m =
  add-references-static Γ (selected-type m)


SelectiveReceive : ∀ {pre} →
  (filter : MessageFilter IS) →
  ActorM i IS
    (SelectedMessage filter)
    pre
    (add-selected-references pre)
```

In figure 5.1, we have defined an actor computation—`receive-42-with-references`—that uses selective receive. The example defines the filter `receive-42` which accepts message of the type `WithReference` where the first field is the natural number `42`. The first message that passes this filter will be passed to `after-receive`. Pattern matching is used in `after-receive` to handle only those messages that can pass the filter. Take special note of the invariants of `after-receive` which have a precondition that depends on the message type and a postcondition that does not. Thanks to `receive-42`, which selects messages of a certain type, are we able to guarantee the type of the reference that will be added to the typing context.

5.2.1 Semantics

The semantics of selective receive follow from the semantics of receive. The only difference is that selective receive has a different definition of what message is next and that we have to define a new way for an actor to be blocked. Core to the new semantics are two data-structures: `SplitList` and `FoundInList`. `SplitList` lets us focus on a specific element in a list and `FoundInList` lets us assert that the focused element matches the filter, or that none of the elements do.

```
record SplitList {a : Level} {A : Set a} (ls : List A) : Set (lsuc a) where
  field
    heads : List A
    el : A
    tails : List A
    is-ls : (heads) ++ (el ∷ tails) ≡ ls
```

```
WithReference : MessageType
WithReference = ValueType ℕ :: [ ReferenceType OtherInbox ]¹

SR-inbox : InboxShape
SR-inbox = SomeMessage :: [ WithReference ]¹

receive-42 : MessageFilter SR-inbox
receive-42 (Msg Z _) = false
receive-42 (Msg (S Z) (n :: _ :: [])) = ⌊ n ≟ 42 ⌋
receive-42 (Msg (S (S ())) _)

after-receive : ∀ {i Γ} →
  (msg : SelectedMessage receive-42) →
  ∞ActorM i SR-inbox
    T₁
    (add-selected-references Γ msg)
    (λ _ → OtherInbox :: Γ)
after-receive sm: Msg Z _ [ () ]
after-receive sm: Msg (S Z) (n :: _ :: []) [ msg-ok ] = return tt
after-receive sm: Msg (S (S ())) _ [ _ ]

receive-42-with-reference : ∀ {i Γ} →
                            ActorM i SR-inbox
                              T₁
                              Γ
                              (λ _ → OtherInbox :: Γ)
receive-42-with-reference =
  selective-receive receive-42 ∞>>=
  after-receive
```

Figure 5.1: A small program that uses selective receive

```
matches-filter : ∀{a} {A : Set a} → (f : A → Bool) → A → Set
matches-filter f v = f v ≡ true

misses-filter : ∀{a} {A : Set a} → (f : A → Bool) → A → Set
misses-filter f v = f v ≡ false

data FoundInList {a : Level} {A : Set a}
                 (ls : List A)
                 (f : A → Bool) :
                 Set (lsuc a) where
  Found : (split : SplitList ls) →
          (matches-filter f (SplitList.el split)) →
          FoundInList ls f
  Nothing : All (misses-filter f) ls →
            FoundInList ls f
```

We create an element of `FoundInList` for the actor's inbox and the message that is `Found` is the message that the actor receives. When reducing selective receive, `Found` provides the proof of `f msg ≡ true` that is needed in `SelectedMessage`. Reducing the computation of selective receive is thus a matter of pairing the message with the proof from `Found`:

```
                split : SplitList inbox
              ok : filter (split .el) ≡ true
      received-message = unname-message (split .el)
    Return sm: received-message [ ok ] → continuation
```

We can not reduce selective receive when none of the messages in the inbox matches the filter. To model this in our notion of weak progress, `IsBlocked` has to be extended with a new constructor: `BlockedSelective`. The new constructor matches actors of the form $\langle a, \texttt{receive}_{\texttt{selective}}\, f, \vec{V}\rangle \mid \forall v \in V : fv \equiv \mathit{false}$

```
BlockedSelective :
  (aac : ActorAtConstructor Selective actor) →
  (point : has-inbox store actor) →
  ∀ inbox →
  {ps : All
        (misses-filter (filter-named (selected-filter actor aac)))
        inbox} →
  InboxForPointer inbox store inbs point →
  InboxInFilterState inbox (Nothing ps) →
  IsBlocked store inbs actor
```

5.3 Selective receive as a library

Fowler, Lindley and Wadler (2017) show that selective receive can be emulated without adding any new primitive operations, provided actors are written in a language with support for lists and algebraic data types. This is an idea that Haller (2012) explore in the context of actors in Scala, where a save queue is used to emulate selective receive in Akka. These insights tells us that we should be able to write some Agda function `selective-receive` that emulates the functionality of SR_{prim}. In that vein we develop a small Agda library that emulates selective receive, which we call SR_{lib}.

SR_{lib} uses the same `MessageFilter` as SR_{prim}, and messages returned from the selective receive construct still come with a proof of having passed the filter. However, SR_{lib} additionally has to store received messages that do not pass the filter, so that they can be processed later. The simple solution is to pair the returned message with a 'save queue', i. e. a list of messages that have been received but not yet matched.

```
record SelRec (IS : InboxShape) (f : MessageFilter IS) : Set₁ where
  constructor sm:_[_]-stash:_
  field
    msg : Message IS
    msg-ok : f msg ≡ true
    waiting : List (Message IS)
```

Recalling the definition of `Receive`, when a reference is received it gets added to the variable context. Our emulation of selective receive can not change the order that messages are read from the inbox, but that does not mean that SR_{lib} is forced to let the variables of received, but not yet matched, messages shadow the other variables. By using the `Strengthen` operation, SR_{lib} can reorder the variable context into two segments: variables we want to have access to and variables that belong to unmatched messages.

Both the precondition and postcondition of `selective-receive` split the variable context into two segments. The precondition is split into accessible variables `Γ` and unmatched messages (`waiting-refs q`). The postcondition is split the same way, with the references in the matched message added to `Γ` and the references of the new list of unmatched messages at the end[2]. We only show the type signature of `selective-receive` here, but the interested reader finds its full implementation in appendix C.

[2] Associativity of `++` dictate that
`add-references Γ (msg m) ++ waiting-refs (waiting m) ≡`
`add-references (Γ ++ waiting-refs (waiting m)) (msg m)`.
The sole reason for explicitly adding the references to `Γ` is to have the types line up better for future use, e. g. subsequent calls to `selective-receive` or when accessing variables in `Γ`

```
selective-receive : ∀ {i IS Γ} →
                    (q : List (Message IS)) →
                    (f : MessageFilter IS) →
                    ∞ActorM i IS
                      (SelRec IS f)
                      (Γ ++ (waiting-refs q))
                      (λ m → add-references Γ (msg m) ++
                             waiting-refs (waiting m))
```

Below we have adapted the example from section 5.2 to SR_{lib}. It has the same functionality as there, but now a save-queue has to be passed around manually. The save-queue is also prevalent in the invariants of the actor, making for less elegant types.

```
WithReference : MessageType
WithReference = ValueType ℕ :: [ ReferenceType OtherInbox ]¹

SR-inbox : InboxShape
SR-inbox = SomeMessage :: [ WithReference ]¹


receive-42 : MessageFilter SR-inbox
receive-42 (Msg Z _) = false
receive-42 (Msg (S Z) (n :: _ :: [])) = ⌊ n ≟ 42 ⌋
receive-42 (Msg (S (S ())) _)


after-receive : ∀ {i Γ} →
  (m : SelRec SR-inbox receive-42) →
  ∞ActorM i SR-inbox
    (List (Message SR-inbox))
    (add-references Γ (msg m) ++ waiting-refs (waiting m))
    (λ q' → OtherInbox :: Γ ++ (waiting-refs q'))
after-receive (sm: Msg Z _ [ () ]-stash: _)
after-receive (sm: Msg (S Z) (n :: _ :: [])
                   [ msg-ok ]-stash: waiting) =
              return₁ waiting
after-receive (sm: Msg (S (S ())) _ [ _ ]-stash: _)


