

A Programming Language For Type-Safe Object Evolution

Alexandra Mirrlees-Black

Supervised by **Fabian Muehlboeck**

A thesis submitted in partial fulfilment for the degree of
Bachelor of Philosophy (Honours) at
The Australian National University

October 2025

© Alexandra Mirrlees-Black 2025

I declare that this work:

- upholds the principles of academic integrity, as defined in the [Academic Integrity Rule](#)
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

Alexandra Mirrlees-Black
24 October 2025

Acknowledgements

This thesis was written on the lands of the Ngunnawal people. I acknowledge and celebrate the First Australians on whose traditional lands I have learned, taught, researched and written on, and pay my respect to the elders past and present.

I would like to give my deepest thanks to my supervisor Fabian Muehlboeck, who has been an unwavering provider of deep guidance, encouragement, support and much-needed choc-chip cookies. I appreciate his willingness to supervise me to achieve my goal of writing a compiler from scratch, and the constant supply of motivation needed to see this project come to fruition.

Thank you to Gregor Richards for the initial motivation to explore object evolution, and to Haoyu Wu for asking in-depth questions, prompting new ideas, as he works towards a formalisation of the work in this thesis. To various colleagues in Systems, thank you for listening to me explain my work, answering my questions and being the sounding board I needed to solve the problems I encountered in my research.

And finally, my friends, my housemates and my family, who have been an incredible support throughout this year, and have made my honours experience a delight.

Abstract

As programs run, the state and functionality of program components change in response to computation, the state of other components and external stimuli like network requests or user interface interactions. Dynamic object-oriented languages often allow for these sorts of changes to be modelled easily and explicitly, by modifying the class of an object as it runs, extending an instance with new fields and methods at run time. However, this is not true of any even moderately popular statically-typed object-oriented language. In these languages, more cumbersome techniques, design patterns and brittle run-time checks must be used instead.

This thesis describes the theory, development and implementation of the MAY programming language (Mirrlees-Black, 2025), a statically-typed object-oriented language that features a type-safe version of object evolution. Object evolution is a technique for adapting the flexible nature of objects in dynamic object-oriented languages, to a statically typed context. Compared to other methods, it avoids the type soundness problems that arise by restricting changes to an objects class to be *monotonic*: they can only occur from a class to one of its subclasses. This ensures the type of aliased references remains sound in the presence of run-time representation changes.

Object evolution can fail when the run-time class of an object is more precise than the static type. To ensure type safety, we introduce a sub-structural extension to our type system to statically track the exact run-time type of an object, with minimal notational overhead. We explore how this type system integrates with core object-oriented features like interfaces, and describe a bidirectional typing algorithm for our sub-structural types. We evaluate this feature through a small case study to demonstrate it is practical for writing non-trivial programs.

Table of Contents

Acknowledgements	v
Abstract	vii
Table of Contents	ix
Table of Figures	xi
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	4
2 Background	7
2.1 Types	7
2.1.1 Bidirectional Type Checking	9
2.2 The Semantics of Objects	12
2.2.1 Classes and Nominal Typing	13
2.2.2 Constructors	14
2.2.3 Run time representation	14
2.3 Object Reclassification	15
2.3.1 Dynamic Object-Oriented Programming	16
2.3.2 Static types, Aliasing and Reclassification	18
2.3.3 Fickle	19
2.3.4 Ego	20
2.3.5 Object Evolution	21
3 Approaches to Safe Object Evolution	25
3.1 Design Constraints for Object Evolution	25
3.1.1 Evolvers Are Just Like Constructors	25
3.1.2 Automatic constructors and evolvers	26
3.1.3 Field overriding	27
3.1.4 Constructor Semantics in the Presence of Evolution	29
3.2 Failure of evolution	31
3.2.1 Option Types	32
3.2.2 A Half Solution: Exact Types	34
3.2.3 Permission Types	36
3.2.4 Exactness and Interfaces	41
3.2.5 Type Checking and Inference	44
4 The MAY Language	47
4.1 Language Overview	47
4.1.1 Primitive types	47

4.1.2	Arrays	48
4.1.3	Option Types	48
4.1.4	Functions and Control Flow	48
4.1.5	Modules	49
4.1.6	Classes and Interfaces	50
4.1.7	Permissions	52
4.2	Compiler Design and Motivation	53
4.3	Case Study	55
4.4	Compiling Object Evolution	60
4.4.1	Optimisations in the presence of object evolution	64
5	Related Work	65
5.1	Object Reclassification and Evolution	65
5.2	Typestate	65
5.2.1	Typestate Checking	66
5.2.2	Typestate-oriented Programming	67
5.3	Object Initialisation	70
6	Concluding Remarks	73
6.1	Future Work	73
6.2	Conclusion	74
	Bibliography	75

Table of Figures

Figure 1	A brief example of how evolution could be used to express the annotation of a syntax tree with types.	2
Figure 2	A demonstration of how changing classes arbitrarily can run into type-soundness problems. The example shows two classes <code>Rectangle</code> and <code>Circle</code> which do not inherit from each other. Initially a rectangle is created, but it is then turned into a circle. However, a second reference to the rectangle existed, and now the <code>getWidth()</code> method is not present at run time.	3
Figure 3	A type system for a simple language featuring: integers, variables and addition. ($e ::= n \mid x \mid e + e$).	8
Figure 4	An extension of the simple type system in Figure 3 to include booleans: <code>true</code> , <code>false</code> and conjunction. ($e ::= \dots \mid \mathbf{true} \mid \mathbf{false} \mid e \wedge e$). .	8
Figure 5	A bidirectional type system for a simple language featuring: integers, floats, variables, addition and type annotations. ($e ::= n \mid f \mid x \mid e + e \mid e : t$)	11
Figure 6	Memory layout of an object with method table (<code>vtable</code>) stored in the first slot of the object. This method table is shared between all instances of the same class.	15
Figure 7	Memory layout of an object with method table (<code>vtable</code>) for class and interfaces.	15
Figure 8	The three stages an object passes through as it is constructed in a constructor and more invariants are established.	22
Figure 9	The two stages an object passes through as it is evolved within an evolver. These stages are the final two stages and invariants in Figure 8	22
Figure 10	An example demonstrating the type unsoundness of writing into an overridden field, by showing how we can use this to get a value of type <code>Twig</code> when we expected a value of type <code>Branch</code> , a more specific type than <code>Twig</code>	27
Figure 11	A code listing showing that within a super-first constructor, evolution occurs implicitly as invariants of an object are established. The points at which the method table of the object is updated to signify that its run-time type has changed are annotated across two classes <code>A</code> and <code>B</code>	29

Figure 12	A code listing demonstrating how in the field-first approach to constructors, there is no implicit evolution, and that all fields will be initialised before any methods are called.	30
Figure 13	An example showing how evolution can fail when the run-time class of an object is different from its static type.	32
Figure 14	An extension to a simple language that already features classes but no null values, which adds safe option types and null values. The syntax, typing rules and subtyping rules are extended for constructing and eliminating optional values.	33
Figure 15	An example program showing that a naïve implementation of exact types does not prevent multiple evolution.	35
Figure 16	An example program show a naïve solution that updates the exactness of a type still fails since it does not account for aliasing.	36
Figure 17	A function or a method can be used to abstract over evolution by passing in and returning owned references.	38
Figure 18	An example showing how a field can be overridden in a subclass in a way that retains its evolvability. In this example Branch extends Twig.	40
Figure 19	Subtyping diagram for the binary operator classes with and without ownership annotations, and syntax tree interfaces. The dashed line shows where evolution can occur.	41
Figure 20	The interface and class structure for a syntax tree and typed syntax tree in MAY syntax. Methods for evolution are part of the interface for SyntaxTree but not TypedSyntaxTree and so do not need to be implemented on TypedBinaryOp.	42
Figure 21	Subtyping diagram for the binary operator classes and syntax tree interfaces showing ownership for both classes and interfaces. The dashed line shows where evolution can occur.	43
Figure 22	An example showing the syntax for declaring interfaces and classes in MAY.	51
Figure 23	An example showing a simple evolver using MAY's syntax.	51
Figure 24	A program that returns the first element of an array of evolvable values with type !C, or null if the array has zero length. It manages permissions by using the swap operator >=<.	52
Figure 25	An example program that illustrates how evolves methods may be defined and invoked.	52
Figure 26	An example showing the use of evolves methods within interfaces and on which classes these methods must be implemented.	53
Figure 27	An example showing how evolves fields and overrides fields can be used in conjunction to describe fields who's type becomes more specific as part of evolution.	53
Figure 28	The definition for the IMP language.	55
Figure 29	A snippet of code from the definition of IMP syntax in MAY.	55

Figure 30	A snippet of code from the <i>typed</i> syntax definition for IMP in MAY.	56
Figure 31	A snippet of code, from the IMP type checker in MAY, showing the <i>typeCheck()</i> methods invoking the appropriate evolver.	56
Figure 32	A snippet of code, from the IMP type checker in MAY, showing the implementation of the IMP-INT and IMP-OP+ rules within the evolvers for the <code>TyLitInt</code> , <code>TyOp</code> and <code>TyAdd</code> classes.	57
Figure 33	58
Figure 34	An implementation of <code>mkSeq</code> that requires the use of the swap operator, bypassing null safety checks, and mutability of an array of evolvable references	60
Figure 35	Hypothetical implementation of <code>mkSeq</code> using a for loop.	60
Figure 36	Object evolution implemented by pre-allocating sufficient memory.	61
Figure 37	Object evolution implemented by forwarding pointers/handles. .	61
Figure 38	Object evolution implemented by optional forwarding pointers. .	63
Figure 39	Object evolution implemented by extension pointers.	63

Introduction

This thesis is an account of the background, theory, development, implementation and related works of a new experimental programming language MAY, for exploring type safe and efficient approaches to *object evolution* in the context of a modern object-oriented language.

Object evolution (Cohen and Gil, 2009) is a language feature that extends the expressiveness of statically-typed object-oriented programming. Rather than fixing the available methods and fields of an object at the point it is created, object evolution is one answer to the question of what a programming language might look like if it allowed the functionality and features of an object to update or *evolve* over time. More precisely, object evolution allows an object to become an instance of a subclass of its current class, adopting its fields and methods, such that any reference to the object transparently observes this update.

Much more powerful variants of this feature are reasonably commonplace in dynamically typed object systems: SMALLTALK, SELF, JAVASCRIPT, PYTHON and Common Lisp's CLOS, to name a few. Yet a feature like object evolution, or its more general form, object reclassification, cannot be found in statically typed object systems outside of a handful of research languages. One possible reason is the inherent challenges in ensuring type safety in the presence of object evolution. A core contribution of this thesis and the MAY language is a type system for performing object evolution safely through the use of permission types.

1.1 Motivation

Object evolution lets programmers cleanly express state transitions in a programming language that would otherwise have to be encoded in brittle ways. Many real world programs, at least implicitly, rely on some notion of state transition that can be handled by object evolution.

```
let syntax_tree : SyntaxTree = parseInput(input);  
let program_type : Type = inferType(syntax_tree);  
return syntax_tree evolves TypedSyntaxTree(program_type);
```

Figure 1: A brief example of how evolution could be used to express the annotation of a syntax tree with types.

Compilers: Throughout this thesis we will consider in depth the motivating example of type-checking a syntax tree represented by object as nodes, by adding a field for the object's type. Compilers represent a real-world example of a program where in several stages, syntax trees are annotated with additional data and become specialised as types and names are resolved. Being able to represent these stages within the type system in an efficient way is important for reducing code duplication and enforcing safety properties (Oliveira, 2024).

Staged Initialisation: Complex data structures or objects that may rely on some external resource for part of their initialisation can have their initialisation effectively staged without the need to use null pointers or excessive data copying. For example, an object representing a record from a database can be initialised and shared with just the key to that data, and then evolved to a full version once the database query completes.

Stateful interfaces: Objects are often used to model stateful processes, such as files that can be accessed in an `Open` state, but once they transition to a `Closed` state may no longer. For monotonic stateful interfaces, evolution can be used to express the transition between these states. This can be more effective than using the traditional State design pattern (Gamma et al., 1995) as the need to copy data between classes and handle wrapper classes is removed.

Lazy computation: Using object evolution, an object can evolve from a general class before some expensive computation is performed into a specific class representing the output of that computation. The first time certain object methods are called, the object can perform some expensive computation and then specialise itself so that future calls to the same method use the cached result.

Encoding the semantics of object evolution is possible in existing popular languages like `JAVA`, but is not supported by the object model or type system. There are two ways to encode this behaviour in a statically typed language like `JAVA`. The first is to use `null` initialisation to declare fields that will later be filled in. In our type-checking example, the field representing the type of a syntax node would start null-initialised, and only be given a non-null value during type checking. This approach is brittle: it becomes the burden of the programmer to check whether or not the field has been initialised or is `null`, and it may not be initially clear that the field only goes from being `null` to non-null and not the other way round.

A more type safe approach would be to use two objects. To emulate something like evolution, we start with an instance of one of the objects, and then allocate a new object containing a copy of the old object's data. In our type checking example, this would mean creating two copies of the syntax tree, first the version without types and then constructing a distinct version with types during type checking. However, any references to the old object now point at stale data, with no way to convey that the object has moved. A new layer of indirection could be added in the form of a wrapper class, to ensure that only one reference to the object exists at runtime, but then all operations have to be forwarded through this wrapper, introducing further complexity. Object evolution provides a semantically cleaner solution to this programming idiom.

Existing typestate-oriented programming (Aldrich et al., 2009; Garcia et al., 2014, 2010) techniques do provide solutions to some of the examples presented, but have to do so at the cost of additional notational and type-system overhead. Typestate-oriented programming takes a much broader approach than object evolution, allowing classes to be changed in a much more flexible way. However to ensure type soundness in the presence of aliasing, these systems necessarily have to both restrict aliasing for objects that may change state and require complex annotations as part of type signatures to track which changes are possible and what their results may be. Figure 2 illustrates how arbitrary class changes break type safety in the presence of aliasing. Evolution avoids many of these safety concerns by restriction how classes may change, only allowing them to extend their functionality. As a result, the systems for tracking evolvability are much simpler than the systems for general typestate, and easily handle aliasing even in the naïve case.

Many of these motivating factors have been explored within the literature through alternative and similar approaches historically (Bejleri et al., 2006, Bierhoff and Aldrich, 2007, Cohen and Gil, 2009, Damiani et al., 2003, DeLine and Fähndrich, 2004a; Drossopoulou et al., 2002, 2001; Fähndrich and Leino,

```
class Rectangle { getWidth() : int; ... }
class Circle { getRadius() : int; ... }

let rectangle = new Rectangle(...);
let other_rectangle = rectangle;
let circle_now = rectangle evolves Circle(...);

// other_rectangle is a runtime Circle now
other_rectangle:getWidth()
```

Figure 2: A demonstration of how changing classes arbitrarily can run into type-soundness problems. The example shows two classes `Rectangle` and `Circle` which do not inherit from each other. Initially a rectangle is created, but it is then turned into a circle. However, a second reference to the rectangle existed, and now the `getWidth()` method is not present at run time.

2003b). However, type safe object evolution promises an interesting set of features and semantics particularly for modern object-oriented programming languages. Much of the initial research into object reclassification and evolution focus on languages that take strong inspiration from JAVA. But we are now researching in a context where languages like KOTLIN (Akhin and Belyaev, 2021), TYPESCRIPT (Bierman et al., 2014) and RUST (Klabnik and Nichols, 2022) have become popular. Evolution has new applications in this context where it can ameliorate some of the challenges resulting from type safety.

Initialisation with null safety: Languages like KOTLIN (Akhin and Belyaev, 2021), TYPESCRIPT (Bierman et al., 2014) and DART (Bracha, 2015) have helped make null safety a standard feature of modern object-oriented programming (OOP) languages, requiring annotations on types that have the capacity to be null, and enforcing that null values are checked where appropriate. This makes writing safe code easier, but also reduces object initialisation flexibility. Object evolution provides a promising answer to the question of how one might interleave initialisation with computation. By grading initialisation into several stages, a class can become progressively more initialised without the need to use null values for uninitialised fields.

Gradual type systems: Dynamically typed object-oriented systems, like those mentioned above, often have ways of updating the class of an object at runtime, with different levels of support. This highlights the ways in which dynamically typed programming is not just statically typed programming without type annotations, but in places, a fundamentally different paradigm. Gradual type systems (Siek and Taha, 2006, Tobin-Hochstadt and Felleisen, 2006), which allow for statically and dynamically typed code to be mixed, hence struggle to include this feature in a sound way. The system of type-safe object evolution presented within this thesis provides an answer for how the dynamic and static paradigms could be brought closer together, for application in gradual type systems.

1.2 Contributions

As part of this thesis we have developed a novel programming language MAY. MAY is a class-based nominal object-oriented language that features object evolution, null-safety and a substructural permission system for tracking evolvability. The implementation of this language is a compiler to QBE's (Carbonneaux, 2024) intermediate language, which can be used to generate optimised machine code for several 64bit platforms. As part of this project we have:

- developed a language based around object evolution for the first time: existing work only presents an extension to an early JAVA implementation with

several limitations. Our implementation is described in [Section 4](#) with syntax and semantics documented in [Section 4.1](#).

- further explored the design space for inheritance based object evolution, including how language design can be influenced by the inclusion of evolution ([Section 3.1](#)), how object evolution interacts with field overriding and interfaces ([Section 3.2.3](#)), as well as the space of possible implementation techniques for object evolution ([Section 4.4](#)).
- developed an affine type system for tracking permission specifically for object evolution. This system is more flexible than standard affine type systems for managing aliasing ([Section 3.2.3](#)). We have implemented a type checking algorithm for this type system that requires minimal programmer annotations ([Section 3.2.5](#)).
- implemented a small case study using object evolution to demonstrate the feature can be practically utilised for real world programs ([Section 4.3](#)).

Background

In this section we provide an account of the required prior work and topics for this thesis. First we give a formal description of type systems and outline standard notation for describing type systems. Following this, we describe bidirectional type checking, a general purpose approach to writing typing algorithms. Next we give an overview of the theory, semantics and implementation of object-oriented languages. Finally we discuss prior work on object reclassification and in particular object evolution. A full treatment of alternative typestate systems and other related work is given after our main contributions in [Section 5](#).

2.1 Types

Type systems are the primary mechanism for enforcing certain kinds of safety properties for programs, systematically ensuring they cannot go wrong in various problematic ways. A principled understanding of types is necessary to understand how we can statically guarantee, without ever running our program, that our programs won't unintentionally crash, fail, or treat data improperly.

In this thesis we will be using standard natural-deduction-style notation for presenting type systems. We begin by describing the notation for a *typing judgement*. We write the expression $\Gamma \vdash e : t$ to mean that in the context Γ , the expression e has the type t . The notation $e : t$ makes this assertion outside of any context. A context in its simplest form is a finite list or set of variables and their associated types, e.g. $\Gamma = x_1 : t_1, \dots, x_n : t_n$. This represents the variables that are “in-scope” for a particular expression, and the type of those variables, whether they be function parameters or locally defined variables. A *type system* is a collection of rules that dictate how we may derive valid typing judgements. We express these rules in a natural deduction system, which makes clear the

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \text{Int}} \text{SMPL-INT} \quad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \text{SMPL-VAR} \\
\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{SMPL-ADD}
\end{array}$$

Figure 3: A type system for a simple language featuring: integers, variables and addition. ($e ::= n \mid x \mid e + e$).

premises (what must be shown) which sit above the line, and the conclusion which sits below.

The example in [Figure 3](#) shows the typing rules for a simple language featuring numbers n , variables x and sums of expressions $e_1 + e_1$. The first, [SMPL-INT](#), states that in any context Γ , a number n has type `Int`. The second, [SMPL-VAR](#) says that if the variable x is in scope with type A , then the expression x has type A . The third, [SMPL-ADD](#) says that in any context Γ , the expression $e_1 + e_2$ has type `Int` when both e_1 and e_2 have type `Int` in the same context Γ . This intuitively captures the idea that we should only be able to add integers together. If our expression language were extended to include booleans `true` and `false` assigned type `Bool` as in [Figure 4](#), then we would be able to show $1 + 2 : \text{Int}$ but not $1 + \text{true} : \text{Int}$.

In expressions with variables, syntax fragments alone are insufficient for determining whether an expression is well-typed. Can we show that $1 + x : \text{Int}$? Well, it will naturally depend on what sort of value x is standing in for. Hence, understanding the type of x in this expression is dependent on the *context* Γ . If x is an integer variable, i.e. our context looks like $\Gamma = x : \text{Int}$, then the program *is* well-typed. But instead if $\Gamma = x : \text{Bool}$, then this is no longer true!

A good type system should only allow us to type the programs that “make sense” semantically. If we can show using our type system that a program is well-typed, i.e. we can assign it a type in the system, then it should not be able to “go wrong” in some way ([Milner, 1978](#)). The formal name for this property is *type safety*. What exactly is meant by “go wrong” is dependent on the semantics of the language, but might include that boolean values cannot be added to integers, or that null values cannot be dereferenced. While we will not be providing proofs of type safety, the informal intuition of this property plays a

$$\begin{array}{c}
\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \text{SMPL-TRUE} \quad \frac{}{\Gamma \vdash \text{false} : \text{Bool}} \text{SMPL-FALSE} \\
\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{Bool}} \text{SMPL-AND}
\end{array}$$

Figure 4: An extension of the simple type system in [Figure 3](#) to include booleans: `true`, `false` and conjunction. ($e ::= \dots \mid \text{true} \mid \text{false} \mid e \wedge e$).

key role in understanding the design decisions made in the type systems we present.

