Classless Object Semantics

by

Timothy Jones

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the requirements for the degree of
Doctor of Philosophy

Victoria University of Wellington
2017

Abstract

Objects have been categorised into classes that declare and implement their behaviour ever since the paradigm of object-orientation in programming languages was first conceived. Classes have an integral role in the design and theory of object-oriented languages, and often appear alongside objects as a foundational concept of the paradigm in many theoretical models.

A number of object-oriented languages have attempted to remove classes as a core component of the language design and rebuild their functionality purely in terms of objects, to varying success. Much of the formal theory of objects that eschews classes as a fundamental construct has difficulty encoding the variety of behaviours possible in programs from class-based languages.

This dissertation investigates the foundational nature of the class in the object-oriented paradigm from the perspective of an 'objects-first', classless language. Using the design of theoretical models and practical implementations of these designs as extensions of the Grace programming language, we demonstrate how objects can be used to emulate the functionality of classes, and the necessary trade-offs of this approach.

We present Graceless, our theory of objects without classes, and use this language to explore what class functionality is difficult to encode using only objects. We consider the role of classes in the types and static analysis of object-oriented languages, and present both a practical design of brand objects and a corresponding extension of our theory that simulates the discipline of nominal typing. We also modify our theory to investigate the semantics of many different kinds of implementation reuse in the form of inheritance between both objects and classes, and compare the consequences of these different approaches.

The history of all hitherto existing society is the history of class struggles.

— Karl Marx & Friedrich Engels, *The Communist Manifesto*

Acknowledgements

This thesis has been a journey and a challenge, and it would not have been possible without the help and support of many people.

- James Noble, somehow both pragmatic adviser and mystic guide through the world of programming languages. I will always maintain that James tricked me into writing a cohesive thesis, and I will always be grateful that he did.

- David Pearce, whose door was always open to me, even if it was actually just ajar to let the undergraduates know that he was in. Drawing on David's whiteboard has solidified my knowledge on a wide range of topics, and his help was invaluable in completing this thesis.

- Kim Bruce and Andrew Black, who have helped me truly understand the concept of design by committee. This thesis was made possible by their fundamental disagreement on the nature of the class. I am also grateful for their feedback on the papers that eventually made up the contents of this thesis, and to Kim and his family for very kindly hosting me in their home.

- Jonathan Aldrich, Tony Hosking, and Alex Potanin, the examiners of this thesis, who provided me with insightful feedback and a lively and fascinating discussion in its defence.

- Paley Li, Roma Klapaukh, Julian Mackay, and Alex da Silva, who were my office-mates and colleagues, and remain my good friends. I hope that we each have the opportunity to pretend to understand what the other is talking about over coffee again in the future.

- My hosts and now friends at Imperial College, Cambridge, and St. Andrews, as well as the many student volunteers that I had the pleasure to work with along the way.

- My parents, eternally supportive, I hope that now I can finally return in the summer and not still have work to do. I am also grateful for the support and encouragement of my grandparents, though perhaps now I can argue that I am overqualified to fix their computers. To my siblings, well done with your own successes and thank you for your help with mine. May you never need new cars again.

- Michael Homer, whose contributions to this thesis almost rival my own. Michael has been my colleague, co-author, and friend throughout, and his insightful observations on the consequences of many language design decisions have directly influenced the content of this thesis.

- Eleanor Beeden, whose support and care has seen me through this entire process. Elle, you have helped me complete this thesis, but more importantly, my time with you has made me a better person. Thank you.

Contents

List of Figures

1 Introduction

Classes have been a fundamental component of object-oriented programming ever since the paradigm's inception in the Simula-67 programming language (Birtwistle et al. 1979). A class defines a common implementation for objects, a mechanism for creating those objects, an efficient mechanism for code reuse, and a type inhabited by objects that it or its inheritors have created. The prevalence of the object-oriented paradigm has seen to it that classes have become one of the primary contexts of programming: practically all code written in the most used language in the world must appear inside class definitions (Arnold, Gosling, and Holmes 2000).

In the literature on the theory of objects, classes have had a less fundamental role. Much of the work formalising object behaviour in the 1990s interpreted objects as record values of behaviours, mapping labels to methods (Abadi and Cardelli 1996; K. B. Bruce 2002). Objects were standalone constructions, and did not require a common definition of their implementation to fulfil the core object-oriented functionality of encapsulated state and behaviour, with message sends for interaction. Object types reflected the messages that objects were capable of accepting, rather than the class that they originated from.

As the theory has swung toward modelling existing object-oriented languages, more formalisms have included class tables and define objects and types purely in terms of these classes (Igarashi, Pierce, and Wadler 2001). Languages have since begun to pivot away from an 'all-in' approach to classes, with structural typing having seen a surge of popularity, often in combination with gradual typing as the static parallel to the duck typing of dynamically-typed languages. The rise of JavaScript in particular has propelled to the fore the idea that classes need not be a fundamental concept in the object-oriented paradigm (ECMAScript Project 2016).

A classless object language is not a new idea — Self is almost 30 years old, after

all (Ungar and Smith 1991). But programming in Self is fundamentally different to modern object-oriented development, and developers have come to expect the features made available by classes, even in JavaScript. This dissertation is an investigation into whether classes are truly redundant as a core feature of a programming language, or whether they have some fundamental importance in the implementation, creation, and typing of objects.

The context of this dissertation is the Grace programming language (Black, K. B. Bruce, and Noble 2016). Grace is a relatively new object-oriented programming language intended for use in education. One of its goals is to minimise the number of built-in language concepts, and to that end the story for classes has always been that they are built out of other features instead of being innate to the language. A class is just a factory which builds objects with the same implementation, and typing is structural rather than nominal (Black, K. B. Bruce, Homer, and Noble 2012).

Much of the research in this dissertation stemmed from the practical implications of attempting to take such a puritan stance towards classes when it came to actually implementing programs. The stories around object construction, code reuse, and typing were not consistent with the mainstream expectations of classes, which posed a problem for Grace both as a teaching language (under the expectation that students will be able to convert their knowledge to other object-oriented languages) and when porting existing libraries. The story around inheritance was particularly inconsistent with standard assumptions of object behaviour.

1.1 Contributions

The contributions of our thesis are:

- A formal model of a classless object-oriented language, *Graceless*, a language that encodes many of the features of Grace. Graceless intentionally models some of the more difficult aspects of Grace, including the object initialisation order and method requests unqualified by a receiver. Graceless forms the context of the remaining contributions.
- The design and implementation of *brand objects* to provide both the dynamic

and static components of nominal typing. The dynamic implementation is able to discriminate on the run-time identity of a brand object, and the static type system is implemented as a Grace dialect, so that brands can be introduced entirely as a library instead of extending the core language. Graceless is extended to formally model this design.

- An investigation into inheritance *between objects* rather than classes. This investigation modifies Graceless to produce formal models of both traditional object inheritance techniques and original attempts to simulate class inheritance, and consider a number of different systems for implementing multiple inheritance as well. The trade-offs in each design are presented and compared, and the solutions of other classless languages are also considered.

The result of our investigations is a family of formal languages that can help us to better understand the role that classes have to play in object-oriented programs, systems, languages, and applications.

We have also implemented most of our formal models using PLT Redex (Felleisen, Findler, and Flatt 2009). Encoding the models into a mechanical form helps to clarify any assumptions made in the text, permits counterexample checking, and makes it easy and automatic to examine the exact semantics of each language.

1.2 Structure

This dissertation is structured into four parts.

- Part I introduces the background of classless object-oriented languages. Chapter 2 discusses the relevant concepts from related work. Chapter 3 describes the Grace language specifically.

- Part II examines the role of types in a classless language. Alongside their role as factories, classes provide nominal types that discriminate which class constructed an object. We investigate alternatives to nominal typing as well as techniques to reproduce nominal typing within Grace's extensible type system. Chapter 4 defines Graceless, a structurally typed classless language.

Chapter 5 adds the ability to express assumptions about the type of Graceless objects using casts. Chapter 6 presents our design and implementation of brand objects for nominal typing.

- Part III considers the meaning of inheritance without classes. By examining a number of different formal models of inheritance we aim to provide consistent definitions for existing object inheritance techniques, and an analysis of the trade-offs inherent to each design. Chapter 7 introduces the relevant concerns for each of the models. Chapter 8 formalises existing techniques for reuse between objects. Chapter 9 defines models for object inheritance that emulate the behaviour of classes. Chapter 10 extends the models with multiple inheritance. Chapter 11 compares and contrasts the outcomes of the models.

- Part IV summarises our thesis. Chapter 12 uses our contributions to draw conclusions on the role of classes as a foundational concept of the object-oriented paradigm. Chapter 13 considers potential future work.

1.3 Publications

Many of the following chapters present existing research contributions of the following papers, written jointly with other authors:

- Jones and Noble (2014), *Tinygrace: A simple, safe, and structurally typed language*. This work is mostly superseded by the content presented in Chapter 4.

- Jones, Homer, and Noble (2015), *Brand Objects for Nominal Typing*. The design and implementation in this work remains mostly the same as the one presented in Chapter 6, but the formal model has been updated.

- Jones, Homer, Noble, and K. Bruce (2016), *Object Inheritance Without Classes*. Much of this work makes up the body of Part III, though the models have been updated to build on the language defined in Chapter 4.

We have also worked on the related publication Homer, Jones, et al. (2014), *Graceful Dialects*; this work predominately appears in the background of the dissertation, and does not form a core contribution.

Part I

Classless Languages

2 Related Work

In this chapter we introduce the research background of our thesis, laying out the relevant related work in programming language theory. The chapter begins by introducing the concept of 'objects-first' classless languages, then discusses formal models of programming languages, relevant typing disciplines, and extensible languages.

2.1 Objects First

Class-based object-oriented languages have a simple story about the relationships between objects and the classes that create them: an object is an instance of a class (Birtwistle et al. 1979). A specialised, 'one-off' object is just an instance of a specialised, one-off, anonymous class. Inheritance is between classes, and new objects are constructed and initialised by their classes.

This simple story comes at the expense of a more complicated story about classes, especially so if classes are themselves objects. More than thirty years ago, Borning (1986) identified eight separate roles that classes can play in most class-based object-oriented languages, each of these roles adding to the complexity of the whole language. This complexity leads inexorably to various kinds of infinite regress in meta-object systems (Goldberg and Robson 1983; Kiczales, des Rivières, and Bobrow 1991; Shaughnessy 2013).

To address this problem, prototype-based object-oriented languages, beginning with Lieberman's work inspired by LOGO (Lieberman 1986) and popularised by Self (Ungar and Smith 1991), adopted a conceptually simpler model in which objects were the primary concept, defined individually, without any classes. Shar-

ing of state and behaviour was handled by *delegation* between objects, rather than inheritance between their defining classes. Special-purpose objects could be defined directly, while new objects could be created in programs by cloning existing objects. Emerald went one step further and aimed to eschew all implicit sharing mechanisms, supporting neither inheritance nor delegation (Black, Hutchinson, et al. 2007), though prefixing an object with the body of another statically resolvable definition was permitted.

2.1.1 Object Inheritance

Under delegation, if an object does not directly know how to respond to a method request, it can hand the request off to another object to try instead. Self implements delegation using special parent* slots, which an object can set to specify which other objects should be used to handle a message the object does not understand directly. By sharing a single implementation of methods in a prototype object, and having many objects delegate to that object, delegation can act as a reuse mechanism in a very similar fashion to traditional class inheritance.

JavaScript's development as a classless language is directly influenced by Self, using prototypes to delegate between objects (ECMAScript Project 2016). Individual objects can be constructed using the literal syntax {x : y}, or the implementation of objects can be structured around *constructor* functions, with methods attached to the function's prototype object. Invoking a function with the new prefix constructs a new object that delegates to the prototype.

```
function Point(x, y) { ··· }
Point.prototype.distanceFromOrigin = function () {
    return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
};
```

Following the example above, the expression new Point(a, b) constructs a new object that delegates to Point.prototype. A call to the distanceFromOrigin method on this new object is not understood directly by the object, since it is a fresh object with no direct implementation, but the object can delegate to the implementation in Point.prototype to successfully respond to the method call.

Constructor functions can 'inherit' from one another by having the child constructor assign the result of calling the parent to its prototype.

```
function GraphicPoint(x, y) { ··· }
GraphicPoint.prototype = new Point();
GraphicPoint.prototype.draw = function () { ··· };
```

A new GraphicPoint(a, b) still responds to distanceFromOrigin, because the language will search transitively up through the chain of delegation objects.

Note that a and b are not available when constructing the intermediate prototype object, so new Point() is not given any arguments. Given that the constructor is likely to have side-effects, it is better to construct an object that will delegate to Point.prototype without actually calling the Point constructor. Modern JavaScript provides the Object.create method for this purpose: the method returns a new object that will delegate to its first argument when called, so the use of new Point() can be replaced with Object.create(Point.prototype).

JavaScript relies heavily on its ability to mutate the structure of an object after it has been constructed to build the implementation of objects imperatively. Because the structure of an object is often not declared before it is constructed, the implementation of methods are typically not present in objects at the point of creation, and are added in afterwards by assigning functions to fields. This is at odds with the declarative nature of classes in languages such as Java (Arnold, Gosling, and Holmes 2000) or C# (C# Project 2015), where the implementation of an object is entirely contained within a statically available class and cannot be changed at run-time.

The E programming language is also a classless object-oriented language (Miller 2006), but the entirety of an object's implementation is described at the object's point of creation. Constructors are just functions that return a new object.

```
def point(x, y) {
    def self { to distanceFromOrigin() { ··· } }
    return self
}
```

Defining an object in E must assign a name, and the definition is not an expression, hence the separate return statement in the example above.

E supports a form of delegation by defining an object with an extends clause, which takes an object to delegate to.

```
def graphicPoint(x, y) {
    def self extends point(x, y) { to draw() { ··· } }
    return self
}
```

E does not require contortions to avoid actually calling the constructor when constructing the parent object like JavaScript did, since the extends clause appears inside of the constructor directly and so has access to all of the relevant arguments. Before the child object is constructed, a parent object is created by calling the point function; when the parent is returned the newly constructed child object will delegate to it if it receives a request for distanceFromOrigin.

2.1.2 Self Binding

In an object-oriented language with classes, the value of a self or this variable in a class is like a hidden parameter on each of the methods. Since the class describes the implementation of many objects, the binding of the variable changes depending on which object is the receiver of any given method call. In a classless object-oriented language the meaning of the variable can be much simpler because a constructor only describes one object at a time, so the value of the self variable is just the value of the object. The E language does not even have a special variable for self references, since every object must explicitly be given a name when it is created.

The story for the self variable in Self and JavaScript is more complicated than just binding the surrounding object. In both languages it is possible to invoke a method with the value of self bound to an object that is not the one holding the method. The delegation mechanism in both languages passes the original receiver of the method call as an implicit parameter to method it delegates to.

In the Point JavaScript example earlier the binding of this is crucial to achieving the expected behaviour. The distanceFromOrigin method refers to this.x and this.y, but these fields do not exist on the object that defines the method, Point.prototype. Only by binding this to be the original receiver of a call to distanceFromOrigin,

an object constructed by the Point constructor, will these fields be present in the method.

JavaScript permits further manipulation of the value of this beyond the implicit binding achieved by delegation. All functions have methods call and apply that both invoke the function with this bound to their first argument. This functionality is often used to simulate super-constructor calls from within constructors:

```
function GraphicPoint(x, y) { Point.call(this, x, y); }
```

The Point constructor assigns x and y to the relevant fields in this, so it must be invoked with the same value of this as in the current context to ensure that the fields appear in the object constructed by the call to GraphicPoint.

2.1.3 Emulating Classes

Programmers using object-based languages have found the need to reintroduce classes — several times over in many of these languages. The Emerald compiler, and later the Self IDE, added explicit support for class-style inheritance. Languages with object inheritance such as Lua, JavaScript, and Tcl have a variety of libraries implementing classes in terms of objects. Most recently classes have been added explicitly to the recent JavaScript standard (ECMAScript Project 2016), to bring some order to the profusion of libraries already offering classes.

The new classes in JavaScript are only syntactic sugar for constructs that already exist in the language. Consider the following JavaScript class:

```
class Point {
    constructor(x, y) { ··· }
    distanceFromOrigin() {
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
    }
}
```

The outcome of running this program is not much different from our earlier Point example, accounting for some JavaScript oddities that the class form tidies up. Translating GraphicPoint into a class that extends Point and calls super in its constructor is also just sugar, and the use of super translates exactly into a use of call on Point.

2.2 Programming Language Formalisms

A formal model for a programming language has a number of benefits. A language formalism subsumes the task of defining a language specification while also providing formal constructs to reason about the design of the language. This allows the statement and proof of properties about the language. For example, the Standard ML programming language is fully formalised in Milner et al. (1997). The formalisation subsequently guides the implementation of the language while also providing correctness proofs for language features. The correct implementation of a formalism for a complete type checking algorithm ensures that a language cannot cause a type error when executing a program (K. B. Bruce 2002; Pierce 2002).