receive-42-with-reference : ∀ {i Γ} →
                            (q : List (Message SR-inbox)) →
                            ActorM i SR-inbox
                              (List (Message SR-inbox))
                              (Γ ++ (waiting-refs q))
                              (λ q' → OtherInbox :: Γ ++
                                      (waiting-refs q'))
receive-42-with-reference q =
  selective-receive q receive-42 ∞>>=
  after-receive
```

5.4 Building on selective receive

Because of time constraints, we had to select one implementation of selective receive to focus our research on. SR_{prim} and SR_{lib} are equally expressive, but other properties make the choice important.

SR_{lib} has the advantage that it keeps the calculus small, but in doing so it sacrifices some simplicity. In particular, the invariants of selective receive are more complex in SR_{lib} than in SR_{prim}. The complexity of the invariants in SR_{lib} stem from it having to capture variables from messages that are received but not yet matched. Those messages are completely hidden from the invariants in SR_{prim}, naturally making for simpler invariants.

If our goal was to prove properties about the calculus, focusing on a minimal calculus would make a lot of sense. We choose instead to focus on what abstractions can be built using selective receive, making ease of use a much higher priority than a small calculus. The abstractions we present are thus built using SR_{prim}. Translating the abstractions into SR_{lib} is possible, although time consuming.

In the following sections we will show how selective receive can capture some important communication patterns. We will start with an emulation of local channels, which forms the basis for our implementations of synchronous calls and active objects.

5.5 Channels

Using selective receive, we can create an abstraction that emulates local channels. A local channel is a message queue with multiple writers and a single reader, much like an inbox. The difference between a local channel and an inbox is that an actor can have multiple local channels, each with their own type. The difference between a local channel and the channels of π-calculus is that the channels of π-calculus can be read by multiple processes.

The importance of having channels that are local is reflected in the implementation of typed channels in Cloud Haskell (Epstein, Black and Jones 2011). In Cloud Haskell, the motivation for local channels is that they expect channels to be used to communicate across a network and that moving the receive port of a channel is difficult in a distributed setting, especially when one of the prime reasons for moving a service to a new server is that the old server has crashed.

We choose to reuse the types for inboxes in the types for channels. The messages that can be received over a channel are thus modelled as an `InboxShape`. New channels can be created at run-time. This means that multiple channels can share the same type and that a channel-identifier must be included in every message to decide what channel the message was sent to. We model this by restricting the message types of a channel to those that includes a channel-identifier as their first value. The channel-

identifier must be unique, but we want to avoid further extensions to `ActorM` and have therefore left identifiers as natural numbers. We assume that the programmer has some means of creating unique tags, for example via UUID's[3].

```
UniqueTag = ℕ
TagField = ValueType UniqueTag

data IsChannelMessage : MessageType → Set₁ where
  HasTag : ∀ MT → IsChannelMessage (TagField :: MT)

record ChannelType : Set₁ where
  field
    channel-shape : InboxShape
    all-tagged : All IsChannelMessage channel-shape
```

When an actor reads from a channel it is actually reading from its inbox using selective receive. We capture this property by requiring that an instance of a channel comes with a proof of the messages types of the channel being a subset of the message types of the receiving actors inbox. Channel instances must furthermore know their identifier, but nothing more.

The subset of message types paired with a channel-identifier forms an instance of a channel, which we call `ChannelSession`. An actor receives messages from a channel via the function `from-channel`, which from the `ChannelSession` can create a `MessageFilter` and perform a selective receive. The received message is then converted from the actor's `InboxShape` to the channel's `InboxShape`.

```
record ChannelSession
         (channel : ChannelType)
         (receiver : InboxShape): Set₁ where
  field
    can-receive : (channel .channel-shape) <: receiver
    tag : UniqueTag


from-channel : ∀ {Γ i receiver} →
               ∀ ct →
               ChannelSession ct receiver →
               ∞ActorM i
                 receiver
                 (Message (ct .channel-shape))
                 Γ
                 (add-references Γ)
```

[3] An Agda program targeting the Haskell backend could create bindings to the Haskell library `https://hackage.haskell.org/package/uuid`

5.6 Initiating channels

For a channel to be useful other actors must know about it. The general pattern when an actor initiates a channel is thus that it sends a message, that contains a channel-identifier together with a reference, to another actor. We capture this idea in `ChannelInitiation`, which specifies the interface for setting up a channel from one actor to another.

```
data IsRequestMessage (IS : InboxShape) : MessageType → Set₁ where
  HasTag+Ref :
    ∀ MT →
    IsRequestMessage IS (TagField :: ReferenceType IS :: MT)

record ChannelInitiation : Set₁ where
  field
    request : InboxShape
    response : ChannelType
    request-tagged : All
                       (IsRequestMessage (response .channel-shape))
                       request
```

A `ChannelInitiation` is split into the interface for requests and the interface for the channel that will be responded to. A request is a message that has a channel-identifier as its first field and a reference to the channel as its second field. Similarly to the `channel-shape` of `ChannelType`, the type of messages in a `request` are constrained to those that have a channel-identifier as its first field and a reference of the channel type as its second field.

To apply the interface specified by a `ChannelInitiation` we must show that the calling actor can open the channel and that the callee can receive the request. Opening a channel is already covered by `ChannelSession` and the second obligation is just a subtype relation between the request interface and the callee.

```
record ChannelInitiationSession
         (ci : ChannelInitiation)
         (caller callee : InboxShape) : Set₁ where
  field
    can-request : (ci .request) <: callee
    response-session : ChannelSession (ci .response) caller
```

The most common scenario for initiating a channel is when the caller is the current actor. We have captured this scenario in the function `initiate-channel`, which sends a message containing the channel-identifier, a reference to itself, and any extra fields, to the caller. Its definition is placed in appendix D.

5.7 Synchronous call

In the beginning of this chapter we showed how Erlang/OTP uses selective receive to implement synchronous calls. After having translated that pattern into M_{act}, we noticed that synchronous calls and local channels had very similar implementations. Since singly typed actors does not capture protocols, the types for synchronous calls and local channels that can be captured by the type-system for . A synchronous call is in fact a local channel that is used to receive a single message.

We highlight in figure 5.2 how the protocol of Erlang/OTP's call corresponds to our functions for local channels. Figure 5.3 shows a straightforward translation of the pattern into Agda.

```
call : ∀ {Γ i caller} →
         (protocol : ChannelInitiation) →
         Request Γ caller protocol →
         ∞ActorM
           i
           caller
           (Message (protocol .response .channel-shape))
           Γ
           (add-references Γ)
call protocol request =
  let
    open ChannelInitiationSession
    open Request
    open ChannelSession
  in do
    initiate-channel _ request
    let rs = request .session .response-session
    from-channel (protocol .response) rs
```

Figure 5.2: Annotated version of Erlang/OTP' call

5.8 Active objects

The active object model, see section 3.2, replaces message passing with asynchronous method invocation. Messages are still sent and received in active objects, but it happens under the hood. To emulate active objects we must take control over the request-reply-loop and invoke the right method for each message. This is similar to the behaviour of Erlang/OTP's gen_server[4] which also takes control of the request-reply-loop, but uses a single method to handle every call.

Following the paradigm of object-oriented programming, an active object is an instantiation of a class. A class is a template that specifies the internal state and

[4] http://erlang.org/doc/man/gen_server.html

```
call : ∀ {Γ i caller} →
         (protocol : ChannelInitiation) →
         Request Γ caller protocol →
         ∞ActorM
           i
           caller
           (Message (protocol .response .channel-shape))
           Γ
           (add-references Γ)
call protocol request =
  let
    open ChannelInitiationSession
    open Request
    open ChannelSession
  in do
    initiate-channel _ request
    let rs = request .session .response-session
    from-channel (protocol .response) rs
```

Figure 5.3: Full implementation of call using the channel abstraction

behaviour of its instances. To emulate active objects we must create a similar specification that captures both the state of the object and its methods.

The methods of a class are essentially named fields that point to a function. These functions receive an implicit variable that is used to access the state of the object and to call the other methods of the object. Of these features, only named fields are inconvenient to emulate. We therefore choose to ignore names and instead place methods in an ordinary list.

A method consists of an interface, or method header, and a function body. It is common that a method returns a value, which corresponds to sending a reply in our model, but it can optionally return nothing. The request-response pattern is already captured by the `ChannelInitiation` from section 5.6 and a method that returns nothing poses no special constraints on the received message. A method head can thus be specified in terms of a `ChannelInitiation` when it returns a value and in terms of a sum of message types when it returns nothing.

```
data ActiveMethod : Set₁ where
  VoidMethod : InboxShape → ActiveMethod
  ResponseMethod : ChannelInitiation → ActiveMethod
```

We are required to construct a type for the inbox of the active object that captures every message type handled by its methods. Since an `InboxShape` is a list, we can do so by concatenating the interfaces of each method. We additionally want to give active objects the ability to call other actors, which will often have replies that do not match the message types of the calling actor's methods. This is captured by

including a set of additional message types in the specification of an active object. The interface for an actor that emulates an active object thus becomes

```
methods : List ActiveMethod
extra-messages : InboxShape
methods-shape methods ++ extra-messages
```

where the function `methods-shape` is defined as follows

```
active-method-request : ActiveMethod → InboxShape
active-method-request (VoidMethod x) = x
active-method-request (ResponseMethod x) = x .request

methods-shape : List ActiveMethod → InboxShape
methods-shape [] = []
methods-shape (am :: lci) =
  let rec = methods-shape lci
      am-shape = active-method-request am
  in am-shape ++ rec
```

We have captured method interfaces, but must also capture the notion of updateable state, and eventually method bodies. Wadler (1992) explains that state may be mimicked by a function that receives an initial state and returns its computed value paired with the final state: $State \longrightarrow (a, State)$. This idea is essential to our definition of method bodies.

The state of an active object must be able to capture references to other actors. In M_{act}, that means that we should be able to infer the actor's `TypingContext` from its state. And lo, our active objects include a function `vars` of type `state-type → TypingContext`. We use this function to encode the invariants of the method bodies, i. e. the precondition of a method is `(vars initial-state)` and the postcondition is `λ (a , new-state) → vars new-state .`

Having defined method interfaces, the interface for an active object actor, its state, and its invariants, the only remaining concept to define is method bodies. The details are different between the body for a `ResponseMethod` and a `VoidMethod`, but the handlers are in both cases functions of the following type:

```
(i : Size) →
(input : input-type) →
(state : state-type) →
∞ActorM i IS
  (return-type input)
  (precondition input state)
  (postcondition input)
```

The definitions of input-type, return-type, precondition, and postcondition all depend on whether the method is a `ResponseMethod` or a `VoidMethod`. Both

cases have computations that return state, but `ResponseMethod` additionally returns the message that should be sent as a response. Both cases have inputs that stem from the request message, but the input to a `ResponseMethod` is slightly modified in order to guarantee that it can not send additional response messages. With both input and output differing between the cases, invariants must also be defined in different ways. We will show how the definitions are derived by explaining the request-response-loop used in our active objects.

The overarching request-response-loop starts when a message targeted to one of the methods is received. We can locate which method has been targeted by looking at the message type tag. With the method located we can translate the message type from being expressed in terms of the actor's inbox to being expressed in terms of the method's interface. The message with its type translated, together with the current state, is applied to the method body. Eventually, the method returns an updated state and, in the case of it being a `ResponseMethod`, a message that is sent as reply.

Specializing the request-response-loop to `VoidMethod`s is simple. Since we know nothing about the request message and there is no response to send, the specialization has to deliver the request message unaltered and the method has to return nothing but the new state. For a `VoidMethod IS'` we thus have that:

```
input-type = Message IS'

return-type : input-type → Set₁
return-type _ = state-type

precondition : input-type → state-type → TypingContext
precondition input state = state-vars var-f input state

postcondition : (input : input-type) →
                return-type input → TypingContext
postcondition input state = state-vars var-f input state
```

where `state-vars` is defined as follows:

```
state-vars : ∀ {IS} {state-type : Set₁} →
               (state-type → TypingContext) →
               Message IS → state-type →
               TypingContext
state-vars vars input state = add-references (vars state) input
```

The specialization for `ResponseMethod` is a bit more involved. There are two factors that contribute to making it so: the method needs to return the new state paired with a response and we want to prevent the method from sending extra replies.

To prevent the method from sending extra replies we must hide the response reference from it. We must at the same time have the reference available when

the method returns, so that we can send the response. To solve this conundrum we can make it so the response reference is still threaded through the invariants of the method, but with the reference temporarily rendered unusable.

To make a reference temporarily unusable we can apply the idea of parametricity. Wadler (1989) famously describe how some behaviours of a polymorphic function can be inferred by its type alone, without inspecting the function body. Their insight is that the operations a function can perform are limited when it has to work for any type. In the same vein, a reference is unusable if its type can be anything.[5]

To turn the response reference into a parameter we must detach it from the request message. Using the `request-tagged` field from the method's `ChannelInitiation`, the function `active-request-type` strips the first two fields from every message type, removing the channel identifier and the reference. To make the invariants computable we include the type of the reference in the input, but this does not give back any power to the method.

```
record ResponseInput (ci : ChannelInitiation) : Set₁ where
  constructor _sent_
  field
    caller : InboxShape
    msg : Message (active-request-type ci)
```

We perform a similar transformation of the message types for responses, where `active-reply-type` strips the channel identifier from each message type. This transformed message type is used in the return value of the method, which pairs a `SendMessage` with the new state:

```
record ActiveReply (ci : ChannelInitiation)
                   (state-type : Set₁)
                   (vars : state-type → TypingContext)
                   (input : ResponseInput ci) : Set₁ where
  field
    new-state : state-type
    reply : SendMessage
              (active-reply-type (ci .response))
              (reply-state-vars vars input new-state)
```

The typing context used in `reply` is the same the as those we saw in the specialization for `VoidMethod`. The only difference between `reply-state-vars` and `state-vars` is the representation of the input. The same is true in the definitions below, where `reply-vars` just extracts the `state` field from the

[5] It is key to parametricity that the type of the polymorphic parameters can not be inferred by some other means. It *is* possible to pattern match on an `InboxShape` to learn how many message types it supports and whether the fields of the message are values or references. However, the types of value fields are polymorphic, so when a polymorphic `InboxShape` is broken down there is still no way to infer what values it supports.

`ActiveReply`. This lets us finally define the specialization for the bodies of `ResponseMethod`

```
input-type = ResponseInput ci

return-type : input-type → Set₁
return-type input = ActiveReply ci state-type var-f input

precondition : input-type → state-type → TypingContext
precondition input state = reply-state-vars var-f input state

postcondition : (input : input-type) →
                return-type input → TypingContext
postcondition input = reply-vars var-f input
```

Putting this all together we get our definition of an active object. An actor can at any time become an active object, provided its interface and invariants are correct, by invoking `run-active-object`. The behaviour that is captured by this implementation is one that arises naturally when writing actor programs and we were able to replace several implementations where control flow was earlier implemented manually. Furthermore, other actors can interact with the active object using the abstractions they feel fit, whether that is synchronous call, channels, or manual (selective) receive.

```
record ActiveObject : Set₂ where
  field
    state-type : Set₁
    vars : state-type → TypingContext
    methods : List ActiveMethod
    extra-messages : InboxShape
    handlers : All
                 (active-method
                   (methods-shape methods ++ extra-messages)
                   state-type
                   vars
                 )
                 methods
```