Many type systems feature a form of *subtyping*, a way of expressing that everywhere that an instance of type A is expected, then an instance of a subtype B will suffice (Liskov and Wing, 1994). We use the notation $B <: A$ to mean that B is a subtype of A . Many programming languages allow an implicit conversion or cast from integers to floating point numbers, an example of subtyping that we could write as $\text{Int} <: \text{Float}$. A more popular example is that of class inheritance in object-oriented languages, which we explore in Section 2.2. This notion of subtyping was popularised by Liskov and Wing (1994) who formalised the subtyping relation to mean that any property which is true of all instances of a supertype, must also be true of all instances of a subtype.

There are two properties that we expect a subtyping relation to have: reflexivity which means for any type t , it should be the case that $t <: t$ (it is a subtype of itself); and transitivity which means for any types t , t' and t'' such that $t <: t'$ and $t' <: t''$, then we know $t <: t''$. To ensure this is the case we often take the *reflexive-transitive closure* over a subtyping relation. This adds in all the necessary extra relations to ensure that these properties hold. For example, if we suppose $A <: B$ and $B <: C$, then the reflexive-transitive closure would include $A <: C$ as well as $A <: A$, $B <: B$, and $C <: C$. We will be using a subtyping relation that is implicitly reflexive and transitive via this construction throughout the thesis.

2.1.1 Bidirectional Type Checking

There is a lot of flexibility permitted by the structure of a type system's derivations and judgements. While type systems may be designed for many purposes, our focus is on the implementation of a programming language to enforce a type system. As it stands, the presentation of our rules, while useful for understanding the logical underpinnings of the theory, do not easily lend themselves to an efficient implementation for type checking — “does the program have this type, for some type?” — and type synthesis/inference — “what type(s) *can* the program have?”. Indeed, in general, type systems provide no guarantee of efficient implementation at all, see JAVA's Turing-complete generics as an example (Grigore, 2017).

Furthermore, from the logical presentation of the rules, it is not clear where the inputs and outputs are present. Can we synthesise a context and type to satisfy the typing constraints for a particular program or can we only check that a term satisfies a type within a context? From an ergonomic perspective, it would be overly cumbersome to have to extensively annotate program source with type annotations. And yet, approaches to complete type inference are brittle to language features such as subtyping that can make inference undecidable (Kfoury et al., 1993).

The discipline of bidirectional typing, first academically documented by [Pierce and Turner \(1998\)](#), is a general purpose approach to typing algorithms that interleaves both synthesis, where possible, and checking, where necessary. Rather than interpret the rules of our type system purely as logical judgements, we can write them as an algorithm, where type information flow can be reasoned about *locally*, within the rule itself. Our presentation follows the work by [Dunfield and Krishnaswami \(2022\)](#) which gives a principled approach to constructing effective bidirectional type systems.

The main benefit of adopting this approach as opposed to alternative type synthesis disciplines like Hindley-Milner ([Milner, 1978](#)), is how broadly effective the approach is, while only requiring minimal type annotations. The approach has become the de facto standard for even complex type theories such as dependent types ([Ferreira and Pientka, 2014](#)). Additionally, in most programs, type annotations should only be necessary on top level definitions (e.g. as part of a function signature), where they are often useful anyway as a form of documentation and for enforcing interfaces and abstractions.

To distinguish between types we can synthesise for a term and types we can check a term against, we split our notation. Additionally we annotate inputs and outputs to these notations to make the information flow clear in the typing rules.

$$\boxed{\begin{array}{c} \Gamma \vdash e \leftarrow A \\ \uparrow \quad \uparrow \quad \uparrow \\ \text{in} \quad \text{in} \quad \text{in} \end{array}}$$

In a context Γ , the term e can be *checked* to have type A .

$$\boxed{\begin{array}{c} \Gamma \vdash e \Rightarrow A \\ \uparrow \quad \uparrow \quad \downarrow \\ \text{in} \quad \text{in} \quad \text{out} \end{array}}$$

In a context Γ , the term e *synthesises* the type A .

Using this approach we can “bidirectionalise” the typing rules presented in the previous section, featuring a generalisation to include a Float type. For now, addition will be compatible with both Floats and Ints, as long as both the left hand side and right hand side are the same. These rule are shown in [Figure 5](#).

$$\begin{array}{c} \frac{}{\Gamma \vdash n \Rightarrow \text{Int}} \text{BI-SMPL-INT} \quad \frac{}{\Gamma \vdash f \Rightarrow \text{Float}} \text{BI-SMPL-FLOAT} \quad \frac{x : t \in \Gamma}{\Gamma \vdash x \Rightarrow t} \text{BI-SMPL-VAR} \\ \frac{\Gamma \vdash e_1 \Rightarrow t \quad \Gamma \vdash e_2 \leftarrow t}{\Gamma \vdash e_1 + e_2 \Rightarrow t} \text{BI-SMPL-ADD} \quad \frac{\Gamma \vdash e \Rightarrow t' \quad t = t'}{\Gamma \vdash e \leftarrow t} \text{BI-SMPL-SUBS} \\ \frac{\Gamma \vdash e \leftarrow t}{\Gamma \vdash e : t \Rightarrow t} \text{BI-SMPL-ANNOT} \end{array}$$

Figure 5: A bidirectional type system for a simple language featuring: integers, floats, variables, addition and type annotations. ($e ::= n \mid f \mid x \mid e + e \mid e : t$)

Given an Int, Float or variable, we can synthesise the correct type, either by looking at the syntax or looking in the context. We can synthesise the addition of two expressions by looking at the left hand side and synthesising a type for it, and then check the right hand side has the same type. Finally the type of the result of the sum is the same as the type of the left hand side which we synthesised a type for. We also introduce a *subsumption* rule **BI-SMPL-SUBS** that lets us switch from checking mode into synthesising mode. If we don't know how to check the type of an expression, for example there is no rule to check the type of 3, only synthesise it, then we may instead synthesise the type Int using **BI-SMPL-INT** and check that this matches the type we expect. Finally, the *annotation* rule **BI-SMPL-ANNOT** lets us annotate our program with types. If there is some expression we need to, but cannot, synthesise a type for, then instead we may annotate its type, check that it has that type and hence produce that type in the conclusion. Note that in this rule, the colon : is being used as part of the language syntax rather than part of the rule syntax.

By taking up the bidirectional discipline, a typing rule now represents a recursive algorithm for determining whether a conclusion has a proof in the type system! First we start with the inputs to the conclusion, pass them to the premises in order (left to right), either check or synthesise a proof for that premise, and then pass the outputs from typing that premise along with the inputs we've collected so far to the next premise. Finally the outputs of the premises can be used to produce the outputs of the conclusion. When a typing rule can easily be converted into an algorithm in this way, we call it *mode-correct*. Rules that introduce inputs to judgements that do not come from anywhere are not considered mode-correct.

Another nice advantage of the bidirectional approach over other approaches to type synthesis is that it neatly is able to handle subtyping. If we wish to add a subtyping relation to our language with the rule $\text{Int} <: \text{Float}$, then we need some way to express that anywhere we expect to see a Float it is OK to instead see an Int. This sounds almost exactly the same as the subsumption rule **BI-SMPL-SUBS** but rather than checking equality between the checked and synthesised types, we can check subtyping!

$$\frac{\Gamma \vdash e \Rightarrow t' \quad t' <: t}{\Gamma \vdash e \Leftarrow t} \text{BI-SMPL-SUBS-2}$$

To caveat, in general subtyping can be complex and requires careful analysis in tandem with other typing rules to ensure subtyping and subsumption work as intended.

One of the tricky parts of designing a bidirectional type system is deciding which rules ought to be checking rules and which ought to be synthesis rules. Existing wisdom around bidirectional type systems ([Dunfield and Pfenning, 2004](#)) dictates that introduction rules: forms for introducing data like pairs or

lists ought to be checked, and forms for eliminating data such as projecting out items of a pair, or even branching on the value of a boolean in an `if` expression ought to be inferred. This wisdom is based on sound reasoning, since it results in a system which is mode-correct, requires minimal type annotations, requires annotations predictably and is *syntax directed*, i.e. by looking at the syntax of a term we can choose which rule we need to use.

However taking this approach can result in programs that synthesise types in places where one might expect the type system ought to check them. Instead, some rules should come in both synthesis and checking pairs (Dunfield and Krishnaswami, 2022). We demonstrate this by considering a fragment of a language including boolean (`Bool`) types and `if` expressions (ternary expressions in languages like `JAVA` and `C`). As an eliminator, we are encouraged to synthesise a type for this rule¹:

$$\frac{\Gamma \vdash b \Leftarrow \text{Bool} \quad \Gamma \vdash e_1 \Rightarrow t \quad \Gamma \vdash e_2 \Leftarrow t}{\Gamma \vdash \text{if } b \{ e_1 \} \text{ else } \{ e_2 \} \Rightarrow t} \text{BI-SMPL-IF-1}$$

The problem with exclusively using this rule arises when we encounter an `if` expression in checking mode rather than synthesis mode. This rule forces us to use subsumption and then synthesis using `BI-SMPL-IF-1`. But, if we are unable to synthesise a type for the expression e_1 , we would require annotations! Since synthesis for e_1 does not depend on any of the other premises, we can include an alternative rule to check e_1 if the whole `if` statement is being checked.

$$\frac{\Gamma \vdash b \Rightarrow \text{Bool} \quad \Gamma \vdash e_1 \Leftarrow t \quad \Gamma \vdash e_2 \Leftarrow t}{\Gamma \vdash \text{if } b \{ e_1 \} \text{ else } \{ e_2 \} \Leftarrow t} \text{BI-SMPL-IF-2}$$

Now if checking an `if` expression, we are permitted to check e_1 as the same type, with no annotations required!

2.2 The Semantics of Objects

The meaning of the phrase “object-oriented”, as with many imprecisely defined terms in politics, is the subject of infighting and religious wars. Dogmas and ideology around the right and wrong way to orient your programs around objects, what an object exactly is and how that definition might integrate with a type system, are pervasive and extensive (Cook, 2009). Within this thesis we deal with a very loose definition of object, roughly inspired by existing stati-

¹In this rule we check that b has type `Bool`, which intuitively makes sense since we know that b has to have type `Bool` in an `if` expression. In general, the type of a value being eliminated on might be more complex, such as a product or sum of two other types. In these context synthesis is *required* so some presentations insist using synthesis here, however we use checking for clarity.

cally typed language implementations like C++ and JAVA, since for our practical purpose this is sufficient.

Cook (2009) characterises objects as data abstraction by interface and procedure abstraction. Alternative techniques for abstraction such as abstract data types which perform data hiding still use concrete functions that operate over known data structures. Objects on the other hand are not guaranteed to have any known data structure at runtime, and calling methods on object is a form of *late binding*, that resolves the procedure that gets called at run time. The specification of which methods are available make up an object's *interface*, and a procedure that accepts an interface should expect to get anything that could implement that interface. In practice, this notion of object is rather broad, and encompasses to an extent, certain features of JAVA's class system, as well as Haskell's type classes, which both specify interfaces that resolve methods by late binding.

2.2.1 Classes and Nominal Typing

Class-based systems are one way of structuring programs around this philosophy of objects. Classes define together both the implementation and interface for an object through a specification of fields (internal data), methods (exposed functions) and constructors. Classes also allow for inheritance and overriding. A class may inherit from another class, adopting all existing field and method definitions, and extending them as suitable. Furthermore specific methods may be overridden to change their implementation in some way. Since a subclass inherits from a base class, any instance of the subclass remains a suitable candidate for the base class' interface. This establishes a *subtyping relation* based on the inheritance hierarchy. We can say that $A <: B$ when A inherits from B . In a program it is sound to replace any B by an A since A possesses the same capabilities as B .

Class-based type systems can either be nominally or structurally typed. In a nominal typing system, like JAVA, the inheritance relation must be made explicit, meaning subtyping is determined based on the name associated with a class, rather than the interface of that class. Structural systems instead determine subtyping entirely based on the interface of a class, allowing classes to be treated as subtypes of another without explicitly inheriting from that class. While a structural system is more powerful, a nominal system is often easier to reason about.

Even in class-based languages, the need for explicit interface definitions separate from a class definition is apparent for describing more general notions of abstraction (Canning et al., 1989). An interface definition specifies a set of methods that characterise the interface. Any class that wishes to implement this interface must at least provide definitions for each of the methods in the interface. Interfaces neatly integrate into an inheritance-based subtyping relation, with any class that implements an interface being considered a subtype of

that interface. Furthermore, many languages allow interfaces to extend other interfaces by requiring that all the base interface's functionality is present in addition to new methods. This further extends the subtyping relation to allow an instance of one interface to be the subtype of another interface.

2.2.2 Constructors

Object systems, and especially class-based object systems often use *constructors* to define how objects are initialised. Unlike methods which are called on existing objects, classes may have one or in some cases several constructors associated with them that are called as part of object instantiation. These are resolved statically rather than lazily like methods. Constructors have the role of initialising the fields of a class and performing any operations to enforce an object's invariants.

Describing exactly what an object “is” when it is being initialised presents a problem (Qi and Myers, 2009). In languages like Java, fields are initialised to default values like `null` or `0`, and so at every point in a constructor the object is considered “fully initialised” from the perspective of the type system. Without default initialisation, consideration must be taken if one wants to avoid accesses to uninitialised data, such as by enforcing fields be initialised before any calls to methods or passing around the `this` pointer.

Handling constructors in the presence of inheritance further interacts with this problem in interesting ways and introduces the main variation between languages in their description of constructors. The principled approach for writing constructors of classes that inherit from a superclass is to invoke the superclass' constructor: `super()`. This initialises the part of the object relevant to the superclass but avoids exposing the hidden internals of the superclass.

However, with inheritance, naïve method calls inside a constructor fail to prevent uninitialised accesses, even after initialising all the fields and calling the super constructor correctly. This is a result of method overriding and late binding, which means the method that gets called could be an arbitrary, overridden method on a subclass. This overridden method could access fields that have not been initialised in the super constructor yet. The two main solutions to this are to either call methods directly, preventing late binding within constructors, or to enforce all fields introduced in the current class have been initialised, before calling the super constructor. The first is the approach taken by C++.

A more detailed discussion of constructor variation is considered in sections 2.3.5 and 3.1, where it can be better discussed in the context of object evolution.

2.2.3 Run time representation

Any representation of a class-based object system needs some way to handle both the data stored in the fields of an object as well as the method implemen-

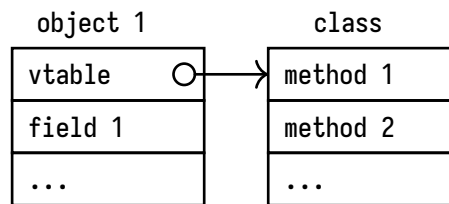


Figure 6: Memory layout of an object with method table (vtable) stored in the first slot of the object. This method table is shared between all instances of the same class.

tations for that object. The standard way to do this is by using a record for the field data that includes a pointer to the record for the method implementations, as shown in Figure 6. The record to the method implementations is called the method table, virtual method table or just vtable. Late binding is performed by looking up the method in an object's method table at run time when the method gets called. The method can be identified because it will be at a static offset in the table. At compile time, each class gets its own record of methods. The trick to handling inheritance is that inherited methods in a subclass go at the same offset as those methods in the base class. That way, when accessing the function pointer at run time, the relevant method is in the correct location whether the object is an instance of the base class or any subclass. If a method is overridden in a subclass, the new method overwrites the entry for that method in the subclass' method table. When a method is called on an object, we call this object the receiver of the method. That function pointer is called with the parameters to the method call along with the receiver.

Implementing method lookups through interfaces can be more difficult in a system that replaces method names with offsets, since there is no way to guarantee that two classes that implement the same interface will be able to put the interface-relevant methods at the same offsets. There is a greater variety of techniques here. We document a relatively simple solution (Alpern et al., 2001, Ramalingam and Srinivasan, 1999), that does not require code to explicitly cast from an object treated as a class to an object treated as an interface. This solution uses a separate set of records representing interface method tables. Finding the appropriate method table is performed using yet another table, this time storing pairs of unique interface identifiers alongside a pointer to the relevant interface method table. . When an interface method is invoked, the record is searched for the entry corresponding to the calling interface, which then points to the interface table which can be indexed to find the correct method.

2.3 Object Reclassification

The existing literature for understanding the sorts of problems posed by incorporating static typing with updating an object's class at run time stems from two different research areas: object reclassification and typestate programming.

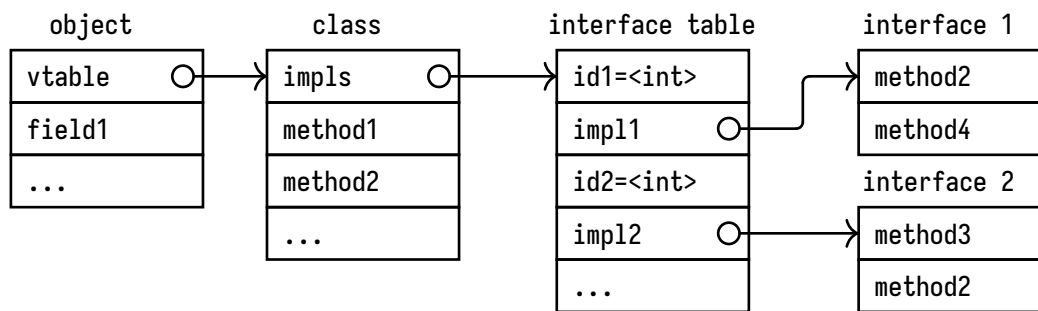


Figure 7: Memory layout of an object with method table (vtable) for class and interfaces.

Object evolution is a result of the work on object reclassification and so we describe it here, however there are many parallels with the research into typestate and typestate oriented programming which we describe as part of the related work in [Section 5](#).

Firstly we provide an overview of dynamic object reclassification in languages with dynamic type systems, since this is the context in which reclassification originates. Then we describe work on the experimental programming language Fickle, in its several iterations, which presents an example of statically-typed object reclassification by restricting how reclassification may occur. We also look at the Ego language which provides a type system for SELF by introducing linear types. Finally we consider the existing work on object evolution, the approach directly drawn upon and refined in this thesis.

2.3.1 Dynamic Object-Oriented Programming

Object-oriented programming (OOP) originated in highly dynamic systems, free of static types, permitting exploratory programming and complex manipulations of run-time state ([Goldberg and Robson, 1983](#)). We document some dynamic object-oriented languages and how they express run-time object reclassification.

The SELF ([Ungar and Smith, 1987](#)) programming language introduced an object-oriented programming model that uses prototypes as opposed to class-based inheritance and instantiation. Prototypes are flexible enough to encompass both inheritance and instantiation, as well as less conventional object-oriented structures. However, the idea relies on the ability to dynamically change the identity and available functionality in one or several objects.

SELF's primitive structure is the *object*: a collection of fields (referred to as slots) associating a name with a value, along with a reference to a parent object. Operations on objects are handled via *message passing*: a slot/field lookup that recursively forwards messages to parent objects if they cannot be resolved directly. If the slot contains code rather than data, that code is executed with the arguments passed as part of the message, along with a reference to the object itself, named `self`. By using objects containing methods, which represent

classes, class objects with parents containing more methods, or collections of many objects sharing a single parent, we can model classes, inheritance and instantiation with a single mechanism.

Much like the languages that inspired it, such as SMALLTALK (Goldberg and Robson, 1983), SELF features a dynamic type system: checking whether a message can be received by an object is performed at run time, as opposed to the static method resolution in statically typed object-oriented languages. No guarantees are statically provided about the presence of method slots.

In order to take advantage of its object system, SELF provides the ability to update any of an object's slots, including its parent object at run time. The pattern for creating new objects centers around making a prototype object by adding fields and methods to an existing object, and then cloning that prototype to form new instances. Hence, SELF is be "prototype-based" rather than "class-based". This flexibility even lets programmers manipulate the methods and fields present on existing objects at run time. These features were heavily motivated by SELF' exploratory programming model, which blurs the lines around run time and instead encourages directly programming within the running system. In this context, being able to modify classes and objects directly at run time is necessary.

While the paradigm of prototypes has not been widely adopted across many programming languages, it remains a feature of JAVASCRIPT despite the addition of traditional classes and inheritance. Hence, JAVASCRIPT, although it lacks the same first-class exploratory programming model, allows for the same level of dynamism as SELF: modifying fields methods, updating object identities and inheritance hierarchies, all at run time.

While not necessarily a first-class feature in the same way, object reclassification was available in the class-based language SMALLTALK (Goldberg and Robson, 1983) that inspired SELF, as well as several modern dynamic languages that feature classes. The primitive `become:` message of SMALLTALK swaps the instance of two objects with each other. One of the original uses of this feature is to model collections that increase in size, by becoming an object with a larger size. This is a more dramatic change than just swapping the class of two objects, all the fields are swapped too, exchanging the very identities of the objects. This is quite different from SELF's notion of object reclassification.

PYTHON (Van Rossum and Drake, 2009) permits at run time the ability to add/update/remove fields on an object, methods on a class, as well as changing an instance's class via its `__class__` field. In Ruby it is possible to add additional methods and fields to objects/classes at run time but it is not possible to change what class an object is an instance of. The Common Lisp Object System (DeMichiel and Gabriel, 1987) features the `change-class` function. This function changes the class of an instance and ensures the slots on the new class are initialised, as well as removing the slots only present on the old class.

This process can be made less arbitrary by defining a `class-changed` method that takes in the old and new instances and initialises the fields of the new instances using the contents of the old.