Languages that are defined first by a formal model before being implemented tend to be small and intended primarily for research. Many formalisms instead model specific features of a language or paradigm. Formal models of features can then be applied to implementations, often discovering edge cases that effect the correctness of the overall language. The formalisation of Cook and Palsberg (1989) for object inheritance with denotational semantics subsequently revealed a number of flaws in the type conformance algorithm of Eiffel (Cook 1989).

The formalisation of general programming concepts are also useful for building more specific models. The ς and $\mathbf{Ob}_{<:}$ calculi of Abadi and Cardelli (1996), both formal models of objects, are the basis for formalisms of a number of more specific features.

Few widely used languages are completely formally specified because of the difficulty in encoding the detailed nuances of large programming languages, which in many cases are specified mostly by their implementation (Owens 2008). Maintaining a language in strict adherence to a formalism will often stymie extensions to the language, as it requires first extending and proving the correctness of the model (Strub et al. 2012).

A number of language formalisms instead approach a more reasonable subset of a language in order to avoid these complications (Igarashi, Pierce, and Wadler 2001; Owens 2008). Smaller formalisms are particularly useful for formally describing extensions to a language, as it means that the correctness of the extension can be determined just for the relevant portions of the language.

Within the object-oriented paradigm, perhaps the most influential formalism derived from an actual programming language has been Featherweight Java (FJ), a formalised subset of the Java programming language (Igarashi, Pierce, and Wadler 2001). The subset removes Java features that complicate the ability to reason about objects and their types, such as variable assignment and side-effects on the wider machine. More complicated concepts like generics are also removed, and can then be added on top of the core calculus in isolation. Featherweight Java programs describe a series of invocations of class constructors, and can be executed by reducing the program to a collection of objects.

The result is a significantly simpler (though less practical) object-oriented language that submits to formalisation more easily, and is more readily extended. The extensibility of the formalism is immediately demonstrated by reintroducing Java's generic types, producing Featherweight Generic Java (FGJ). Featherweight Java has since been used to formally define Java extensions such as multi-methods (Bettini, Capecchi, and Venneri 2007), traits (Liquori and Spiwack 2008), and features (Apel, Kästner, and Lengauer 2008), as well as formalise more of Java proper such as generic wildcards (Torgersen, Ernst, and Hansen 2005; Cameron, Drossopoulou, and Ernst 2008) or mutable and immutable state (Mackay et al. 2012).

The approach of using a core calculus that abstracts away complications means that any properties proven for the calculus do not necessarily translate into properties on the larger language. Despite the soundness result for FJ, FGJ, and its many extensions, the Java type system itself is not sound, recently discovered in Amin and Tate (2016). As pointed out by Summers (2009), language features often combine in unexpected ways, and using a simplified calculus can hide the appearance of unsafe behaviour in the larger language. The unsoundness of Java is the outcome of an interaction between wildcards and null pointers: modelling null pointers is not a particularly difficult challenge, but it was still elided from FJ and from other calculi used to reason about wildcards such as Tame FJ (Cameron, Drossopoulou, and Ernst 2008).

2.2.1 Verifying Languages

Machine-checked verification is an important part of developing formal models, as it offers the potential of being a quicker or less fallible method of ensuring a proof of correctness than checking proofs by hand, though the overhead of translating proofs to their mechanised form can often void these benefits. Tools like Coq (The Coq Development Team 2016) make use of dependent typing, a more expressive form of typing that allows values to influence the type other values hold, and totality, a guarantee that a function will terminate with a value explicitly in its codomain, among other techniques. These properties allow for the expression of complex assertions that can be formally verified to hold. Other languages with these properties, such as Agda (Norell 2007), can use these strong reasoning properties to verify their programs.

F* is a dependently typed dialect of ML for distributed programming which contains proof checking of a similar form to Coq and Agda (Swamy et al. 2011). Strub et al. (2012) use the language as a basis to introduce the concept of self-certification, wherein the language is able to verify its own correctness. Producing a formal model of a language is beneficial, but it makes it difficult to continue to extend the language as the new formalism must be translated into a mechanically verifiable format and run through a proof checker before the new implementation can be confirmed as correct. Self-certification attempts to mitigate this problem by automatically verifying the new implementation. F* verified its initial theorem proving abilities both in Coq *and itself*, and was thus emancipated from using Coq as a proof checker because each new version of the language could be verified by the previous one.

The PLT group have also developed the Racket language Redex for the semantic modelling of software (Felleisen, Findler, and Flatt 2009). Redex allows for the expression of software models, including constraints and invariants about its structure. While this does not produce a complete formal model and Redex does not definitively prove the correctness of the defined models, it allows for comprehensive testing of software semantics without the complications of dependent typing and mechanised proofs.

2.3 Type Systems

Types are a particularly common form of formal methods in programming languages (Pierce 2002), and are applicable across a variety of paradigms. Different languages use different type systems, with different definitions of how values conform to types and different algorithms for ensuring that a program correctly conforms throughout.

Modern object-oriented languages broadly adhere to two different typing disciplines: static and dynamic. Languages with a static type system require that a program is well-typed before it will compile (though, as pointed out in §2.2, this does not necessarily guarantee that the program behaves correctly (Amin and Tate 2016)). Languages with a dynamic type system perform no such check when compiling, and instead ensure type correctness at run-time (Pierce 2002).

The concept of *subtyping* is prevalent in object-oriented languages, and broadly indicates that one type (the sub-type) contains only objects in another (the super-type). That two types are in a subtyping relation is often denoted with $T_1 <: T_2$, where T_1 is the sub-type and T_2 is the super-type (Cardelli, Martini, et al. 1994; Abadi and Cardelli 1996).

Although there are many different typing disciplines for object-oriented languages, subtyping is almost always relevant because of the *interface segregation principle*, wherein a client should not be forced to depend on the entirety of an object's interface in order to use the desired subset of that object's functionality (Martin 1996). Subtyping lets us express the simplest type necessary so that it is inhabited by any object that fulfils the interface that we require.

The behaviour of subtyping in object-oriented languages is often accompanied by the *Liskov substitution principle*, which requires that anywhere an object of type T is expected, it is safe to provide an object whose type is any sub-type of T (Liskov 1987). The substitution principle forms the core of the object polymorphism that makes object-oriented programming so useful — including mechanisms of reuse such as class inheritance — because subtyping allows the client to describe any object with the relevant interface while ensuring that its exterior behaviour is as expected.

Programs in a language that satisfy the substitution principle can rely on ab-

stractions of an object's interface rather than its internal implementation. This applies equally to statically-typed and dynamically-typed languages, as although the interfaces are often not explicitly stated in dynamically-typed languages, they are implied by the use of an object (specifically, the method calls made on the object).

2.3.1 Nominal Typing

The most popular statically typed object-oriented programming languages are class-based and use nominal types (K. B. Bruce 2002). Along with their implementation, classes simultaneously declare types that are associated with the class. Objects are instances of a type if and only if they are instances of the associated class (either directly, or as an instance of a subclass). The name 'nominal' refers to the fact that a class can only implement a type if it is explicitly named in its inheritance chain (K. B. Bruce 2002). Java (Arnold, Gosling, and Holmes 2000), C# (C# Project 2015), and C++ (Stroustrup 2007) are prime examples of modern languages that use a static nominal type system.

Dynamically typed object-oriented languages adhere to the notion that static checking is often too restrictive, and they postpone type errors to run-time. Rather than associating types with classes, dynamic typing associates type by the structure of an object. When an object is the receiver of a method call, it produces a run-time type error if it has no relevant method. Objects are not given explicit types, and instead the language assumes that each object implements the required interface.

Deferring type checking to run-time by raising type errors is often referred to as *duck typing*, under the maxim that "if it walks like a duck, swims like a duck, and quacks like a duck, then it may as well be a duck". More concretely, it is best practice to *not* check whether an object is a duck, but only check that it has the relevant methods to make it a duck (Martelli 2000). Languages with duck typing are often accompanied by object reflection mechanisms that allow the program to query whether a method appears on an object without actually invoking it; it has been argued that reflecting on an object in this way violates the principles of object-orientation, and that methods should only be discoverable through invocation (Abadi and Cardelli 1996).

The Python (Python Project 2016), Ruby (Ruby Project 2012), and JavaScript

```
// Library code.
class TheirClass {
    public String getName() { return "John"; }
}
// Client code.
interface Named {
    String getName();
}
class MyClass {
    public printName(Named named) {
        System.out.println(named.getName());
    }
}
```

Figure 2.3.1: Nominal types in Java

(ECMAScript Project 2016) programming languages all use dynamic typing. All of these languages also include some run-time nominal information as well, as it is possible to query whether an object was constructed by a particular class (or constructor function in the case of JavaScript).

Under nominal typing, the structure of an object is only relevant in that the interface of a class dictates its shape. The primary issue with this approach is that two objects can have an identical structure, and yet have no shared type (excepting the trivial top type, usually from an 'Object' class) (Dubochet and Odersky 2009; Gil and Maman 2008). In standard nominal type systems, such as those found in Java and C#, it isn't possible to add type relationships between classes outside of their declarations. Client code cannot add to the types of library classes, and so are unable to add potentially essential type relationships between otherwise unrelated objects (Baumgartner and Russo 1997; Büchi and Weck 1998; Läufer, Baumgartner, and Russo 2000; Malayeri and Aldrich 2008).

Consider the Java example in Figure 2.3.1, which shows a class from an external library and the code of a developer using it. An instance of TheirClass can't be passed to printName, despite having the requisite method. There is no way to specify that TheirClass also satisfies Named, because it is defined outside of the developer's own code. While there are other solutions to this problem within the

```
// Library code.
class TheirClass {
    def name : String = "John"
}
// Client code.
class MyClass {
    def printName(named : { def name : String }) {
        println(named.name)
    }
}
```

Figure 2.3.2: Structural types in Scala

realm of nominal typing, we first consider how a different method of typing objects — structural typing — can avoid this kind of problem.

2.3.2 Structural Typing

Structural typing of objects approaches the type checking of object-oriented programs in a different manner to nominal typing. A structural type definition describes an interface for an object, and any object which implements that interface is implicitly an instance of the type (Cardelli, Donahue, et al. 1989; Cardelli 1988). This resembles duck typing, but values are annotated with the type they are expected to satisfy rather than having this type inferred by usage. Consider Figure 2.3.2, which reworks the example from Figure 2.3.1 into the Scala programming language, making use of the structural type system.

The Scala example emphasises that the name a type is given is unimportant, as we completely remove the Named nominal interface. Instead we directly insert the type definition for an object with a name method that returns a string: { def name : String }. The Scala compiler will allow an instance of TheirClass to be passed as the named argument in a call to printName, because the structure of the object matches the interface given.

Note that an object may have *more* methods than specified by the type, as it still satisfies the type's interface. Cardelli (1988) explains that structural subtyping has an advantage over nominal subtyping in that structural types have meaning inde-

pendent of the placement of their definition in a program, such as in the example above. Structural subtyping is closely linked with the form of object inheritance found in languages such as Java and Scala, as an instance of a class will always satisfy the types of its superclasses. The crucial difference between structural and nominal subtyping is that an explicit inheritance relationship between two objects is *not* required for a structural subtyping relationship to exist.

Structural subtyping supports both *width* and *depth* subtyping (Cardelli 1988). Width subtyping requires only that an object have *at least* the methods described in the structural type, and may be 'wider' than what the type describes by having more methods that are not described in the type: this corresponds to the interface segregation principle (Martin 1996), as it permits describing the minimum methods required.

Depth subtyping requires that the type annotations on method signatures in a structural sub-type only be *as specific* as the corresponding annotations in the super-type, with *covariant* return types (the types in the sub-type are subtypes of the types in the super-type) and *contravariant* parameter types (the types in the sub-type are supertypes of the types in the super-type).

Structural subtyping of objects appears in Modula-3 (Nelson 1991; Cardelli, Donahue, et al. 1989), as well as the OCaml (Leroy et al. 2016) and Go (Go Project 2016) programming languages. OCaml's object types are defined by the methods that appear on an object — any object with at least the methods in a type is an instance of that type — and Modula-3 includes fields in this definition as well. Go features primitive, method-less structures which are nominally typed, and allows the definition of methods on pointers to these nominal types to create object-like pointers. Structural subtyping in Go is achieved with 'interface' types: any pointer to a nominal type which has at least the methods in an interface type is automatically a subtype of the interface.

Structural subtyping is not without its own failings. The most notable issue is 'accidental conformance', where an object is an instance of a type it was not intended to satisfy (Läufer, Baumgartner, and Russo 2000). Consider a structural type representing a geometric line:

type Line { x_1, y_1, x_2, y_2 : Number }

Line is a reasonable type for a representation of a line. Line is also a reasonable type for any number of other shapes — the opposing points of a rectangle, for instance. With purely structural subtyping, there is no way to distinguish these types from one another. Malayeri and Aldrich (2007) point out that nominal types also encode design intent, explaining that without the ability to explicitly segregate types, it becomes harder to enforce certain constraints.

Structural subtyping is also not the only solution to the issues with nominal typing that we have considered. Wehr, Lämmel, and Thiemann (2007) introduce the ability to perform retroactive interface implementation in JavaGI, which allow the relationship between the Named and TheirClass types from Figure 2.3.1 to be made explicit. The relationship is no longer implicit and the types are all still nominal, preserving the developer's intent within class relationships. Retroactive implementation resembles Haskell's type-class feature (Hall et al. 1996), though over object interfaces rather than functional data types.

2.4 Gradual Typing

Static and dynamic type systems play to different strengths. Knowing types in advance allows the compiler to detect errors more easily before a program is run and optimise the resulting execution, and type annotations provide a convenient form of in-code documentation. It is pointed out by Abadi, Cardelli, et al. (1991) that it isn't always possible to determine the type of data at compile time, and we have already seen an instance where a static nominal type system has prevented a structurally correct program from running. Modern compiler techniques such as just-in-time compilation are capable of producing high-performance code without the presence of type information (Castanos et al. 2012; Jantz and Kulkarni 2013).

An alternative to either system is to instead use optional typing, which combines both techniques, introducing a new 'dynamic' or 'unknown' type whose values are dynamically checked. Optional typing for objects arose both from the desire to add stronger type guarantees to existing dynamic systems such as Smalltalk (Bracha and Griswold 1993; Graver and Johnson 1990) and to add more flexibility to existing statically typed languages (Henglein 1994; Baars and Swierstra 2002; Noort, Achten, and Plasmeijer 2010). Attempts to add dynamic typing to static

$$\boxed{T \sim T} \qquad \frac{}{T \sim T} \quad \frac{T_{11} \sim T_{21} \qquad T_{12} \sim T_{22}}{(T_{11} \to T_{12}) \sim (T_{21} \to T_{22})} \quad \frac{}{T \sim ?} \quad \frac{}{? \sim T}$$

Figure 2.4.1: Consistency relation of $\lambda^{?}_{\to}$

systems have resulted in dynamically typed variables explicitly being marked as dynamic (Abadi, Cardelli, et al. 1991; C# Project 2015), which is at odds with the dynamic style of no type annotations at all (Siek and Taha 2006). There have been a number of recent examples of optional typing on top of or as a competitor to JavaScript, including TypeScript (Bierman, Abadi, and Torgersen 2014) and Dart (Dart Project 2015).

Gradual typing is a form of optional typing that aims to ease the transition of a program from an untyped prototype to a larger typed application. The name refers to the concept of gradually introducing types into a dynamically- or partially-typed program. Gradual typing distinguishes itself from optional typing by performing run-time checks to ensure that assumptions that were unable to be tested at compile-time are upheld throughout the execution of a program. The extensible language Racket uses gradual typing for interoperability between standard code and the Typed Racket extension (Felleisen, Findler, Flatt, et al. 2015; Takikawa et al. 2012); interoperation is achieved by bridging the typed and untyped code with run-time contracts (Findler and Felleisen 2002; Strickland, Tobin-Hochstadt, et al. 2012; Strickland and Felleisen 2010).

2.4.1 Consistency

Formal models of gradual typing work by introducing the dynamic type ? into an existing static type system, replacing type equality with *type consistency*, and translating programs in the gradual calculus into a cast calculus to enforce unchecked assumptions at run-time (Siek and Taha 2006). Type consistency, written $\sim$, is a relation between two types which are equal in all of their *known* parts; unknown parts are always consistent with any other part. The consistency relation of the $\lambda^{?}_{\to}$ calculus of Siek and Taha (2006) is defined in Figure 2.4.1.

All types are consistent with the ? type, and types with the same top-level shape are consistent if their parts are also consistent. Type consistency is reflexive and symmetric, but — unlike equality — it is not transitive. Transitivity would cause consistency to degenerate into a total relation, where every type is consistent with every other type. For instance, Int $\sim$? and ? $\sim$ Bool, so transitivity would give us the nonsensical relation Int $\sim$ Bool.