```
run-active-object : {i : Size} →
                    (ac : ActiveObject) →
                    (state : ac .state-type) →
                    ∞ActorM (↑ i)
                            (active-inbox-shape ac)
                            (ac .state-type)
                            ((ac .vars) state)
                            (ac .vars)
run-active-object ac state .force =
  receive-active-method ac ∞>>= λ { m .force →
  handle-active-method ac m state ∞>>= λ state' →
  run-active-object ac state'
  }
```

5.9 Guide to examples

We have written several programs to exercise the capabilities of M_{act} and the abstractions built on it. The code is available on GitHub[6] and contains information for setting up a local environment. We recommend that you explore the code interactively, using Emacs in agda2-mode[7], by introducing goals (write {!*term*!} around any term) and using commands like 'goal type and context' to see what types the terms in the surrounding context have.

Examples/PingPong 'Ping Pong', an actor 101 example, implemented without use of any abstractions. Messages that are received when an actor is in the wrong state are discarded.

Examples/SimpleActor The example from section 4.2. Designed to use every feature of `ActorM`, while still being simple.

Libraries/SelectiveReceive The implementation of SR_{lib}. Also available in appendix C.

Libraries/Call An implementation of synchronous calls in SR_{lib}. It is implemented as a stand-alone, without using a channel abstraction. There exists a similar implementation for SR_{prim}.

Selective/Examples/PingPong 'Ping Pong', now implemented using SR_{prim}. With selective receive in place, the actors do not have to throw away messages that arrive at an inconvenient moment.

[6] `https://github.com/Zalastax/singly-typed-actors` with the canonical commit hash being d7ed5ab556eb6d2a7921ea38aa91a5b8e2e236f1

[7] `http://agda.readthedocs.io/en/v2.5.3/tools/emacs-mode.html`

Selective/Examples/Fibonacci: The Fibonacci sequence implemented via two collaborating actors. The protocol[8] comes from the ABCD group's list of session types use cases[9], from which we also implement their bookstore and chat protocols.

Selective/Examples/Bookstore: A fairly faithful implementation of the 'Bookstore' protocol[10]. As we know, the type system for singly typed actors can not guarantee the absence of deadlocks, but it can still encode advanced protocols. Selective receive lets us write code where messages arriving in the wrong order is not a problem. The implementation was made before other abstractions were in place and therefore uses SR_{prim} without any further abstractions.

Selective/Examples/Chat: A fairly faithful implementation of the 'Chat' protocol[11]. This version was also made before other abstractions were in place. A reimplementation of the protocol, using active objects, also exists.

Selective/Libraries/Call: Synchronous call implemented without the channel abstraction. The synchronous call of SR_{lib} is a direct translation from this code.

Selective/Libraries/Channel: The implementation of channels that we have presented here.

Selective/Libraries/Call2: The implementation of synchronous call that we have presented here.

Selective/Libraries/ActiveObjects: The implementation of active objects that we have presented here.

Selective/Libraries/ReceiveSublist: Provides the function `receive-sublist` that lets an actor receive the messages that fall in a certain range of the actors `InboxShape`. I. e. for an actor with the `InboxShape` `xs ++ ys ++ zs` it will return `Message ys`. This function is used in active objects to only receive messages that are ment for its methods.

Selective/Examples/ChatAO: Reimplementation of the 'Chat' protocol, now using active objects. Active objects reduce a fair bit of boilerplate code, making the code that can use it easier to read.

[8] https://github.com/epsrc-abcd/session-types-use-cases/tree/master/fibonacci

[9] https://github.com/epsrc-abcd/session-types-use-cases/

[10] https://github.com/epsrc-abcd/session-types-use-cases/tree/master/Bookstore/description

[11] https://github.com/epsrc-abcd/session-types-use-cases/tree/master/Bookstore/description

Library testing: Our libraries come with simple programs that serve as minimal tests of the library's functionality. These test programs are: `Examples/TestSelectiveReceive`, `Examples/TestCall`, `Selective/Examples/TestCall`, `Selective/Examples/TestCall2`, `Selective/Examples/TestAO`.

6

Discussion

Modeling distributed systems is a difficult endeavour. A complete model should capture communication latency, failures, bandwidth, liveness, and so on. The singly typed actor model captures very few of these properties. However, this does not make the model a failure.

The singly typed actor model bridges the gap between actors and static type safety, and it does so using means that are available today. The industry has shown that singly typed actors can be practically implemented and Fowler, Lindley and Wadler (2017), followed by this thesis, show that they are theoretically sound as well.

The request-response pattern is very common in distributed programming. We have shown in chapter 5 how request-response can be encoded using selective receive and that the type of the response can be translated to the tight representation we expect. Unfortunately, having to capture the type of every message that should ever be received in a single type is still a significant nuisance.

The big issue in M_{act}, and in typing actors in general, is that actors have a single inbox. Even with selective receive, a single inbox per actor leads to wide types and a poor programming experience. These issues are very difficult to resolve and we instead argue for pursuing the local channels of Vyšniauskas (2015), even though they can be emulated in M_{act}. We believe that local channels can easily replace inboxes in the actor model and that doing so will result in simpler types.

While recommending local channels we want to highlight Vyšniauskas's (2015) conclusion that 'the full-ownership restriction does not mix well with the general formulation of the π-calculus'. Vyšniauskas (2015) ponders whether their π_{dist} would benefit from a formulation not based on the π-calculus, and to that we answer that λ_{act} (or M_{act}), extended with multiple inboxes, is a strong contender. λ_{act} is a calculus that is well-suited for distribution and it is easy to extend and work with. To researchers who wish to explore this route, we urge you to consider using Agda (or a similar language) in your mechanizations; dependent types are very well suited for this type of work. Dependent types gives you the freedom to create intricate typing relations, just by writing a function. This freedom lets you explore ideas about types quickly and relieves you from the burden of writing a special compiler that can capture these ideas.

The following sections present ideas for future work in the context of extending M_{act}.

6.1 Serialization

A distributed system involves values communicated between different computers and this inevitably requires serializing the data into a stream of bits (and deserializing it at the other end). For simplicity's sake we will assume the usage of Haskell's ByteString and access to Haskell's Binary class from the binary package. These assumptions lets us build on the work of Epstein, Black and Jones (2011) which requires that transmitted data implements the type class Serializable. The Serializable type class ensures that an item is Binary, i. e. it can be encoded and decoded to and from binary, and that it is Typeable, i. e. that we can get a concrete representation of its type. Agda uses an alternative to type classes called instance arguments (Devriese and Piessens 2011), where Serializable would be represented as a record that is provided similarly to implicit arguments.

The first operator in M_{act} that requires serialization is `Send`. For this we need to implement Serializable for messages, which requires serialization of the message's tag and fields. The tag is an index into a list of message types and can be serialized by first converting the index to a natural number. Serializing the message fields requires serialization of values and references. Values can be serialized by constraining them to only those that have an instance of Serializable, whereas references require a bit more thought.

References in M_{act} contain the name of the referenced actor and a subtyping relation. Names are just natural numbers, which we know how to serialize. The subtyping relation in M_{act} is a subset relation, which depending on your representation of subsets can be simple or very hard to serialize. A subset relation represented as a function between indices would be a representation that is difficult to serialize, whereas the representation we have, a list of indices into the other list, can be serialized as a list of integers.

The second operator that requires serialization is `Spawn`. `Spawn` needs `ActorM` to be serializable, which in the case of bind requires serialization of functions. Serializing functions is a complicated topic, in that it requires not only runtime support for serializing the code, but also requires serialization of free variables. Epstein, Black and Jones (2011) provide the solution of capturing closures as static code paired with its arguments as a serializable environment. Their solution can only mark code as static if it is a top-level definition, but we suspect that this is a small limitation in the context of actors. We have in fact already noted a natural tendency to define the spawned actor as its own top-level definition due to the spawned actor often having a fair amount of code.

6.2 Evolving interfaces

We have highlighted that request-response patterns lead to wide interfaces that must accommodate the needs of their actor for the actor's whole lifetime. Selective receive lets the actor handle only the messages that are relevant at the moment, but a wide interface still poses a significant nuisance that we would like to avoid.

The interface of an actor in M_{act} is fixed for its whole lifetime, making the model less flexible than a dynamically typed counterpart. It would make sense to let actors of M_{act} grow their interface in accordance to the subtyping rules we have already established since this would increase flexibility without invalidating old references.

To implement evolving interfaces one needs to consider what should happen when a message is sent via a reference that uses one of the older interfaces. The old references can not be modified, so the system will have to prove the subtype relation behind the scenes before the message is added to the inbox. We believe that the simplest method to accomplish this is to tag each message with a version number and to remember every evolution that an actor has performed. The version number can then be used to extract which evolutions lie between the message's interface and the current one. The extracted evolutions thus form a path of subtyping relations that can be applied one after another, thanks to transitivity of subtyping.

If we could ignore old references it would make sense to also let an interface shrink. A long running actor would then be able to temporarily extend its capabilities in order to follow a protocol and when the protocol is complete, forget this temporary extension to decrease noise in the types. However, if the other actors do not follow the protocol, or if the protocol is bad, the actor that shrunk its interface might get sent a message that it no longer understands. Implementing shrinking interfaces would thus require either adding some run-time checks à la dynamic typing, or modelling protocols as types as well, e. g. using session types.

We are not sure how well statically typed sessions can be adapted to actors with a single inbox since there is a big risk that types will easily become unwieldy. Current research has focused on the channels of π-calculus, which instead leads to problems with distribution. We believe that these problems can both be avoided by taking the best of both models: the distributability of actors and the granular typing of channels. This idea is somewhat explored already in Epstein, Black and Jones's (2011) Cloud Haskell, which supports typed channels that are fully local.

The need for multiple inboxes is also discussed by de'Liguoro and Padovani (2018). They present the *mailbox calculus*—a mild extension of the asynchronous π-calculus—and *mailbox types*—a new kind of behavioral types. We find it especially interesting that their model subsumes actors and that their approach can compositionally ensure mailbox conformance and deadlock freedom of a system. In contrast, a multiparty session approach, such as that by Charalambides, Dinges and Agha (2016), requires global types that are then extracted into local ones. We

are eager to see what future work will spawn from their mailbox approach, especially the possibility of efficiently distributable and type safe programming models.

6.3 Frame rule

When computations are chained in M_{act}, the postcondition of a computation must match exactly with the precondition of the following computation. A function that is not written with this idea in mind could therefore end up not being callable even though every variable it needs is in the context, just not in the right place. We have written two similar functions that illustrate this problem. The function `send-nat` is written with invariants in mind, whereas `send-nat-frame` is explicit about a specific shape of the variable context.

```
NatMessage : MessageType
NatMessage = [ ValueType ℕ ]¹

send-nat : ∀ {i IS ToIS pre} →
             (canSendTo : ToIS ∈ pre) →
             (NatMessage ∈ ToIS) →
             ∞ActorM (↑ i) IS T₁ pre (λ _ → pre)
send-nat canSendTo p = canSendTo ![t: p ] [ lift 42 ]ᵃ