Naturally, within a dynamically typed system, there is less need to restrain the flexibility of an object system to handle dynamic reclassification updates. While dynamic languages may restrict flexibility to improve run-time performance, or encourage particular programming patterns, they do not have to restrict their flexibility to ensure type safety. We now explore some of the ways reclassification has been adopted in experimental static type systems, both in the way they introduce complex types to enforce safety and restrict reclassification to avoid type system overhead. We conclude this analysis by looking at the existing work on object evolution, the technique we have developed on in this thesis.

2.3.2 Static types, Aliasing and Reclassification

Object reclassification as seen in dynamically typed object-oriented languages does not play nicely with static typing. The static type given to an object reference represents a contract stating which methods and fields will be available on an object at runtime. If the underlying class of an object changes arbitrarily at run time, then the contract of the object, in the form of its type, needs to be updated. However, when multiple references to an object exist across a program, and an update needs to only occur at one of them to affect the rest of them (since the underlying object is the same), it is impossible to update all the contracts, i.e. types, appropriately.

For example, suppose we define a class `Circle` with a `getRadius()` method. With arbitrary reclassification, we may change the class of an object from `Circle` to `Square`. Even if we update the static type of the object at the location we change it, there could be other references to the same object that still have the type `Circle`. We call these sorts of references *aliases*, since although the variable (i.e. alias) is different, the underlying object is the same. In the presence of aliasing we might be able to call the `getRadius()` method on our object, which would no longer be sound since the object is not a circle anymore!

```
class Shape { fun getXPosition() : int { ... } ... }
class Circle < Shape { fun getRadius() : int { ... } ... }
class Square < Shape { fun getSideLength() : int { ... } ... }
```

Being able to handle these sorts of dynamic reclassification operations in the most general way possible within a static type system requires completely understanding the run-time memory layout of a program, which is impossible without running the program. Therefore, approaches to incorporating dynamic reclassification with static types, have to accept some restrictions balancing the scope of reclassification with the precision of types.

2.3.3 Fickle

The Fickle language and its successors Fickle₁ and Fickle₃ (Damiani et al., 2003; Drossopoulou et al., 2002, 2001) are a family of statically typed and type safe languages featuring dynamic object reclassification, allowing an instance of one class to become an instance of another class at runtime. This change is not totally arbitrary, however, and the management of this dynamism is the key insight Fickle provides.

Fickle is a JAVA-like class-based language. It features two different types of classes, root classes and state classes. Root classes are much like standard JAVA classes, exhibiting inheritance and featuring fields and methods, but are not available as targets of reclassified, i.e. an object cannot become, at runtime, an instance of an arbitrary root class. Instead objects may be reclassified into state classes that are subclasses of the same root class. In order to ensure type safety, state classes are much more restricted: they must be the subclass of a root class and cannot explicitly be used as the type of a field or other heap location. Fickle₃ avoids the syntactic need for the root/state class distinction. The underlying theory for Fickle₃ is more expressive than this presentation, but ultimately uses the same principles.

The root/state distinction is imposed as a way of handling problems that arise as a result of aliasing. Fickle's static type system needs to ensure that only fields/methods that actually exist on an object can be accessed, even in the presence of the fields and methods on that object actively changing as a result of reclassification. It is entirely plausible that several references to an object may exist, and if that object experiences reclassification, we have to restrict all aliases to that object. The solution, in most cases, is to enforce that objects can only be referenced as their root class. Root classes are guaranteed to be stable: only the underlying state class, but not the root class, can change.

Fickle and its successors provide limited and safe access to the underlying state class, so that a program can actually vary according to its current state. Local variables can be declared to have the type of a state class, however, if there's ever a chance that the variable might no longer reference that specific state class as a result of reclassification, then that privilege is taken away from the variable. This manifests as changing its type back into the respective root class. In order to track whether reclassifications may occur, Fickle uses an effect system, present as annotations on each method/function definition stating which root classes it may reclassify. Calling such a method, or performing reclassification in the function body bumps up all relevant local variable types to the root class. The static type checker also asserts that these annotations are present on all functions that either call functions with annotations, or perform reclassification. Because only local variables can have a state class type, with all fields having root class types, we ensure that updating the types of the local variables is sufficient.

2.3.4 EGO

The EGO programming language (Bejleri et al., 2006) takes a different approach, directly modelling an object system closely matching SELF, but with static types. To permit dynamic updates to the available slots of an object, EGO has to introduce linearity into its type system as a way of taming object aliasing.

Linear programming was first introduced into pure functional programming, as a way of safely modelling stateful operations without accidentally duplicating or destroying “the world” (Wadler, 1990). The theory of linear functional programming was itself based upon Girard’s linear logic (1987). The key invariant that a linear type system enforces is that only one reference to an object may exist at a time. It does so by ensuring that once a variable holding a unique reference is consumed by, say, passing it to a function, that variable essentially no longer exists in the type system. Further, a linear type system more strictly requires that the variable is used exactly once. In contrast, an affine type system requires that a variable is used at most once.

The following LIN-APP rule captures the spirit of linear logic, without the complexity of EGO’s type system or the full linear typing rules. The rule gives a linear version of function application, with the conventional rule APP is shown beside it. In the conventional rule, the left hand side must have a function type $U \rightarrow V$, and the right hand side the same type as the argument U , with the result of application having the same type as the result of the function V . In all three judgements, the context Γ is the same, since function application does not change what variables are in scope. The linear version is spiritually similar, but where a function arrow \rightarrow is used in the conventional rule, a linear function arrow \multimap is used instead, and the context changes across each rule, with t and u being checked in disjoint contexts Γ and Δ respectively, while the application holds with combined context Γ, Δ . The disjoint contexts ensure that a variable is only used either on the left or on the right, but not in both. This ensures variables cannot be duplicated or destroyed on the expression level. The linear arrow \multimap is used to express this linearity holds within the function body of t as well, contrasting with the regular arrow \rightarrow which makes no such guarantees.

$$\frac{\Gamma \vdash t : U \multimap V \quad \Delta \vdash u : U}{\Gamma, \Delta \vdash tu : V} \text{LIN-APP} \quad \frac{\Gamma \vdash t : U \rightarrow V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V} \text{APP}$$

By using a more complex version of Wadler’s original linear types (1990), EGO can perform SELF-style object reclassification because it knows that certain object references will always be unique. This means that there cannot possibly be any aliased references that might break soundness, safely allowing the static type change. EGO, much like Wadler’s system, does not enforce linearity for *all* objects as this would too severely restrict the space of programs which could be written; the ability to share references to data is a useful property for flexible program design. Naturally, these non-linear objects cannot be the target of reclassification.

Ego's partial linear types still can be quite restrictive. Sometimes, it is useful to be able to share objects and alias them despite the desire to reclassify them again later. [Kehrt and Aldrich \(2008\)](#) introduce a version of ego featuring borrowing, which allows an object to temporarily give up its linearity and create short lived references.

2.3.5 Object Evolution

The work of [Cohen and Gil \(2009\)](#) introduces a refinement on object reclassification that restricts the operation to be *monotonic*, but as a result allows a more flexible type system that avoids updating a variable's type across method calls or controlling aliasing. As we have discussed, objects feature a subtyping relation that allows instances of an object of type B to be treated as objects of type A, if B extends A. This reclassification is *monotonic*, in the sense that objects may only change from a superclass to a subclass, but not the other way round. We follow [Cohen and Gil \(2009\)](#) and refer to this reclassification as *evolution*. This nomenclature emphasises the monotonicity of the operation.

Monotonicity may seem like too severe a restriction on the capabilities of reclassification, as opposed to arbitrary reclassification. [Cohen and Gil \(2009\)](#) surveyed various examples of programs that use reclassification to identify how regularly they employed non-monotonicity. Across various domains, they found 10 distinct examples, out of which only 3 were non-monotonic, the rest being representable using evolution. Furthermore, as outlined in [Section 1.1](#), we believe there are now more examples of object evolution's applicability in the context of modern OOP languages with regards to safe staged initialisation for example.

Using evolution rather than reclassification also resolves many of the difficulties with handling aliasing. As we have seen, general dynamic reclassification is unsafe, however it *is safe* to perform if the new type of the object is a subtype of its original type. Intuitively, any subtype will satisfy the same contract as the original type, so it is safe to substitute it in place. For example, the operation that evolves a `SyntaxTree` into its subtype `TypedSyntaxTree` via the addition of typing information is safe to perform, since any aliased reference to the original `SyntaxTree` maintains the invariant that it can treat it as a `SyntaxTree`.

Evolution is performed through the use of *evolvers*, which are defined with strong parallels to constructors. Where a constructor performs the actions necessary to initialise an object, instantiate its fields and enforce its invariants, an evolver performs the actions necessary to ensure the object is fully initialised in its new identity, including establishing new invariants and instantiating new fields. We can understand the precise distinction more formally by considering objects as passing through three phases in the body of a JAVA-like constructor for a class B, which extends A, shown in [Figure 8](#). This style of constructor calls the super constructor before initialising any fields. At the entry point of the constructor, we can assume nothing about the state of the receiver **this**

```

class B < A {
  γ : object;
  constructor(x : object, γ : object) {
    // this : object
    super(x);
    // this : A
    this.γ = γ;
    // this : B
  }
}

```

Figure 8: The three stages an object passes through as it is constructed in a constructor and more invariants are established.

and so we are permitted to treat it as the most general possible type: `object`, and nothing more specific. After we call the super constructor `super(...)`, the receiver satisfies the constraints of the inherited class and so we may treat it as an `A`. Finally, once we have initialised all the fields on the receiver, we know that it satisfies the invariants required to be treated as a `B`.

We draw analogy when describing evolvers. An evolver has to establish the new invariants of a class when an object evolves. However, the difference is that in an evolver, *we already know the object satisfies the invariants of its superclass*, because it *is an instance of the superclass*. This means that the structure and semantics of an evolver are exactly that of a constructor, but lacking the initial `super(...)` constructor call. This is illustrated by the example in [Figure 9](#). In this example, we still have to initialise the new field `γ` on `B`, and if we wanted to could invoke arbitrary methods and functions. But because the object `this` starts as an `A`, we do not need to call the `A` constructor.

We introduce the syntax `e evolves C(...)` to our language to represent evolution. This expression evolves the result of evaluating `e` into a `C`. To do this, it calls `C`'s evolver, passing in the arguments "...". The result of this expression is the same as the value of `e`, except that it has now been evolved to become a `C`. Furthermore, the type of the result of evolution is `C`, rather than the less specific type of `e`. The rule `EVO` formalises a typing rule for the style of evolution in [Cohen and Gil \(2009\)](#). We require that the type `A` of the expression `e` is a

```

class B < A {
  γ : Object;
  constructor(γ : Object) evolves A {
    // this : A
    this.γ = γ;
    // this : B
  }
}

```

Figure 9: The two stages an object passes through as it is evolved within an evolver. These stages are the final two stages and invariants in [Figure 8](#).

supertype of the target class B , and we give the result of evolution the type B directly. We also introduce some new meta-syntax here, using the green/thick over-line notation $\overline{e_i}$ to represent lists indexed by i . We use this to represent the list of arguments to the evolver $\overline{e_i}$, and have a corresponding i -indexed list of premises referencing the formal parameter types of the evolver B_i^{evo} , ensuring the arguments match the type of the parameters. If evolution fails due to e not being evolvable into C , then an exception is thrown, much like if evolution were treated as a downcast operation.

$$\frac{\Gamma \vdash e : A \quad B <: A \quad \overline{\Gamma \vdash e_i : B_i^{\text{evo}}}}{\Gamma \vdash e \text{ evolves } B(\overline{e_i}) : B} \text{EVO}$$

Cohen and Gil (2009) discuss various other design decisions they took in their version of object evolution. This includes the automatic declaration of evolvers and implicit chaining of evolution. We describe and consider these features in context where we evaluate them in Section 3.1.

Here we have discussed evolution by inheritance, but Cohen and Gil (2009) describes two other approaches to object evolution, one using mixins and the other using shakeins. Mixins and shakeins introduce ways of constructing subtyping relations without using inheritance. Mixins describe a process for extending a class programmatically, describing how to generically extend any class that satisfies some interface. Shakeins have not been particularly well adopted, but are roughly a generalisation of mixins that generically specify patterns for instrumenting classes to create subclasses. As inheritance is the most popular mechanism for expressing subtyping in object-oriented programming languages, we consign our focus in this thesis to exclusively the inheritance approach.

Having presented the principles of type systems, object orientation and in particular, object evolution, we may now consider the main ideas of this thesis.

Approaches to Safe Object Evolution

In this section we present the design and theory behind our language MAY. This section exists in two halves. Firstly, in [Section 3.1](#) we document the language design considerations for a language featuring object evolution, including evolver semantics ([Section 3.1.1](#)), automatic evolvers and evolution ([Section 3.1.2](#)), field overriding ([Section 3.1.3](#)) and finally constructor semantics ([Section 3.1.4](#)). In the second half, we take an in-depth look at how object evolution can fail and the ways this can be made safer. We consider introducing option types ([Section 3.2.1](#)), present an optimistic but ultimately flawed solution ([Section 3.2.2](#)) and finally use this to motivate our system of permission types ([Section 3.2.3](#)). We present a bidirectional type checking algorithm for our permission types in [Section 3.2.5](#).

3.1 Design Constraints for Object Evolution

[Cohen and Gil \(2009\)](#) present three approaches to object evolution, looking at evolution via inheritance, mixins and shakeins. Our work exclusively follows the inheritance based approach but in doing so we independently explored many of the design considerations in such a system. These are not detailed by [Cohen and Gil \(2009\)](#) and so we choose to describe them here since they play a role in influencing how we provide safe object evolution. In this way, we hope we can document the design space around inheritance based evolution.

3.1.1 Evolvers Are Just Like Constructors

Evolution is an operation that takes an object of one class and returns the same object, but now an instance of a subclass. As a two-class-dependent operation, it is worth considering whether the evolver should be defined on the superclass the object starts as, or as part of the subclass it becomes. We believe that

attaching evolvers to a subclass is the most compelling choice, and that the advantages of a superclass approach can still be met through traditional object-oriented design patterns.

The most compelling reason we see for including evolvers on a subclass is their similarity to constructors. Evolvers need to be able to initialise the internal fields and invariants of a class, and while this could be performed from the superclass, it necessarily entangles the superclass with the internal details of the subclass, breaking principles of encapsulation and abstraction.

Writing evolvers on subclasses also makes evolution more flexible. Classes are traditionally open to inheritance. However if the superclass must detail within its definition all the classes that may evolve from it, then it would be closed to evolution. While placing the barrier that a class is open to inheritance but not to evolution is possible, it is hard to argue that this is a desirable restriction in general. By placing the evolver on the subclass, evolution remains open, and any subclass can add a new evolver.

On the other hand, it seems useful to be able to express within a class declaration that its instances may be evolved. For example, we may wish to say through a class definition that it represents an abstract syntax tree that does not necessarily have type information, but that it can evolve to include it. Additionally, constructors have limited capabilities: they have to create new objects, they can only create instances of a specific class and cannot appear as part of an interface. Using an evolvers-like-constructors approach gives us similar limitations: evolvers can't appear on interfaces, calling an evolver fixes the target class and it guarantees that the object must evolve. While constructors are more flexible in the presence of general evolution, once we introduce permissions to our system in [Section 3.2.3](#) we have to take back this flexibility, so it is not something we want to rely on.

To solve this problem, we cite Item 1 of *Effective Java*, a standard style guide for `JAVA` programs: use factory methods instead of constructors where appropriate ([Bloch, 2017](#)). Given the similarity between constructors and evolvers, we extend the approach to using “factory evolution methods” over direct evolvers where appropriate. These are methods that internally perform evolution on an object. We can then use these to specify on a class that it evolves into a subclass by including a method that invokes the correct evolver. Furthermore, this method could choose between various different evolution targets and could appear as part of an interface. This resolves many of the issues that result from requiring object evolution be specified on subclasses.

3.1.2 Automatic constructors and evolvers

We believe there is value in not performing automatic insertion of evolvers in the presence of existing constructors, in contrast to [Cohen and Gil \(2009\)](#) which present automatic evolvers as part of their system. The parallel that exists

between evolvers and constructors means that it is possible to add/remove the call to the `super()` constructor, when it has no arguments, to translate between the definition of an evolver and constructor. This can be used to automatically insert appropriate evolvers/constructors on classes given the other has been defined. While this may work on the implementation level, and can help reduce code duplication, it has consequences for the interface provided by a class. A class' public interface may intentionally hide a constructor/evolver to enforce the class is used in a particular manner. Furthermore, even if at the type-system level, automatic evolvers/constructors are safe, there are no guarantees that the invariants imposed by the implementation are maintained. This might break abstractions in ways programmers had not anticipated, which is undesirable. Hence, we have chosen not to include this feature in our language.

The other implicit behaviour documented by [Cohen and Gil \(2009\)](#) is automatic transitive evolution. Suppose we have a class hierarchy with three classes: $C <: B <: A$. Transitive evolution allows us to take an instance x of A and evolve it directly into a C using an implicit evolution first into B . However, this is only clearly possible when the evolver from A into B takes no arguments. While transitive evolution adds a nice flexibility to the language, it is fragile in its requirement that the evolver has no arguments, so it is not a feature we choose to include. Instead, a solution for explicit evolution directly from A to C would be more appropriate, or in the spirit of the previous section, the use of a method.

3.1.3 Field overriding

One feature we believe is strongly motivated by the inclusion of object evolution is the ability to override fields in subclasses. Typical object-oriented languages permit method overriding as a way of changing the definition of a method within a subclass, specialising it to that subclass. As an analogue for fields, we might consider fields that get overridden to be specialised to the subclass and as a result get a new specialised type, i.e. a subtype. In practice, this feature is not present in most object-oriented languages, due to the challenges they pose with mutability and initialisation, but we believe evolution motivates an even greater need to incorporate field overriding. As a class evolves, it might want to evolve one or more of its fields along with it, so that both it and its fields become a subtype of the original type. Our motivating example of a syntax tree relies on this: in order to say that during type checking the sub-trees of a syntax tree have also been typed, the types of the fields representing the sub-trees must be a subtype on the subclass. This represents that they've been changed from the supertype `SyntaxTree` to the subtype `TypedSyntaxTree`. When we do this, walking the typed tree does not have to assert that the sub-trees have evolved, this can be enforced by the language.

Field overriding is unfortunately incompatible with mutability. It is always safe to read the field of an object from some superclass, even if that field was

```

class Twig {...}
class Branch < Twig {...}
class Sapling { mut arm : Twig; ... }
class Tree < Sapling { overrides mut arm : Branch; ... }

let tree = new Tree(new Branch());
let sapling : Sapling = tree;
sapling.arm = new Twig();
// Expects Branch but gets Twig!
let x : Branch = tree.arm;

```

Figure 10: An example demonstrating the type unsoundness of writing into an overridden field, by showing how we can use this to get a value of type `Twig` when we expected a value of type `Branch`, a more specific type than `Twig`.

overridden to be more specific in a subclass, because we may always treat that more specific value as if it had the less specific type given in the superclass. Hence we say that reading a field is a *covariant* operation, since the subtyping on the read operation is in the same direction as the subtyping on the surrounding class. On the other hand, we cannot write into an object's field if, at run time, the object is a subclass where the field was overridden to be a subtype. If we were to write into the field, a value less specific than expected by the subclass, we would break the type constraint imposed on that field. Further reads from that field would get us a value with a more specific type than its actual run-time representation, breaking type soundness, as shown in [Figure 10](#). This is a result of updating a field being a *contravariant* operation: the subtyping on the write operation is in the opposite direction to the subtyping on the surrounding class. We would need to require any subclasses accept a *supertype* of the original field's type for writing, since it has to at least be able to handle anything of that satisfies the field type on the superclass. This fundamental disagreement between reading and writing fields means that we cannot have both general mutability and overriding of fields. Instead, we work around this problem by marking certain fields as mutable, and then only allowing fields which are immutable to be overridden.

There is one specific instance in which we know it is safe to mutate an overridden field: inside an evolver. Within an evolver, we know the run-time type of an object exactly, and hence know that it is impossible for us to violate the typing invariants of some potential subclass. This is good news, as we would need to be able to mutate the contents of an overridden field when object evolution occurs to ensure that it satisfies the new object's invariants. However once we start updating these overridden fields within an evolver, they are no longer truly immutable. Hence, it would be unwise from a usability standpoint to claim within the definition of a class that a field is immutable if in fact it is entirely possible for that field to change. Hence an appropriate third kind of mutability, one we mark as **evolves**, is necessary on a field in order to override it.

This indicates that while any user of the class cannot update the field, they also may not rely on the field remaining constant, given the presence of evolution.

3.1.4 Constructor Semantics in the Presence of Evolution

Cohen and Gil (2009) describe the relation between JAVA-style constructors and evolution. We make this relationship even closer by employing a strategy that avoids late-binding of methods in constructors.

As we have described in Section 2.3.5, within a constructor, objects pass through stages as they are initialised, following the inheritance hierarchy. Each class begins its constructor by calling its super constructor, recursively calling the constructor of classes in inheritance order. As each constructor returns, the object becomes progressively more initialised. In a language without default initialisation, we avoid accessing uninitialised fields on a subclass by ensuring only methods that are valid for the object's invariants may be called, as described in Section 2.2.2. Furthermore, within the body of a constructor, the type of the `this` receiver and available methods first become that of the superclass after calling the `super()` constructor and then finally `this` becomes the constructed object after initialising all the fields. This approach models the run-time type of the object updating as it is constructed. We call this the *super-first* approach to constructors, since in this style, constructors begin with a call to `super()`.