Siek and Taha (2007) define $\mathbf{Ob}^{?}_{<:}$, a gradual type system for objects that combines $\lambda^{?}_{\rightarrow}$ and the object calculus $\mathbf{Ob}_{<:}$ of Abadi and Cardelli (1996). Transitivity is also an issue for subtyping as it is for type equality: the appearance of the dynamic type would allow all types to be subtypes of one another. $\mathbf{Ob}^{?}_{<:}$ introduces the concept of consistent-subtyping, written $\lesssim$.

As with consistency, consistent-subtyping resembles the standard subtyping relationship but is not transitive in the presence of the dynamic type. Siek and Taha (2007) define the consistent-subtyping judgement $T_1 \lesssim T_2$ as subtyping between T_1 and T_2 with the unknown parts of each type 'masked' from one another, but the judgement is equivalent to finding some other type T_3 such that $T_1 <: T_3$ and $T_3 \sim T_2$. The dynamic type appears at every point in the consistent-subtyping lattice, so it is both a top and bottom type ($T \lesssim ?$ and $? \lesssim T$), but the type is neutral to regular subtyping, so only reflexivity applies ($? <: ?$).

2.4.2 Casts

As well as statically checking programs, it is important that a gradual type system also ensure type correctness at run-time. The dynamic type allows any object to claim to be of any type, and it is up to the run-time system to enforce the type correctness of this claim. Run-time checking of higher-order type assertions can be achieved with a contract (Wadler and Findler 2009; Takikawa et al. 2012), which the gradual typing literature calls a *cast*.

A cast allows the expression of more specific claims about the invariants of data and behaviours than types usually provide, but are often difficult to determine at compile-time and are instead enforced at run-time (Meyer 1986; Findler and Felleisen 2002). A gradual typing system needs to add casts at the boundaries of the static and dynamic worlds, to ensure that objects cannot masquerade under a

```
// Will raise a type error if the argument is not a string.
method printString(string : String) {
    print(string)
}

// Dynamically typed.
var number := 5

// Raises a type error as the value crosses from dynamic to static typing.
printString(number)
```

Figure 2.4.2: Run-time error in a gradually-typed language

type that they are not an instance of.

Consider the gradually typed program given in Figure 2.4.2. The printString method has a precondition that its argument be a string. When the number variable is passed to the method, it crosses the boundary between dynamic typing (as it is declared without a type) and static typing (as the parameter is of type String), which means the run-time system must enforce the precondition. Note that this is also a good example of how the run-time semantics of gradual typing differs from duck typing: this program will run without error under duck typing, whereas gradual typing will cause a run-time error on the invocation of printString.

Assumptions about higher-order type properties are more difficult to maintain. If the method in Figure 2.4.2 were expecting a function that returned a String, it would be able to check if the given argument was a function, but not what it returned without actually calling it. The same higher-order assumptions exist on structural types, where the object can be inspected for the presence of the relevant methods, but not what types the methods accept and return. These assumptions must be deferred until the function or method is actually called, at which point the deeper types apply.

When a cast needs to continue enforcing higher-order assumptions, it wraps and replaces the value it is checking and chaperones the value to observe its behaviour (Strickland, Tobin-Hochstadt, et al. 2012). Whenever a relevant action is performed on the cast, such as a method call, the action is passed to the underlying value and the types are checked as it is performed. The types checked by a cast

may also include higher-order assumptions, in which case more chaperone casts are generated.

The consistent relations allow the typing judgement to make assumptions about types that include the dynamic type ?. In order to ensure that these assumptions are upheld during the reduction of a program, formal gradual languages do not define reduction directly on the gradual calculus, and instead convert gradual terms into a term in a *cast calculus* instead. Terms in the $\lambda^{?}_{\rightarrow}$ calculus are transformed into terms in the $\lambda^{\langle\tau\rangle}_{\rightarrow}$ calculus; terms in the $\mathbf{Ob}^{?}_{<:}$ calculus are transformed into terms in the $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ calculus.

An $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ cast $\langle T_2 \Leftarrow T_1 \rangle\, t$ wraps a term t, with a source type of T_1 and target type of T_2. The modern literature now uses the syntax $t : T_1 \Rightarrow T_2$ (Cimini and Siek 2017). To be well-typed, the body of the cast must satisfy the source type, and the source and target types must be consistent. This means that a single cast by itself is always valid: only when reduction attempts to merge two casts together can the assumptions of a cast be found invalid.

The process of *cast insertion* doubles as a type system for a gradual language and the procedure that converts from gradual to cast calculus terms. The cast insertion procedure of Siek and Taha (2007) is expressed as $\Gamma \vdash t \rightsquigarrow t' : T$, taking an $\mathbf{Ob}^{?}_{<:}$ term t and a typing environment Γ, and producing an $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ term t' and type T. For typing purposes, t' can be thrown away; for evaluation purposes, t can be thrown away. The procedure determines where assumptions are being made (effectively, where consistency is being used instead of equality), and inserts casts into the resulting cast calculus term to explicitly encode these assumptions.

In further work, inference for gradual types is explored by the $\lambda^{?\alpha}_{\rightarrow}$ calculus of Siek and Vachharajani (2008) and the inference algorithm of Rastogi, Chaudhuri, and Hosmer (2012), as well as the investigation of Garcia and Cimini (2015) into the potential meaning of omitting a type annotation in a gradually typed language, ultimately subsuming the earlier work on gradual inference. Recent developments have investigated generating gradually typed languages directly from existing statically-typed languages (Cimini and Siek 2016; Garcia, Clark, and Tanter 2016; Cimini and Siek 2017).

```
// Will raise a type error if the argument is not a list.
method printFirstString(strings : List⟦String⟧) {
    // Will raise a type error if the resulting element is not a string.
    // Blames the call to printFirstString.
    print(strings.first)
}

// Dynamically typed.
var intList := list(1, 2, 3)

// Does not raise a type error here, but inside the method instead.
printFirstString(intList)
```

Figure 2.4.3: Run-time error identified by blame

2.4.3 Blame

One of the interesting challenges of gradual typing is the production of relevant run-time type error information (Findler and Felleisen 2002). Wadler and Findler (2009) introduce a 'blame' calculus which annotates casts with blame labels, such as $t :^{\ell} T_1 \Rightarrow T_2$. When a cast fails, the blame label is used to produce the original source of the error. For casts between simple types this is an irrelevant addition, as if such a cast fails then it does so immediately and is easily blamed. Casts enforcing higher-order types such as functions or generic types cannot always be completed at the point where the cast is made, and so blame labels are used to ensure that if a cast fails in the future, the blame can be tracked back to the correct cast.

In the example presented in Figure 2.4.2, a run-time type error is produced at the same point as the typing fault. It is not always possible to enforce the typing precondition as an object crosses the typing boundary (Ina and Igarashi 2011). We extend the previous example so that the method operates on a list of strings rather than a single string in Figure 2.4.3, and supply it with a list of the wrong element type. As intList is passed to the printFirstString method, the run-time system can confirm that it is a list, but it cannot know what the elements of the list are without retrieving them. Rather than producing a type error at the invocation of printFirstString, it can only detect the fault at the call to strings.first.

In order to achieve correct blame placement of the resulting type error, the cast maintaining the higher-order assumptions of the type of strings must remember where it originated from. When the call to first reveals that the list does not always contains strings, the caller of printFirstString is blamed instead of the body of the method. A cast records the point where it crosses a typing boundary, and reports that particular cast as the assertion which has failed.

2.4.4 Gradual Guarantee

The gradual guarantee is one of four *correctness criteria* defined for gradually typed languages (Siek, Vitousek, Cimini, et al. 2015). The guarantee was developed following the observation that a typecase form in a structurally typed language would cause type annotations on methods to become relevant outside of enforcing typing assumptions at run-time (Boyland 2014).

Consider two structural types named Graphic and Gunslinger. Both types contain a signature named draw, but they each accept a parameter of a different type: Graphic accepts a Widget, while Gunslinger expects a Gun. The difference in their parameter types means that neither type is a subtype of the other: in a typecase that examines parameter types, a Graphic is distinguishable from a Gunslinger.

In a gradually typed language, an object that is intended to be a Graphic may omit its parameter type, indicating that it should be interpreted as the dynamic type ?. Such an object is no longer distinguishable as either a Graphic or Gunslinger: a typecase that discriminates on both types must pick one. Later, as the program is developed, the type may be filled in with one of the two valid parameter types, and this may cause a typecase to now pick a different path with this new information.

The gradual guarantee effectively states that the addition or loss of any precision in types alone to a gradually typed program cannot affect its behaviour. Since the addition of an incorrect type could cause a program to fail with a type error, the guarantee is formally expressed for removing precision: if a gradually typed program reduces to a value without error, then the term produced by making any types in that program less precise reduces to the same value. The typecase described above fails the guarantee because making the parameter type less precise could cause it to branch differently.

The guarantee is intended to uphold the 'gradual' component of gradual typing. For a program to be gradually developed from an untyped prototype to a larger typed application, the gradual addition of types should not affect the behaviour of a (correct) program.

2.5 Hybrid Type Systems

Nominal and structural typing both have advantages (Malayeri and Aldrich 2008; Gil and Maman 2008). Structural typing decouples an object's type — the set of methods to which it can respond — from the object's implementation (usually a class). Structural types can be declared at any time, in any part of the program, and still be relevant to any object with the appropriate interface. Any object that conforms to a structural type can be used wherever an instance of that structural type is required, even though the object's definition did not declare that it implemented the type — among the reasons that Go adopted structural typing (Go Project 2016).

Being based solely on objects' interfaces rather than their implementations, structural types correspond to the conceptual model of object-oriented programming where individual objects communicate only via their interfaces, with their implementations encapsulated (Cook 2009). The clear separation between structural types and their implementing classes, and the ease of defining types independently from classes works well with gradual and pluggable typing (Bracha 2004; Andreae et al. 2006), so programmers can begin by writing programs without types, and then add types later as the need increases.

Some modern languages have adopted structural subtyping. In Go, for example, types are declared as interfaces, and an object conforms to a type if the object declares at least the methods required by the interface (Go Project 2016). As well as Go's interfaces, Emerald is structurally typed, as is OCaml's object system and Trellis/OWL (Black, Hutchinson, et al. 2007; Leroy et al. 2016; Schaffert et al. 1986). Structural types have also been used to give types post-hoc to dynamically typed languages: Strongtalk originally supported structural types for Smalltalk, and Diamondback Ruby uses structural types for Ruby (Bracha and Griswold 1993; Furr et al. 2009).

On the other hand, nominal subtype relationships must be designed and de-

clared by programmers, meaning they can capture programmers' intentions explicitly. Nominal subtyping can make finer distinctions between objects than structural subtyping: a structural system cannot distinguish between two different classes that have the same external interface, whereas a nominal system can distinguish between every implementation of every interface. Because nominal types can distinguish between different implementations (classes), compilers and virtual machines can optimise object allocation and method execution for particular implementations — for example, allocating machine integers and compiling arithmetic without any method dispatch.

The reality is that most statically typed object-oriented languages use nominal subtyping. From Simula (Birtwistle et al. 1979) and C++, through to Java, C# and Dart, an instance of one type can only be considered an instance of another type if the subtyping relationship is declared in advance, generally at the time the subtype is declared. In many of these languages, interfaces can be used to describe the required structure of an object in the same way as structural types, but objects do not implicitly satisfy interfaces and must be constructed by a class that explicitly declares that it implements the interface.

As most languages are nominally typed, most of the major platforms for object-oriented languages (the Java Virtual Machine and the Common Language Runtime) are themselves nominally typed, so interoperability with VMs and languages is assisted by nominal typing. Pedagogically, nominal subtyping ensures every type has a name, so compilers and IDEs (especially their error messages) can refer to types by name, making teaching and debugging easier. Every nominal type has an explicit, unique, declaration in the program, a declaration that describes its relationships with all its supertypes, so class and type hierarchies can be understood in a straightforward manner. These advantages are among the reasons that Strongtalk, for example, moved from structural to nominal typing (Bracha and Griswold 1993).

Given that nominal and structural typing both have advantages, there have been attempts to combine them both in a single language. The Whiteoak language (Gil and Maman 2008) begins with Java's nominal type system and adds in support for structural types. Around the same time, Scala 2.6 (Odersky 2014) added structural types, again on top of a language with a nominal type system. The Unity language design similarly adds structural types onto a nominal class hierarchy (Malay-

```
struct Amount {
    int amount;
    int add(Amount other) {
        return this.amount + other.amount;
    }
}
```

Figure 2.5.1: Structural types in Whiteoak

eri and Aldrich 2008). All of these additions begin with a nominal-typed language, where interface and class are already conflated, and attempt to add structural types as refinements onto nominal types.

The dichotomy between structural and nominal subtyping has been studied from the earliest applications of types to object-oriented languages (Black and Palsberg 1994). Simula, the first object-oriented language, is nominally typed: a subclass must be explicitly declared as inheriting (being prefixed) by its super-class (Birtwistle et al. 1979). Most object-oriented languages (C++, Java, C#, etc) followed Simula's lead, though OCaml supports structural subtyping for objects, as does Go (Leroy et al. 2016; Go Project 2016).

Whiteoak introduces a number of structural typing features on top of the standard Java type system, allowing the definition of structural types for both methods and fields, and nominal types conform to structural types when they implement their interface. The new structural types can also provide default methods for objects explicitly declared to be of that type. In the Whiteoak example given in Figure 2.5.1, any object with an integer field amount is an instance of the Amount type, and any variable declared with the type Amount has a method add, regardless of whether the underlying object actually has one or not.

Dubochet and Odersky (2009) added structural subtyping to Scala on top of its existing nominal type system with a similar feature set. The Java platform has no support for structural subtyping, so both implementations have to translate the structural types into nominal types in order to generate valid Java bytecode. The easiest solution is to use whole-program-analysis and produce a new Java type for every intersection between a nominal and structural type.

Gil and Maman (2008) describe generating unique Java types for nominal and

structural types as 'executable blowup' and consider it an unacceptable solution. Whiteoak achieves compatibility with the Java 5 platform by building wrapper classes for each instance of upcasting from a nominal type to a structural one, whereas Scala makes use of the more recent 'invokedynamic' feature of the Java Virtual Machine (Lindholm et al. 2013) to avoid producing additional classes on each use.

The path-dependent types in DOT and System D (Rompf and Amin 2016; Amin 2016) achieve a level of pseudo-nominality, in that without equal bounds on a type member the type system cannot reason about the type exactly, and can only equate it to itself, just as with nominal types. In particular, the bounds $\bot..T$ behave exactly as a nominal type whose structural information is T, and two distinct members with these same bounds are not considered the same type, even if the members themselves are declared to be the same. The type members of DOT permit a simulation of nominality through abstraction, but if the language were extended with a run-time matching construct (i.e. instanceof) this feature could only match on structural types, since there are no true nominal types.

2.5.1 Brands

Most early theoretical analysis of type systems for object-oriented languages used structural types (Cook, Hill, and Canning 1990; Cardelli and Wegner 1985; K. B. Bruce 1994; Pierce and Turner 1994). Later references such as Palsberg and Schwartzbach (1994), K. B. Bruce (2002), and Pierce (2002) discuss structural and nominal (sub)typing, but they do not address the question of how both kinds of types can best be integrated into a single, practical, language design.

Malayeri and Aldrich (2008) introduce 'brands' in the Unity programming language, which combines nominal and structural subtyping. Brands are essentially nominal classes, but the type system is extended so that structural constraints can be added. For instance, a standard hierarchy for a windowing system may describe a Window brand as its top:

```
abstract brand Window (···)
```

As in a nominal class hierarchy, there will be a number of brands extending Window, such as ResizableWindow or MovableWindow. In a nominal setting, in order to get a window with a scroll bar a method must request an argument of type ScrollBarWindow,

and every class that has a scroll bar must implement this interface. With brands, a method can instead request any subtype of Window that has a scrollBar method:

```
method scroll(win : Window({scrollBar : ScrollBar})) : unit
```

The notion of nominal brands on structural types originated in Modula-3 (Nelson 1991). Record types in Modula-3 generally use structural equivalence, but can be annotated with a brand to give nominal equivalence. Modula-3 brands can also be given explicitly, e.g. for type safety between programs or across networks. Even with structural equivalence, Modula-3 record types do not support subtyping: there is no type relationship between a record type with a particular set of fields, and a second record type with a subset (or superset) of those fields — only between two record types whose field types are identical. Modula-3's object types are "essentially SIMULA classes" (Cardelli, Donahue, et al. 1989) and, like SIMULA, use nominal subtyping. Neither the Cardelli, Donahue, et al. (1989) formalisation of the Modula-3 type rules, nor the Baby Modula-3 of Abadi (1994), nor the *Theory of Objects* (Abadi and Cardelli 1996) model Modula-3's branded types.