send-nat-frame : ∀ {i IS ToIS} →
                   (NatMessage ∈ ToIS) →
                   ∞ActorM (↑ i) IS T₁ [ ToIS ]¹ (λ _ → [ ToIS ]¹)
send-nat-frame p = Z ![t: p ] ([ lift 42 ]ᵃ)
```

We added the `Strengthen` operator which lets us forget about variables, but using `Strengthen` to call `send-nat-frame` results in losing the other variables forever. We see from this example that M_{act} lacks an operator for local reasoning, i. e. the *frame rule* from separation logic.

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \mathsf{Modifies}(C) \cap \mathsf{Free}(R) = \emptyset$$

Figure 6.1: Frame rule

The frame rule, see figure 6.1, codifies a notion of local behaviour (O'Hearn, Reynolds and Yang 2001). The idea is that the precondition P in $\{P\}C\{Q\}$ specifies the properties that are sufficient for C to run and that running C with precondition P will establish the postcondition Q. A program that executes safely in the small state P can also execute in a bigger state ($P*R$), and this execution will not affect the additional part of the state ($Q * R$), as long as the variables modified ($\mathsf{Modifies}(C)$) are not occurring in the free variables of R.

Adding a frame rule operator to M_{act} would provide a means to call functions that are written without invariants in mind. The frame rule would thus let us call `send-nat-frame` without losing variables. The frame rule could also simplify functions that have invariants in mind by alleviating them from specifying variables that are irrelevant for the function. E. g. in SR_{lib}, Γ is not used by `selective-receive` and could be left out.

6.4 Time, failures, and delays

We have conveniently ignored some important but difficult problems when designing M_{act}, namely timeouts, failures, and delays between sending and receiving messages. Adding a clock to M_{act} should be fairly simple, but you have to decide whether it should use a wall-clock, logical local time, or logical global time (Raynal and Singhal 1996). The receive construct should also be extended with a timeout facility, which gives back control to the actor if no message was received in the timeout interval.

Failure detection and timeouts are heavily connected in asynchronous distributed systems. In fact, Chandy and Misra (1986) showed that detection of process failure is impossible without using timeouts. Errors not related to processes being unreachable should be possible to model using the operations already available in M_{act}, e. g. by the usage of error messages. We therefore suggest that extensions of M_{act} should focus on adding timeouts, leaving other error handling to be implemented as a library.

Even though M_{act} is asynchronous, to call it distributed would not be entirely honest. What M_{act} fails to model is that every non-local action has to be distributed to its target and that distribution has an inherent latency. To make M_{act} properly distributed, sending a message should be separated from delivering the message to the inbox, so that latency can be properly accounted for.

With each message having an independent latency, messages between actors could arrive in a different order from the order they were sent in. Agha (1990) shows that such arrival order non-determinism can easily be handled and they propose a scheme for restoring order of communication. Their solution is that if actor a wish that actor b should process messages in the same order as a sent them, then a should tag each message with an ordinal that gets increased for each message. The actor b in turn remembers how many messages it has processed and buffers messages that arrive before the message it is supposed to process next. Alternatively, the model could of course be extended to buffer and reorder messages behind the scenes.

7

Conclusion

We have seen how the concept of a coinductive parameterized monad can be used to model actors in Agda (section 4.2). In section 4.8 we explain the semantics of M_{act}, showing how the model captures type-safety if we ignore the possibility of deadlocks.

Shallowly embedding M_{act} in Agda was a powerful move which has allowed us to explore problems that are relevant and of significant size. Dependent types in particular have let us describe advanced abstractions without resorting to tricks, which one often has to do in less powerful type systems.

We demonstrated how selective receive simplifies control flow and can be used to build powerful abstractions (chapter 5). This shows that our intuition that selective receive can emulate many communication patterns was correct, but we can still not call that result a complete success. In chapter 6 we describe why a single inbox per actor is problematic, even when paired with selective receive. We suggest that the actor model should be extended to support multiple inboxes or local channels, resolving the type granularity problem without compromising distributability.

Exploring local channels is the future work we are most excited about. In particular, we would like to see the mailbox types (de'Liguoro and Padovani 2018) adapted to a programming model—perhaps as an EDSL in Agda.

Bibliography

Abel, Andreas (2010). 'MiniAgda: Integrating Sized and Dependent Types'. In: *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*. Ed. by Ekaterina Komendantskaya, Ana Bove and Milad Niqui. Vol. 5. EPiC Series. EasyChair, pp. 18–32. URL: http://www.easychair.org/publications/paper/51657.

Abel, Andreas and James Chapman (2014). 'Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types'. In: *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*. Ed. by Paul Levy and Neel Krishnaswami. Vol. 153. EPTCS, pp. 51–67. DOI: 10.4204/EPTCS.153.4. URL: https://doi.org/10.4204/EPTCS.153.4.

Abel, Andreas and Brigitte Pientka (2013). 'Wellfounded recursion with copatterns: a unified approach to termination and productivity'. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, pp. 185–196. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500591. URL: http://doi.acm.org/10.1145/2500365.2500591.

Agha, Gul A. (1990). *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press. ISBN: 978-0-262-01092-4.

Atkey, Robert (2009). 'Parameterised notions of computation'. In: *J. Funct. Program.* 19.3-4, pp. 335–376. DOI: 10.1017/S095679680900728X. URL: https://doi.org/10.1017/S095679680900728X.

Bengtson, Jesper and Joachim Parrow (2009). 'Formalising the pi-calculus using nominal logic'. In: *Logical Methods in Computer Science* 5.2. URL: http://arxiv.org/abs/0809.3960.

Berghofer, Stefan and Christian Urban (2007). 'A Head-to-Head Comparison of de Bruijn Indices and Names'. In: *Electr. Notes Theor. Comput. Sci.* 174.5, pp. 53–67.

DOI: 10.1016/j.entcs.2007.01.018. URL: https://doi.org/10.1016/j.entcs.2007.01.018.

Bove, Ana, Peter Dybjer and Ulf Norell (2009). 'A Brief Overview of Agda - A Functional Language with Dependent Types'. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, pp. 73–78. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_6. URL: https://doi.org/10.1007/978-3-642-03359-9_6.

Brady, Edwin (2015). *Embedded Domain Specific Languages in Idris*. URL: https://www.cs.ox.ac.uk/projects/utgp/school/idris-tutorial.pdf.

Chandy, K. Mani and Jayadev Misra (1986). 'How Processes Learn'. In: *Distributed Computing* 1.1, pp. 40–52. DOI: 10.1007/BF01843569. URL: https://doi.org/10.1007/BF01843569.

Charalambides, Minas, Peter Dinges and Gul A. Agha (2016). 'Parameterized, concurrent session types for asynchronous multi-actor interactions'. In: *Sci. Comput. Program.* 115-116, pp. 100–126. DOI: 10.1016/j.scico.2015.10.006. URL: https://doi.org/10.1016/j.scico.2015.10.006.

Charousset, Dominik, Raphael Hiesgen and Thomas C. Schmidt (2016). 'Revisiting actor programming in C++'. In: *Computer Languages, Systems & Structures* 45, pp. 105–131. DOI: 10.1016/j.cl.2016.01.002. URL: https://doi.org/10.1016/j.cl.2016.01.002.

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, October 26, 2015*. Ed. by Elisa Gonzalez Boix et al. ACM, pp. 1–12. ISBN: 978-1-4503-3901-8. DOI: 10.1145/2824815.2824816. URL: http://doi.acm.org/10.1145/2824815.2824816.

Coquand, Thierry (1993). 'Infinite Objects in Type Theory'. In: *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*. Ed. by Henk Barendregt and Tobias Nipkow. Vol. 806. Lecture Notes in Computer Science. Springer, pp. 62–78. ISBN: 3-540-58085-9. DOI: 10.1007/3-540-58085-9_72. URL: https://doi.org/10.1007/3-540-58085-9_72.

Dardha, Ornela, Elena Giachino and Davide Sangiorgi (2017). 'Session types revisited'. In: *Inf. Comput.* 256, pp. 253–286. DOI: 10.1016/j.ic.2017.06.002. URL: https://doi.org/10.1016/j.ic.2017.06.002.

de Boer, Frank S. et al. (2017). 'A Survey of Active Object Languages'. In: *ACM Comput. Surv.* 50.5, 76:1–76:39. DOI: 10.1145/3122848. URL: http://doi.acm.org/10.1145/3122848.

De Koster, Joeri, Tom Van Cutsem and Wolfgang De Meuter (2016). '43 years of actors: a taxonomy of actor models and their key properties'. In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*. Ed. by Sylvan Clebsch et al. ACM, pp. 31–40. ISBN: 978-1-4503-4639-9. DOI: 10.1145/3001886.3001890. URL: http://doi.acm.org/10.1145/3001886.3001890.

de'Liguoro, Ugo and Luca Padovani (2018). 'Mailbox Types for Unordered Interactions'. In: *CoRR* abs/1801.04167. arXiv: 1801.04167. URL: http://arxiv.org/abs/1801.04167.

Devriese, Dominique and Frank Piessens (2011). 'On the bright side of type classes: instance arguments in Agda'. In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu and Olivier Danvy. ACM, pp. 143–155. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034796. URL: http://doi.acm.org/10.1145/2034773.2034796.

Dijkstra, Edsger W. (1975). 'Guarded Commands, Nondeterminacy and Formal Derivation of Programs'. In: *Commun. ACM* 18.8, pp. 453–457. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

Epstein, Jeff, Andrew P. Black and Simon L. Peyton Jones (2011). 'Towards Haskell in the cloud'. In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, pp. 118–129. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034690. URL: http://doi.acm.org/10.1145/2034675.2034690.

Fowler, Simon (2016). 'An Erlang Implementation of Multiparty Session Actors'. In: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*. Ed. by Massimo Bartoletti et al. Vol. 223. EPTCS, pp. 36–50. DOI: 10.4204/EPTCS.223.3. URL: https://doi.org/10.4204/EPTCS.223.3.

Fowler, Simon, Sam Lindley and Philip Wadler (2017). 'Mixing Metaphors: Actors as Channels and Channels as Actors'. In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik,

11:1–11:28. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.11. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.11.

Geuvers, Herman (2009). 'Proof assistants: History, ideas and future'. In: *Sadhana* 34.1, pp. 3–25.

Gifford, David K. and John M. Lucassen (1986). 'Integrating Functional and Imperative Programming'. In: *LISP and Functional Programming*, pp. 28–38.