In the example in Figure 11, the `increment()` method is overridden by class B. Calling the overridden version depends on the field `y` being initialised. Hence, even if invoked via a call to `new B()`, within the constructor for A, the call to `increment()` cannot invoke the method defined on B. Naïvely, the method table of `this` might have been B's table, which would contain the wrong `increment()` method. Instead the method table is updated incrementally during object construction so that it points to the method table of the class that has been fully initialised. The type of the receiver `this` is hence updated as the object becomes more initialised.

Crucially, this is essentially the same semantics as object evolution, but implicitly. Rather than requiring multiple calls to evolvers that progressively initialise the object, this happens implicitly through the use of `super()` calls. We could model this approach using evolvers by first creating an instance of the topmost object in the inheritance hierarchy, and then initialising down to the object we actually want to create.

As mentioned in Section 2.2.2, this is only one approach to type-safe object constructors. We can also consider the *fields-first* approach. Suppose instead we want to recover dynamic dispatch on methods within a constructor to hit the method of the actual class being constructed. Rather than progressively refining the type of the object being constructed, swapping out its method table in the process, we can initialise its method table exactly once. To do this,

```

class A {
  x : int;
  constructor() {
    this.x = 0;
    // Always calls A.increment()
    this:increment();
  }
  fun increment() {
    this.x += 1;
  }
}

class B < A {
  y : bool;
  constructor() {
    // this : Object
    super();
    // this : A

    // Calls A.increment()
    this:increment();
    this.y = false;
    // B fully initialised
    // Update this's method table
    // this : B

    // Calls B.increment
    this:increment();
  }
  overrides fun increment() {
    super:increment();
    this.y = !this.y;
  }
}

```

Figure 11: A code listing showing that within a super-first constructor, evolution occurs implicitly as invariants of an object are established. The points at which the method table of the object is updated to signify that its run-time type has changed are annotated across two classes A and B.

we need to maintain the invariant that a method is only called once all fields have been initialised, which is achieved by only permitting calls to the `super()` constructor once all fields introduced in the current class have been initialised. Intuitively, at the entry point of a super constructor, the fields of all subclasses have already been initialised. The only contract it needs to fulfil is that all its fields are initialised, as well as the superclass' fields, after which it may call any method. Figure 12 show how our example from Figure 11 results in the `increment()` method on B being called safely in all cases within the constructor.

The trade-off between these approaches is the different kinds of flexibility they enable. The super-first approach allows for calling methods on `this`, as if it were a superclass, during field initialisation. As a result, super-methods can be used to initialise fields. Furthermore, there is an elegant symmetry with evolvers, letting us assign similar semantics and functionality to both. On the other hand, the field-first approach means that method calls within a constructor always respect late binding. While, the super-first approach is similar enough to evolution that it can be emulated through it, the field-first expresses a new kind of construction that is not available in within super-first.

Field overriding does not play particularly nicely with super-constructor-calling object initialisation, since any call to a `super()` constructor, along with

```

class A {
  x : int;
  constructor() {
    // All subclass fields
    // are initialised

    this.x = 0;
    // Object is now fully
    // initialised, so can
    // do dynamic dispatch
    this:increment();
  }
  fun increment() {
    this.x += 1;
  }
}

class B < A{
  y : bool;
  constructor() {
    // this : Object
    // Initialise fields
    this.y = false;
    // Then call super constructor
    super();
    // this : B

    this:increment();
  }
  overrides fun increment() {
    super:increment();
    this.y = !this.y;
  }
}

```

Figure 12: A code listing demonstrating how in the field-first approach to constructors, there is no implicit evolution, and that all fields will be initialised before any methods are called.

an initialisation of the overridden field, means the field must be initialised multiple times. However, the field-first approach is unsound on classes with overriding. In this approach, fields are initialised in the *reverse* order of the subclasses, because fields are initialised before calling the super constructor. As a result, the least specific form of the field must be initialised last. This can result in a value that does not satisfy the overridden type being assigned to the field. It does not mean this approach is totally incompatible with object evolution, but it is incompatible with classes that override fields.

While our MAY compiler currently takes the super-first approach, due to the symmetry it enables, and its semi-compatibility with overridden fields, both seem like reasonable design decisions if one limits constructors to classes without overridden fields. Further research may identify which is more useful in practice.

3.2 Failure of evolution

We now move to a discussion of how object evolution fails and in particular, what we can do about that. Evolution based on inheritance, as presented by [Cohen and Gil \(2009\)](#), is not guaranteed to succeed. Suppose we have a class hierarchy with A and its two direct subclasses B and C, which are not subclasses of each other. Since we are working in the presence of subtyping, at run time a reference to a value of type A may actually be a reference to a run-time B. If we were to try and evolve such a value, we would be turning a run-time value that

```

class Sapling { ... }
class GumTree < Sapling { ... }
class PineTree < Sapling { ... }

let sapling = new Sapling();
let gum_tree = sapling evolves GumTree();

// Fails as 'sapling' is actually a GumTree!
let pine_tree = sapling evolves PineTree();

```

Figure 13: An example showing how evolution can fail when the run-time class of an object is different from its static type.

is really a B into a C, but this is impossible as neither is a subtype of the other! This is illustrated in [Figure 13](#).

In a type system featuring exceptions, it is possible for the runtime to perform the necessary checks for type evolution at runtime and then throw an exception if the evolution is not possible. For example, [Cohen and Gil \(2009\)](#) treat object evolution like a JAVA downcast, which fails if the object is not of the correct type.

However, exceptions, at least anecdotally, seem to be falling out of favour as part of good software engineering practice. This is at least partially evidenced by their absence from newer imperative languages like Go ([Meyerson, 2014](#)) and RUST ([Klabnik and Nichols, 2022](#)), for example. Exceptions provide non-local and hence unpredictable control flow that makes reasoning about the safety properties of code more challenging. They simplify happy-path code but make failure cases implicit and hence hard to glean directly from code. Exceptions have their place, but for the purposes of investigating strong type systems they present a non-solution.

3.2.1 Option Types

Our initial development of object evolution makes failure checks explicit through the use of *option types*, also known as a form of *null safety*. By providing a type system featuring explicit marking of optional references, i.e. references that can be null, we make handling null references explicit. Then, when evolution fails, we have it return a null value, marking the return type as optional. This forces failure of evolution to be handled at the point it may occur. Our system of optional references was inspired by languages like C# ([Ecma International, 2023](#)), KOTLIN ([Akhin and Belyaev, 2021](#)) and ZIG which treat optional values (values that can also be null) as a special type, as opposed to sum types which have some similar features.

The introduction of option types already improves the safety properties of object-oriented languages, by requiring explicit checks for null pointers rather than raising null pointer exceptions, and encoding in the type system when these checks are necessary and when they are not. Experimental tests of JAVA

$$\begin{array}{c}
t ::= \dots \mid ?C \\
e ::= \dots \mid \mathbf{null} \mid \mathbf{if?} \ x := e \{ e' \} \ \mathbf{else} \{ e'' \} \\
\frac{}{\Gamma \vdash \mathbf{null} : ?C} \text{NULL} \quad \frac{\Gamma \vdash e : ?C \quad \Gamma, x : C \vdash e' : t \quad \Gamma \vdash e'' : t}{\Gamma \vdash \mathbf{if?} \ x := e \{ e' \} \ \mathbf{else} \{ e'' \}} \text{IF-NULL} \\
\frac{}{C <: ?C} \text{SUB-OPT} \quad \frac{C_1 <: C_2}{?C_1 <: ?C_2} \text{SUB-OPT-MONO}
\end{array}$$

Figure 14: An extension to a simple language that already features classes but no null values, which adds safe option types and null values. The syntax, typing rules and subtyping rules are extended for constructing and eliminating optional values.

code show that 75% of object references are meant to be non-null by design, and using optional types encodes this invariant into the type system (Chalin and James, 2007).

In Figure 14, we present an extension for optional types, following the rules presented by Fähndrich and Leino (2003a), on top of an existing nominally-typed class-based object-oriented language. The existing language includes types t of classes represented by capital letters like C , expressions e which include variables, and a subtyping relation $<:$.

Firstly, we extend the syntax for types to include not just the names of classes C , but also optional classes $?C$ with the prefix question mark operator. Additionally, we add a `null` value and an `if?` expression for constructing null values and eliminating them respectively. The `if? $x := e \{ e' \} \ \mathbf{else} \{ e'' \}$` expression is evaluated by first evaluating e , checking if it is null, executing e'' if it is, and executing e' if it is not but with the variable x bound to the value e evaluated to. An alternative approach like Ceylon's flow sensitive typing (King, 2011) would allow us to use a traditional `if` statement with `$e == \mathbf{null}$` as its condition. Flow typing lets us update the type of e on the `if` and `else` branches appropriately.

The `NULL` typing rule allows us to type `null` with any optional class. This might appear to cause type checking problems, but in practice we can almost always provide a type for `null` using bidirectional type-checking. The `IF-NULL` rule follows the semantics of the `if?` expression. It ensures that e is indeed an optional type $?C$, checks e' and e'' have the same return type and additionally checks e' with x in the context, with the non-optional type C .

We also provide two new subtyping rules so that optional classes play nicely with non-optional classes and their inheritance structure. Certainly if a value is a non-optional instance of a class C then we can interpret it as if it were an optional $?C$, giving us `SUB-OPT`. Additionally, if we know a value is an optional $?C_1$ and we can treat a C_1 as a C_2 , i.e. $C_1 <: C_2$, then we may treat the value as a $?C_2$. Either the value at run time is null and so certainly is a $?C_2$, or it is a

C_1 which is by assumption a subtype of C_2 . This is expressed in the rule **SUB-OPT-MONO**.

Once we have a system for null-safety in place, we can use it to give a safe type to object evolution. The **EVO-OPT** rule is much like the original **EVO** rule, but the potential failure is now encoded by the resulting type of the **evolves** expression being $?B$.

$$\frac{\Gamma \vdash e : A \quad B <: A \quad \overline{\Gamma \vdash e_i : B_i^{\text{evo}}}}{\Gamma \vdash e \text{ evolves } B(\overline{e_i}) : ?B} \text{EVO-OPT}$$

3.2.2 A Half Solution: Exact Types

The use of optional types certainly guarantees a safe way to perform object evolution, but comes at the price of having to perform mandatory null checks. If we can know statically where evolution will happen and encode this into the type system, we can instead erase these null checks and ensure evolution will *always* succeed.

It does not seem unreasonable that we would know where evolution could occur. For example, when checking a syntax tree and turning it into a typed syntax tree, we know exactly when object evolution occurs: during type checking! Similarly, if performing some form of delayed initialisation, in which the full initialisation of an object occurs after some computation or acquiring some resource, then we know evolution occurs once we are ready to complete initialisation.

In order to perform the run-time evolution of some object x , an instance of class A , into class B , a direct subclass of A , we have to know one thing: that x is *exactly* an instance of an A . That is, at runtime, x is an instance of A and none of its subclasses. We define the word *exact* to refer to this case. Additionally we introduce a new syntax for types $!C$ for any class C to refer to the type of expressions that are exactly a C .

This then would allow us to naively introduce an alternative typing rule **EVO-EXACT** for evolution along with the additional notation $B <!A$ for when B is a direct subclass of A .

$$\frac{\Gamma \vdash e : !A \quad B <!A \quad \overline{\Gamma \vdash e_i : B_i^{\text{evo}}}}{\Gamma \vdash e \text{ evolves } B(\overline{e_i}) : !B} \text{EVO-EXACT}$$

In principle, this rule should never fail, since as long as we can show that e has exactly type A then we can always evolve it into exactly type B . In practice, as we shall see, ensuring that $!A$ really is a run-time A is harder than we have made out so far, as a result of aliasing. We defer these problems for now while we outline the spirit of exact types.

```

let x : !A = new A();
let y = x evolves B();
// Cannot evolve x more than once!
let z = x evolves C();

```

Figure 15: An example program showing that a naïve implementation of exact types does not prevent multiple evolution.

In order to create these exact types, we can modify the typing rule for creating objects to **NEW-EXACT**. The same notation is used for checking argument types, with the formal parameter types of the constructor written A_i^{con} .

$$\frac{\Gamma \vdash e_i : A_i^{\text{con}}}{\Gamma \vdash \text{new } A(\overline{e}_i) : !A} \text{NEW-EXACT}$$

The traditional subtyping rules do not, and in fact, should not carry over to these exact types. While we might know that $B <: A$ for distinct A and B , the only way for $!B <: !A$ to hold is if $B = A$! Otherwise we would be able to say that where we expect something to be exactly an A it was actually exactly a B , breaking what we meant by exactness.

While we are not able to take any old A and treat it as an exact $!A$, the converse is true. Wherever we expect an instance of A we are permitted to use something that is exactly an instance of A . This gives us the subtyping rule that lets us mix exact and inexact types,

$$\frac{}{!A <: A} \text{SUB-EXACT}$$

Things become considerably more complicated once we want to assign these exact types to local variables. The program in [Figure 15](#) illustrates the problem concisely. The type system has no way to know that the object x evolved from an A into a B on line 2, so on line 3, if x still has type $!A$, we can evolve once more into a C , which doesn't make any sense at run time! We can see how this went wrong: while x was indeed exactly an A on the first line, as the type annotation shows, by the conclusion of the second line, x is no longer exactly an A , because its type has changed via evolution! The half-fix we might at this point consider, which is not wrong, but not yet complete, is to either prevent accessing the variable x again, or perhaps less harshly, update its type to A , so that it no longer claims to exactly have the type A when it does not, in fact, have it anymore.

Yet this solution fails to account for *aliasing*. [Figure 16](#) shows this by aliasing the variables $x1$ and $x2$ to be the same exact reference to $!A$. Then we evolves $x1$ into a B but $x2$ into a C . Since these are the same object at run time, this operation would fail and should have been prevented by the type system. To handle this, we need to introduce a system for alias tracking.

```

let x1 : !A = new A();
let x2 : !A = x1;

let y1 = x1 evolves B();
// x2 aliases x1, and x1 has already evolved!
let y2 = x2 evolves C();

```

Figure 16: An example program show a naïve solution that updates the exactness of a type still fails since it does not account for aliasing.

3.2.3 Permission Types

In order to maintain the contract that a variable with type $!A$ has exactly type A , we need to ensure that no aliased reference to the object can evolve it. We do this by tracking not just types but whether a value has the unique permission to be evolved. Thankfully for us, this permission coincides with knowing the type of a variable exactly. When we know the type of a variable exactly, then we ought to be able to evolve it. And conversely, if the variable has the permission to be evolved, then we will know the type exactly because no one else has the permission to evolve it.

Since these concepts coincide, both with each other and also existing work around linearity and ownership, we will refer to the type $!A$ as an *evolvable* reference, an *owned* reference, or just an *exact* reference. The intuition to have for how we treat locations or variables with type $!A$ is that at each step of the program, there may be at most location with type $!A$ that points to each object. This is in many ways similar to the linear or affine type systems discussed in relation to languages like Ego (Bejleri et al., 2006), and more closely aligns with Featherweight Typestate (Garcia et al., 2014, 2010) which we discuss in Section 5.

It might seem at this point that we’ve gone to a lot of effort to implement a monotonic system of object reclassification, with the initial motivation of avoiding a type system that has to keep track of aliasing, and especially a type system that has to expose a permission type system to programmers. And yet, we are now proposing that very thing once again to assert new safety properties that have arisen as a result of the path we have taken. Is this not hypocritical? In some sense it is. The requirement to think carefully about how to orchestrate the language’s features to handle ownership carefully is going to be necessary in our system. However, the control of our ownership system does *not* extend to restrictions on mutability or aliasing, only to object evolution. We posit that this system is more natural than one with a more iron-clad but strict ownership discipline. At the very least, our system presents an interesting exploration of the intersection of evolution and permissions.

We enforce the aliasing restriction of evolvability almost entirely with a pair of rules that control the type of locally scoped variables, and specifically how they change. In order to cope with the idea that the context can change,

we introduce a new form of typing judgement, with both an input and output context. We write

$$\Gamma \vdash e : t \dashv \Gamma'$$

to mean that under context Γ , e has type t and the context is modified to be Γ' , following the notation of [Garcia et al. \(2010\)](#). For now we present these as typing judgements, and in [Section 3.2.5](#) give an algorithmic presentation to these rules using the bidirectional method outlined in [Section 2.1.1](#).

When reading the type of a variable out from the local scope, we have two options, either we use the evolvability of that variable in some way, whether to evolve the object or transfer its evolvability, or we access just the normal shared reference to that object without any evolution permissions. This maintains the invariant that exactly one evolvable reference exists, since either we consume it, or we defer its consumption.

We use the $\text{erase}(t)$ function to represent a type that has its permission erased. So for example $\text{erase}(!A) = A$ and $\text{erase}(?!A) = ?A$ but if no ownership is present, then it is the identity $\text{erase}(?A) = ?A$. Now we may state the two new variable rules.

$$\frac{}{\Gamma, x : t \vdash x : t \dashv \Gamma, x : \text{erase}(t)} \text{VAR-OWN}$$

$$\frac{}{\Gamma, x : t \vdash x : \text{erase}(t) \dashv \Gamma, x : t} \text{VAR-SHARE}$$

The [VAR-OWN](#) rule handles the case where we want to consume the ownership in some way, taking the prior context $\Gamma, x : t$ and erasing the ownership in the final context $\Gamma, x : \text{erase}(t)$. The permission is not lost however as it is present in the type t we may give to x . The [VAR-SHARE](#) rule is dual, erasing the ownership in the type assigned to the expression x , but preserving it in the context. We note that we are using a set based interpretation of contexts here so $\Gamma, x : t$ does not imply x was the most recently added variable to the context, but that we can reorder the context into disjoint parts Γ and $x : t$.

With a new notation for typing contexts available to us, it is worth reexamining how existing typing rules fit into this new model, especially if they don't directly have any influence on the ownership. Here we show an example of the [if](#) rule [IF-PERM](#) which illustrates typing under both sequencing and branching of computation.

$$\frac{\Gamma \vdash e_b : \text{Bool} \dashv \Gamma' \quad \Gamma' \vdash e_1 : t \dashv \Gamma'' \quad \Gamma' \vdash e_2 : t \dashv \Gamma''}{\Gamma \vdash \text{if } e_b \{ e_1 \} \text{ else } \{ e_2 \} : t \dashv \Gamma''} \text{IF-PERM}$$

The semantics for evaluating an [if](#) expression are to first evaluate the condition e_b and then to either evaluate expression e_1 or e_2 , and finally join control flow back together. This is reflected in the flow of contexts and ownership: e_b is able to use any of the evolvable references available in Γ since it executes

first, and passes on the updated post-context Γ' . Then when checking e_1 and e_2 , we have to use this updated context so that we do not consume ownerships consumed within e_b . However, since only one of e_1 and e_2 is executed, we can type check both with the context Γ' . Finally, we join control flow back up which requires us to merge the contexts we get after checking e_1 and e_2 . A more sophisticated merging operation might be more flexible, but for now we assert that the same ownerships were erased in both branches.

The alias-tracking evolvable types are an extension of the half-solution description of exact types from Section 3.2.2. Hence, we keep the rules around subtyping, **new** and **evolves** for evolvable types, but extend them to include pre- and post-contexts.

$$\frac{}{!A <: A} \text{SUB-PERM} \quad \frac{\overline{\Gamma_i \vdash e_i : A_i^{\text{con}} \dashv \Gamma_{i+1}}}{\Gamma_0 \vdash \mathbf{new} A(\overline{e_i}) : !A \dashv \Gamma_n} \text{NEW-PERM}$$

$$\frac{\Gamma \vdash e : !A \dashv \Gamma_0 \quad B < !A \quad \overline{\Gamma_i \vdash e_i : B_i^{\text{evo}} \dashv \Gamma_{i+1}}}{\Gamma \vdash e \mathbf{evolves} B(\overline{e_i}) : !B \dashv \Gamma_n} \text{EVO-PERM}$$

Since evolvable references are just types, it still makes sense to be able to have function parameter and return types be evolvable references. In order to pass a value into an owned parameter of a function, we have to consume the ownership of the argument, because the ownership is passed with the value into the function. Within the body of the function, the formal parameter is treated as an owned local variable and thus can have its ownership used exactly once, as with any other local variables. Returning an owned value works without change, as long as the value being returned really is an owned value.

By default, calling a method on an object does not give up ownership of that object, yet if we wish for an object to be able to evolve itself, we must have a way of doing that. Since the receiver of a method is passed as a parameter, we may annotate the method to say it is passed with its ownership, having the desired effect. Figure 17 illustrates how this can be used with **MAY** syntax, noting that the **evolves** prefix on the method *evolveMe()* indicates that the receiver is passed in with ownership. Importantly, in subclasses, unlike with regular inheritance, these **evolves** methods are not inherited and available on the subclass. This prevents us from accidentally invoking *evolveMe()* on some $x : !B$ which would break the subtyping property which ensures $!B \not<: !A$. Thankfully this does not interfere with inheritance in general, since these methods are only available when the type of the receiver is known exactly.

Variables and function arguments are not the only way we can store references to objects and hence evolvable objects. Evolvable objects could be stored in the fields of classes, global variables (if present) and as items in a primitive array type. The language we have developed does not have global variables, but does have primitive arrays/slices. To have a complete picture of permissions,

```

fun evolveA(a : !A) : !B {
    return a evolves B();
}
class A {...}
class B < A {...}

class A {
    evolves fun evolveMe() : !B {
        return this evolves B();
    }
}
class B < A {...}

```

Figure 17: A function or a method can be used to abstract over evolution by passing in and returning owned references.

we need to understand how it integrates with non-local storage locations. We use arrays as our example here.