The Tagging Language of Glew (1999) introduces 'tags' in the context of type dispatch. Tags can be used to implement class- and exception-casing in much the same way as brands. The underlying type system is not structural, and is populated by primitive sequence and function types instead. The language formalism goes into depth on the existence of tags at run-time, including populating the heap and run-time matching. Strongtalk (Bracha and Griswold 1993) is an optional (and arguably pluggable) type system for Smalltalk: in the original version of Strongtalk, the types were structural with optional brands, again very similar to our design, although a later version of Strongtalk abandoned brands and adopted declared subtyping and matching relationships (Bracha 1996).

A design for *trademarks* were proposed for ECMAScript 6 (Horwat and Miller 2011) that provides a very similar model of branding for the language. Trademarks are split between a branding object and a guard object, the former for tagging an object as branded, and the latter for identifying which objects are branded. The design shows how a program can hide the branding object while exposing the guard to prevent fraudulent branding: the branding object acts as a *capability* for the guard's precondition (Miller 2006). As a dynamically-typed language, combining

static reasoning about trademarks with a static structural type system in JavaScript would be useful.

2.5.2 Tagged Objects

Recent work includes the Tagged Objects theory of Lee et al. (2015), and the resulting practical extension to the Wyvern language (Nistor et al. 2013). Tagged Objects adds nominal 'tags' on top of an existing structural type system (along with other common functional constructs). This approach focuses on the type theory of tags, and provides new primitive type and matching constructs as an extension to the language, with new static typing rules. Unlike DOT, this mechanism *does* support true nominality with run-time matching, and the construction of a tag includes the relevant structural information, so any object that is declared with a tag must satisfy the structural requirement of that tag. In this sense, a tag encodes the same concept as interfaces in Java or C#.

Tags are constructed using a newtag$[\tau]$ function, with the type τ representing the type that any values tagged with the new tag object are required to satisfy. The result of applying newtag is a new tag value c of type τ tag. A value c is effectively a reference, and appears in a store S during reduction, but the actual objects in the language are not allocated on the store, so S exists purely for the purposes of recording the identity of a tag object (and its corresponding type τ).

Any tag at a name x can be applied to a term e with an application of new(x, e) so long as e can be typed with the corresponding τ of x. An application of new is a value once its argument e has been reduced to a value as well, with the type tagged x. This hides the value and type of e, up to the corresponding τ of x, since e must have at least the type τ. The extract function extracts the value from a tag application and discards the tag: extract(new(x, v)) reduces to v. The value must be extracted before it can be used as an object of type τ.

Existing tags can be extended with the subtag$[\tau](x)$ function, the result is also a fresh reference c, where c is associated with τ in the store, but this value has the type τ tag extends x. Values tagged with a sub-tag are acceptable where values with the super-tag are expected, so tagged x_1 <: tagged x_2 if x_1 has the type τ_1 tag extends x_2 and x_2 is also a tag. In order to ensure that extracting the values

```
let intOption : ⊤ tag = newtag[⊤] in
let none : ⊤ tag extends intOption = subtag[⊤](intOption) in
let some : int tag extends intOption = subtag[int](intOption) in
let x : tagged intOption = new(some; 5) in
match(x; some; y. extract(y) + 1; −1)
```

Figure 2.5.2: Optional integer type example modified from Lee et al. (2015)

is safe in the presence of this subtyping, the sub-tag's required type τ must also be a subtype of the super-tag's required type. Anything extracted from the sub-tag must be compatible with the type of things extracted from the super-tag.

Tags can be detected during execution with a match construct that branches on the presence of a given tag object; matching against a tag x, then in the body of the branch that is entered if the tag is present the object is bound to a variable with the type tagged x. The match construct makes it possible to encode both nominal class discrimination such as instanceof in Java, as well as pattern matching on algebraic data types. The authors present an example of encoding the latter to describe the type of an 'optional' int using tags none and some. This adds a null-like form to the type, but the cases *must* be discriminated before access to the underlying value (if present) is permitted.

A modified form of the example from Lee et al. (2015) is presented in Figure 2.5.2. First, a common super-tag of the two cases must be declared, then the none and some tags are constructed as sub-tags. The value 5 is tagged with some, which is well-typed because 5 is an int. Finally, the match construct discriminates on the brand some, binding the value of x to y in the body if x is tagged some (or potentially a sub-tag of some). The result of running this program is 6, since the match succeeds and so the value 5 is extracted and incremented.

The purpose of the Tagged Objects theory is to act as a foundation for object-oriented languages in the same vein as Featherweight Java, but without the (often implicit) class table that appears in FJ (Lee et al. 2015; Igarashi, Pierce, and Wadler 2001). To this end, classes can be represented as a dependent sum of a tag value and a constructor: the sum must be dependent because the type of the constructor depends on the *value* of the tag. The type of a class whose objects satisfy the

interface τ and whose constructor takes input τ' is the sum:

$$\sum_{x:\tau\ \mathsf{tag}} (\tau' \to \mathsf{tagged}\ x)$$

Any such class that constructs its objects with a function f can be represented by the term:

$$\mathsf{let}\ x = \mathsf{newtag}[\tau]\ \mathsf{in}\ \langle x, \lambda y.\ \mathsf{new}(x; f\ y)\rangle$$

Any use of the underlying object is an extract away.

Nominal subtyping relationships between these classes can be declared as extends relationships between their tags, so a class that extends another class x has the type:

$$\sum_{y:\tau\ \mathsf{tag}\ \mathsf{extends}\ \mathsf{fst}(x)} (\tau' \to \mathsf{tagged}\ y)$$

Such a class can be represented as before, but using the subtag function:

$$\mathsf{let}\ y = \mathsf{subtag}[\tau](\mathsf{fst}(x))\ \mathsf{in}\ \langle y, \lambda z.\ \mathsf{new}(y; f\ z)\rangle$$

The result is that classes can be passed around as first-class values, and the types of these class objects express the declared nominal subtyping relationships between them. Accessing the fst of any class provides the tag to match on objects constructed by that class, encoding Java's instanceof and a safe form of type casting.

The one caveat is that, in order to use a tag as a type with the tagged form, the tag itself must be in scope. This means that that tag is also available to be used in an application of new. The Tagged Objects language provides no mechanism for exposing the *type* of tagged objects without also exposing the mechanism for tagging other values. As such, while the behaviour of Java-like classes can be encoded in the theory of Tagged Objects, the language cannot actually enforce that a tagged object was constructed by a particular class. This is mitigated by the τ type associated with a tag, such that any tagged value is required to at least implement the interface associated with the class, which is effectively the same as inheriting from the class and then overriding all of the methods. This equivocation breaks down in the presence of more complicated features such as side effects in a constructor, though.

2.6 Pluggable Typing

Static type checking can catch a variety of potential errors at compile-time, but there are a number of common errors which can be addressed statically that many standard type systems do not address (Flanagan et al. 2002; Chalin et al. 2005; James and Chalin 2009). For instance, the most common object-oriented languages feature a null pointer that is not checked by their type system, which can often result in the infamous null pointer exception. Extended static checking (ESC) is a technique for addressing more specific forms of errors at compile time (Chalin et al. 2005; Xu 2006). Flanagan et al. (2002) introduce ESC/Java as an extension to the Java programming language. The extension allows for statically verified assertions in a program, including non-null variables and basic reasoning about number values.

A number of other, more specific extensions to Java's static checking system exist. Clarke, Potter, and Noble (1998) add ownership types to constrain the exposure of an object's representation, and Pearce (2011) adds purity checking on object methods to constrain method side-effects. The issue with these myriad extensions is that they are inherently incompatible. As they directly extend the language by modifying the compiler, a developer wishing to use combinations of the static checkers must manually compose them.

Bracha (2004) argues that the choice of a programming language should be independent from the choice of a type system in an introduction to pluggable type systems. He claims that mandatory typing can hinder the development and stability of software, whereas optional typing allows for the separation of the type checker from the language itself. The reintroduction of type checking to such a language can be thought of the application of a plugin to the compilation process, and there is no reason that multiple such plugins might not be introduced.

This concept is explored further by Andreae et al. (2006) with JavaCOP, an extension to the Java language that allows for the development of annotations and assertions about their application in a declarative format. These assertions are then checked on top of Java's standard type checking. They produce a static solution to the null pointer error with a NonNull annotation and an associated checker that guarantees annotated variables cannot hold the null value.

The restriction of JavaCOP's assertion language to a simple declarative format ensures that the framework cannot cause the type checking stage of the compiler to execute forever (Markstrum et al. 2010). Papi et al. (2008) do not adhere to this restriction and allow for either a declarative and procedural approach to confirming assertions. Java's own Annotation Processing Tool, though primarily intended for performing compile-time processing of Java source files, also allows warnings and errors to be communicated to the Java compiler.

Ultimately, the Java language was not designed with an extensible type system in mind. Attempts to add extended static checkers require extensions to the underlying compiler. A language which is inherently extensible would be a more suitable candidate for pluggable types.

2.7 Extensible Languages

An extensible language contains constructs that allow a developer to modify its syntax or semantics (Standish 1975). Extensibility is often associated with metaprogramming, as it allows programs to operate on themselves (Kiczales, des Rivières, and Bobrow 1991), but meta-level operations are not necessary for a language to be considered extensible. For instance, while many languages feature built-in control structures such as if-then-else and while-do, other languages define the structures in terms of other constructs, and allow for the modification or invention of them.

Lisp macros are a canonical example of extensibility. There is no special syntax for control structures in Lisp, as control flow is specified with the use of macros. Macros rewrite their call sites in such a way that their arguments are lazily evaluated: they are not evaluated unless they are needed. For instance, conditional branching can be achieved by defining a macro that ensures that a branch value will not be evaluated unless the condition is true, but when called the result is indistinguishable from an ordinary function call:

```
(if list
    (print "Non-empty list")
    (print "Empty list"))
```

The Smalltalk programming language achieves a similar effect without the use of

metaprogramming (Ingalls 1978; Goldberg and Robson 1983). The language achieves the same functionality using methods and blocks. Blocks provide a way of defining code that can be run later, acting as an explicit form of lazy evaluation. The if-then-else structure becomes an operation performed on a boolean object, taking two block arguments:

```
list isEmpty
    ifTrue: [ Transcript show: "Empty list" ]
    ifFalse: [ Transcript show: "Non-empty list" ]
```

This allows objects other than boolean to be used in an if-then-else construct, because they just have to implement an ifTrue:ifFalse: method. Different iterable objects can each implement forEach:, and so on.

The Racket programming language is a Lisp dialect and a descendant of Scheme developed by the PLT group with a focus on extensibility (Felleisen, Findler, Flatt, et al. 2015). The syntax and semantics of Racket can be modified within the confines of a single module, to the extent that a module can be implemented in an entirely different language. Each module declares itself as being written in a particular language with the #lang directive. This separation of languages can be used to form a 'syntactic tower', where one language can be used to define the implementation of another, which in turn can be used to produce further languages.

The base language of Racket closely resembles its ancestors, including its use of dynamic typing. The extensible features of the language subsume the basic features of pluggable typing, and have been used to create Typed Racket, a language with static typing (Felleisen, Findler, Flatt, et al. 2015). Typed Racket is still able to interact with other modules not in the typed language, introducing a form of gradual typing to Racket as well (Takikawa et al. 2012). The boundary of static and dynamic typing now exists at the boundary of these interacting modules, requiring run-time checks and chaperones (Strickland, Tobin-Hochstadt, et al. 2012). The development of tools like PLT Redex (Felleisen, Findler, and Flatt 2009) demonstrate that Racket is capable of addressing at least semantic modeling.

3 Grace

This chapter introduces the relevant features of the Grace programming language (Black, K. B. Bruce, Homer, and Noble 2012; Black, K. B. Bruce, and Noble 2016), which will be the context for the remainder of the thesis. We discuss the core of the language, including the nature of objects and the type system governing their use, as well as explaining the language's annotation and dialect features.

3.1 The Core Language

The Grace programming language is an object-oriented language intended for education (Black, K. B. Bruce, Homer, and Noble 2012; Black, K. B. Bruce, Homer, Noble, et al. 2013). Grace aims to supplant the use of enterprise languages in this field, such as Java, by reducing the boilerplate necessary to produce a working program while also introducing modern, demonstrably useful software engineering concepts. Grace features a number of techniques for gradually building programs so that the simplest programs are small while also allowing for the construction of reasonably large and complex applications around object-oriented principles.

The core primitive of Grace is the object. An object in Grace is essentially a set of methods which can be *requested* to invoke the contained code. Requesting a method is the same as sending a message in Smalltalk or calling a method in most object-oriented languages. We use the term 'requested' to make it clear that, in good object-oriented form, only the receiver of a method request is responsible for determining which method is actually invoked.

The language uses an expression to create objects with the object keyword, and methods are defined inside this literal with the method keyword. An object with a

single method, square, can be created with the following code:

```
object {
    method square(x) {
        return x * x
    }
}
```

Methods are identified by their name and arity (number of parameters), and no object may have more than one method with the same identifier. The variable self always refers to the closest surrounding object, and outer can be used to refer to surrounding objects whose self value is shadowed by a nearer object.

Objects can be stored in local variables that are either constant, with the def keyword, or variable, with var. Assignment to these variables uses = and := respectively, to distinguish between binding a constant value to a name and assigning a value in a mutable variable. Object fields are created by defining a method which gets or sets a locally defined variable, and it is possible to automatically generate these methods (see §3.4). Note that the return keyword is unnecessary in the example: Grace methods always return the expression on their final line.

Arbitrary code can appear inside of an object expression, and will be run imperatively when the object is constructed. Fields declared inside of an object are accessible immediately following the construction of an object, but they are uninitialised until they are assigned a value by the imperative sequence of the object body.

The language takes a pragmatic approach to object initialisation: access to uninitialised variables raises a run-time error. A safe initialisation scheme, such as Delayed Types, Masked Types, Hard Hats, Freedom Before Committment, or the Billion Dollar Fix should be able to avoid the pitfalls of uninitialised references (Fähndrich and Xia 2007; Qi and Myers 2009; Gil and Shragai 2009; Zibin et al. 2012; Summers and Müller 2011; Servetto et al. 2013), we discuss how the language permits such extension in §3.5.

Methods are requested with the dot operator, and parameters are passed as comma separated values in parentheses, as in most mainstream OO languages. The receiver of the request can be omitted, in which case the method will be resolved

in the surrounding scope. There are three major differences to mainstream syntax:

1. If there are no arguments, then the parentheses can be left off. The expressions a.apply() and a.apply are equivalent.

2. Method names can be made up of multiple words, each with their own parameters, as in the Smalltalk language (and the languages it influenced, including Self and Objective-C). Such a method is considered 'mixfix', and is useful both for making the purpose of argument values more clear (such as in substringFrom(0) to(5)) and for the definition of control structures that appear no different from the built-in ones. All of Grace's control structures such as if() then() else() and for() do() are used as requests to mixfix methods.

3. Operators are also standard object methods — another feature derived from Smalltalk — but are syntactically the same as infix operators in most languages. Operators can either be infix or prefix. There is also a special 'field assignment' operator, which is defined with a standard method name followed by the assignment operator, :=. This method is requested as though assigning a field with the given name: obj.field := 5.

The combination of these features is intended to allow the full customisation of what are traditionally built-in features without much difference in syntax.

Grace also has syntax for class declarations. The syntax is mostly the same as in a standard class-based language such as Java, but the constructor and the definition are combined to encourage the notion of classes as factory objects (Black, K. B. Bruce, Homer, Noble, et al. 2013). For instance, a class named dog would be defined as follows:

```
class dog(name) {
    method bark {
        print("The dog named {name} barked")
    }
}
```

Classes are not a primitive construct of the language, but are instead syntactic sugar for a combination of objects and methods, enforcing the Grace ideal of having

a single underlying primitive: the object. Grace objects are not defined in terms of classes, and object inheritance does not establish relationships between classes. The definition above introduces a method named dog, and when this method is requested, a new object with the method bark is returned.

The use of name in the string above is another feature of Grace. Expressions can be interpolated directly into a string literal by surrounding them with braces. The contained expression will be evaluated, converted into a string, and then concatenated onto the strings either side of it. The expression dog("Charlie").bark will print "The dog named Charlie barked".

A 'block' in Grace is akin to anonymous functions and lambdas from other languages. A block — denoted by braces – is like a first-class method, a value whose contained code can be invoked by requesting its apply method. If an argument list contains just a single block, then the parentheses can be left off and the braces take their place in delimiting the arguments. The omission of the parentheses allows method requests to resemble the control structures of C and Java.