Haller, Philipp (2012). 'On the integration of the actor model in mainstream technologies: the scala perspective'. In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*. Ed. by Gul A. Agha et al. ACM, pp. 1–6. ISBN: 978-1-4503-1630-9. DOI: 10.1145/2414639.2414641. URL: http://doi.acm.org/10.1145/2414639.2414641.

Haller, Philipp and Martin Odersky (2009). 'Scala Actors: Unifying thread-based and event-based programming'. In: *Theor. Comput. Sci.* 410.2-3, pp. 202–220. DOI: 10.1016/j.tcs.2008.09.019. URL: https://doi.org/10.1016/j.tcs.2008.09.019.

Hawblitzel, Chris et al. (2017). 'IronFleet: proving safety and liveness of practical distributed systems'. In: *Commun. ACM* 60.7, pp. 83–92. DOI: 10.1145/3068608. URL: http://doi.acm.org/10.1145/3068608.

He, Jiansen (2014). 'Type-parameterized actors and their supervision'. PhD thesis.

He, Jiansen, Philip Wadler and Philip W. Trinder (2014). 'Typecasting actors: from Akka to TAkka'. In: *Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28-29, 2014*. Ed. by Philipp Haller and Heather Miller. ACM, pp. 23–33. ISBN: 978-1-4503-2868-5. DOI: 10.1145/2637647.2637651. URL: http://doi.acm.org/10.1145/2637647.2637651.

Hewitt, Carl, Peter Boehler Bishop and Richard Steiger (1973). 'A Universal Modular ACTOR Formalism for Artificial Intelligence'. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. Ed. by Nils J. Nilsson. William Kaufmann, pp. 235–245. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

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

– (1978). 'Communicating Sequential Processes'. In: *Commun. ACM* 21.8, pp. 666–677. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

Honda, Kohei (1993). 'Types for Dyadic Interaction'. In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, pp. 509–523. ISBN: 3-540-57208-2. DOI: 10.1007/3-540-57208-2_35. URL: https://doi.org/10.1007/3-540-57208-2_35.

Honda, Kohei, Vasco Thudichum Vasconcelos and Makoto Kubo (1998). 'Language Primitives and Type Discipline for Structured Communication-Based Programming'. In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, pp. 122–138. ISBN: 3-540-64302-8. DOI: 10.1007/BFb0053567. URL: https://doi.org/10.1007/BFb0053567.

Honda, Kohei, Nobuko Yoshida and Marco Carbone (2008). 'Multiparty asynchronous session types'. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, pp. 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472. URL: http://doi.acm.org/10.1145/1328438.1328472.

Hu, Raymond, Nobuko Yoshida and Kohei Honda (2008). 'Session-Based Distributed Programming in Java'. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, pp. 516–541. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_22. URL: https://doi.org/10.1007/978-3-540-70592-5_22.

Igried, Bashar and Anton Setzer (2016). 'Programming with monadic CSP-style processes in dependent type theory'. In: *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*. Ed. by James Chapman and Wouter Swierstra. ACM, pp. 28–38. ISBN: 978-1-4503-4435-7. DOI: 10.1145/2976022.2976032. URL: http://doi.acm.org/10.1145/2976022.2976032.

Johnsen, Einar Broch, Reiner Hähnle et al. (2010). 'ABS: A Core Language for Abstract Behavioral Specification'. In: *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*. Ed. by Bernhard K. Aichernig, Frank S. de Boer and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science.

Springer, pp. 142–164. ISBN: 978-3-642-25270-9. DOI: 10.1007/978-3-642-25271-6_8. URL: https://doi.org/10.1007/978-3-642-25271-6_8.

Johnsen, Einar Broch, Olaf Owe and Marte Arnestad (2003). 'Combining active and reactive behavior in concurrent objects'. In: Citeseer.

Kobayashi, Naoki, Benjamin C. Pierce and David N. Turner (1999). 'Linearity and the pi-calculus'. In: *ACM Trans. Program. Lang. Syst.* 21.5, pp. 914–947. DOI: 10.1145/330249.330251. URL: http://doi.acm.org/10.1145/330249.330251.

Levy, Paul Blain, John Power and Hayo Thielecke (2003). 'Modelling environments in call-by-value programming languages'. In: *Inf. Comput.* 185.2, pp. 182–210. DOI: 10.1016/S0890-5401(03)00088-9. URL: https://doi.org/10.1016/S0890-5401(03)00088-9.

Maksimovic, Petar and Alan Schmitt (2015). 'HOCore in Coq'. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, pp. 278–293. ISBN: 978-3-319-22101-4. DOI: 10.1007/978-3-319-22102-1_19. URL: https://doi.org/10.1007/978-3-319-22102-1_19.

Martin, Robert C (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Martin-Löf, Per and Giovanni Sambin (1984). *Intuitionistic type theory*. Vol. 9. Bibliopolis Napoli.

McBride, Conor and James McKinna (2004). 'The view from the left'. In: *J. Funct. Program.* 14.1, pp. 69–111. DOI: 10.1017/S0956796803004829. URL: https://doi.org/10.1017/S0956796803004829.

Milner, Robin, Joachim Parrow and David Walker (1992). 'A Calculus of Mobile Processes, I'. In: *Inf. Comput.* 100.1, pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4. URL: https://doi.org/10.1016/0890-5401(92)90008-4.

Moggi, Eugenio (1991). 'Notions of Computation and Monads'. In: *Inf. Comput.* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4.

Mostrous, Dimitris and Vasco Thudichum Vasconcelos (2011). 'Session Typing for a Featherweight Erlang'. In: *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*. Ed. by Wolfgang De Meuter and Gruia-Catalin Roman. Vol. 6721. Lecture Notes in Computer Science. Springer, pp. 95–109. ISBN: 978-3-

642-21463-9. DOI: 10.1007/978-3-642-21464-6_7. URL: https://doi.org/10.1007/978-3-642-21464-6_7.

Musser, David R. and Carlos A. Varela (2013). 'Structured reasoning about actor systems'. In: *Proceedings of the 2013 Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*. Ed. by Nadeem Jamali et al. ACM, pp. 37–48. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541334. URL: http://doi.acm.org/10.1145/2541329.2541334.

Nanevski, Aleksandar et al. (2008). 'Ynot: dependent types for imperative programs'. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by James Hook and Peter Thiemann. ACM, pp. 229–240. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411237. URL: http://doi.acm.org/10.1145/1411204.1411237.

Neykova, Rumyana and Nobuko Yoshida (2017). 'Multiparty Session Actors'. In: *Logical Methods in Computer Science* 13.1. DOI: 10.23638/LMCS-13(1:17)2017. URL: https://doi.org/10.23638/LMCS-13(1:17)2017.

Nordström, Bengt, Kent Petersson and Jan M Smith (1990). *Programming in Martin-Löf's type theory*. Oxford University Press.

Norell, Ulf (2008). 'Dependently typed programming in Agda'. In: *International School on Advanced Functional Programming*. Springer, pp. 230–266.

O'Hearn, Peter W., John C. Reynolds and Hongseok Yang (2001). 'Local Reasoning about Programs that Alter Data Structures'. In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, pp. 1–19. ISBN: 3-540-42554-3. DOI: 10.1007/3-540-44802-0_1. URL: https://doi.org/10.1007/3-540-44802-0_1.

Perera, Roly and James Cheney (2016). 'Proof-relevant pi-calculus'. In: *CoRR* abs/1604.04575. arXiv: 1604.04575. URL: http://arxiv.org/abs/1604.04575.

Pierce, Benjamin C. (1997). 'Foundational Calculi for Programming Languages'. In: *The Computer Science and Engineering Handbook*. Ed. by Allen B. Tucker. CRC Press, pp. 2190–2207. ISBN: 0-8493-2909-4.

– (2002). *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8.

Raynal, Michel and Mukesh Singhal (1996). 'Logical Time: Capturing Causality in Distributed Systems'. In: *IEEE Computer* 29.2, pp. 49–56. DOI: 10.1109/2.485846. URL: https://doi.org/10.1109/2.485846.

Sirjani, Marjan et al. (2004). 'Modeling and Verification of Reactive Systems using Rebeca'. In: *Fundam. Inform.* 63.4, pp. 385–410. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05.

Srinivasan, Sriram and Alan Mycroft (2008). 'Kilim: Isolation-Typed Actors for Java'. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, pp. 104–128. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_6. URL: https://doi.org/10.1007/978-3-540-70592-5_6.

Summers, Alexander J. and Peter Müller (2016). 'Actor Services - Modular Verification of Message Passing Programs'. In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, pp. 699–726. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1_27. URL: https://doi.org/10.1007/978-3-662-49498-1_27.

Svenningsson, Josef and Emil Axelsson (2012). 'Combining Deep and Shallow Embedding for EDSL'. In: *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Vol. 7829. Lecture Notes in Computer Science. Springer, pp. 21–36. ISBN: 978-3-642-40446-7. DOI: 10.1007/978-3-642-40447-4_2. URL: https://doi.org/10.1007/978-3-642-40447-4_2.

Takeuchi, Kaku, Kohei Honda and Makoto Kubo (1994). 'An Interaction-based Language and its Typing System'. In: *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*. Ed. by Constantine Halatsis et al. Vol. 817. Lecture Notes in Computer Science. Springer, pp. 398–413. ISBN: 3-540-58184-7. DOI: 10.1007/3-540-58184-7_118. URL: https://doi.org/10.1007/3-540-58184-7_118.

Turbak, Franklyn, David Gifford and Mark A Sheldon (2008). *Design concepts in programming languages*. MIT press.

Vitek, Jan, ed. (2008). *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Vol. 5142. Lecture Notes in Computer Science. Springer. ISBN: 978-3-540-70591-8. DOI: 10.1007/

978-3-540-70592-5. URL: https://doi.org/10.1007/978-3-540-70592-5.

Vyšniauskas, Ignas (2015). 'πdist: Towards a Typed π-calculus for Distributed Programming Languages'. MA thesis. Universiteit van Amsterdam.

Wadler, Philip (1989). 'Theorems for Free!' In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. Ed. by Joseph E. Stoy. ACM, pp. 347–359. ISBN: 0-201-51389-7. DOI: 10.1145/99370.99404. URL: http://doi.acm.org/10.1145/99370.99404.

– (1992). 'Monads for functional programming'. In: *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. Ed. by Manfred Broy. Vol. 118. NATO ASI Series. Springer, pp. 233–264. ISBN: 978-3-642-08164-4. DOI: 10.1007/978-3-662-02880-3_8. URL: https://doi.org/10.1007/978-3-662-02880-3_8.

Wilcox, James R. et al. (2015). 'Verdi: a framework for implementing and formally verifying distributed systems'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, pp. 357–368. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737958. URL: http://doi.acm.org/10.1145/2737924.2737958.

Wright, Andrew K. and Matthias Felleisen (1994). 'A Syntactic Approach to Type Soundness'. In: *Inf. Comput.* 115.1, pp. 38–94. DOI: 10.1006/inco.1994.1093. URL: https://doi.org/10.1006/inco.1994.1093.

Yasutake, Shohei and Takuo Watanabe (2015). *Actario: A framework for reasoning about actor systems.*

Yonezawa, Akinori, Jean-Pierre Briot and Etsuya Shibayama (1986). 'Object-Oriented Concurrent Programming in ABCL/1'. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, pp. 258–268. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28722. URL: http://doi.acm.org/10.1145/28697.28722.

Yoshida, Nobuko et al. (2013). 'The Scribble Protocol Language'. In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*. Ed. by Martín Abadi and Alberto Lluch-Lafuente. Vol. 8358. Lecture Notes in Computer Science. Springer, pp. 22–41. ISBN: 978-3-319-05118-5. DOI: 10.1007/978-3-319-05119-2_3. URL: https://doi.org/10.1007/978-3-319-05119-2_3.

A

Examples of monadic computation

Example program for the identity monad:

```
bind-suc : ℕ → MonadID ℕ
bind-suc n = return (suc n)

is-3+ : ℕ → MonadID Bool
is-3+ 0 = return false
is-3+ 1 = return false
is-3+ 2 = return false
is-3+ _ = return true

prog : ℕ → MonadID Bool
prog n =
       return n >>=
       bind-suc >>=
       bind-suc >>=
       is-3+

run-0 : run-monadID (prog 0) ≡ false
run-0 = refl

run-1 : run-monadID (prog 1) ≡ true
run-1 = refl
```

Example program for the Maybe monad:

```
_safe-div_ : ℕ → ℕ → Maybe ℕ
dividend safe-div zero = nothing
dividend safe-div (suc divisor) = just (dividend div suc divisor)

divmania : ℕ → ℕ → ℕ → ℕ → Maybe ℕ
divmania a b c d = (a safe-div b) >>= λ a/b →
                   (c safe-div d) >>= λ c/d →
                   a/b safe-div c/d

result-0 : divmania 0 1 1 1 ≡ just 0
result-0 = refl
```

```
result-1 : divmania 100 3 10 3 ≡ just 11
result-1 = refl

result-2 : divmania 10 3 2 5 ≡ nothing
result-2 = refl
```

B

Actor helper functions

```
--
-- ========== Helpers for ActorM ==========
--

open ∞ActorM public

-- coinduction helper for Value
return₁ : {A : Set (lsuc lzero)} → ∀ {i IS post} → (val : A) →
          ∞ActorM i IS A (post val) post
return₁ val .force = Return val

-- universe lifting for return₁
return : {A : Set} → ∀ {i IS post} → (val : A) →
         ∞ActorM i IS (Lift A) (post (lift val)) post
return val = return₁ (lift val)

_>>=_ : ∀ {i IS A B pre mid post} → (m : ∞ActorM i IS A pre mid) →
  (f : (x : A) → ∞ActorM i IS B (mid x) (post)) →
  ∞ActorM i IS B pre post
(m >>= f) .force = m ∞>>= f

_∞>>_ : ∀ {i IS A B pre mid post} → (m : ∞ActorM i IS A pre (λ _ → mid)) →
  (n : ∞ActorM i IS B mid post) →
  ActorM i IS B pre post
m ∞>> n = m ∞>>= λ _ → n

[mid:_]_>>=_ : ∀ {i IS A B pre post} → ∀ mid →
  (m : ∞ActorM i IS A pre mid) →
  (f : (x : A) → ∞ActorM i IS B (mid x) (post)) →
  ∞ActorM i IS B pre post
[mid: mid ] m >>= f = _>>=_ {mid = mid} m f

_>>_ : ∀ {i IS A B pre mid post} → (m : ∞ActorM i IS A pre (λ _ → mid)) →
  (n : ∞ActorM i IS B mid post) →
  ∞ActorM i IS B pre post
(m >> n) .force = m ∞>> n

[mid:_]_>>_ : ∀ {i IS A B pre post} → ∀ mid →
  (m : ∞ActorM i IS A pre (λ _ → mid)) →
```

```
  (n : ∞ActorM i IS B mid (post)) →
  ∞ActorM i IS B pre post
[mid: mid ] m >> f = _>>_ {mid = mid} m f

-- coinduction helper for spawn
spawn : ∀ {i IS NewIS A pre postN} →
  ActorM i NewIS A [] postN →
  ∞ActorM (↑ i) IS T₁ pre λ _ → NewIS :: pre
spawn newAct .force = Spawn newAct

-- spawn potentially infinite actors
spawn∞ : ∀ {i IS NewIS A pre postN} →
  ∞ActorM (↑ i) NewIS A [] postN →
  ∞ActorM (↑ i) IS T₁ pre λ _ → NewIS :: pre
spawn∞ newAct = spawn (newAct .force)

-- coinduction helper and neater syntax for message sending
_![t:_]_ : ∀ {i IS ToIS pre MT} →
  (canSendTo : ToIS ∈ pre) →
  (MT ∈ ToIS) →
  All (send-field-content pre) MT →
  ∞ActorM (↑ i) IS T₁ pre (λ _ → pre)
(canSendTo ![t: p ] fields) .force = Send canSendTo (SendM p fields)

-- coinduction helper for Receive
receive : ∀ {i IS pre} → ∞ActorM (↑ i) IS (Message IS) pre (add-references pre)
receive .force = Receive

self : ∀ {i IS pre} → ∞ActorM (↑ i) IS T₁ pre (λ _ → IS :: pre)
self .force = Self

-- coinduction helper for Strengthen
strengthen : ∀ {i IS xs ys} → ys ⊆ xs → ∞ActorM (↑ i) IS T₁ xs (λ _ → ys)
strengthen inc .force = Strengthen inc

begin : ∀ {i IS A pre post} → ∞ActorM i IS A pre post → ActorM i IS A pre post
begin = _∞>>_ (return tt)

⊠-of-values : List Set → InboxShape
⊠-of-values [] = []
⊠-of-values (x :: vs) = ([ ValueType x ]¹) :: ⊠-of-values vs
```

C

Full implementation of selective receive as a library

This constitutes the full implementation of selective receive as a library. The bulk of the code is concerned with moving references in the variable context, and if we ignore that part of the code `selective-receive` is actually fairly short.

```
module Libraries.SelectiveReceive where

open import ActorMonad public
open import Prelude
open import Data.List.Properties using (++-assoc ; ++-identityʳ)

waiting-refs : ∀ {IS} → (q : List (Message IS)) → ReferenceTypes
waiting-refs [] = []
waiting-refs (x :: q) = add-references (waiting-refs q) x

record SplitList {a : Level} {A : Set a} (ls : List A) : Set (lsuc a) where
  field
    heads : List A
    el : A
    tails : List A
    is-ls : (heads) ++ (el :: tails) ≡ ls

matches-filter : ∀{a} {A : Set a} → (f : A → Bool) → A → Set
matches-filter f v = f v ≡ true

misses-filter : ∀{a} {A : Set a} → (f : A → Bool) → A → Set
misses-filter f v = f v ≡ false

data FoundInList {a : Level} {A : Set a}
                 (ls : List A)
                 (f : A → Bool) :
                 Set (lsuc a) where
  Found : (split : SplitList ls) →
          (matches-filter f (SplitList.el split)) →
          FoundInList ls f
  Nothing : All (misses-filter f) ls →
            FoundInList ls f
```