Initialising and writing to a location is straightforward. Given some array $a : []!A$ we are always permitted to write/overwrite some entry $a[i]$ with another $b : !A$ as long as we erase the permission from that b . We would lose the ownership of the entry being overwritten, but this seems reasonable in the context of overwriting the value anyway. To initialise an array of evolvable values, we may provide a list of references directly $[x_1, \dots, x_N]$, but are not permitted to use any kind of evolvable default-value initialiser.

In order to take an exact reference to an object out of some location in memory, it is not sufficient to read that location. An array of owned references $a : []!A$ has the same type after reading out the first value $a[0]$. The permission system does not track arrays of evolvable types, like $[]!A$, which allows multiple references to the array to exist at once. But it also means we cannot safely erase the permission from the whole array, which would likely be undesirable for consuming just a single index anyway. Under these constraints, no method for reading a value from an array with its evolvability will satisfy the invariant that exactly one evolvable reference may exist for an object at once, since there is another reference left behind in the array. We introduce a swap operator $x \succ=< y$ in the style of [Harms and Weide \(1991\)](#) to enable safely reading heap locations to evolvable references. We are allowed to swap $a[0]$ with some other exact instance $b : !A$ in the expression $a[0] \succ=< b$, which exchanges the references they contain. This lets us retain the invariant that exactly one evolvable reference exists at a time, because we have replaced evolvable references with each other. This technique of using a swap operator to explicitly control aliasing has been used in other systems featuring uniqueness and permissions ([Garcia et al., 2010](#), [Haller and Odersky, 2010](#)).

It is tempting to provide access to an evolvable `this` pointer within a constructor or an evolver, to allow for evolution within that constructor/evolver. Indeed, it might seem that within a constructor or evolver we know the type of the object exactly, but this isn't quite true. For standard constructors there is no guarantee the constructor corresponds exactly to its class, because constructors can be called as super constructors. Calling an evolver in the constructor for an A to turn it into a B , only for the object actually being constructed as a C

```

class Sapling {
  evolves arm : !Twig;
  constructor(child : !A) {
    this.child = child;
  }
  ...
}
class Tree {
  overrides arm : !Branch
  constructor() evolves A {
    this.arm = this.arm evolves Branch();
  }
  ...
}

```

Figure 18: An example showing how a field can be overridden in a subclass in a way that retains its evolvability. In this example Branch extends Twig.

<: A, completely invalidates our safety properties. Therefore we cannot hold ownership over the receiver in a constructor. This pitfall is not unique to the ownership approach. We can statically prevent it in this ownership approach, but the problem does not disappear when using null-types to check evolution at runtime.

The second problem, which again applies both to evolvers and constructors, is the return type of these constructs. In order to have the permission to create an object and then evolve it multiple times, it is necessary that each time we call an evolver or a constructor we get back an exact type. If we could change the runtime type of an object inside a constructor or evolver, we would no longer reliably be able to do this. Constructors and evolvers could in theory be extended to mark a custom return type, but this sort of defeats the purpose of them, and instead methods make for a more optimal approach (Bloch, 2017).

Our presentation of field overriding in Section 3.1.3 integrates well with object evolution. When no evolution permission is present on a field’s type, overriding behaves as normal. When the permission *is* present, this indicates that the class itself has the permission to evolve the field. But this isn’t something it should be able to do arbitrarily, since the type of the field would have to change correspondingly with its evolution. Hence, we expose the permission exactly within the evolver in a subclass that overrides the field, where it can be suitably updated. Even though subtyping does not apply to exact types $!A \not\prec !B$, we allow field overriding to perform subtyping along with evolvability. We can do this because outside of an evolver, the evolvability of the field is totally inaccessible. This is a consequence of the immutability of override-able fields that prevents reading out or inspecting the evolution permission outside an evolver. Inside an evolver, overridden fields behave slightly strangely, in that they may be accessed as their old type, but have to be written to as their new

type. [Figure 18](#) illustrates very briefly what evolving a field looks like in this system.

3.2.4 Exactness and Interfaces

In the previous section we explored how we were able to incorporate a permission system for tracking exact types with class-based hierarchies. However many modern languages, even those without explicit object-orientation, include interfaces as a way of defining abstractions ([Ramalingam and Srinivasan, 1999](#)). Since no runtime object is ever exactly an instance of an interface, since interfaces don't have notions like initialisation, or often even fields, it feels as if exact typing and ownership should be mutually exclusive from any discussion of interfaces. However, we have found the opposite, incorporating interfaces into our theory of exact typing results in a richer and more expressive theory.

Once again we consider the motivating problem of typing syntax trees. In our previous descriptions of this approach, we have implicitly hidden behind the idea of an “untyped syntax tree” and a “typed syntax tree” as things we can talk about in our language. We have also stated that our typed syntax trees should be subtypes of untyped syntax trees. But syntax trees are made up of concrete syntax nodes, and it is those that are incarnated into our language in the form of classes and objects. So how can we represent our syntax tree, typed and untyped, that we need in order to write down what the type of the sub-nodes of a syntax node, and what it is we evolve when type checking the tree?

The construct we need here is an interface, and specifically an interface that can carry the permissions required for evolution. We consider an example class for a kind of syntax node: a `BinaryOp` in both its untyped and typed (`TypedBinaryOp`) forms, for representing binary operators. Intuitively, we should treat any `BinaryOp` as a `SyntaxTree`, and any `TypedBinaryOp` as a `TypedSyntaxTree`. Furthermore, every `TypedBinaryOp` is an appropriate untyped `BinaryOp` by erasing the type information, and similarly we may take `TypedSyntaxTree` to be a `SyntaxTree`, this gives us the inheritance diagram in [Figure 19](#), including the evolvable types `!BinaryOp` and `!TypedBinaryOp`. The corresponding code is in [Figure 20](#).

Every syntax node can evolve into a typed syntax node, and so we lift this property onto the interface itself. But to do this we need a notion of *evolvable interfaces*. An evolvable interface does not indicate an ability to be evolved, or an exactness that mirrors a runtime notion of types, since an interface cannot have either of these properties. Instead, ownership of an interface reflects ownership of the underlying object. Thus the permission is tracked in the same way as it would be if it appeared on a class annotation. An owned interface is available to receive methods that require an owned receiver, which lazily binds to the underlying class. There the actual evolution permission is exposed back to the type system through the concrete method that gets called.

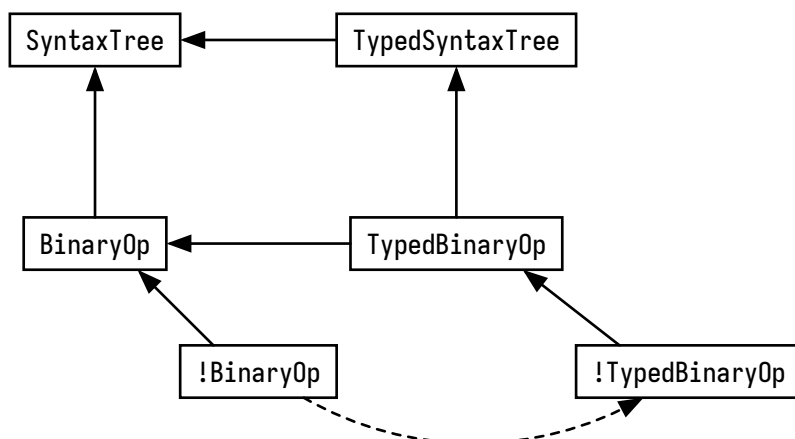


Figure 19: Subtyping diagram for the binary operator classes with and without ownership annotations, and syntax tree interfaces. The dashed line shows where evolution can occur.

While this answers how we might pass from owned interfaces back to owned classes by preserving the uniqueness of the evolution permission, going from an owned object to an owned interface is slightly more subtle. It makes sense that we may treat a `!BinaryOp` as a `!SyntaxTree` and include this as a subtyping relation, since we want to be able to call `evolves` methods on `BinaryOp` through the interface, for the purposes of type checking for example. The situation is not quite so clear for `TypedBinaryOp`, which should no longer be evolvable by type checking. By transitivity of subtyping, `TypedBinaryOp` implements `SyntaxTree`. However, just as an `evolves` method is not inherited by subclasses, `TypedBinaryOp` should not be required to implement the `evolves` method `typeCheck()` on `SyntaxTree`. To keep this system sound while also omitting the method from `TypedBinaryOp`, we can't have the relation `!TypedBinaryOp <: !SyntaxTree`, even if we have `TypedBinaryOp <: SyntaxTree`.

```

interface SyntaxTree {
  evolves fun typeCheck() : !TypedSyntaxTree
}

class BinaryOp implements SyntaxTree {
  evolves fun typeCheck() : !TypedSyntaxTree { ... }
}

interface TypedSyntaxTree { ... }
class TypedBinaryOp < BinaryOp
  implements TypedSyntaxTree { ... }
  
```

Figure 20: The interface and class structure for a syntax tree and typed syntax tree in MAY syntax. Methods for evolution are part of the interface for `SyntaxTree` but not `TypedSyntaxTree` and so do not need to be implemented on `TypedBinaryOp`.

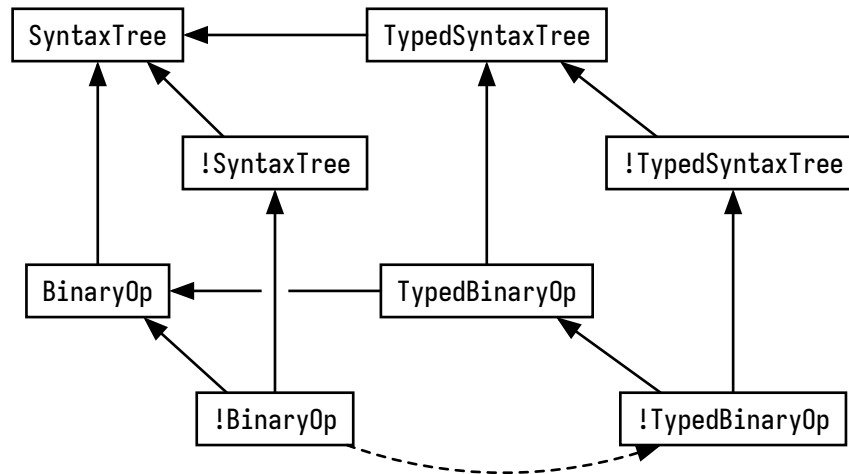


Figure 21: Subtyping diagram for the binary operator classes and syntax tree interfaces showing ownership for both classes and interfaces. The dashed line shows where evolution can occur.

To make the distinction clear between when we have subtyping on evolvable classes and interfaces, we distinguish between interfaces that are implemented *directly*, i.e. in the class' signature, and those implemented *indirectly*, implicitly through the superclass or another interface. In Figure 20, TypedBinaryOp directly implements TypedSyntaxTree since it explicitly appears in the **implements** clause, but SyntaxTree is indirectly implemented because it does not appear explicitly but is still implemented by transitivity. Using this distinction, we require only classes that *directly* implement an interface to provide implementations of all its **evolves** methods. Consequently, we get an owned subtyping relation $!C <: !I$ for class C and interface I only when C implements I directly. To ensure this rule holds in the presence of transitivity, we restrict subtyping on owned interfaces in the same way as classes: $!I <: !J$ only when $I = J$.

Figure 21 modifies the original inheritance hierarchy of Figure 19 to include the owned interfaces and the subsequent inheritance relations we derive from these rules. We observe that the subtyping relations (but naturally not the evolution relation) on the bottom face of the cube, on the class level, lift exactly to the top face of the cube along the vertical edges to the interface level. Additionally, the subtyping structure on owned types is now more rich than just the trivial discrete order with no relations. Between two classes and between two interfaces, the owned type ordering remains discrete, but from class to interface the relation may include arbitrarily many of the subtyping relations present in regular classes to regular interfaces. This gives us a subtyping relation of height at most 1.

3.2.5 Type Checking and Inference

To implement our permission types in the MAY compiler, we need to be able to algorithmically perform type checking and synthesis in a way that requires minimal annotations. We use a bidirectional-style typing algorithm for this purpose.

The algorithm has to infer when a local variable should be accessed in a way that consumes its evolution permission, or if instead it should be accessed in its erased form. In Section 3.2.3, we introduced two typing rules for variables that handle these cases: **VAR-OWN** and **VAR-SHARE**. But, just by looking at the syntax of the term, it is unclear which rule should be applied. However, if we know the type the variable should have, then we *can* tell. If the variable access should have a type *with* evolution permissions then we have to use the **VAR-OWN** rule, since the **VAR-SHARE** rule always erases any permissions. This indicates that we can use a bidirectional type system. If we are *checking* the type of a variable has the evolves permission, we use the **VAR-OWN** rule, but conservatively within *synthesis* we use the **VAR-SHARE** rule.

$$\frac{}{\Gamma, x : t \vdash x : t \dashv \Gamma, x : \text{erase}(t)} \text{VAR-OWN}$$

$$\frac{}{\Gamma, x : t \vdash x : \text{erase}(t) \dashv \Gamma, x : t} \text{VAR-SHARE}$$

Firstly, we extend the description of bidirectional rules we gave for standard typing judgements to our stateful typing contexts. Our typing rules now take in the initial context Γ as input, and produce the final context Γ' as output.

Γ	\vdash	e	\Leftarrow	A	\dashv	Γ'
\uparrow		\uparrow		\uparrow		\downarrow
in		in		in		out

In a context Γ , the term e can be *checked* to have type A and produces context Γ' .

Γ	\vdash	e	\Rightarrow	A	\dashv	Γ'
\uparrow		\uparrow		\downarrow		\downarrow
in		in		out		out

In a context Γ , the term e *synthesises* the type A and produces context Γ' .

We can use this to give two typing rules for variables, one that checks the type of a variable x , and one that synthesises. The checking rule ensures that the type of the variable t is suitable for the type it is being checked as t' . The second premise also ensure the checking type t' is an evolvable type, to avoid needlessly consuming the permission of $x : t$. If any of these conditions are not met, the **VAR-SYN** rule can be used instead.

$$\frac{t <: t' \quad t' \neq \text{erase}(t')}{\Gamma, x : t \vdash x \Leftarrow t' \dashv \Gamma, x : \text{erase}(t)} \text{VAR-CHK}$$

$$\frac{}{\Gamma, x : t \vdash x \Rightarrow \text{erase}(t) \dashv \Gamma, x : t} \text{VAR-SYN}$$

In order for our bidirectional system to be effective, we have to ensure that where ever a variable is used with the intent of consuming its permission, we do so. For example, this means that evolution should always check its left hand side. The **EVO-SYN** rule is the bidirectional checking rule for evolution. The class D is the strict superclass of C , which we use to check e is an evolvable $!D$. Then each argument x_i is checked in order, ensuring it has the parameter type C_i^{evo} .

$$\frac{C <!D \quad \Gamma \vdash e \Leftarrow !D \dashv \Gamma_0 \quad \overline{\Gamma_i \vdash e_i \Leftarrow C_i^{\text{evo}} \dashv \Gamma_{i+1}}}{\Gamma \vdash e \text{ evolves } C(\overline{e_i}) \Rightarrow !C \dashv \Gamma_n} \text{EVO-SYN}$$

The function application rule is similar in that it checks each of its arguments. This is standard for function application but also ensures that it is possible to infer when a variable is passed as the argument to a function along with its permission. In **APP-SYN**, the function type is written $(\overline{t_i})^n \rightarrow t_{\text{out}}$ to mean that the function takes n inputs each of type t_i , and returns a value of type t_{out} .

$$\frac{\Gamma \vdash f \Rightarrow (\overline{t_i})^n \rightarrow t_{\text{out}} \dashv \Gamma_0 \quad \overline{\Gamma_i \vdash x_i \Leftarrow t_i \dashv \Gamma_{i+1}}}{\Gamma \vdash f(\overline{x_i}) \Rightarrow t_{\text{out}} \dashv \Gamma_n} \text{APP-SYN}$$

Finally, it is important that eliminators like **if** have both checking and synthesis variants. This way we can pass checking through to the expressions e_i to correctly consume their permission. In the presentation of **IF-SYN** and **IF-CHK**, we implicitly check that the output contexts and types for both branches are the same.

$$\frac{\Gamma \vdash b \Rightarrow \text{Bool} \dashv \Gamma' \quad \Gamma' \vdash e_1 \Rightarrow t \dashv \Gamma'' \quad \Gamma' \vdash e_2 \Leftarrow t \dashv \Gamma''}{\Gamma \vdash \text{if } b \{ e_1 \} \text{ else } \{ e_2 \} \Rightarrow t \dashv \Gamma''} \text{IF-SYN}$$

$$\frac{\Gamma \vdash b \Rightarrow \text{Bool} \dashv \Gamma' \quad \Gamma' \vdash e_1 \Leftarrow t \dashv \Gamma'' \quad \Gamma' \vdash e_2 \Leftarrow t \dashv \Gamma''}{\Gamma \vdash \text{if } b \{ e_1 \} \text{ else } \{ e_2 \} \Leftarrow t \dashv \Gamma''} \text{IF-CHK}$$

In our experience, this system has been effective at inferring permission type usage without additional annotation on variable usages in a way that is predictable.

Armed with an understanding of the influences of object evolution on language design, and the theory of type safe object evolution, we explore the implementation of object evolution in our language **MAY**.

The MAY Language

The language features described within this thesis have been developed as part of a bespoke experimental programming language called MAY (Mirrlees-Black, 2025). It is a simple core language that is designed for easy control and modification, suited to our experimentation. The language is implemented as a compiler from MAY source files to QBE's (Carbonneaux, 2024) intermediate language, which can then be compiled to an executable binary on several 64 bit platforms. The compiler itself is written in the OCAML language. We give a more precise overview of the MAY language in Section 4.1, followed by the design and design choices in our compiler in Section 4.2. An analysis of some case studies writing code in the MAY compiler is given in Section 4.3. Approaches to implementing object evolution are described in Section 4.4.

4.1 Language Overview

MAY is a JAVA-style expression-oriented language featuring nominal subtyping, classes and interfaces, null-safety, arrays and naturally, object evolution. In this section we document the syntax and semantics of the language as a reference point. While syntax has been implicitly and explicitly introduced throughout the thesis so far, we provide a complete description of all the language's features here.

4.1.1 Primitive types

MAY has a handful of non-object built-in primitive types:

- **int**: represented as sequences of digits, e.g. 123. ints compile to 64 bit integers.
- **char**: represented either as a character/escape sequence surrounded by a single quotes, e.g. 'a' or the ascii value of a character terminated by c, e.g. 97c. chars compile to 8 bit integers.
- **bool**: has values **true** and **false**.
- **unit**: has exactly one value ().
- **noreturn**: the bottom type, has no values.

- **object**: the supertype of all objects.

The primitive unary and binary operators are:

- Standard arithmetic operators and comparisons over integers: `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`
- Referential equality: `==`, `!=`
- Boolean short circuiting operators: **and**, **or**
- Unary boolean negation: `!`
- Unary integer negation: `-`

4.1.2 Arrays

MAY has both immutable arrays, using the syntax `[]T` and mutable arrays `[]mut T` for any type `T`. Arrays can be created in three ways:

- `new [length]T(defaultValue)`: to define an array of type `[]mut T` with length `length` and default value `defaultValue`.
- `[x1, ..., xN]`: to define an array of length `N` with contents `xI` at index `I`.
- `"string contents"`: to define an immutable array of type `[]char` with bytes given by the contents of the string

Arrays can:

- Be indexed by an integer starting at 0: `"hello"[1] == 'e'`
- Be sliced by a pair of integers `String.equal("hello"[1..4], "e11")`.
- Have their length taken as if it were a field `"hello".len == 5`.

4.1.3 Option Types

MAY also features option types, given the notation `?T`. Only reference types (arrays and objects) can be annotated as optional. Values of type `T` will be implicitly cast to type `?T` where necessary, such as when passing a non-optional object to a function which accepts an optional argument. The following syntax relates to option types:

- `null` has type `?T` for any `T`. If this `T` cannot be inferred, a type annotation is required.
- `if? x = value { ... } else { ... }` tests whether `value` is null or not. If it is not null, then the value is assigned to the variable `x` in the first scoped block `{ ... }` which is then executed, otherwise the second scoped block is executed.
- `x or else y` evaluates to `x` if `x` is not null. Otherwise it evaluates to `y`.
- `x.?` asserts to the compiler that `x` is non null, aborting the program if it is.

4.1.4 Functions and Control Flow

Functions are defined using the `fun` keyword, with the return type annotated after the parameter list. Functions that do not return a value may use the `unit` return type. The last expression in a function block is implicitly returned, however `return` statements are also available for explicit returns.

```
fun minimum(a : int, b : int) : int {  
  if a <= b { a } else { b }  
}
```

MAY's expression oriented semantics means that control flow structures like `if` can appear both in statement and expression position. As shown in this example, an `if` expression evaluates to the final value in either its left or right hand blocks.