Using blocks to delay the evaluation of a block of code when requesting a method is similar to the custom control structures of Smalltalk, but Grace allows the definition of these methods in the local scope, providing for syntax closer to the C-like structures such as if-else:

```
if (isTrue) then {
    doTrue
} else {
    doFalse
}
```

The method if() then() else() invoked above is a regular method that applies one of the given blocks depending on the value of the first argument. Developers can define their own control structures that are no different from the standard ones. The following code defines a control structure that runs a block if its first argument is empty:

```
method ifEmpty(list) then(block) {
    if (list.isEmpty) then { block.apply }
}
```

Blocks can also take parameters, introduced by the → symbol. This allows control structures that introduce variables to name and assign those values. This is useful for the standard 'for each' control structure, represented in Grace by the for() do() method. The method takes an iterable object and passes each element to the apply method of the given block. The block receives the arguments passed to apply by defining parameters:

```
for(list) do { element →
    process(element)
}
```

Use of the return keyword in a block signals a return to the enclosing method, not the block. This is important because it allows situations like a conditional return, as in the following:

```
method sum(list) {
    if (list.isEmpty) then { return 0 }
    ...
}
```

As with all values in Grace, blocks are just objects. It's possible to define an object that behaves exactly like a block without the use of the literal syntax.

Each file of Grace code constitutes a module. Modules are imported by designating a path to the containing file, and a name to give the module. Modules are also objects, and all top-level public definitions in a file are exposed on the resulting object when the module is imported (Homer, K. B. Bruce, et al. 2013). A module runs when it is first imported, and subsequent imports receive a pointer to the existing object.

3.2 Inheritance

An object in Grace can inherit the implementation of another object when it is created. The inherit clause is a statement that may appear at the top of an object expression, and the clause contains an expression which is evaluated to build the parent object. Consider an object albie that inherits from the dog class defined above.

```
def albie = object {
    inherit dog("Albie")
    method fetch { ··· }
}
```

This object understands both the fetch and bark methods, the latter of which was inherited from the definition of dog.

Unlike the delegation of Self or JavaScript discussed in §2.1.1, Grace objects cannot inherit from an arbitrary object. The expression in an inherit clause must be a request to a *manifest* method, which means that the actual method that will be called can be statically resolved. The manifest method must construct and return a new object when it is called, and the definition of that object must also be manifest.

These strict requirements on which expressions can be inherited from ensure that the structure of the object that will be inherited is always statically-known, including non-public definitions. This is important in determining which definitions in the inheriting object are new, and which are overriding existing definitions in the parent.

It is also important to know the exact structure of any inherited objects because inherited definitions can shadow local definitions. The albie object above can call its own bark method without qualifying the request with self. If there were some other bark definition surrounding the definition of albie, then the inherited definition takes precedence. In order to correctly resolve references in the presence of inheritance, either for typing or compilation, all of the inherited definitions must be statically-known.

The request in an inherit clause does not actually construct the object that the method would return if requested outside of the clause, but rather the identity of the inheriting object is implicitly passed to the invoked method and the inheriting and inherited objects' definitions are merged together before any initialisation code is run. The inheritance of dog in albie does not create a separate dog object and then inherit from it; instead the bark method is inserted directly into albie. Any initialisation code that might have run inside of dog is run inside of albie instead.

Grace's inheritance semantics are very different from most classless object-oriented languages, which typically permit delegation between arbitrary objects. We explain

and discuss the specific semantics of inheritance in Grace in much more detail in §9.3 and §10.2.

3.3 Types

Types in Grace are both gradual and structural (Black, K. B. Bruce, Homer, and Noble 2012). Unlike most object oriented languages, classes in Grace do not introduce types, with the intention of clearly separating the concepts of factories and types that are intertwined in many class-based languages. Type names are introduced with the type keyword and assigned a structural type value, which lists the signatures of the methods for an object of that type. Parameter types are delimited with the : operator followed by the type. Method return types are delimited with the → symbol.

A Vehicle type might be expected to have a number of wheels, and be able to drive a certain distance. A type that represented such a vehicle would have a method wheels that returns a Number and a method drive which takes a number as a parameter, like so:

```
type Vehicle = {
    wheels → Number
    drive(kilometers : Number) → Done
}
```

The return type of the drive method is Done, a type that indicates no useful return value. This type is like void in Java, except that methods must return a value, so any method that only performs side-effects and does not need to return a value can instead return the sentinel value done.

As types are gradual, it is also acceptable for the return type of drive to be left off entirely. This may be marked explicitly using the Unknown type, equivalent to the gradual dynamic type ?. Every object satisfies the Unknown type, so it is always permitted to omit a type.

Because Grace is structurally typed, any object that implements methods with the signature specified in the Vehicle type satisfies that type. Unlike nominal type systems, objects may be instances of a type without any knowledge of the type at

their point of creation.

Grace also supports generic types (Black, K. B. Bruce, Homer, Noble, et al. 2013). Generic type parameters can be introduced on a type declaration using Oxford brackets ⟦⟧. The following example defines an Iterable type over values of a generic type E:

```
type Iterable⟦E⟧ = {
    iterator → Iterator⟦E⟧
}
```

Generics can also be applied to methods. The next example illustrates an identity function which uses generics to avoid losing type information about its argument:

```
method identity⟦T⟧(value : T) → T { value }
```

Types can also be combined with the ∩ and ∪ operators. The form A ∩ B is a structural type that contains every method that appeared in either A or B. The form A ∪ B is a non-structural type whose inhabitants are the union of the inhabitants of A with the inhabitants of B. This allows a greater degree of flexibility than interface extension from Java, and encourages the separation of concepts into types, as they can easily be composed together to produce the required interfaces. For example:

```
type NumberOrString = Number ∪ String

type Sized = { size → Number }
type Collection⟦E⟧ = Sized ∩ Iterable⟦E⟧
```

Types are reified as objects at run-time, including both named types and generic types. The body of the identity method above can refer to T as an object and call methods on it, or even return it (though this would not satisfy the type of the signature).

3.3.1 Patterns

Type casting is an important component of the Java type system, so much so that Featherweight Java — a bare-bones core calculus — includes it in its type system. It is particularly important for down-casting a general type to a more specific instance. Branching on a more specific type of an object is a common Java idiom.

For instance:

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.bark();
} else if (animal instanceof Cat) {
    Cat cat = (Cat) animal;
    cat.meow();
}
```

Grace does not have a special mechanism for casting between types, and instead encodes the idiom above by using pattern matching, which subsumes both the check and the casting assignment.

```
match(animal)
    case { dog : Dog → dog.bark }
    case { cat : Cat → cat.meow }
```

Pattern matching isn't restricted to types, and any object that is an instance of the Pattern type can be used. For instance, literals such as numbers and strings can also be matched against, branching if the matcher and the matchee are equal.

The interface of Pattern defines a match method that accepts an object and determines if the pattern matches it. Homer, Noble, et al. (2012) show how this allows pattern matching against any value, allowing the match–case form to subsume Java's switch statements and idioms such as the one above.

When types are reified as objects at run-time, they are given an interface that conforms to the Pattern type. The match method on types subsumes the behaviour of Java's instanceof operator: the Java expression pet instanceof Dog can be written as Dog.match(pet) in Grace.

```
if (Number.match(amount)) then {
    print("{amount} is a number")
}
```

Both ∩ and ∪ are present as operators on the Pattern type, which is the interface used for pattern matching. Literals are another example of a pattern, and the match only against values equal to them. It is valid to combine different patterns through

the combination operators: for instance the expression String ∪ 5 will produce a pattern that matches against any object that is either a string or the number five. The introduction of non-type values to a pattern means that the resulting value cannot be used as a static type.

Grace does not have a special mechanism for casting between types, but gradual typing allows the language to simulate this as any object can be stored into a variable with the dynamic type and then be retrieved as any other type. This behaviour can be codified into a method using Grace's generic types:

```
method cast⟦T⟧(value : Unknown) → T { value }
```

The type to cast the value into is specified as the generic type. This allows a call to the cast method to specify what the resulting type is expected to be. For instance, we can attempt to cast a spoon to a fork as follows:

```
def spork : Fork = cast⟦Fork⟧(spoon)
```

Note that the cast will be checked at run-time and raise a run-time type error if the spoon is not also a fork. In contrast, pattern matching ensures that the types match before performing a type cast, and so is safe.

3.4 Annotations

Java has two different mechanisms for specifying properties of methods and classes. Built-in concepts like visibility (publicity) and concurrency synchronisation are all denoted by keywords. Properties that are not assigned a keyword, such as deprecated code or explicit method overriding, are instead defined with annotations. As in the Ceylon programming language (King 2016), Grace denotes all of these properties with user-definable annotations. Annotations can be attached with the is keyword:

```
object { method square(x : Number) is public { x * x } }
```

While the use of the name public has (effectively) the same meaning as the keyword in Java, the use of an access modifier in Grace is closer to the use of an annotation in Java, where it would be labelled with an @ symbol, such as @Public. The following Java method illustrates the differences:

```
@Override public synchronized void doSomething() { ... }
```

In the equivalent Grace code, all of these properties are annotations, as none of them are a special component of the language. Any of them could be replaced by different annotations that have the same effects:

```
method doSomething is override, public, synchronized { ... }
```

Annotations can be applied to more than just methods. Variables defined within an object expression become fields with corresponding accessor methods, but the generated methods are always hidden by default. Constant definitions can be marked public, and variables can be marked either readable, writable, or public, and these annotations change the visibility of the relevant accessor methods. The following declaration translates into an object with publicly accessible name and age:= methods:

```
object {
    def name : String is public = "Bert"
    var age : Number is writable := 34
}
```

Different implementations of Grace (see §3.6) interpret annotations differently. One implementation builds custom annotations into the platform directly and ignores any annotation that is not already defined beforehand. Others require that an annotation expression return an object of the right type, and the type is used to determine what the annotation should do. Our practical implementations were implemented assuming the latter; this is discussed further in §6.5.

3.5 Dialects

As Grace is a general-purpose educational language, it is intended to be capable of teaching a variety of different programming skills. While some courses may benefit from the introduction of types after the more fundamental skills are taught, others may require that every value be statically typed. A course on functional programming might enforce that all data be immutable and introduce more standard functions for transforming over structures.

Grace solves this issue using dialects. Dialects are similar to Racket's languages (Felleisen, Findler, Flatt, et al. 2015) in that they modify the context of a module without changing the entire program, but aren't capable of changing the basics of the language as fundamentally as Racket languages. The major difference is that a dialect can't change the syntax of the language, so uses of object or type are still the same. Dialects can restrict usage of language constructs: a program that only uses dynamic types can disable the use of the type declaration and assignment of types, while another program might require type annotations on every method and variable. Dialects also define the methods that are in the surrounding scope of a module, which allows a dialect to specify the available control structures.

Dialects are built as a module. Dialects affect only those modules that use them, and modules of different dialects can freely import each other. They are declared at the top of a module with the dialect keyword (in the same way as Racket's #lang). All publicly available methods on the object used in the declaration become locally available in the module:

```
dialect "functional"

// Introduced by the dialect
map { x → x + 1 } over(list(1, 2, 3))
```

Dialects also enable pluggable typing by defining 'checker' methods over a module. A checker method receives the Abstract Syntax Tree (AST) of the module it is checking, which is the data structure resulting from parsing. With the AST a checker method can programatically descend through components of the module and ensure that no restricted features are in use. Checker methods are defined by naming a method check, and multiple checker dialects can be combined by passing the AST to each checker method in the desired order. This is an advantage of encoding dialects as standard Grace objects.

An example of a checker method is defined in Figure 3.5.1; the check ensures that every declaration in the source code is explicitly given a type. It descends over every descendant of the top AST node and for any node that is a declaration it evaluates whether that the node has an implicit type annotation. If any node does not have an explicit type, the method reports an error on the node, and continues to descend over the tree. If any node has a report, the check has failed and the

```
method check(root) {
    ast.accept(object {
        inherit visitor
        method visitDeclaration(node) {
            if (Declaration.match(node) ∧
                    node.typeAnnotation.isImplicit) then {
                node.report("Unannotated Error",
                    "{node} has no type annotation")
            }
            super.visitDeclaration(node)
        }
    })
}
```

Figure 3.5.1: An example checker method that requires type annotations

module will refuse to execute.

In Homer, Jones, et al. (2014), we have also built a dialect for writing dialects, where rules can be expressed directly on node types instead of having to use a visitor object to traverse the AST. The example in Figure 3.5.1 would instead be:

```
rule { node : Declaration →
    if (Declaration.match(node) ∧
            node.typeAnnotation.isImplicit) then {
        node.report("Unannotated Error",
            "{node} has no type annotation")
    }
}
```

The rule form maps more closely to the conceptual goal of a checker dialect of writing rules for specific kinds of nodes.

Note that although dialects are built with Grace code, they themselves need to be compiled and executed before the compilation of the module using that dialect can finish compiling. This fits in with the notion of Grace as an interpreted language, where the code is the executable program, rather than having the code precompiled into an executable. The notion of compile-time and run-time is particularly blurred in this case, as any amount of code might need to be run before

compilation of the entire program can be completed.

Dialects cannot change the meaning of the code they are applied to. If the dialect could attach annotations to the given code, then it could solve a number of arguments about default settings. The default accessibility of methods is one such argument brought up in regard to language design: non-public by default is argued as a safer option in a software engineering perspective, but public by default is argued as simpler and requiring less boilerplate. If a dialect was able to change the default accessibility of methods then developers could choose which one to use within the same language.

There are also potential applications for dialects capable of transforming modules in typing. Currently, leaving off a type annotation from a declaration causes its type to become Unknown. A dialect capable of altering type information could instead modify def declarations to infer the type of their assignment (similar to auto in C++11 (Stroustrup 2007), var in C# (C# Project 2015), and let in Rust (Rust Project 2016)), and default the return type of methods to Done (Grace's nearest equivalent to void). There is also potential for performing flow-based inference on var declarations (Pearce 2013).

The danger in allowing dialects to have this much power is that they could change the meaning of the code too radically, making it difficult to understand code written in an unfamiliar dialect. One of the driving design concerns of Grace is that it should be easy to read and comprehend, and providing this ability would likely impact general comprehension.

3.6 Implementation

Grace has a number of implementations: *Minigrace*, a compiler written in Grace itself; *Kernan*, an interpreter written in C#, and our own prototype implementation *Hopper*, an interpreter written in JavaScript. Kernan and Minigrace attempt to follow the Grace specification, whereas Hopper is more focused on allowing the implementation of research concepts on top of Grace. All three implementations are available as free software.

Minigrace compiles Grace code to other languages, which it can then compile or interpret using other compilers. The compiler is self-hosting, which allows it

to compile itself to either backend. As such, the compiler is available as either a native application or a browser script. Unlike Minigrace, Kernan interprets Grace by walking a similar AST to the one passed to a checker method in a dialect.

Hopper is a continuation-passing interpreter, intended to run in a web browser (Jones 2016). Hopper can yield to the event loop of a browser at any time, and does so at regular intervals. This prevents long-running and non-terminating executions from locking up a browser's interface, because the execution pauses regularly to allow the browser to respond to events. The constant yielding also means that Hopper can simulate lightweight threading easily by queuing different evaluations for execution at the same time.

The Hopper interpreter is the basis of our practical implementation work for this thesis. We have implemented both the structural type checking described in Chapter 4 (Homer, Jones, et al. 2014) and branded type checking described in Chapter 6 (Jones, Homer, and Noble 2015) as dialects in Hopper. We have also implemented the most comprehensive run-time type checking framework of the three implementations, attempting to preserve the properties demonstrated in Chapter 5. The history of Hopper also contains implementations for many of the inheritance semantics described in Part III, as the specification for Grace's inheritance evolved over time.

Part II

Type Systems

4 Graceless

In order to investigate the semantics of classless object-oriented languages, and analyse the simulation of class features within them, we begin by defining a core model of the Grace programming language. Grace is a useful language to model object features in, as it is not class-based, and it does not permit mutation of an existing object's structure, so we can consider object-oriented concerns and a reasonable environment for typing without being overcome by objects open for extension with trivially mutable structure, as found in languages such as JavaScript. Future chapters extend this model in different ways to construct the different semantic models; within this chapter we consider the features of classes that the model is already capable of simulating.

Our core model is a combination of our existing models of Grace: *Tinygrace* (Jones and Noble 2014), a much smaller language with structural types, whose primary purpose was to compare the evaluation of simple Grace programs to Featherweight Java; and *Graceless*, which was really a family of languages used to investigate object inheritance (Jones, Homer, Noble, and K. Bruce 2016, which we return to in Part III, recast in the updated language defined in this chapter) with more of the features of the full Grace language, including references, mutable objects, and unqualified method lookup, but no types. In this chapter, we present an updated model of the Graceless language extended with structural and union types as well as some extra dynamic features, showing that the resulting type system enforces safe execution of programs.

Graceless is a purely object-oriented language, as there are no primitive forms other than objects. As in Grace, objects are defined using object constructor forms with methods and initialisation code contained inside. Graceless is intended as an encoding of parts of the Grace language, rather than a direct model of Grace itself,

and it is not a strict subset of the language as it contains forms and behaviour that are not directly available in Grace. We introduce these forms in §4.1, and we discuss the features of Grace they are intended to encode.

As a tool for our investigation into the semantics of classless languages, the design of Graceless is intended to be both expressive enough to encode the practical features that appear in many classless languages, and extensible enough to permit the modifications that we will investigate both in this chapter and those that follow. That said, Graceless is not intended to be a foundational calculus for classless object-oriented languages, and it intentionally includes complications from the full Grace language that might otherwise be compiled away in an intermediate language. The purpose of these complications is to examine how the complexities of practical object-oriented languages interact with the lack of classes and attempts to simulate class-like behaviour.

4.1 Syntax

The grammar for Graceless is defined in Figure 4.1.1. Graceless is a classless object-oriented language: objects are constructed using object constructors that directly describe the structure and implementation of the object to be created, and the language includes the implementation for each individual object directly in the store in lieu of a class table. Execution predominantly proceeds through method requests to an object. A method request to an object can be provided explicitly, or the receiver can be omitted and implicitly determined from the nearest surrounding object constructor, so methods appear alongside variables in the local scope, and their identifiers share a namespace. Other features include method updates, raising and rescuing objects to signal exceptional outcomes, and branching based on a reflective examination of an object. Graceless code can also be imperatively sequenced between semicolons.

We proceed by considering each class of syntax separately, beginning with terms. For all forms, an overbar such as $\overline{\mathrm{t}}$ indicates a (possibly empty) sequence of the form under the bar, usually separated with commas when it is not immediately unambiguous.

Grammar

$$w, x, y, z \in \textsc{Var}, \quad T \in \textsc{Type}, \quad S \in \textsc{Struct}, \quad D \in \textsc{Decl},$$
$$d \in \textsc{Def}, \quad t \in \textsc{Term}, \quad v \in \textsc{Value}, \quad m \in \textsc{Name}, \quad n \in \mathbb{N}, \quad s \in \textsc{Subst},$$
$$a \in \textsc{Ident}, \quad \Gamma \in \textsc{Env}, \quad \sigma \in \textsc{Var} \rightharpoonup \textsc{Seq}(\textsc{Def})$$

T	$::=$	$\bigcup \overline{S}$	*(Type)*
S	$::=$	$\mathsf{type}\ \{\ \overline{D}\ \}$	*(Structural type)*
D	$::=$	$m(\overline{z : T}) \to T$	*(Signature)*
a	$::=$	$\langle m, n \rangle$	*(Signature identifier)*
d	$::=$	$\mathsf{method}\ m(\overline{z : T}) \to T\ \{\ t\ \}$	*(Definition)*
t	$::=$	$\mathsf{object}\ \{\ \overline{d}\ t\ \} \mid r m(\overline{t}) \mid w \leftarrow d \mid \Uparrow t \mid t \mathbin{\overline{\Uparrow}} b \mid t \ni a\ b\ b \mid t; t \mid w$	*(Term)*
r	$::=$	$\epsilon \mid t.$	*(Receiver)*
w	$::=$	$\mathsf{self} \mid y$	*(Internal reference)*
b	$::=$	$\{\ z \to t\ \}$	*(Block)*
v	$::=$	y	*(Value)*
m	$::=$	$z \mid z{:=}$	*(Method name)*
s	$::=$	$v/z \mid w./a$	*(Substitution)*

Environments

Γ	$::=$	$\cdot \mid \Gamma, z : T$	*(Typing environment)*

Evaluation contexts

E	$::=$	$F \mid F[E \mathbin{\overline{\Uparrow}} b]$	*(Term context)*
F	$::=$	$\square \mid G$	*(Rescue-free context)*
G	$::=$	$F.m(\overline{t}) \mid v.m(\overline{v_i}, F, \overline{t}) \mid m(\overline{v}, F, \overline{t}) \mid \Uparrow F \mid F \ni a\ b_1\ b_2 \mid F; t$	*(Sub-context)*

Figure 4.1.1: Graceless grammar

4.1.1 Terms

The metavariables w, x, y, and z refer to different sets of variables: like the DOT calculus (Rompf and Amin 2016), we encode store references as variables, and distinguish between these *concrete* store variables and normal *abstract* variables with the y and z metavariables. The metavariable x includes both concrete and abstract variables, so encompasses the set of all variables. The variable self is an abstract variable, and so may be bound by z, but self is bound implicitly in the body of each object constructor and may not be used in any binding location explicitly.

We assume Barendregt's variable convention when binding self (Barendregt 1981; Urban, Berghofer, and Norrish 2007): each binding is (implicitly) distinct, and we can always (implicitly) distinguish between two different bindings of self in the same scope. The convention means that two bindings of self do not shadow one another, which corresponds to Grace's (explicit) outer keyword for distinguishing local references. Consider this Grace term:

```
object { object { outer; self } }
```

In the inner object, access to both surrounding objects is still available thanks to the outer keyword. In Graceless outer is replaced by self referring to a class of variables that happen to share the same name on paper. Effectively this is like tagging each use of self with De-Bruijn indices to indicate which surrounding object the self refers to (and this is what we use to encode this form in our Redex implementation), such as:

```
object { object { self_1; self_0 } }
```

These distinctions are omitted from the formal language.

The metavariable w is an *internal* reference to some surrounding object constructor or object in the store, using self or a store reference y. Internal references are used to qualify operations that are only permitted from inside of the object they are operating on.

Terms are bound by the metavariable t. The terms that also appear in Grace are object constructors object { $\overline{d}$ t }, method requests $rm(\overline{t})$, and an imperative sequence $t_1; t_2$ (where the semicolon is more often a newline in Grace). Object constructors contain a set of definitions $\overline{d}$, which are methods method $m(\overline{z : T_i}) \rightarrow$

T { t }. A method name m is either an abstract variable name z or the special case 'setter' name z:= from Grace. Graceless does *not* provide a mechanism for multi-part method names, though we will occasionally write them when discussing encodings of Grace programs and assume that this is encoded by some translation from many parts to one. The syntax of types T are discussed in §4.1.2. The method body t is to be evaluated in the context of the parameters $\overline{z}$ in the event that the method is called.

The body of an object constructor also contains an arbitrary term t, which is *initialisation* code to be run in the context of the constructed object after it has been constructed. The term is not returned by evaluating the constructor (which always returns a reference to the newly allocated object): the term exists purely for the purposes of mutating the surrounding object during its initialisation. Unlike Grace, the term is compulsory in Graceless, but can effectively be ignored by writing self, since this must be in scope and referencing it does nothing; we imply this convention in some of the following examples by hiding the term. Object constructors can be nested inside one another, both in the bodies of methods and in the initialisation code.

A method request has an optional receiver indicated by the metavariable r, which is either empty (ϵ) or a term t followed by a period. When a request has a receiver, we say it is *qualified* by the receiver, otherwise it is *unqualified*. Unqualified requests implicitly refer to a receiver through the closest method in the surrounding objects, as in the following program:

```
object { method id(a : T) → T { a } id(self) }
```

The request is equivalent to writing:

```
object { method id(a : T) → T { a } self.id(self) }
```

A request always refers to method name m and zero or more arguments in parentheses. As in Grace, the parentheses may be omitted if there are no arguments, so zero-argument method calls may look like the field access of other object-oriented languages like Java: m and m() are equivalent terms. We include this in Graceless because it has an important consequence on the language when the request has no receiver, namely that such a method request is indistinguishable from a regular variable reference. As a result, Graceless does not include a reference

to an abstract variable z in its definition of a term because this is subsumed by a unqualified zero-argument request where the method name is z.

As in Grace, Graceless method definitions may be overloaded on the value of their arity, so that a single object can have many definitions of methods with the same name so long as they all accept a different number of parameters. As a result, method names contain insufficient information to completely identify a particular method in an object, and signatures contain more information than is necessary, since the type annotations on a method are not part of its identifying information. In order to provide a syntactic form that describes precisely enough information to identify a method, the metavariable $\mathfrak{a}$ binds any pair of method name $\mathfrak{m}$ and arity count $\mathfrak{n}$.

Concrete variables $\mathfrak{y}$ are also valid as terms, and are the only form of values v. Evaluation occurs in the context of an object store σ, which is a partial function from concrete variables $\mathfrak{y}$ to a set of definitions $\mathfrak{d}$ that make up the implementation of the object. Similarly, the store type environment Σ maps concrete variables to the signatures D of the corresponding definitions $\mathfrak{d}$.

The remaining term forms simulate some Grace functionality that does not appear directly in its syntax, or simplifies the way that it appears in the syntax. We discuss the corresponding Grace form that each feature encodes as we introduce them.

Raise and rescue The syntax $\Uparrow \mathfrak{t}$ is used to *raise* the value of $\mathfrak{t}$, exiting normal control flow and immediately producing the raised value as the result of the program. A raise can be *rescued* by the form $\mathfrak{t} \overline{\Uparrow} \mathfrak{b}$, which can handle a raise in the execution of $\mathfrak{t}$ by diverting execution into the block $\mathfrak{b}$. A block is a simple single-parameter binding form $\{ z \rightarrow \mathfrak{t} \}$, like the block syntax of Grace, but the parameter cannot be given a type annotation. If a block $\{ z \rightarrow \mathfrak{t} \}$ is tasked with handling the raise of a value v, then the body of the block $\mathfrak{t}$ is evaluated with z bound to v. There is no type annotation because any value can be raised, and the rescue does not discriminate on the type of the raised value when it captures a raise.

The raise and rescue Graceless forms are similar to the raise method on exception objects and the variety of try/catch methods in the standard dialect. The intention is that these Grace constructs can be encoded using the Graceless raise

and rescue. The raise method in an exception can be implemented as:

```
method raise → None { ⇑ self }
```

A try() catch() method can then be implemented as:

```
method try(t : Block) catch(c : Block) → Object {
    t.apply ⇈ { z → c.apply(z) }
}
```

These two methods encode the basic behaviour of Grace exceptions, though it does not account for a catch block discriminating on the *kind* of exception that was raised: we consider mechanisms to solve this problem in §6.2.3.

Method updates The method update syntax $w \leftarrow d$ replaces a method with the same identifier as d in the object referenced by w. An update will only modify an existing method, and the signature of the method that it replaces must be exactly the same as the signature of d. The use of w as the receiver of the update ensures that this operation is internal to the object constructor that it is updating, since the only valid abstract variable that can appear there is some self reference. The following is not a syntactically valid use of a method update:

```
method modify(a : T) → T { a ← (method m → T′ { t }) }
```

Featuring only internal method updates differs from many classless encodings of object-oriented languages that feature method updates (Abadi and Cardelli 1996; Siek and Taha 2007; Siek, Vitousek, and Bharadwaj 2012), where any object can be externally updated from outside of that object's methods. In Graceless, an object is responsible for its own behaviour and so only it has the permission to do so, with the exception that objects constructed by nested constructors implicitly have permission to modify any surrounding objects as well, since the outer self is in scope.

```
object {
    method m → T { t }
    object { outer ← (method m → T { t′ }) }
}
```

Since this update still occurs within the definition of the updated object, it is still considered to be an internal operation.

Method updates simulate field updates in the body of a generated setter or at the point of a field's initialisation in Grace. The updates can simulate much more than this, because they can update any method instead of only field getters, but since they are always internal to an object this is equivalent to an object changing its own behaviour in reaction to some event. An object cannot manually change the behaviour of another object without its consent, except when the former is syntactically nested in the latter.

Method updates are necessary to encode both var and def field definitions in Grace. First we consider the var form; consider the following Grace object constructor:

```
object { var a : T }
```

This is trivially encoded in Graceless with:

```
object {
    method a → T { ⇑ uninitialised }
    method a:=(v : T) → Done { self ← (method a → T { v }); done }
}
```

The encoding assumes that there is some surrounding method definition uninitialised that returns a representation of an UninitialisedFieldError packet, along with done and some accompanying Done type. Requesting the method a on this object before it is assigned produces an error, whereas requesting the method a:= modifies the body of a to return the given value instead.

If a var should be assigned to at its definition site, this can be encoded in Graceless as a request to a:= at the corresponding point in the initialisation code:

t_1; var a := t_2; t_3

The assignment is encoded in Graceless as:

t_1; self.a:=(t_2); t_3

The term t_2 will be reduced to a value before requesting a:=, ensuring that the getter method does not evaluate the term every time it is called.

A def field is more difficult to encode, because it does not have a corresponding setter method to request at its point in the initialisation code.

```
t1; def b = t2; t3
```

Updating the method directly does not first reduce the term to a value, so the following encoding does not evaluate t_2 during initialisation:

```
t1; self ← (method b → T { t2 }); t3
```

Graceless does not have a let construct to bind a term's value to a name, but, as in the DOT calculus, we can simulate this feature with a request on an object constructor (Rompf and Amin 2016).

```
t1; object {
    method apply(v : T) → ⊤ { self ← (method b → T { v }) }
}.apply(t2); t3
```

By forcing the term t_2 through a method request first, the updated method can refer to the value of the term as a reference to the parameter v.

Reflective matching Finally, a *match* t ∋ a b_1 b_2 checks to see if the value of the term t contains a method with the identifier a, and branches down b_1 if it does, or b_2 if it does not. The block parameter is bound to the value of t in either case. Because a match only looks for a matching method identifier and not a full signature, it does not tell you anything about the type annotations on the method in the case that it does exist, only that such a method is present. The behaviour of a match distinguishes the Graceless match form from Tinygrace, which matched on a full type, but allows it to express the actual behaviour of Grace implementations. We explain the reason for this difference in §4.2.

The match construct allows for the encoding of pattern matching against a structural type as described in Homer, Noble, et al. (2012), present in all implementations of Grace. The match method in the reified object of the type type { m_1 → T_1, m_2 → T_2 } can be encoded in Graceless as:

```
method match(a : Object) → Boolean {
    a ∋ ⟨m1, 0⟩ { b → b ∋ ⟨m2, 0⟩ { c → true } { c → false } } { b → false }
}
```

The matching only works on a shallow level, identifying if a method with the appropriate name and arity is present, so matching against a structural type only performs a shallow test. When typing the body of the match/case construct in Grace, the type checker can only assume that the shallow description of the pattern applies to the object.

Despite the fact that several implementations of Grace also perform only a shallow match to test if an object satisfies a type, we cannot fully encode the match method of a block object. The method only executes the body of its block if the input satisfies the shallow structure of its parameter type, and *assumes* that the input satisfies the rest of the parameter type during the execution with a contract that enforces the non-shallow aspects of the type. For instance, this is a request to match on a block in Grace:

```
{ x : type { go → T } → x.go }.match(t)
```

The block will test if the value of t has a go method and, if successful, will execute x.go under the assumption that this results in an object of type T; that is, it should check that the result matches the shallow description of T, and continue to assume that it is of type T if it does (though implementations of Grace vary on whether this the contract behaviour is actually implemented, as observed by Boyland (2014) in Minigrace). In the form we have presented so far, Graceless has no syntax for expressing assumptions about the type of an object, but we address this is Chapter 5.

4.1.2 Types

Graceless features a similar type syntax to Grace, but it encodes some of the type operators of Grace differently to more easily reason about the resulting types. The structural type syntax of Grace groups together signatures like a normal structural type, and these types can be combined with either the union ∪ or intersection ∩ operators. In Graceless, these operators translate their inputs into a normal form, which is a set of structural types combined together in a union relationship, and unions, structural types and signature declarations occupy different syntactic classes (T, S, and D, respectively). The existing class of structural types is sufficient to express intersection, but unions need their own syntax in the normal form.

Splitting the type syntax into a hierarchy like this makes it simpler to define

relations over types, because we can define a single operator as an overloaded set of mutually dependent relations over each part of the hierarchy, and do not need to worry about relations between different parts of the hierarchy: we do not need to consider how to compare a union type with a structural type, for instance. There are additional complications with completeness of subtyping that this hierarchy side-steps as well, which we discuss in §4.2.4.

As such, the syntax of types is split into a mutually coinductive hierarchy, beginning at the metavariable T for a set of structural types $\bigcup\overline{S}$ in a union relation. We often treat $\overline{S}$ like a set since both ordering and duplicates are irrelevant. Not all types in Grace are the result of a union, but we interpret them all as in the union syntax for the normal form of Graceless. A union type can be between an arbitrary number of structural types, including zero or one, so a structural type in Grace is a single structural type in the union syntax by itself.

An empty union corresponds to the empty type (None in Grace), and a union with a single structural type corresponds directly to that structural type as though it was not in a union. In the text we often omit the $\bigcup$ symbol on a single structural type and still consider it a full type T (so type $\{\cdots\}$ in T is actually $\bigcup$type $\{\cdots\}$). In both the text and the following rules, we use the symbol $\bot$ to mean the bottom type ($\bigcup$) that is uninhabited, and the symbol $\top$ to mean the top type $\bigcup$type $\{\}$ that is inhabited by every value.

Where T encompasses a union of a set of structural types $\overline{S}$, a structural type type $\{\ \overline{D}\ \}$ effectively represents an intersection of a set of signature types $\overline{D}$. A signature D is the header of the methods d that were discussed in §4.1.1, of the form $m(\overline{z : T_i}) \to T$. An object inhabits a structural type type $\{\ \overline{D}\ \}$ if it provides methods with at least the signatures $\overline{D}$. Parameter names are not significant, and any equivalence we draw between the parameter names of two different signatures is only up to alpha-equivalence.

Coinductive types In a signature $m(\overline{z : T_i}) \to T$, the type annotations $\overline{T_i}$ and T loop back around to our definition of types as unions. Graceless does not have an explicit syntax for recursive types through mu-binders (Pierce 2002), but as described above the mutual dependence between T, S, and D is *coinductive*, so the syntactic objects themselves can be infinite in size. We can immediately encode

recursive types without any explicit μ-bindings.

As a translation from Grace code, Graceless represents an encoding where any recursive types, which must have been expressed in Grace's finite syntax, have already been *unfolded* into a syntax tree of infinite depth. The infinite form is trivially represented in many modern programming languages by constructing the tree representation up to its point of repetition, and then mutating the child reference at this point to refer back to the top of the tree, or defining the tree in a directly coinductive way such as with Haskell's lazy let binding.

As an example of this coinductive type syntax, we will define a Graceless type *List* that represents a linked list containing values of some type T. Such a list is either an empty object to represent the end of the list (represented by the type with no signatures type {}), or a pair of head (the value of type T) and tail (the remaining list). The type is not particularly useful, since it contains every object thanks to the appearance of type {} in the union, but it will suffice to investigate the structure of the syntax.

If Graceless had explicit μ-binders, we could define *List* with a finite representation:

$$\mathit{List} = \mu X.\bigcup \texttt{type}\ \{\},\ \texttt{type}\ \{\ \texttt{head} \rightarrow \texttt{T},\ \texttt{tail} \rightarrow \texttt{X}\ \}$$

Unfolding this type by one step moves the μ-binder into the return type of tail, repeating the whole type again therein:

$$\mathit{unfold}(\mathit{List}) = \bigcup \texttt{type}\ \{\},\ \texttt{type}\ \{\ \texttt{head} \rightarrow \texttt{T},\ \texttt{tail} \rightarrow \mathit{List}\ \}$$

The reference *List* in this definition is not its own syntactic object, but indicates that *List* appears at that point in this unfolding. The total type looks like this:

$$\bigcup \texttt{type}\ \{\},\ \texttt{type}\ \{\ \texttt{head} \rightarrow \texttt{T},\ \texttt{tail} \rightarrow \mu X.\bigcup \texttt{type}\ \{\},\ \texttt{type}\ \{\ \texttt{head} \rightarrow \texttt{T},\ \texttt{tail} \rightarrow \texttt{X}\ \}\ \}$$

These two types are semantically equivalent because under an infinite series of unfolding they would produce the same type, but they are syntactically distinct and require extra reasoning in any syntax-directed rules for type relations in order to produce an equirecursive realisation of type equality. If we consider the syntax trees of these two types, we can immediately see the syntactic distinction between

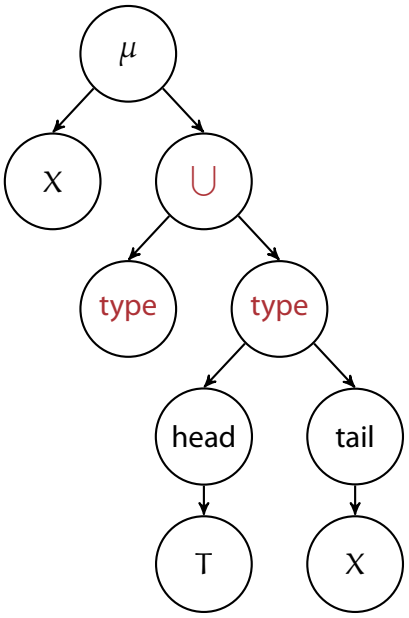

Figure 4.1.2: Syntax tree for the inductive *List* type

them and that their representation as syntactic objects is finite. The syntax tree for *List* is visualised in Figure 4.1.2, with the point of recursion expressed by the use of the variable X inside the tail node.

The outcome of *unfold*(*List*) is visualised in Figure 4.1.3. As in the textual representation, the unfolded tree contains the entire original tree where the reference to X was. While these two trees are different, they represent the same type, and must be manually unfolded when reasoning about their relationships with other types.

A coinductive syntax lets us avoid the syntactic encumbrance of μ-binders by defining the infinite unfolding of the type directly. Coinduction avoids the problem of semantically equivalent but syntactically distinct types, and maps more closely to the meaning of the type. In Graceless, the *List* type would be defined as:

$$List = \bigcup \text{type } \{\}, \text{type } \{ \text{ head} \rightarrow \text{T}, \text{tail} \rightarrow List \}$$

As before, the reference to *List* in this type is not a syntactic object in its own right, but an indication that we should include the value of *List* at that point in the type. The definition is different from when we did this in the unfolding of the finite rep-

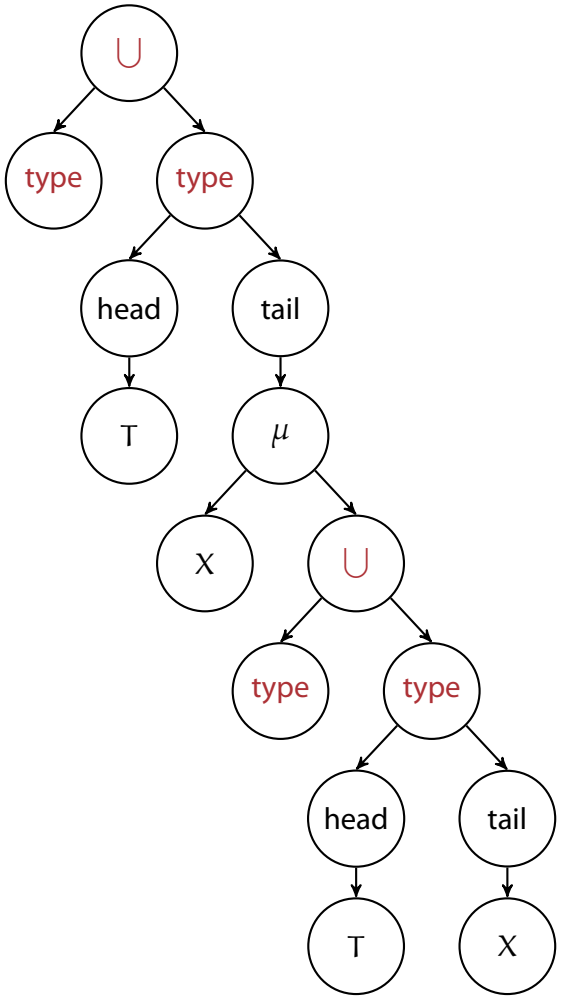

Figure 4.1.3: Syntax tree for the unfolding of *List*

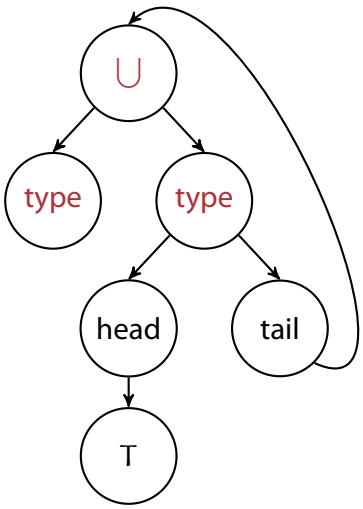

Figure 4.1.4: Syntax tree for the coinductive *List* type

resentation because we are referencing the value of the type *in the definition of the type itself*, so the resulting syntactic object has an infinite depth. Figure 4.1.4 illustrates this definition as a syntax graph. Interpreted as a tree, it has its own root node as a child of the tail node.

The primary trade-off for expressing the type syntax coinductively is that, although we know that the types are regular, we cannot detect the actual points of repetition in a type of infinite depth without explicit μ-binders to mark them. As a result, any relations over types must also be coinductive, and cannot be decidable, because that would require descending over an infinitely deep structure in finite time. We do not view this as particularly significant because the points of repetition *will* be detectable in a practical implementation (probably through reference identities), so although our relations over types are not algorithmic, they can be translated into an algorithm for a practical implementation in a straightforward way.

The structure of this syntax includes two implicit well-formedness criteria. The first criteria is *regularity*, which implies that any infinitely recursive type must be the result of an infinite unfolding of some finite representation (such as with μ-bindings). Because Grace's syntax is finite, any Graceless type translated from Grace is the result of such an unfolding and so automatically satisfies this criteria. Type regularity ensures that any infinite representation of a type is made up of

well-defined points of repetition, even if these points are not represented explicitly in the syntax as they are with μ-binders.

The second well-formedness criteria is *contractivity*, which ensures that while a type may have infinite depth, it cannot have infinite *breadth*: the number of direct children of any syntax node must still be finite. The set of types in a union must be finite, as well as the set of signatures in a structural type and the sequence of parameters in a signature. This criteria is not automatically satisfied by types with μ-bindings or the type syntax in Grace because it is possible to express infinite breadth in their syntax, such as the type:

$$\mathit{BadType} = \mu \mathsf{X}.\bigcup \mathsf{type}\ \{\}, \mathsf{X}$$

Unfolding this type increases its breadth, not its depth, so an infinite unfolding produces a type with infinite breadth. Contractivity subsumes a ban on obvious nonsense type syntax such as μX.X as well, since these forms cannot be infinitely unfolded into any sensible coinductive type.

Contractivity is enforced by the definition we have provided in the mutual dependence of the type syntax. The hierarchy only repeats at the appearance of T in a signature: any part of the type syntax that represents breadth is represented with the multiplicity syntax ($\overline{\mathsf{S}}$, $\overline{\mathsf{D}}$, and $\overline{\mathsf{T}}$), which does not permit the set or sequence that it binds to have an infinite cardinality, regardless of the size of its individual elements. If we had μ-binders in the language then the *BadType* syntax would not be valid syntax, because a recursive type reference X would only be bound by the metavariable T, but in *BadType* it appears in an S position.

The type syntax also has explicit well-formedness rules, mostly related to ensuring sensible signatures in a structural type. We return to discussing types, including the well-formedness judgment, in §4.2.

4.1.3 Substitution

Unlike most formal languages, Graceless has more than one form of substitution, so we explicitly elucidate this syntax in the grammar behind the metavariable s. An application of substitution still follows the syntax [s]t, performing the substitution described by s in the term t. We provide an informal description of these

substitutions here. The syntax v/z is a standard *value* substitution of a value v at any unshadowed reference of an abstract variable z.

The non-standard syntax denotes a *qualifying* substitution $w./a$, which corresponds to resolving the receiver of an unqualified request in local scope. Consider the following program:

```
object {
    method run { ··· }
    run
}
```

When this object is constructed, the object is allocated behind a concrete reference y, and the initialisation code of the object constructor is sequenced with a return of this reference:

```
run; y
```

The problem with this sequence is that we no longer have enough information to determine which object was intended to receive the unqualified request to run, because the object that formed the surrounding scope has been removed by the reduction step.

A qualifying substitution solves the problem of resolving the receiver of an unqualified request by adding a receiver to every unqualified request of the specified identifier a. The object constructor above actually reduces to a substitution:

$$[y./\langle \mathsf{run}, 0\rangle]\mathsf{run}; y$$

The substitution qualifies the request to run with the receiver y.