```
find-split : {a : Level} {A : Set a}
             (ls : List A) (f : A → Bool) →
             FoundInList ls f
find-split [] f = Nothing []
find-split (x :: ls) f with (f x) | (inspect f x)
... | false | [ neq ] = add-x (find-split ls f)
  where
    add-x : FoundInList ls f → FoundInList (x :: ls) f
    add-x (Found split el-is-ok) = Found (record {
      heads = x :: heads split
      ; el = el split
      ; tails = tails split
      ; is-ls = cong (_::_ x) (is-ls split)
      }) el-is-ok
      where open SplitList
    add-x (Nothing ps) = Nothing (neq :: ps)
... | true | [ eq ] = Found (record {
      heads = []
      ; el = x
      ; tails = ls
      ; is-ls = refl
      }) eq

add-references++ : ∀ {IS} → (xs ys : ReferenceTypes) →
                   (x : Message IS) →
                    add-references (xs ++ ys) x ≡
                    add-references xs x ++ ys
add-references++ xs ys (Msg {MT} x x₁) =
                  sym (++-assoc (extract-references MT) xs ys)

waiting-refs++ : ∀ {IS} → (xs ys : List (Message IS)) →
                          waiting-refs (xs ++ ys) ≡
                          waiting-refs xs ++ waiting-refs ys
waiting-refs++ [] _ = refl
waiting-refs++ (x :: xs) ys with (waiting-refs++ xs ys)
... | q with (cong (λ qs → add-references qs x) q)
... | r = trans r (halp xs ys x)
  where
    halp : ∀ {IS} → (xs ys : List (Message IS)) (x : Message IS) →
      add-references (waiting-refs xs ++ waiting-refs ys) x ≡
      add-references (waiting-refs xs) x ++ waiting-refs ys
    halp xs ys x = add-references++ (waiting-refs xs) (waiting-refs ys) x

move-received : ∀ {IS} → ∀ pre →
                (q : List (Message IS)) →
                (x : Message IS) →
                  (pre ++ (waiting-refs (q ++ [ x ]¹))) ⊆
                  (add-references (pre ++ waiting-refs q) x)
move-received pre q (Msg {MT} x x₁) rewrite
                    (waiting-refs++ q [ Msg x x₁ ]¹) |
```

```
                    (++-identityʳ (extract-references MT)) =
                    ⊆-trans move-1 move-2
  where
    move-1 : (pre ++ waiting-refs q ++ extract-references MT) ⊆
             ((pre ++ waiting-refs q) ++ extract-references MT)
    move-1 = ⊆++comm' pre (waiting-refs q) (extract-references MT)
    move-2 : ((pre ++ waiting-refs q) ++ extract-references MT) ⊆
             (extract-references MT ++ (pre ++ waiting-refs q))
    move-2 = ⊆-move (pre ++ waiting-refs q) (extract-references MT)

accept-received : ∀ {IS} → ∀ pre →
                  (q : List (Message IS)) →
                  (x : Message IS) →
                    (add-references pre x ++ waiting-refs q) ⊆
                    (add-references (pre ++ waiting-refs q) x)
accept-received pre q (Msg {MT} x x₁) = ⊆++comm
                                          (extract-references MT)
                                          pre
                                          (waiting-refs q)

open SplitList

accept-found : ∀ {IS} → ∀ Γ →
               (q : List (Message IS)) →
               (split : SplitList q) →
                  (add-references Γ (el split) ++
                    waiting-refs (heads split ++ tails split)) ⊆
                  (Γ ++ waiting-refs q)
accept-found Γ q record {
  heads = heads
  ; el = Msg {MT} x y
  ; tails = tails
  ; is-ls = is-ls
  } rewrite (sym is-ls) =
  ⊆-trans
    (⊆-inc
      (extract-references MT ++ Γ)
      (Γ ++ extract-references MT)
      (waiting-refs (heads ++ tails))
      (⊆-move (extract-references MT) Γ)
    )
    (⊆-trans
      (⊆++comm
        Γ
        (extract-references MT)
        (waiting-refs (heads ++ tails))
      )
      (⊆-skip
        Γ
        (extract-references MT ++ waiting-refs (heads ++ tails))
```

```
        (waiting-refs (heads ++ Msg x y :: tails))
        final-move
      )
    )
  where
    final-move : (extract-references MT ++ waiting-refs (heads ++ tails)) ⊆
                 waiting-refs (heads ++ Msg x y :: tails)
    final-move rewrite
       (waiting-refs++ heads tails) |
       (waiting-refs++ heads (Msg x y :: tails)) =
       ⊆-trans
         (⊆++comm'
           (extract-references MT)
           (waiting-refs heads)
           (waiting-refs tails)
         )
         (⊆-trans
           (⊆-inc
             (extract-references MT ++ waiting-refs heads)
             (waiting-refs heads ++ extract-references MT)
             (waiting-refs tails)
             (⊆-move (extract-references MT) (waiting-refs heads))
           )
           (⊆++comm
             (waiting-refs heads)
             (extract-references MT)
             (waiting-refs tails)
           )
         )

MessageFilter : (IS : InboxShape) → Set₁
MessageFilter IS = Message IS → Bool

record SelRec (IS : InboxShape) (f : MessageFilter IS) : Set₁ where
  constructor sm:_[_]-stash:_
  field
    msg : Message IS
    msg-ok : f msg ≡ true
    waiting : List (Message IS)

open SelRec

selective-receive : ∀ {i IS Γ} →
                      (q : List (Message IS)) →
                      (f : MessageFilter IS) →
                      ∞ActorM i IS
                        (SelRec IS f)
                        (Γ ++ (waiting-refs q))
```

```
                          (λ m → add-references Γ (msg m) ++
                                 waiting-refs (waiting m))

selective-receive {IS = IS} {Γ} q f = case-of-find (find-split q f)
  where
    case-of-find : ∀ {i} →
                     FoundInList q f →
                     ∞ActorM i IS
                       (SelRec IS f)
                       (Γ ++ waiting-refs q)
                       (λ m → add-references Γ (msg m) ++
                              waiting-refs (waiting m))
    case-of-find (Found split x) .force =
      strengthen (accept-found Γ q split) ∞>>
      return₁ (record {
        msg = el split
        ; msg-ok = x
        ; waiting = (heads split) ++ (tails split)
        })
    case-of-find (Nothing ps) .force =
      receive ∞>>=
      handle-receive
      where
        handle-receive : ∀ {i}
                           (x : Message IS) →
                         ∞ActorM i IS
                           (SelRec IS f)
                           (add-references (Γ ++ waiting-refs q) x)
                           (λ m → add-references Γ (msg m) ++
                                  waiting-refs (waiting m))
        handle-receive x with (f x) | (inspect f x)
        handle-receive {i} x | false | p =
          strengthen (move-received Γ q x) >>
          selective-receive (q ++ [ x ]¹) f
        handle-receive x | true | [ p ] =
          strengthen (accept-received Γ q x) >>
          return₁ ret-v
          where
            ret-v : SelRec IS f
            ret-v = record { msg = x ; msg-ok = p ; waiting = q }
```

D

Definition of initiate-channel

```
lookup-all : ∀ {a p} {A : Set a} {P : A → Set p} {ls : List A} {x : A} →
               x ∈ ls →
               All P ls → P x
lookup-all Z (px :: pxs) = px
lookup-all (S px) (px₁ :: pxs) = lookup-all px pxs

translate-⊆ : ∀ {a} {A : Set a} {ls ks : List A} {x : A} →
               (ls ⊆ ks) →
               (x ∈ ls) →
               (x ∈ ks)
translate-⊆ [] ()
translate-⊆ (x₂ :: subs) Z = x₂
translate-⊆ (x₂ :: subs) (S px) = translate-⊆ subs px
```

```
extra-fields-shape : ∀ {IS Mt} →
                     IsRequestMessage IS Mt →
                     MessageType
extra-fields-shape (HasTag+Ref Mt) = Mt

extra-fields : ∀ {IS Mt} →
               (Γ : TypingContext) →
               IsRequestMessage IS Mt →
               Set₁
extra-fields Γ irm = All
                     (send-field-content Γ)
                     (extra-fields-shape irm)

record Request (Γ : TypingContext)
               (caller : InboxShape)
               (ci : ChannelInitiation) : Set₁ where
  field
    {callee} : InboxShape
    var : Γ ⊢ callee
    {MtTo} : MessageType
    chosen-field : MtTo ∈ (ci .request)
    fields : extra-fields
               Γ
               (lookup-all chosen-field (ci .request-tagged))
    session : ChannelInitiationSession ci caller callee
```

```
suc-send-field-content : ∀ {Γ IS F} →
                         send-field-content Γ F →
                         send-field-content (IS :: Γ) F
suc-send-field-content {F = ValueType x} sfc = sfc
suc-send-field-content {F = ReferenceType x}
  ([ actual-is-sendable ]>: actual-handles-requested) =
    [ S actual-is-sendable ]>: actual-handles-requested

initiate-channel-fields :
  ∀ {Γ caller ci} →
  (request : Request Γ caller ci) →
  All (send-field-content (caller :: Γ)) (Request.MtTo request)
initiate-channel-fields {ci = ci} record {
  chosen-field = chosen-field
  ; fields = fields
  ; session = session
  } with (lookup-all chosen-field (ci .request-tagged))
... | HasTag+Ref _ =
  let open ChannelSession
      open ChannelInitiationSession
      channel = session .response-session
      channel-tag = lift (channel .tag)
      reference = [ Z ]>: (channel .can-receive)
  in channel-tag :: reference :: ∀map suc-send-field-content fields


initiate-channel : ∀ {Γ i receiver} →
                   (ci : ChannelInitiation) →
                   Request Γ receiver ci →
                   ∞ActorM i receiver T₁ Γ (λ _ → Γ)
initiate-channel ci request =
  let
    open Request
    open ChannelInitiationSession
  in do
    self
    let
      protocol-to-callee = translate-⊆ (request .session .can-request)
      to = S (request .var)
      which = protocol-to-callee (request .chosen-field)
    to ![t: which ] initiate-channel-fields request
    strengthen (⊆-suc ⊆-refl)
```