Variables are always mutable and can be defined within a function using the `let` keyword. They can optionally feature type annotations to check the right hand side as the given type instead of inferring it.

```
fun main() : unit {  
  let x = 5;  
  x = 10;  
  let y : int = x;  
  return;  
}
```

The other primitive control flow structure may has is `while` loops, given the following syntax:

```
fun factorial(x : int): int {  
  let total = 1;  
  while x >= 1 {  
    total = x * total;  
    x = x - 1;  
  }  
  return total;  
}
```

4.1.5 Modules

Code can be organised using modules, which may contain functions, class definitions and global constants, as well as other modules:

```
module A {  
  module B {  
    const magic_number = 42;  
  }  
  
  fun getMagicNumber() : int {  
    B.magic_number  
  }  
}
```

Code can be loaded from other files into a named module as

```
import MyLibrary from "lib/library.may";
```

Code can be ordered and split across files arbitrarily, including in a circular fashion:

```

// In "odd.may"
import Even from "even.may";
fun odd(i : int): bool {
  if i == 0 { false } else { Even.even(i - 1) }
}

// In "even.may"
import Odd from "odd.may";

fun even(i : int): bool {
  if i == 0 { true } else { Odd.odd(i - 1) }
}

```

As an escape hatch, calling out to C function can be done using the **extern fun** syntax. This is used in the very sparse standard library to define primitive functions like string printing.

```
extern fun print([]char) : unit = "internal_may_print_string";
```

4.1.6 Classes and Interfaces

As an object-oriented language, MAY features both classes, defined using the **class** keyword and interfaces, defined using the **interface** keyword. Classes inherit from other classes by noting their superclass after the **<** operator, and specify the interfaces they implement after the **implements** keyword. Subtyping in MAY is *nominal* and so these annotations are required to establish inheritance relationships.

A class' fields in MAY are defined by stating the field name and type separated by a colon, terminated by a semicolon. By default, fields are immutable and private. However they can be marked as **public** to change their visibility, **mut** to be mutable, **evolves** to allow the field to evolve when the whole class evolves, or **overrides** when the field in the superclass is **evolves**, to give a more specific type for the field, .

Methods are defined by declaring functions within a class scope. The receiver of the method is accessible through the implicit **this** parameter. Methods are public by default but can be marked as **private**. To override the method of a superclass, the **overrides** keyword must be prefixed.

Constructors must call the super constructor **super(...)** when the class extends another class, and must always initialise all class fields defined on its class. Once this contract has been fulfilled, the type of the constructed receiver **this** is promoted to the type of the current class. If the super constructor has been called but not all class fields have been initialised, then the constructed receiver has satisfied the initialisation requirements of at most the superclass, and so is given the type of the superclass.

Interfaces are made up of several method signatures: function definitions with no body. Interfaces are also able to implement other interfaces, requiring

```

interface Drawable { fun draw() : unit; }
class Rectangle implements Drawable {
  // Fields
  width : int;
  height : int;

  constructor(width : int, height : int) {
    this.width = width;
    this.height = height;
  }
  // Methods
  fun draw() : unit { ... }
}

class Square < Rectangle {
  constructor(sideLength : int) {
    super(sideLength, sideLength);
  }
  overrides fun draw() : unit { ... }
}

```

Figure 22: An example showing the syntax for declaring interfaces and classes in MAY

that any class that implements this interface, must also implement all parent interfaces. Interfaces do not expose fields.

MAY has no notion of “static” variables/methods associated with a class/interface. Instead, static methods should be defined as functions outside a class definition.

Instances of classes are created by using `new Class(args...)` syntax. Fields are accessed using “.” dot notation like `rect.width`. Methods are accessed, to avoid ambiguity, through “:” colon notation, such as `rect:draw()`. Within a method definition, fields and methods must be explicitly accessed through the receiver `this` rather than implicitly resolved.

Classes can include evolvers, specified by annotating `evolves c` after the parameter list of a constructor. Currently, the class name must exactly match the parent class, following the single step approach to evolution we have taken. However, the syntax is flexible enough to allow for multiple evolution constructors to exist from various different parent classes and is a feature that may be explored in future research. An evolution constructor must initialise all fields declared in the current class, as well as any fields with `overrides` annotations.

The evolver is invoked when an object undergoes object evolution, which is performed using the syntax `untyped_num evolves TypedInt(int_type)`. This invokes the `TypedInt` evolver on the `untyped_num` term with the argument `int_type`. When

```

class TypedInt < Num {
  type : Type
  constructor(Type type) evolves Num { this.type = type; }
}

```

Figure 23: An example showing a simple evolver using MAY's syntax

MAY is invoked without ownership, the type of this evolves expression is ? TypedInt to indicate the result is either a TypedInt or that evolution failed.

4.1.7 Permissions

When the may compiler is invoked using the `-ownership` flag, the syntax and typing rules are extended and updated to use the permission system developed in [Section 3.2.3](#) for tracking evolvability. With this enabled, a class type can be annotated with a bang/exclamation mark `!C` to indicate that it is evolvable. Invoking constructors by `new C()` now return a value of type `!C`, and evolution of the form `x evolves C()` requires `x` to have an evolvable type, and returns an exact `!C`.

Local variables with exact types implicitly pass on their permission when used for evolution or passed to a function taking evolvable parameters. Other kinds of storage locations (arrays and class fields) do not permit reading out values with evolvability permission. While values can be mutated using the normal assignment operator, reading requires the use of the exchange or swap operator `>=<`, to ensure uniqueness of permissions. An example of using the swap operator is shown in [Figure 24](#).

Methods and method signatures in an interface can be marked with an `evolves` prefix to require that the method be called on an exact reference and hence marks the receiver as exact. This lets methods perform object evolution on themselves. `evolves` methods are not inherited by subclasses. This is shown for methods in [Figure 25](#). When interfaces mark a method signature as `evolves`, classes which *directly* implement the interface, such as `UntypedFunCall` in [Figure 26](#), must implement the method. However indirect implementers, such

```

fun f(array1: []!C): ?!C {
  if array1.len == 0 {
    null
  } else {
    let c = new C();
    array1[0] >=< c;
    c
  }
}

```

Figure 24: A program that returns the first element of an array of evolvable values with type `!C`, or `null` if the array has zero length. It manages permissions by using the swap operator `>=<`.

```

class C {
  constructor C() {}
  evolves fun evolveToD(): !D {
    this evolves D()
  }
}

fun f() : !D {
  let c : !C = new C();
  c:evolveToD()
}

```

Figure 25: An example program that illustrates how **evolves** methods may be defined and invoked

```

interface SyntaxTree { evolves fun typeTree(): !TypedSyntaxTree; }
interface TypedSyntaxTree implements SyntaxTree { ... }
class UntypedFunCall implements SyntaxTree {
  // Must implement `typeTree`
}
class TypedFunCall < UntypedFunCall implements TypedSyntaxTree {
  // Does not have to implement `typeTree`
}

```

Figure 26: An example showing the use of **evolves** methods within interfaces and on which classes these methods must be implemented.

as `TypedFunCall` in the example, do not have to implement these **evolves** method signatures.

MAY allows fields to be marked as **evolves**, shown in [Figure 27](#). This allows the field to evolve when the class itself evolves, taking on a subtype of the original field in a subclass. In a syntax tree that becomes typed, we can express that the sub-trees of a syntax node evolve into typed trees when the whole operator becomes typed. In subclasses, these fields may be overridden, which requires that they are updated within an **evolves** constructor. An **evolves** field cannot be mutable, and only has an accessible evolution permission within the constructor where it is initially assigned, and the evolver where it is consumed. Within the current MAY compiler, and source examples presented in this chapter, the evolution permission is implicit.

4.2 Compiler Design and Motivation

The compiler design is fairly standard, featuring lexing, parsing, type checking and code-generation phases. The lexer is defined using `ocamllex` in `lib/lexer.ml`. The parser is defined using `menhir` ([Régis-Gianas, 2016](#)) in `lib/menhir_parser.mly`. Type checking is mostly contained to the `lib/check.ml` file. Code-generation to

```

class UntypedOperator implements SyntaxTree {
    evolves lhs : SyntaxTree;
    evolves rhs : SyntaxTree;
    ...
}

class TypedOperator < UntypedOperator implements TypedSyntaxTree {
    overrides lhs : TypedSyntaxTree;
    overrides rhs : TypedSyntaxTree;
    type : Type;

    constructor() evolves UntypedOperator {
        this.lhs = this.lhs.typeTree();
        this.rhs = this.rhs.typeTree();
        this.type = inferTypeOp(this.lhs, this.rhs);
    }
    ...
}

```

Figure 27: An example showing how `evolves` fields and `overrides` fields can be used in conjunction to describe fields whose type becomes more specific as part of evolution.

QBE (Carbonneaux, 2024) is specified in the `lib/qbe_backend.ml` file. The choice to use QBE was primarily motivated by the relatively small scale of the project over alternatives like LLVM (Lattner and Adve, 2004). This results in an easily distributable compiler and fast compile times, while still achieving reasonable performance.

The compiler can be invoked in one of two modes: with option-based evolution and with ownership-based evolution. Within our testing, we have given option-based evolution files the suffix `.may`, and the ownership based evolution files the suffix `.mayo` as an abbreviation of “MAY with Ownership”. Both languages feature optional types, however only the ownership version features ownership types.

Our main motivation in developing a new language as opposed to implementing the feature within an existing language is that having total control over the language design and runtime essentially gives us a sandbox within which we can explore the design considerations effectively. We certainly model MAY’s class system on existing mainstream statically-typed object-oriented languages like JAVA and C#, however such languages are complex and highly feature-full. Both languages feature exception handling, generics and reflection that interact with object evolution in complex ways. Changes to these languages would have to preserve the semantics of existing programs, or at least change them in a coherent way.

Furthermore, we want to understand the performance characteristics of various implementations of object evolution and testing these within complex language runtimes that make assumptions about object layouts adds barriers

to experimentation. Writing MAY as a standalone language implementation has been a considerable undertaking, but has also enabled a flexibility in design that could not be so easily achieved through the use of an existing tool.

4.3 Case Study

As a case study, we have implemented a type checker and pretty printer for the IMP v1 language (Winskel, 1993) inside MAY. The language is used to teach the Principles of Programming Languages (COMP3610 / COMP6361) course at the Australian National University. In this section, we demonstrate how object evolution with ownership can be effectively used to safely perform in-place type-checking of a syntax tree. However, we also acknowledge some of the existing limitations with the current ownership system and the tricks we used to get around them. The full program source text can be found in the compiler repository at `test/end_to_end/imp.mayo`.

The IMP language, specified in Figure 28, is an imperative language featuring a store of integers n labelled by locations ℓ . Integers can be summed and compared, locations can be dereferenced and assigned to, statements can be sequenced and we have structured control flow using **if** and **while** constructs.

The abstract syntax tree (AST) is represented within MAY using the visitor pattern (Gamma et al., 1995). Each syntax former is associated to a class that implements the `Ast` interface, and in order to implement the `Ast` interface a class must be able to accept an instance of an `AstVisitor` interface and call out to one of its `visit()` methods. In Figure 29 we give part of the `Ast` and `AstVisitor` definitions and the classes corresponding to literal integers n , `LitInt`, and binary operators $E \text{ op } E$, `Op`. To allow syntax trees to become typed syntax trees, we mark each sub-tree field as **evolves**.

IMP is a typed language featuring 3 types: `int`, `bool` and `unit`. Typed AST nodes, with examples shown in Figure 30, inherit from their untyped

$$\begin{aligned}
 n &\in \mathbb{Z} \\
 b &\in \{\mathbf{true}, \mathbf{false}\} \\
 \ell &\in \{\ell_0, \ell_1, \dots\} \\
 op &::= + \mid \geq \\
 E &::= n \mid b \mid E \text{ op } E \\
 &\quad \mid !\ell \mid \ell := E \\
 &\quad \mid \mathbf{skip} \mid E; E \\
 &\quad \mid \mathbf{if } E \mathbf{ then } E \mathbf{ else } E \\
 &\quad \mid \mathbf{while } E \mathbf{ do } E
 \end{aligned}$$

Figure 28: The definition for the IMP language

```

interface Ast {
  fun accept(AstVisitor) : unit;
  evolves fun typeCheck(Checker, Type) : !Typed.TyAst;
}
interface AstVisitor {
  fun visitLitInt(LitInt) : unit;
  fun visitOp(Op) : unit;
  // fun visit...(...) : unit;
}
class LitInt implements Ast {
  public value : int;
  constructor(value : int) { this.value = value; }
  fun accept(v : AstVisitor) : unit { v.visitLitInt(this); }
  // ...
}
class Op implements Ast {
  public evolves lhs : Ast;
  public evolves rhs : Ast;
  public op : Token;
  constructor(lhs : !Ast, op : Token, rhs : !Ast) {
    this.lhs = lhs;
    this.rhs = rhs;
    this.op = op;
  }
  fun accept(v : AstVisitor) : unit { v.visitOp(this); }
  // ...
}

```

Figure 29: A snippet of code from the definition of IMP syntax in `MAX`.

counterparts, but satisfy the `TyAst` (short for Typed AST) interface with the corresponding `TyAstVisitor`. Note that at this stage, the untyped `Op` node is specialised into either the typed `TyAdd` or `TyGt` node, depending on the operator. Within each typed syntax node, the fields corresponding to sub-trees of the original syntax tree (`Ast`) are overridden to be typed nodes (`TyAst`).

All untyped syntax trees can be type checked, which we represent with a `typeCheck()` method on the `Ast` interface, shown in [Figure 29](#). This method is marked as `evolves` to indicate that it consumes the evolution permission, and returns a `!TyAst`: the evolved AST node and its evolution permission. Because only untyped AST nodes explicitly implement this interface, typed AST nodes do not need to implement the `typeCheck()` method. Each untyped syntax tree implements this method by calling the evolver to the typed tree, with some examples in [Figure 31](#). There is some complexity in the `Op` case, as the operator first evolves to `TyOp` before evolving to either `TyAdd` or `TyGt`.

The evolvers for each node handle the logic for type checking according to the typing rules. Two of the rules—`IMP-INT` and `IMP-OP+`—are shown below, and the implementation of these rules is provided in [Figure 32](#).

$$\frac{}{\Gamma \vdash n : \text{int}} \text{IMP-INT} \quad \frac{\Gamma \vdash E : \text{int} \quad \Gamma \vdash E' : \text{int}}{\Gamma \vdash E + E' : \text{int}} \text{IMP-OP+}$$

⋮

```

interface TyAst implements Ast {
  fun acceptTy(TyAstVisitor) : unit;
  fun getType() : Type;
}
interface TyAstVisitor {
  fun visitLitInt(TyLitInt) : unit;
  fun visitAdd(TyAdd) : unit;
  fun visitGt(TyGt) : unit;
  // ...
}
class TyOp < Op {
  public overrides evolves lhs : TyAst;
  public overrides evolves rhs : TyAst;
  type : Type;
  fun getType() : Type { this.type }
  // ...
}
class TyAdd < TyOp implements TyAst { ... }
class TyGt < TyOp implements TyAst { ... }
class TyLitInt < LitInt implements TyAst {
  type : Type;
  fun getType() : Type { this.type }
  // ...
}

```

Figure 30: A snippet of code from the *typed* syntax definition for IMP in MAY.

The fun `expect(Type, Type): unit` method defined on the `Checker` class asserts that two types are equal. If they are not, then the program crashes with an error message. This is the first current limitation with the ownership based type system for object evolution: its inability to handle failures within the evolver itself. A more sophisticated system would not require the `expect()` method to halt the program in the event of an error, and instead ought to be capable

```

class LitInt implements Ast {
  //...
  evolves fun typeCheck(c : Checker, type : Type) : !TyAst {
    this evolves TyLitInt(c, type)
  }
}

class Op implements Ast {
  //...
  evolves fun typeCheck(c : Checker, type : Type) : !TyAst {
    let tyOp = this evolves TyOp(c, type);
    if this.op.isAdd() {
      tyOp evolves TyAdd(c)
    } else if this.op.isGt() {
      tyOp evolves TyGt(c)
    } else {
      let ignore: !TyOp = tyOp; // compiler limitation
      Std.String.fail("Unknkown OP token")
    }
  }
}

```

Figure 31: A snippet of code, from the IMP type checker in MAY, showing the `typeCheck()` methods invoking the appropriate evolver.

```

class TyLitInt < Untyped.LitInt implements TyAst {
  // ...
  constructor(c : Checker, type : Type) evolves Untyped.LitInt {
    c:expect(c:types().int, type);
    this.type = type;
  }
}
class TyOp < Untyped.Op {
  // ...
  constructor(c : Checker, type : Type) evolves Untyped.Op {
    this.type = type;

    this.lhs = this.lhs:typeCheck(c, c:types().int);
    this.rhs = this.rhs:typeCheck(c, c:types().int);
  }
}
class TyAdd < TyOp implements TyAst {
  // ...
  constructor(c : Checker) evolves TyOp {
    c:expect(c:types().int, this.type);
  }
}

```

Figure 32: A snippet of code, from the IMP type checker in MAY, showing the implementation of the **IMP-INT** and **IMP-OP+** rules within the evolvers for the `TyLitInt`, `TyOp` and `TyAdd` classes.

of passing that error on to the callee where it can be handled appropriately. Exceptions thrown in evolvers fail in the presence of ownership, since the object may be left in a state of limbo. If an exception is thrown in `TyOp`'s evolver when type checking the right-hand side, then the operator will have been partially evolved. At this point during the program's execution, the run-time type of the left-hand side is `!TyAst`, but the run-time type of the right-hand side is `!Ast`. Hence, `this` neither has type `!TyOp` nor `!Op`, since both types require the exact types of the left- and right-hand side to be the same.

This limitation is less present within the permission-free option-based approach to object evolution. The same *semantic* problem arises with half-evolved object, but it does not break soundness as there are no exact types in this system. In this system, object fields also have no restriction on *when* they are evolved, making it possible to evolve the fields evolving the object itself. This permits gracefully handling errors without failure within evolvers occurring. Further research may determine whether ergonomic approaches like this are available to evolvable types.

The final interesting piece of code is the type checker's entry point, shown in [Figure 33](#). Since we have not implemented parsing, the IMP program is represented directly by constructing the objects corresponding to the untyped syntax. The type checker is declared along with the in-scope location names and passed to the checker.

The `mkSeq()` function is a shorthand for chaining an array of commands into `Seq` nodes, which only have a left- and right- hand side. The implementation

```

fun main() : unit {
  let ast: !Untyped.Ast =
    mkSeq(
      [ new Assign("l1", new LitInt(0))
        , new Assign("l2", new LitInt(10))
        , new Assign("l3",
          new Op(
            new Deref("l1")
            , Tokens.mkAdd()
            , new Deref("l2")
          ))
      ])
  ast:accept(new AstPrinter());
  Std.String.println("\nType Checking...");

  // Create Checker with the in-scope locations
  let c = new Checker(["l1", "l2", "l3"]);
  let typed_ast: !TyAst = ast:typeCheck(c, c:types().unit);

  typed_ast:acceptTy(new TyAstPrinter());
  Std.String.println("\nDone!");
}

```

Figure 33:

of the `mkSeq()` function, shown in [Figure 34](#), represents another difficulty with handling permissions: the need to use the swap operator `>=<` in order to extract the evolution permission out of an array. Since the `Seq` node requires evolvable left and right hand side, this is necessary. The swap operator is used with optional typing to swap array items with `null`. This requires locally breaking safety guarantees and using the `?.?` operator to explicitly cast `?!Ast` to `!Ast`. This technique also requires that the array passed to the function be mutable and potentially contain null values. Both of these requirements are reasonably undesirable.

```

fun mkSeq(stmts: []mut ?!Ast) : !Ast {
  if stmts.len == 0 { return new Skip(); }
  let node: ?!Ast = null;
  node >=< stmts[0];
  let i = 1;
  while i < stmts.len {
    let rhs: ?!Ast = null;
    rhs >=< stmts[i];
    let lhs: ?!Ast = null;
    lhs >=< node;
    let newNode: ?!Ast = new Seq(lhs.?, rhs.?);
    node >=< newNode;
    i = i + 1;
  }
  node.?
}

```

Figure 34: An implementation of `mkSeq` that requires the use of the swap operator, bypassing null safety checks, and mutability of an array of evolvable references

```
fun mkSeq(stmts: ![]Ast) : !Ast {
  let node: !Ast = new Skip();
  for! stmt in stmts {
    let oldNode = new Skip();
    node >=< oldNode;
    node = new Seq(oldNode, stmt);
  }
  node
}
```

Figure 35: Hypothetical implementation of `mkSeq` using a for loop.

To this problem we have potential solutions. By introducing a new syntactic construct for consuming the values of an array we can avoid the need to mutate the array while still being able to pull the permission out of each entry. For example, a for loop (called `foreach` in some languages) could remove the ownership from the array and pass it through to each item during iteration. To correctly enforce the uniqueness of the permission on the array, we allow arrays to accept an ownership permission. Figure 35 shows what the `mkSeq()` example might look like if we had for loops in the language. There may be problems with this implementation that we have not fully explored yet.

We hope future case studies will continue to influence the design decisions involved in safe evolution, just as we have been influenced by this case study.

4.4 Compiling Object Evolution

Many different approaches exist for representing object evolution efficiently at runtime, and exploring the performance characteristics of each approach remains incomplete. We provide some estimates on the theoretical performance of each approach however accurately measuring these costs requires a realistic benchmarking suite. The main challenge in compiling object evolution is that the size of an object must be dynamically increased to permit additional fields, in a way transparent to aliasing—all references have to observe the result of evolution. Extending the method table or overriding methods on the other hand, can be performed by just replacing the old method table pointer with the new one.

The first approach we consider is the one the current MAY compiler implements: pre-allocating the memory that might be needed for object evolution at construction time. This is shown in Figure 36. When we know at compile time the entire structure of the inheritance tree and the presence of evolvers on each class, we can compute the maximum amount of memory for storing fields that might be needed by object evolution at each class. When no evolution is present within a program, this method incurs no overhead. Additionally, little overhead is encountered if in most cases objects evolve to their maximum

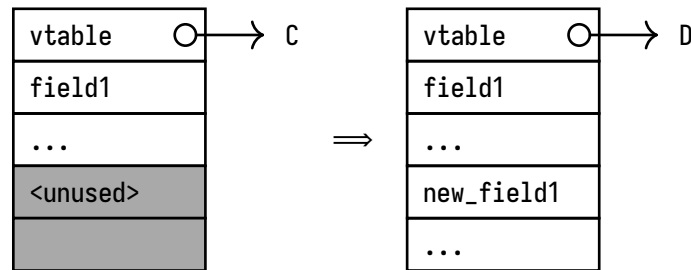


Figure 36: Object evolution implemented by pre-allocating sufficient memory.

size and do so relatively quickly. However the obvious problem introduced by this approach is the introduction of fragmentation. We can consider the pathological case where a small object can evolve to a much larger object. If it only does so very rarely and the small object is used frequently, then we pay an unnecessary overhead in memory use for the possibility that each small object might have to become much larger. Additionally, this approach fails in the presence of dynamic class loading and dynamic linking without careful linker extensions. In these cases we have to be able to compile code without having complete knowledge of the inheritance hierarchy and thus we cannot compute the maximum size of an object.

The following three approaches were presented by [Cohen and Gil \(2009\)](#).

We could theoretically leverage the properties of a moving/copying garbage collector to perform object evolution. Such a collector can move objects in memory and so by invoking the garbage collector each time we need to expand the size of an object we are sure to update all the pointers to the old object so that they now point to the new object. This preserves the performance of code without object evolution, but incurs a substantial cost in order to perform evolution.

Forwarding pointers are an alternative to direct pointer references ([Brooks, 1984](#)). Instead of an object reference pointing directly to its location in memory, it points to an *indirect* or *forwarding* pointer that points to the object's location. The forwarding pointer is the unique direct referent of the object, and so when the object is reallocated, only the forwarding pointer needs to be updated to the object's new location, shown in [Figure 37](#). All other references to the object will observe the evolution indirectly, since the location of the forwarding pointer does not need to change. When these forwarding pointers really do not move (often in the context of a moving garbage collector) they are also known as *handles* ([Kalibera and Jones, 2011](#)).

While this solves the problem of unnecessary memory usage, we have to copy objects in order to evolve them and now pay the cost of double pointer lookups for all object accesses. This is an unconditional read barrier since we have to perform an additional dereference to read an object's fields. While we haven't measured this cost ourselves, we know we can expect around a

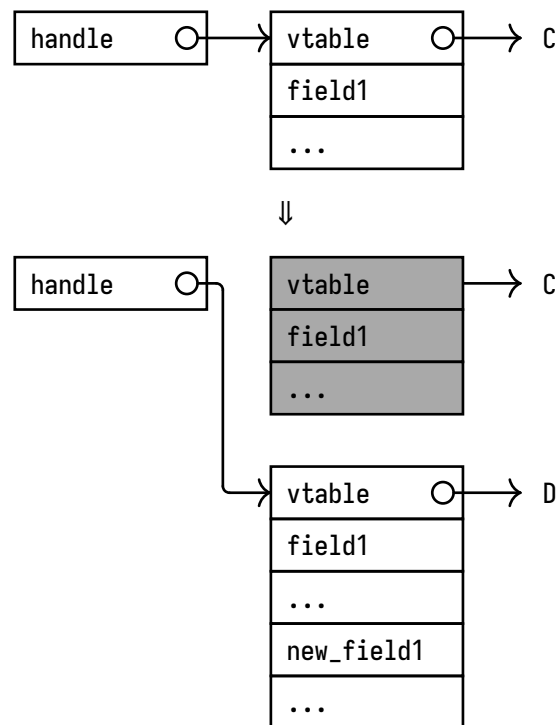


Figure 37: Object evolution implemented by forwarding pointers/handles.

10% overhead (Yang et al., 2012). Naturally, this will be dependent on precise program behaviour.

A simple optimisation on this technique is to allocate the forwarding pointer and object together, with the forwarding pointer immediately preceding the object itself. This way, if the object has not evolved, the forwarding pointer will point back on itself. Since we had to load the pointer itself into cache to read it, we know that the method table and early fields of the object will also lie in cache now. This is the approach taken by Brooks (1984) for LISP.

A slightly more complicated solution is to make forwarding pointers optional. Only when an object evolves do we create a forwarding pointer at the location of the old object. This way, all old references to the object now point to a forwarding pointer, which is automatically dereferenced when present to get to the real object, shown in Figure 38. This necessitates a conditional read barrier, a piece of code that runs on every object access to check whether that object has become a forwarding pointer. Yet we only have to pay the double pointer indirection cost once per reference, since if we see a forwarding pointer, we can patch the reference to point to the new object, skipping the forwarding pointer in future. This approach, without this patching optimisation, seems to be the approach taken in Cohen and Gil (2009) and is the final approach of theirs we present.

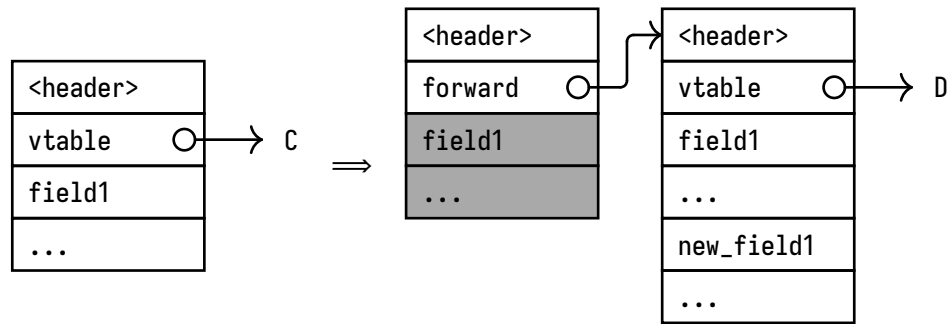


Figure 38: Object evolution implemented by optional forwarding pointers.

Objects can be initially allocated with the space for an extra pointer after its fields. When an object evolves and requires additional space to store its new fields, we update this pointer to a fresh allocation, shown in Figure 39. This eliminates the need to copy-on-evolution that the handle/forwarding pointer based approaches have. However it raises the question of how to compile inheritance for subclasses constructed directly. For a representation that is uniform with evolution, the fields of an object could be represented as an intrusive linked list, and so the fields of the class C_n in an inheritance hierarchy $C_n <: C_{n-1} <: \dots <: C_1$ would live at the n th item in the list. Evolution in this context is just appending the new class's fields to this list.

This probably has too large an overhead to be practical, and so instead we can make these extension pointers optional, just as with forwarding pointers. By default the fields can be allocated in a block, but in case the object was evolved it is necessary to check a flag in the object header to know if instead the pointer to the remaining fields must be used. Additionally, if using a moving GC we can fix up the evolved case to look like the preallocated case at GC time, however we can delay all these fix ups until the GC runs rather than eagerly running the GC when evolving.

The original SELF Just-In-Time compiler used a rather unique strategy for allowing the size of objects to change at run time (Chambers et al., 1989). The approach takes the idea that “all references to the old object should see the evolved object” literally. When the size of an object changes, the process' memory is scanned completely, and any pointers to the old object are updated to the new allocation's address directly in memory. This operation is naturally

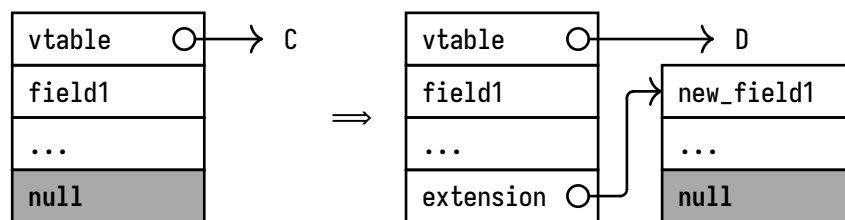


Figure 39: Object evolution implemented by extension pointers.

$O(n)$ in the heap size of the process. There are optimisations considered by [Chambers et al. \(1989\)](#) such as using a segregated heap for pure data and references, as well as tagging pointers to distinguish pointers vs integers without a look-up. While this approach is expensive for object evolution, it results in theoretically no overhead at all when object evolution is not used. However, we consider the cost of evolution in this system too large to bear if we want to genuinely encourage it as a technique one may structure a program around.

4.4.1 Optimisations in the presence of object evolution

Object evolution prevents certain optimisation techniques from being considered that might have been applicable in an object-oriented language without it. Devirtualisation is a technique for reducing the cost of virtual method calls that require multiple dereferences and are hard for branch predictors to effectively reason about ([Ishizaki et al., 2000](#)). Languages like C++ make it undefined behaviour to update the method table of an object, and so compilers are free to optimise repeated accesses to the same method table by accessing the table once and reusing the results ([Padlewski, 2017](#)). This optimisation is not sound in MAY since the method table can change unpredictably between accesses.

We can in theory partially reclaim some of this optimisation in MAY with Ownership. Since owned types !A only change their type in a predictable way, we can still perform this optimisation for types which are owned. We have not implemented this optimisation in MAY and instead only consider it as a theoretical note.

Alternative devirtualisation techniques that don't rely on the method table pointer being stable can still be applied in the presence of evolution. For example, we could still improve branch predictor performance by using a polymorphic inline cache ([Hölzle et al., 1991](#)). This tests and branches explicitly on the method table's pointer (avoiding dereferencing it), and calling the method directly.

In the dynamic context we have already discussed languages like SELF and JAVASCRIPT which feature prototypical inheritance and hence rely on dynamic object reclassification. The semantics of these languages mandate that fields and methods can be added to an object at run-time. At the same time, considerable research has been invested into ensuring these languages run as fast as possible with original research on SELF pioneering Just-In-Time (JIT) compilation ([Chambers et al., 1989](#)). By analysing run-time behaviour, JIT compilers can make optimisations for the common case, such as assuming that an object does not evolve, and then include an efficient check which falls back to a slow path if it has. Since object evolution is a special case of reclassification, the same JIT techniques would apply to MAY.

Related Work

In this section, we give an overview of the related work and describe its relationship to our work. We examine related work on object evolution and reclassification, *typestate* and its applications to protocol correctness as well as *typestate* oriented programming, and finally consider object initialisation.

5.1 Object Reclassification and Evolution

While most of the in-depth detail on reclassification and evolution can be found in [Section 2.3](#) there are two other related works we discuss here. [Sieve et al. \(2025\)](#) integrates a knowledge base with dynamic object reclassification to allow for declarative reclassification. The system focuses on modelling real-world objects like water pumps using OOP and reclassifying objects by finding a suitable class in the presence of a changing context. Aliasing seems to be handled in a similar way to Fickle ([Damiani et al., 2003](#); [Drossopoulou et al., 2002, 2001](#)) but with restrictions around objects that have methods with activation records on the program stack.

[Ciaffaglione et al. \(2021\)](#) give a prototype-based object calculus λObj^\oplus designed to handle reclassification through evolution. Their structurally typed calculus allows objects to extend themselves through the addition of new methods. To ensure safety of evolution in the presence of subtyping, the types of objects both include records of possible methods with their types, and the methods that are actually accessible. As a functional object-oriented calculus, λObj^\oplus is immutable and does not tackle the challenges that mutability raises.

5.2 Typestate

While our work directly incorporates ideas of object reclassification, these ideas are in principle quite similar to *typestate*. *Typestate* extends the notion of a

“type” to capture the state of program objects and how they change during program execution. Just as we’ve explored in the context of object reclassification, aliasing presents a problem to checking typestate statically. We consider the evolution of typestate techniques, how these approaches have tackled aliasing, and how this compares to our work.

Typestate (Strom and Yemini, 1986) has its origins in detecting uninitialised variable access. The core principle is that the type of a variable can change across the scope in which it is declared, in this case to indicate if it has been initialised and/or finalised/deallocated. As a result we derive temporal safety properties not present within traditional type systems. The NIL language (Strom and Yemini, 1986) which initially introduced typestate effectively forbade aliasing, except where it could be resolved at compile time, such as between local variables. This was effective at ensuring correctness, but highly reduces the expressiveness of programs written in NIL.

5.2.1 Typestate Checking

Typestate has been effectively used as a technique for proving properties about stateful protocols in object-oriented languages, with regards to both their definition and their usage (DeLine and Fähndrich, 2004a). An object-oriented file is an example of a stateful protocol used in object-oriented languages. Files can either be in an open state or a closed state, and transition from open to closed by a `close()` method. In order to read or write to the file, it must be in the open state, but in the closed state these operations are not available. In a statically typed programming language like Java these states are encoded implicitly and only exposed to the programmer through documentation and throwing exceptions.

Early systems like Fugue (DeLine and Fähndrich, 2004b) were able to perform simple kinds of typestate checking using annotations inside another OOP language. These indicated the properties fields possessed when an object was in certain states, as well as the state of an object on method entry and exit. For the file example, these annotations would ensure that read and write operations were only performed when the file is in the “open” state, and that once the file transitions to the “closed” state, via the `close()` method, no file operations would be performed. However, Fugue placed heavy restrictions on the aliasing of objects, and would lose any ability to reason about them in the presence of aliasing. Once an object transitions from `NotAliased` to `MaybeAliased`, the typestate must not change. Additionally, Fugue struggled to handle class-based inheritance in the presence of typestate annotations.

Approaches to typestate that are monotonic have been presented as an alternative to the systems for managing aliasing, in much the same way that object evolution is a monotonic version of dynamic object reclassification. Fähndrich and Leino (2003b) introduces a system they call *heap monotonic typestate* which provides a system for enforcing program invariants over the heap in a way that

may only become stronger, not weaker. This way, once an invariant has been established, it can never be broken, and so arbitrary aliasing of that invariant is permitted. Their system uses annotations on class fields and methods that state certain properties, such as enforcing a field is non-null in certain states, and that certain states can only be reached after being in other states. Although this approach is mostly designed in the context of aliasing, the system does have the power to allow non-monotonic heap updates when non-aliasing can be shown.

An alternative approach to overcoming aliasing restrictions is to perform whole program analysis (Fink et al., 2008). This sacrifices modularity and reduces checker performance but removes the need to annotate programs with sophisticated aliasing properties. Additionally, the lack of modularity makes it harder for programmers to reason about why certain stages of typestate checking fail.

The Plural system (Bierhoff and Aldrich, 2008) was developed for performing typestate checks on top of the Java language. It was built around the theory of *access permissions* for typestate programs (Bierhoff and Aldrich, 2007), which had the goal of expressing and verifying protocol APIs in the presence of aliasing. The system assigns object references one of 5 permissions that vary in the access the current reference has (read-only or read/write) and the presence of can permission of other references (non-existent, read-only or read/write). These permissions can be split and joined between references by using fractional permissions (Boyland, 2003), and govern how particular references may update state and invariants. In practice, this permission system can be onerous to reason about effectively, due to the many potential annotations and the unintuitive handling of fractional permissions.

5.2.2 Typestate-oriented Programming

Typestate-oriented programming (TSOP) (Aldrich et al., 2009) is an approach to structuring programs explicitly around their typestate, rather than treating typestate as a separate checking stage over a more traditional object-oriented language. The Plaid language is the first example of such a typestate-oriented programming language. Plaid extends the notion of objects to include states, similar to Fickle's notion of state classes. The language structures the available methods and fields around these states, instead of keeping them implicit in the code structure but verified by annotations. Furthermore, the language lifts the annotations that describe how the receiver and argument states change into the typing annotations for parameters. In order to handle aliasing, Plaid uses a system of permissions adapted from the more complex set of access permissions in Plural. These permissions govern how the typestate of an object may change. A unique permission, which has restricted aliasing, allows performing a guaranteed state change, but an `immutable` permission allows for arbitrary aliasing, without state changes. Using a shared permission allows for aliasing

but requires dynamically testing the state at runtime and that states must be conserved when known, but does not provide any guarantees as to whether the state might change later.

Approaches for TSOP have typically avoided introducing fractional permissions or using non-modular approaches to splitting and joining permissions, due to their unnatural semantics and fragility. [Naden et al. \(2012\)](#) introduced a system for borrowing permissions that is general enough to handle both the `unique` and `immutable` permissions used in `Plaid`. Borrowing allows references even with `unique` permissions to be temporarily shared within a block of code, but ensure that at the termination of the block, the invariant initially established by the permission has been reestablished, i.e. a shared `unique` reference must be `unique` again. To track this, the existing permissions are combined with a local permission, which when annotated on a reference, guarantees it cannot be stored to the heap. This ensures that borrowed references are only present within local variables and hence go out of scope at the end of a block. Borrowing doesn't have an analogue within our system for evolution permission, because in a sense, every evolvable reference is automatically "borrowed" when it is used without its evolution permission. Furthermore, the ability to lend out an evolvable reference and recover it later is not a particularly useful property to have, due to the monotonicity of evolution. Lending out an evolvable reference is only useful if evolution is performed, otherwise one would not lend it out at all, but then if evolution *did* occur, the permission will have been consumed and the underlying type must have changed.

The key principles of TSOP were formalised into the Featherweight Typestate calculus ([Garcia et al., 2014, 2010](#)) inspired by the formalisation of nominal class-based object-oriented principles into Featherweight `JAVA` ([Igarashi et al., 2001](#)). The calculus limits the language to the features of Featherweight `JAVA`: classes, methods and fields, but also includes typestate features for changing states, assigning permissions and borrowing/sharing them. The calculus conflates the notions of state and class, resulting in a system that is very similar to full dynamic object reclassification. Naturally, this requires handling permissions carefully, which is done using a more expressive system than in `Plaid` or `Plural`. Permissions are paired not only with an objects current type/state but also the state guarantee that all other references rely on. This results in a system where typestate can be changed in the presence of aliasing, given sufficient permissions and as long as the guarantee provided to other references is not broken. This gives the system more expressivity while requiring fewer types of permission. This approach can also be viewed as a generalisation of our evolution permission system, but a generalisation that results in a lot of additional complexity in the surface language and the type system. Compared to type-safe object evolution, Featherweight Typestate also lacks an equivalent notion of evolvers or evolvable fields, instead necessitating the less-flexible swap operation to extract a permission back out of an object's fields.

In order to reduce the burden placed on the programmer while working in a TSOP language, [Wolff et al. \(2011\)](#) introduced a gradual type system that mixes the static guarantees afforded by the type system with static checks that can be performed in the dynamic parts of the program. This eliminates the need to always/initially express programs along with explicit descriptions of their state transitions and permissions, as these properties can be checked at runtime. Additionally, a gradual type system allows for programs that might not otherwise be expressible in the rigid static system, but are still valid and safe programs to be included in the language since they may rely on dynamic parts of the program. Gradual typing ([Siek and Taha, 2006](#), [Tobin-Hochstadt and Felleisen, 2006](#)) introduces the notion of a dynamic type `dyn` into an otherwise static type system, and in order to ensure the static type system remains sound, dynamic checks are performed at the interface between dynamic and static code. Since typestate systems involve permissions, being able to check the run-time type requires not only checking the data representation, but also the necessary properties of the set of pointers in the heap and the permissions those pointers have. As part of the verification effort, the minimal Gradual Featherweight Typestate calculus was developed as a gradual extension of the Featherweight Typestate calculus ([Garcia et al., 2014](#)). While we do not yet have a gradual version of our evolution type system, the same considerations will apply. We are less motivated by reducing verbosity, since our system has low notational overhead, but there are certainly programs which cannot be written in a permission based system because they rely on *not* being able to infer where evolution will occur. A gradual type system would be able to resolve this as long as at run time we can check whether other evolvable references exist.

TSOP has seen usage for blockchain smart-contract projects such as Obsidian ([Coblentz et al., 2020](#)). Smart-contracts are programs that can run on the blockchain to add automated but tamper-resistant functionality. By using typestate and linear types, Obsidian can be used to model blockchain assets with greater assurance of program correctness, which is vital when handling financial data in an immutable system.

Ownership and affine type systems have grown in popularity due to the success of the Rust programming language ([Klabnik and Nichols, 2022](#)), where they have primarily been used as a way of tracking object lifetimes statically and guaranteeing memory safety without a garbage collector. Rust’s powerful metaprogramming system combined with trait constraints (similar to interfaces or type-classes) have lent themselves to expressive domain specific languages (DSLs). This includes a system for typestate in Rust ([Duarte and Ravara, 2021](#)) that uses the built-in Borrow Checker to handle alias tracking. Rust’s ownership system is an appealing type system for handling unique references and managing mutability, which align themselves neatly with the aliasing requirements for a general typestate system. However a Rust-style system lacks much of the flexibility afforded by our type-safe evolution system.

Because evolution relies on weaker aliasing guarantees to ensure type safety, type-safe evolution is less suited to being translated into a Rust library.

5.3 Object Initialisation

Object evolution is a useful approach for phased initialisation as it lets us express how we may extend objects with data that only becomes available later. [Fähndrich and Xia \(2007\)](#) introduce a notion of delayed types to allow for expressing circular initialisation patterns without requiring temporary null values. This system does not allow them to express our notion of staged initialisation by evolution, but we, on the other hand, struggle to express their circular data structures without resorting back to null values.

[Reynaud et al. \(2021\)](#) provides a modal system for recursive definitions in OCAML. This system has to handle recursive `let rec` definitions that may place the variable being recursively defined within arbitrary computation. The primary focus here is avoiding the ability to write down unsound definitions, such as those of the form `let rec x = 1 + x;;` but permitting even definitions of circular data structures like `let rec ones = 1 :: ones;;`. Unlike [Fähndrich and Xia \(2007\)](#), modes are not exposed to the surface syntax and so do not permit modular definitions of recursive structures. The nature of constructors-as-code present in object-oriented languages makes these non-modular approaches less extendable, however the underlying theory would be ultimately useful for constructing recursive data structures effectively.

[Qi and Myers \(2009\)](#) give a powerful heap-monotonic-typestate solution to object initialisation, known as masked types. Masked types specify constraints on the initialisation status of an object, such as which fields have been initialised, that update as the object becomes more initialised. These constraints extend to include conditional masks, which express that the initialisation status of one field can depend on another, and mask effects that encode the effect of method calls on initialisation. This system allows complex self-referential data-structures to be constructed safely without null types. The use of masked types allows for object evolution's approach to staged initialisation, since it gives a type to objects that have uninitialised fields. However, the system's completeness comes at the cost of complexity and syntactic overhead. Additionally, placing the names of class fields into the program's types prevents abstraction which is not desirable for general programs, even if it is necessary for initialisation.

[Summers and Müller \(2011\)](#) instead introduce a distinction between references to partially-initialised and fully-initialised objects, along with commitment points where objects transition between these stages. The system requires minimal syntactic and semantic overhead, particularly compared to masked

types, but still permits annotations that give rise to modular, sound and potentially circular initialisation. However, in doing so, this approach loses the ability to capture evolution-style initialisation.

Concluding Remarks

In this section we discuss some of the key areas we think would be fruitful for future work, and then close with an overview of the work in this thesis and some final reflections.

6.1 Future Work

The MAY language presents exciting new possibilities for exploring object evolution further, and the type systems we use to manage it. One notable absence from this thesis is that we have not proved type-safety. This is work that has now been completed by colleagues.

In [Section 4.4](#), we gave an overview of several different memory layouts for objects that can grow. We have not yet implemented or benchmarked these approaches to understand their performance implications. Understanding the exact performance characteristics of each, in scenarios with varying evolution patterns, requires a sufficiently large benchmarking suite of MAY programs.

The design of the language still needs more exploration, so that we can better understand where users may run up against limitations of the type system. By developing programs using object evolution and type safe evolution we will further uncover interesting ways of exploiting evolution to write effective programs.

The MAY type system is still missing several features present in the JAVA or C# type systems that prevent it from feeling like a “real language”. Most notably are generics and parametric polymorphism. While such features likely integrate simple evolution, the behaviour of permission types in these systems likely requires more care.

There also remain usability issues around the handling of owned references within arrays, evolvable fields, and classes that have the ability to own themselves. Exploring and evaluating something similar to the for loop example in

Section 4.3 could help ameliorate the array issues. Currently, accessing evolvable fields is only possible within an evolver, yet this is quite restrictive. We believe we might be able to reason about evolvable fields outside of evolvers, perhaps by allowing evolvers to return additional values. Having this flexibility would let us decouple consuming the evolution permission of a field from the evolution of the class itself to supplement error handling.

6.2 Conclusion

In this thesis we have explored the design and implementation of our language *MAY*, featuring type-safe object evolution. Object evolution is a promising balance between type safety and the alias control for reclassification. Our language makes use of inheritance based evolution and evolvers to define the semantics of object evolution. Field overriding integrates naturally with object evolution to allow for composite structures to be updated in place. In order to ensure type safety of object evolution we have introduced a permission system based around affine types, that is flexible and features minimal overhead. We have demonstrated how the permission system can integrate with an object-oriented language featuring interfaces to support idiomatic patterns of object evolution.

Object evolution is not a new idea, but it is not one that has been deeply explored within programming language theory or implementation. We hope that even if there is some difficulty in adopting an affine permission system within general purpose OOP languages, that we can at least invigorate an interest into this feature as a useful part of a semantics of objects. Indeed, the ability for an object to change its behaviour at runtime is arguably one of the core properties of object orientation, at least if one takes *SMALLTALK* or *SELF* to be cornerstones of pure object orientation.

While the contents of this work has many theoretical elements, the focus has always been on implementing a realistic language within the scope of this project. Programming is (at least for now) a human activity and by implementing this compiler we contribute toward our understanding of programming as a human experience.

Bibliography

- Akhin, M., Belyaev, M., 2021. Kotlin language specification [WWW Document]. URL <https://kotlinlang.org/spec/> (accessed 10.14.25). (Cited on pages 4 and 32)
- Aldrich, J., Sunshine, J., Saini, D., Sparks, Z., 2009. Typestate-oriented programming, in: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA '09. Association for Computing Machinery, Orlando, Florida, USA, pp. 1015–1022.. <https://doi.org/10.1145/1639950.1640073> (Cited on pages 3 and 67)
- Alpern, B., Cocchi, A., Fink, S.J., Grove, D., Lieber, D., 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless, in: Northrop, L.M., Vlissides, J.M. (Eds.), Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001. ACM, pp. 108–124.. <https://doi.org/10.1145/504282.504291> (Cited on page 15)
- Bejleri, A., Aldrich, J., Bierhoff, K., 2006. Ego: Controlling the power of simplicity. Proc. Foundations of Object-Oriented Languages. (Cited on pages 4, 20 and 36)
- Bierhoff, K., Aldrich, J., 2008. PLURAL: checking protocol compliance under aliasing, in: Companion of the 30th International Conference on Software Engineering, ICSE Companion '08. Association for Computing Machinery, Leipzig, Germany, pp. 971–972.. <https://doi.org/10.1145/1370175.1370213> (Cited on page 67)
- Bierhoff, K., Aldrich, J., 2007. Modular typestate checking of aliased objects, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07. Association for Computing Machinery, Montreal, Quebec, Canada, pp. 301–320.. <https://doi.org/10.1145/1297027.1297050> (Cited on pages 4 and 67)

- Bierman, G., Abadi, M., Torgersen, M., 2014. Understanding typescript, in: European Conference on Object-Oriented Programming. pp. 257–281. (Cited on page 4)
- Bloch, J., 2017. Effective JAVA, 3rd ed. Pearson Education. (Cited on pages 26 and 40)
- Boyland, J., 2003. Checking interference with fractional permissions, in: International Static Analysis Symposium. pp. 55–72. (Cited on page 67)
- Bracha, G., 2015. The DART Programming Language, 1st ed. Addison-Wesley Professional. (Cited on page 4)
- Brooks, R.A., 1984. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware, in: Boyer, R.S., Schneider, E.S., Jr., G.L.S. (Eds.), Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984. ACM, pp. 256–262.. <https://doi.org/10.1145/800055.802042> (Cited on pages 61 and 62)
- Canning, P.S., Cook, W.R., Hill, W.L., Olthoff, W.G., 1989. Interfaces for Strongly-Typed Object-Oriented Programming, in: Bosworth, G. (Ed.), Conference on Object-Oriented Programming: Systems, Languages, And Applications, OOPSLA 1989, New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings. ACM, pp. 457–467.. <https://doi.org/10.1145/74877.74924> (Cited on page 13)
- Carbonneaux, Q., 2024. QBE 1.2 [WWW Document].. URL <https://c9x.me/compile/> (Cited on pages 4, 47 and 54)
- Chalin, P., James, P.R., 2007. Non-null references by default in java: alleviating the nullity annotation burden, in: Proceedings of the 21st European Conference on Object-Oriented Programming, Ecoop'07. Springer-Verlag, Berlin, Germany, pp. 227–247. (Cited on page 33)
- Chambers, C., Ungar, D.M., Lee, E., 1989. An Efficient Implementation of SELF - a Dynamically-Typed Object-Oriented Language Based on Prototypes, in: Bosworth, G. (Ed.), Conference on Object-Oriented Programming: Systems, Languages, And Applications, OOPSLA 1989, New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings. ACM, pp. 49–70.. <https://doi.org/10.1145/74877.74884> (Cited on pages 63 and 64)
- Ciaffaglione, A., Gianantonio, P.D., Honsell, F., Liquori, L., 2021. A prototype-based approach to object evolution. J. Object Technol. 20, 1–24.. <https://doi.org/10.5381/JOT.2021.20.2.A4> (Cited on page 65)
- Coblentz, M.J., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B.A., Sunshine, J., Aldrich, J., 2020. Obsidian: Typestate and Assets for

-
- Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.* 42, 1–82.. <https://doi.org/10.1145/3417516> (Cited on page 69)
- Cohen, T., Gil, J., 2009. Three approaches to object evolution, in: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09*. Association for Computing Machinery, Calgary, Alberta, Canada, pp. 57–66.. <https://doi.org/10.1145/1596655.1596665> (Cited on pages 1, 4, 21, 22, 23, 25, 26, 27, 29, 31, 32, 61 and 62)
- Cook, W.R., 2009. On understanding data abstraction, revisited, in: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09*. Association for Computing Machinery, Orlando, Florida, USA, pp. 557–572.. <https://doi.org/10.1145/1640089.1640133> (Cited on pages 12 and 13)
- Damiani, F., Drossopoulou, S., Giannini, P., 2003. Refined Effects for Unanticipated Object Re-classification: Fickle_s, in: Blundo, C., Laneve, C. (Eds.), *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003*, Bertinoro, Italy, October 13-15, 2003, *Proceedings, Lecture Notes in Computer Science*. Springer, pp. 97–110.. https://doi.org/10.1007/978-3-540-45208-9_9 (Cited on pages 4, 19 and 65)
- DeLine, R., Fähndrich, M., 2004a. Typestates for objects, in: *European Conference on Object-Oriented Programming*. pp. 465–490. (Cited on pages 4 and 66)
- DeLine, R., Fähndrich, M., 2004b. The Fugue protocol checker: Is your software baroque. (Cited on page 66)
- DeMichiel, L.G., Gabriel, R.P., 1987. The Common Lisp Object System: An Overview, in: Bézivin, J., Hullot, J.-M., Cointe, P., Lieberman, H. (Eds.), *Ecoop'87 European Conference on Object-Oriented Programming*, Paris, France, June 15-17, 1987, *Proceedings, Lecture Notes in Computer Science*. Springer, pp. 151–170.. https://doi.org/10.1007/3-540-47891-4_15 (Cited on page 17)
- Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P., 2002. More dynamic object reclassification: Fickle_{ll}. *ACM Trans. Program. Lang. Syst.* 24, 153–191.. <https://doi.org/10.1145/514952.514955> (Cited on pages 4, 19 and 65)
- Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P., 2001. Fickle : Dynamic Object Re-classification, in: Knudsen, J.L. (Ed.), *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, Budapest, Hungary, June 18-22, 2001, *Proceedings, Lecture Notes in Computer Science*. Springer, pp. 130–149.. https://doi.org/10.1007/3-540-45337-7_8 (Cited on pages 3, 19 and 65)

- Duarte, J., Ravara, A., 2021. Retrofitting Typestates into Rust, in: Proceedings of the 25th Brazilian Symposium on Programming Languages, SBLP '21. Association for Computing Machinery, Joinville, Brazil, pp. 83–91.. <https://doi.org/10.1145/3475061.3475082> (Cited on page 69)
- Dunfield, J., Krishnaswami, N., 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 1–38.. <https://doi.org/10.1145/3450952> (Cited on pages 10 and 12)
- Dunfield, J., Pfenning, F., 2004. Tridirectional typechecking, in: Jones, N.D., Leroy, X. (Eds.), Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. ACM, pp. 281–292.. <https://doi.org/10.1145/964001.964025> (Cited on page 11)
- Ecma International, 2023. ECMA-334: C# language specification [WWW Document].. URL https://ecma-international.org/wp-content/uploads/ECMA-334_7th_edition_december_2023.pdf (accessed 10.21.25). (Cited on page 32)
- Ferreira, F., Pientka, B., 2014. Bidirectional Elaboration of Dependently Typed Programs, in: Chitil, O., King, A., Danvy, O. (Eds.), Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014. ACM, pp. 161–174.. <https://doi.org/10.1145/2643135.2643153> (Cited on page 10)
- Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E., 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17.. <https://doi.org/10.1145/1348250.1348255> (Cited on page 67)
- Fähndrich, M., Leino, K.R.M., 2003b. Heap monotonic typestates, in: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO). (Cited on pages 4 and 66)
- Fähndrich, M., Leino, K.R.M., 2003a. Declaring and checking non-null types in an object-oriented language, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And Applications, OOPSLA '03. Association for Computing Machinery, Anaheim, California, USA, pp. 302–312.. <https://doi.org/10.1145/949305.949332> (Cited on page 33)
- Fähndrich, M., Xia, S., 2007. Establishing object invariants with delayed types, in: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (Eds.), Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And Applications, OOPSLA 2007, October

-
- 21-25, 2007, Montreal, Quebec, Canada. ACM, pp. 337–350.. <https://doi.org/10.1145/1297027.1297052> (Cited on page 70)
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA. (Cited on pages 2 and 55)
- Garcia, R., Tanter, É., Wolff, R., Aldrich, J., 2014. Foundations of Typestate-Oriented Programming. ACM Trans. Program. Lang. Syst. 36.. <https://doi.org/10.1145/2629609> (Cited on pages 3, 36, 68 and 69)
- Garcia, R., Wolff, R., Tanter, E., Aldrich, J., 2010. Featherweight typestate. (Cited on pages 3, 36, 37, 39 and 68)
- Girard, J.-Y., 1987. Linear logic. Theoretical computer science 50, 1–101. (Cited on page 20)
- Goldberg, A., Robson, D., 1983. Smalltalk-80: The Language and Its Implementation. Addison-Wesley. (Cited on pages 16 and 17)
- Grigore, R., 2017. JAVA generics are turing complete, in: Castagna, G., Gordon, A.D. (Eds.), Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. ACM, pp. 73–85.. <https://doi.org/10.1145/3009837.3009871> (Cited on page 9)
- Haller, P., Odersky, M., 2010. Capabilities for Uniqueness and Borrowing, in: D'Hondt, T. (Ed.), ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, Lecture Notes in Computer Science. Springer, pp. 354–378.. https://doi.org/10.1007/978-3-642-14107-2_17 (Cited on page 39)
- Harms, D.E., Weide, B.W., 1991. Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Trans. Software Eng. 17, 424–435.. <https://doi.org/10.1109/32.90445> (Cited on page 39)
- Hölzle, U., Chambers, C., Ungar, D.M., 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, in: America, P. (Ed.), Ecoop'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings, Lecture Notes in Computer Science. Springer, pp. 21–38.. <https://doi.org/10.1007/BFB0057013> (Cited on page 64)
- Igarashi, A., Pierce, B.C., Wadler, P., 2001. Featherweight JAVA: a minimal core calculus for JAVA and GJ. ACM Trans. Program. Lang. Syst. 23, 396–450.. <https://doi.org/10.1145/503502.503505> (Cited on page 68)
- Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., Nakatani, T., 2000. A study of devirtualization techniques for a JAVA™ Just-In-Time compiler, in: Rosson,

-
- M.B., Lea, D. (Eds.), Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, October 15-19, 2000. ACM, pp. 294–310.. <https://doi.org/10.1145/353171.353191> (Cited on page 64)
- Kalibera, T., Jones, R.E., 2011. Handles revisited: optimising performance and memory costs in a real-time collector, in: Boehm, H.-J., Bacon, D.F. (Eds.), Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011. ACM, pp. 89–98.. <https://doi.org/10.1145/1993478.1993492> (Cited on page 61)
- Kehrt, M., Aldrich, J., 2008. A theory of linear objects, in: Proceedings of the International Workshop on Foundations of Object-Oriented Languages. FOOL. p. 116. (Cited on page 21)
- Kfoury, A.J., Tiuryn, J., Urzyczyn, P., 1993. The Undecidability of the Semi-unification Problem. *Inf. Comput.* 102, 83–101.. <https://doi.org/10.1006/INCO.1993.1003> (Cited on page 9)
- King, G., 2011. The Ceylon Language: 1.4.3 Compile-time safety for null values and flow-sensitive typing [WWW Document].. URL https://web.mit.edu/ceylon_v1.3.3/ceylon-1.3.3/doc/en/spec/html_single/#compiletimesafety (accessed 10.14.25). (Cited on page 33)
- Klabnik, S., Nichols, C., 2022. The RUST programming language. No Starch Press. (Cited on pages 4, 32 and 69)
- Lattner, C., Adve, V., 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, in: CGO. San Jose, CA, USA, pp. 75–88. (Cited on page 54)
- Liskov, B., Wing, J.M., 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 1811–1841.. <https://doi.org/10.1145/197320.197383> (Cited on page 9)
- Meyerson, J., 2014. The go programming language. *IEEE software* 31, 104. (Cited on page 32)
- Milner, R., 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 348–375.. [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4) (Cited on pages 8 and 10)
- Mirrlees-Black, A., 2025. The MAY Programming Language, Source and Executable [WWW Document].. <https://doi.org/10.5281/zenodo.17429520> (Cited on pages 7 and 47)
- Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K., 2012. A type system for borrowing permissions, in: Proceedings of the 39th Annual ACM SIGPLAN-

- SIGACT Symposium on Principles of Programming Languages, POPL '12. Association for Computing Machinery, Philadelphia, PA, USA, pp. 557–570.. <https://doi.org/10.1145/2103656.2103722> (Cited on page 68)
- Oliveira, P., 2024. Type-Safe Tree Transformations for Precisely-Typed Compilers. (Cited on page 2)
- Padlewski, P., 2017. Devirtualization in LLVM, in: Murphy, G.C. (Ed.), Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, And Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017. ACM, pp. 42–44.. <https://doi.org/10.1145/3135932.3135947> (Cited on page 64)
- Pierce, B.C., Turner, D.N., 1998. Local Type Inference, in: MacQueen, D.B., Cardelli, L. (Eds.), POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998. ACM, pp. 252–265.. <https://doi.org/10.1145/268946.268967> (Cited on page 10)
- Qi, X., Myers, A.C., 2009. Masked types for sound object initialization, in: Shao, Z., Pierce, B.C. (Eds.), Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. ACM, pp. 53–65.. <https://doi.org/10.1145/1480881.1480890> (Cited on pages 14 and 70)
- Ramalingam, G., Srinivasan, H., 1999. Object model for Java [WWW Document].. URL <https://patentimages.storage.googleapis.com/87/99/79/eb74436f5be05e/US5907707.pdf> (accessed 10.20.25). (Cited on pages 15 and 41)
- Reynaud, A., Scherer, G., Yallop, J., 2021. A practical mode system for recursive definitions. Proc. ACM Program. Lang. 5.. <https://doi.org/10.1145/3434326> (Cited on page 70)
- Régis-Gianas, F.P.Y., 2016. Menhir Reference Manual [WWW Document].. URL <http://moscova.inria.fr/~fpottier/menhir/manual.pdf> (accessed 10.14.25). (Cited on page 53)
- Siek, J.G., Taha, W., 2006. Gradual Typing for Functional Languages, in: Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06. pp. 81–92. (Cited on pages 4 and 69)
- Sieve, R., Kamburjan, E., Damiani, F., Johnsen, E.B., 2025. Declarative Dynamic Object Reclassification, in: Aldrich, J., Silva, A. (Eds.), 39th European Conference on Object-Oriented Programming (ECOOP 2025), Leibniz International Proceedings in Informatics (Lipics). Schloss Dagstuhl – Leibniz-

-
- Zentrum für Informatik, Dagstuhl, Germany, pp. 1–31.. <https://doi.org/10.4230/LIPICs.ECOOP.2025.29> (Cited on page 65)
- Strom, R.E., Yemini, S., 1986. Tpestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 157–171.. <https://doi.org/10.1109/TSE.1986.6312929> (Cited on page 66)
- Summers, A.J., Müller, P., 2011. Freedom before commitment: a lightweight type system for object initialisation, in: Lopes, C.V., Fisher, K. (Eds.), *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And Applications, OOPSLA 2011, Part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011.* ACM, pp. 1013–1032.. <https://doi.org/10.1145/2048066.2048142> (Cited on page 70)
- Tobin-Hochstadt, S., Felleisen, M., 2006. Interlanguage migration: from scripts to programs, in: Tarr, P.L., Cook, W.R. (Eds.), *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA.* ACM, pp. 964–974.. <https://doi.org/10.1145/1176617.1176755> (Cited on pages 4 and 69)
- Ungar, D.M., Smith, R.B., 1987. SELF: The Power of Simplicity, in: Meyrowitz, N.K. (Ed.), *Conference on Object-Oriented Programming Systems, Languages, And Applications, OOPSLA 1987, Orlando, Florida, USA, October 4-8, 1987, Proceedings.* ACM, pp. 227–242.. <https://doi.org/10.1145/38765.38828> (Cited on page 16)
- Van Rossum, G., Drake, F.L., 2009. *PYTHON 3 Reference Manual.* CreateSpace, Scotts Valley, CA. (Cited on page 17)
- Wadler, P., 1990. Linear Types can Change the World!, in: Broy, M., Jones, C.B. (Eds.), *Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990.* North-Holland, p. 561. (Cited on page 20)
- Winskel, G., 1993. *The formal semantics of programming languages - an introduction, Foundation of computing series.* MIT Press. (Cited on page 55)
- Wolff, R., Garcia, R., Tanter, É., Aldrich, J., 2011. Gradual tpestate, in: *Proceedings of the 25th European Conference on Object-Oriented Programming, Ecoop'11.* Springer-Verlag, Lancaster, UK, pp. 459–483. (Cited on page 69)
- Yang, X., Blackburn, S.M., Frampton, D., Hosking, A.L., 2012. Barriers reconsidered, friendlier still!, in: Vechev, M.T., McKinley, K.S. (Eds.), *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16,*

2012. ACM, pp. 37–48.. <https://doi.org/10.1145/2258996.2259004> (Cited on page 62)