```
y.run; y
```

As a result, we don't need to provide any other dynamic semantics for an unqualified request, because any unqualified request that correctly refers to a method definition in scope is guaranteed to be qualified by the time it comes to evaluate it. Attempting to evaluate an unqualified request is an error in the same way as evaluating a free variable.

The fact that variable references and unqualified zero-argument requests are indistinguishable means that both parameters and method definitions can shadow

each other, so either kind of substitution can be stopped by either kind of definition. A substitution v/z would normally proceed into an object:

$$[v/\mathsf{x}]\mathsf{object}\,\{\,\mathsf{x}\,\} \equiv \mathsf{object}\,\{\,\mathsf{v}\,\}$$

If the object defines a method with identifier $\langle z, 0\rangle$, then this is no longer the case, as the method shadows the variable.

$$[v/\mathsf{x}]\mathsf{object}\,\{\,\mathsf{method}\,\mathsf{x}\,\{\,\cdots\,\}\,\mathsf{x}\,\} \equiv \mathsf{object}\,\{\,\mathsf{method}\,\mathsf{x}\,\{\,\cdots\,\}\,\mathsf{x}\,\}$$

The same is true for qualifying substitutions into a method. A qualifying substitution normally proceeds into the body of a method:

$$[w./\langle \mathsf{x}, 0\rangle]\mathsf{method}\,\mathsf{m}(\mathsf{y} : \mathsf{T}_1) \to \mathsf{T}_2\,\{\,\mathsf{x}\,\} \equiv \mathsf{method}\,\mathsf{m}(\mathsf{y} : \mathsf{T}_1) \to \mathsf{T}_2\,\{\,w.\mathsf{x}\,\}$$

If the method has a parameter with the same name as a zero-argument qualifying substitution, then it stops the substitution from entering the body of the method.

$$[w./\langle \mathsf{x}, 0\rangle]\mathsf{method}\,\mathsf{m}(\mathsf{x} : \mathsf{T}_1) \to \mathsf{T}_2\,\{\,\mathsf{x}\,\} \equiv \mathsf{method}\,\mathsf{m}(\mathsf{x} : \mathsf{T}_1) \to \mathsf{T}_2\,\{\,\mathsf{x}\,\}$$

Shadowing is otherwise consistent with what is expected: parameters stop value substitutions to variables of the same name, and method definitions in an object stop qualifying substitutions to requests of the same method identifier.

Future chapters further complicate this story by adding additional forms of substitution, as well as making it possible for the substitution syntax to appear directly in a term, to the extent that a substitution operation may need to perform a substitution inside of another substitution s. We explain these complications when they become relevant.

4.1.4 Evaluation Contexts

The definitions of the evaluation contexts E, F, and G also deserve some explanation. The contexts are split into these three definitions in order to avoid raising a value through a rescue without handling it, while still having a context for performing a congruence reduction on any sub-term. G allows a raise to terminate a program without also accepting an already terminated program; F extends G with a direct hole to handle rescuing a raised value without skipping over an intervening rescue; and E extends F to permit congruence evaluation inside the body of a rescue.

As is standard, we abuse the evaluation context syntax to treat an application $\mathsf{E}[\mathsf{t}]$ as a term, rather than manually describing a relation between such an application and its corresponding term $\mathsf{t}_1 \ltimes \mathsf{E}[\mathsf{t}_2]$ and proving that the relation is well-moded in both directions (Harper 2012). The application $\mathsf{E}[\mathsf{t}]$ descends through any appearance of a recursive context as defined by E, inserting t for a hole $\square$.

The context F is defined in a standard way, with either a direct hole $\square$ or a recursive definition such as $\mathsf{F}.\mathfrak{m}(\overline{\mathsf{t}})$, indicating the sub-term to evaluate appears in the receiver of the request. F does not include the rescue syntax, so a rescue cannot appear in the context either as the directly matched term or some surrounding context of the matched term.

In order to define E as an extension of F with the body of a rescue form included, it is not sufficient to add the definition $\mathsf{E} \Uparrow \mathsf{b}$ alongside F, because F only recurses on F: such a definition would only allow E to find a sub-term in a rescue if it appeared as the other-most form of the term. As soon as E delegated to F, it would not be able to consider a rescue. E would include $\square \Uparrow \mathsf{b}$, but not $\mathfrak{m}(\square \Uparrow \mathsf{b})$, because F only permits the context F in the arguments of a request. The context E extends F with $\mathsf{F}[\mathsf{E} \Uparrow \mathsf{b}]$, which provides the expected behaviour: a rescue can be matched in any hole which is valid for F, at which point the actual hole for the application of E can recurse on E again. Our example $\mathfrak{m}(\square \Uparrow \mathsf{b})$ can appear in E, because it can match the rescue in an F hole, and the body of the rescue then recurses on E which permits a direct hole to appear there.

We use these evaluation contexts in the definition of reduction, defined and discussed in §4.3.

4.2 Types

In §4.1.2 we discussed the syntax of types, including the infinite nature of the syntax. In this section, we define semantic well-formedness criteria for types, the type combinators for union and intersection, signature subtraction from types, and the subtyping relation.

$$\boxed{\vdash T} \qquad \text{(W-Uni)}\ \frac{\vdash \overline{S}}{\vdash \bigcup \overline{S}} \qquad \boxed{\vdash S} \qquad \text{(W-Str)}\ \frac{\overline{\mathit{identify}(D)\ \text{unique}} \qquad \vdash \overline{D}}{\vdash \mathsf{type}\ \{\ \overline{D}\ \}}$$

$$\boxed{\vdash D} \qquad \text{(W-Sig)}\ \frac{\vdash \overline{T_i} \qquad \vdash T}{\vdash m(\overline{x_i : T_i}) \to T}$$

$$\begin{aligned} &\mathit{identify} : (\textsc{Decl} \cup \textsc{Def}) \to \textsc{Ident} \\ &\mathit{identify}(m(\overline{x : T_i}) \to T) \quad = \langle m, |\overline{x}| \rangle \\ &\mathit{identify}(\mathsf{method}\ D\ \{\ t\ \}) = \mathit{identify}(D) \end{aligned}$$

Figure 4.2.1: Type well-formedness

4.2.1 Well-Formedness

The well-formedness relations for types are defined in Figure 4.2.1. These rules ensure that types are sensible on top of the implicit well-formedness criteria specified in §4.1.2. Well-formedness is defined for union types $\vdash T$, structural types $\vdash S$, and signatures $\vdash D$, and simply ensures that the identifiers of the signatures of any structural type are unique. Like the syntax itself, this judgment is defined coinductively, allowing it to express well-formedness for infinitely deep types.

The auxiliary function *identify* is also defined, which computes the method identifier of either a signature or method. We use the function here to describe the well-formedness property, but it appears in the rules that follow to both ensure this well-formedness is upheld by the types assigned by the typing judgment and to compare methods and signatures in other ways.

4.2.2 Type Combinators

Grace features several type combinators as binary operators. We define the two primary combinators intersection $\cap$ and union $\cup$, and use them in the typing rules that follow in this chapter.

Since we already have a syntax for type unions, the union combinator $\cup$ is simple to define, as it takes the structural types in either type and puts them together.

$$\bigcup \overline{S_1} \cup \bigcup \overline{S_2} = \bigcup \overline{S_1, S_2}$$

$$\cap : \textsc{Type} \times \textsc{Type} \to \textsc{Type}$$
$$\bigcup \overline{S_i}^{i \le n} \cap \bigcup \overline{S_j}^{j \le m} = \bigcup \overline{S_i \cap S_j}^{(i,j) \le (n,m)}$$

$$\cap : \textsc{Struct} \times \textsc{Struct} \to \textsc{Struct}$$
$$\mathsf{type}\ \{\ \overline{D_i}\ \} \cap \mathsf{type}\ \{\ \overline{D_j}\ \} = \mathsf{type}\ \{\ (\overline{D_i} \cap \overline{D_j})\ \}$$

$$\cap : \textsc{Seq}(\textsc{Decl}) \times \textsc{Seq}(\textsc{Decl}) \to \textsc{Seq}(\textsc{Decl})$$
$$(D_1, \overline{D_i}) \cap (\overline{D_j}, D_2, \overline{D_k}) = (D_1 \cap D_2), (\overline{D_i} \cap (\overline{D_j}, \overline{D_k}))$$
$$\textbf{where } \mathit{identify}(D_1) = \mathit{identify}(D_2)$$
$$(D_1, \overline{D_i}) \cap \overline{D_j} = D_1, (\overline{D_i} \cap \overline{D_j})$$
$$\cdot \cap \overline{D_j} = \overline{D_j}$$

$$\cap : \textsc{Decl} \times \textsc{Decl} \rightharpoonup \textsc{Decl}$$
$$m(\overline{z_i : T_{i1}}) \to T_1 \cap m(\overline{z_i : T_{i2}}) \to T_2 = m(\overline{z_i : (T_{i1} \cup T_{i2})}) \to (T_1 \cap T_2)$$

Figure 4.2.2: Type intersection combinator

Any duplicate structural types are removed from the result under our treatment of the structural types in a union as a set, but this isn't strictly important. The result is a type that is inhabited by all of the values inhabited by either of the two combined types.

Defining the intersection combinator $\cap$ is more complicated. Although the structural type syntax $\mathsf{type}\ \{\ \overline{D}\ \}$ acts like an intersection of the signatures $\overline{D}$, we need to consider intersection between union types, and including all of the signatures in both arguments may violate the well-formedness relation defined above since the signatures may share identifiers. The intersection combinator performs a 'structural union' by including all of the signatures in either type, but also intersects those signatures that have equivalent identifiers to ensure that method identifiers remain unique in the resulting type. The intersection combinator is defined in Figure 4.2.2. The combinator is defined as a series of overloaded operators, some of which are partial.

Applying the operator between two union types distributes the intersection between the structural types in the two unions to produce a union of the intersection

of every possible pairing under intersection. As with the union combinator, because we treat the types in a union as a set, any duplicates that result from this operation are removed. Taking the intersection of two structural types combines with intersection the sequence of signatures together into a new structural type.

Intersecting a sequence of signatures intersects those signatures that have the same identifier, and includes all of the other signatures unchanged that cannot. The form $\cdot$ is an empty sequence. Two signatures can be intersected if they have the same identifier (the same name and arity). The result of intersecting two signatures is a signature with the same name, parameter types that are the pairwise combination of the parameter types of the inputs under the union combinator $\cup$, and return type that is the intersection of the return types of the two signatures.

We can demonstrate that union and intersection are identities with the bottom and top types, respectively.

Lemma 1 (Union identity with $\bot$)**.**

$$\overline{\mathsf{T} \cup \bot = \mathsf{T}}$$

Proof. Immediate from the definition of $\cup$, since $\bot$ contains no structural types in its union. □

Lemma 2 (Intersection identity with $\top$)**.**

$$\overline{\mathsf{T} \cap \top = \mathsf{T}}$$

Proof. Results in distributing $\mathsf{S} \cap \mathsf{type}\ \{\}$ where S are the structural types in the union of T. $\mathsf{S} \cap \mathsf{type}\ \{\} = \mathsf{S}$ is immediate from the definition of $\cap$, since $\mathsf{type}\ \{\}$ contains no signatures. □

These lemmas will come in handy for proving more in-depth properties of Graceless and its extensions.

4.2.3 Signature Subtraction

The signature subtraction operator $\mathsf{T} - \alpha$ removes any structural type from the union type T that directly contains a signature identified by α. The operator is de-

$$- : \textsc{Type} \times \textsc{Ident} \to \textsc{Type}$$
$$\bigcup \overline{S_i} - a = \bigcup \overline{S_i - a}$$

$$- : \textsc{Struct} \times \textsc{Ident} \rightharpoonup \textsc{Type}$$
$$\mathsf{type}\,\{\,\overline{D}\,\} - a = \begin{cases} a \in \overline{\mathit{identify}(D)} & \text{undefined} \\ a \notin \overline{\mathit{identify}(D)} & \mathsf{type}\,\{\,\overline{D}\,\} \end{cases}$$

Figure 4.2.3: Signature subtraction

fined in Figure 4.2.3 as an overloaded operation on both union types T and structural types S. On structural types, the operation fails if a appears in the identifiers of the contained signatures, and is otherwise the identity function. On union types, the operation uses signature subtraction on its component structural types to effectively filter out those that contain a: the subtractions on structural types that are undefined are removed from the union, and the remaining types are unchanged. If none of the structural types contain a signature identified by a, then the operation is the identity, and if all of them contain such a signature then the result is the $\bot$ type.

Signature subtraction is used to type the 'else' branch of a match construct: this branch only executes if a signature corresponding to the matching identifier is not in the matched object, so we can remove any variants from a union type that do contain such a signature. Because structural types only describe what *does* appear in an object, and not what *does not*, signature subtraction can often be a more powerful tool than intersection: when we discover that a signature is in a term, then the presence of that signature is the *only* information that we gain, and none of the existing in the type variants are invalidated; but when we discover that a signature is not in a term, then we can gain a lot more information about the type of that term by eliminating whole variants of its type.

As with the type combinators, we show an identity lemma for subtraction.

Lemma 3 (Subtraction identity if a is absent)**.**

$$\frac{T = \bigcup \overline{\mathsf{type}\,\{\,\overline{D}\,\}} \qquad \overline{a \notin \overline{\mathit{identify}(D)}}}{T - a = T}$$

$$\boxed{T <: T} \quad \text{(S-Uni)} \quad \frac{\forall S_i.\ \exists S_j.\ S_i <: S_j}{\bigcup \overline{S_i} <: \bigcup \overline{S_j}}$$

$$\boxed{S <: S} \quad \text{(S-Str)} \quad \frac{\forall D_j.\ \exists D_i.\ D_i <: D_j}{\mathsf{type}\ \{\ \overline{D_i}\ \} <: \mathsf{type}\ \{\ \overline{D_j}\ \}}$$

$$\boxed{D <: D} \quad \text{(S-Sig)} \quad \frac{\overline{T_{i2} <: T_{i1}} \qquad T_1 <: T_2}{m(\overline{x_i : T_{i1}}) \to T_1 <: m(\overline{x_i : T_{i2}}) \to T_2}$$

Figure 4.2.4: Subtyping judgment

Proof. Immediate from the definition of subtraction. □

4.2.4 Subtyping

The subtyping relation $T_1 <: T_2$ holds if the sub-type T_1 requires at least the structure described by the super-type T_2, so anywhere a value of type T_2 is expected, a value of type T_1 will suffice. The relation is defined in Figure 4.2.4, also as a series of overloaded judgements on each part of the type syntax hierarchy.

Rule S-Uni requires that, for every structural type S_i in the union of the sub-type, there is a structural type S_j in the union of the super-type such that $S_i <: S_j$. A single structural type in the super-type may satisfy any number of structural types in the sub-type. The rule gives us the desired behaviour for our definition that $\bot = (\bigcup)$ because $\bot <: T$ for any type T, since there are no structural types in the sub-type and the rule becomes immediate.

Rule S-Str enforces the role of a structural type as an intersection of signatures, as this rule is effectively the dual of Rule S-Uni: for every signature D_j in the *super-type*, there is a signature D_i in the *sub-type* such that $D_i <: D_j$. Again, this justifies the definition that $\top = \bigcup \mathsf{type}\ \{\}$ because $T <: \top$ for any T, since either T is $\bot$, or for every structural type S in T then $S <: \mathsf{type}\ \{\}$ as there are no signatures in $\mathsf{type}\ \{\}$ that must be satisfied.

The simplicity of these two rules makes a good justification for the rigidity of our type syntax hierarchy, where we might otherwise have included the combinators $\cup$ and $\cap$ directly in the syntax, and provided subtyping rules to reason about

them. Rather than have three rules each for these combinators, as well as separate rules for reasoning about structural types, we have offloaded any complication into the definition of the $\cap$ function and have only a single rule each for unions and structural types to consider when reasoning about subtyping.

Rule S-SIG defines the subtyping relation between signatures, which determines if two signatures are compatible: $D_1 <: D_2$ holds if the signature D_1 subsumes the information in D_2. For signatures, this means that their identifiers are the same (equivalent name and arity), with standard contravariant subtyping of the parameter type annotations (the parameter types in the super-signature are each a subtype of the corresponding parameter type in the sub-signature) and covariant subtyping of the return type annotations (the return type in the sub-signature is a subtype of the return type in the super-signature).

As with well-formedness, the recursive use of subtyping in Rule S-SIG is defined coinductively, to match the coinductive definition of the type syntax. A derivation of the subtyping relation can be infinite in the same way as the type syntax is permitted to be infinite, repeating with a regular form at an application of Rule S-SIG. As a result, we do not require an assumption set to reason about infinitely-sized recursive types, and in fact we *cannot* do this because we cannot detect the points of repetition in a coinductively-defined recursive type in finite time. As discussed in §4.1.2, the downside is that the subtyping we present here is technically not decidable, but this is easily recoverable in a practical implementation that uses assumptions between trivially equatable objects to simulate finding each point of repetition in the derivation.

We now present a few simple lemmas about the properties of the subtyping judgment.

Lemma 4 (Subtyping is reflexive)**.**

$$\frac{\vdash T}{T <: T}$$

Proof. Trivial coinduction over the proof that $\vdash T$. □

Lemma 5 (Subtyping is transitive)**.**

$$\frac{\vdash T_1, T_2, T_3 \qquad T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3}$$

Proof. Simple mutual coinduction over the proofs that $T_1 <: T_2$ and $T_2 <: T_3$. □

We can also formally prove the properties of $\top$ and $\bot$ discussed above.

Lemma 6 ($\bot$ is the bottom of the subtyping lattice)**.**

$$\frac{}{\bot <: T}$$

Proof. $\overline{S_i}$ is empty, so Rule S-Uni is immediate. □

Lemma 7 (type {} is the top of the structural subtyping lattice)**.**

$$\frac{}{S <: \text{type } \{\}}$$

Proof. $\overline{S_j}$ is empty, so Rule S-Str is immediate. □

Lemma 8 ($\top$ is the top of the subtyping lattice)**.**

$$\frac{}{T <: \top}$$

Proof. For every S in the union of T, $S <:$ type {} by Lemma 7, so Rule S-Uni applies. □

The bottom type $\bot$ is also uniquely situated at the bottom of the lattice.

Lemma 9 ($\bot$ is the unique bottom type)**.**

$$\frac{T <: \bot}{T = \bot}$$

Proof. By Rule S-Uni, for every structural type in T, there must be a corresponding structural type in $\bot$. Since $\bot$ contains no structural types, the rule can only hold if T also contains no structural types, and so is also $\bot$. □

A similar lemma that the type $\top$ is the unique top type does not hold, since type {} in a union with any other structural types is also a super-type of every other super-type, including $\top$.

We also have a lemma that states that if a structural type has a single signature, then any other structural type that is a sub-type of the first must have a compatible signature.

Lemma 10 (Structural subtype implies declaration subtype)**.**

$$\frac{\mathsf{type}\ \{\ \overline{\mathsf{D}_i}\ \} <: \mathsf{type}\ \{\ \mathsf{D}\ \}}{\exists \mathsf{D}_i.\ \mathsf{D}_i <: \mathsf{D}}$$

Proof. Simple case analysis of the proof that $\mathsf{type}\ \{\ \overline{\mathsf{D}_i}\ \} <: \mathsf{type}\ \{\ \mathsf{D}\ \}$. □

Moreover, such a proof guarantees that the identifiers of the two methods are the same.

Lemma 11 (Declaration subtype implies identifier equality)**.**

$$\frac{\mathsf{D}_1 <: \mathsf{D}_2}{\mathit{identify}(\mathsf{D}_1) = \mathit{identify}(\mathsf{D}_2)}$$

Proof. Immediate from Rule S-Sig. □

It is also important that union and intersection behave correctly with respect to subtyping: union always produces a super-type of both of its inputs, and intersection always produces a sub-type.

Lemma 12 (Union produces super-type)**.**

$$\frac{\mathsf{T}_1 \cup \mathsf{T}_2 = \mathsf{T}_3}{\mathsf{T}_1 <: \mathsf{T}_3 \qquad \mathsf{T}_2 <: \mathsf{T}_3}$$

Proof. Every structural type in the unions of T_1 and T_2 appears in the union of T_3, so Rule S-Uni can satisfy both goals through Lemma 4 (modified to apply to structural types, which is part of the proof for that lemma). □

Lemma 13 (Intersection produces sub-type)**.**

$$\frac{\mathsf{T}_1 \cap \mathsf{T}_2 = \mathsf{T}_3}{\mathsf{T}_3 <: \mathsf{T}_1 \qquad \mathsf{T}_3 <: \mathsf{T}_2}$$

Proof. Intersection produces a union with every possible pairing of structural types across T_1 and T_2, and Rule S-Uni requires that we demonstrate that every one of these pairings has a corresponding structural super-type in both T_1 and T_2 in order to satisfy both of the goals: for any structural intersection $S_1 \cap S_2$, we can show that both S_1 and S_2 are structural super-types. By Rule S-Str, for every signature in both S_1 and S_2, there is some corresponding signature in the intersection of their sequences of signatures. In the case that the identifier of a signature in one sequence does not appear in the other sequence, this requirement is satisfied by Lemma 4 (modified to apply to signatures, which is part of the proof for that lemma). If two signatures in either sequence have the same identifier, then we need to show that the intersection of those signatures is a sub-type of each signature: if $D_1 \cap D_2$ is defined, then both D_1 and D_2 are signature super-types. This requirement is satisfied by Rule S-Sig, coinductively applying Lemma 12 for the parameter types and this lemma for the return type. □

Subtraction also always produces a sub-type of the minuend.

Lemma 14 (Subtraction produces sub-type)**.**

$$\frac{T_1 - a = T_2}{T_2 <: T_1}$$

Proof. Any structural types the appear in the union of T_2 are unmodified from T_1; the subtraction only removes structural types. Rule S-Uni applies with Lemma 4 (again applied to structural types). □

4.3 Dynamic Semantics

We now turn to the evaluation of Graceless programs, which are pairs of an object store σ and a term t. The dynamic semantics of Graceless programs is presented in Figure 4.3.1. Reduction is split into two relations, $\longrightarrow$ and $\longmapsto$: the former handles computation, and the latter uses the evaluation contexts E and F to perform congruence reductions by either applying $\longrightarrow$ in the hole of a context with Rule E-Cng or terminating the program on an unrescued raise with Rule E-Rse.

$\boxed{\sigma \mid t \longrightarrow \sigma \mid t}$

(E-Seq)
$$\frac{}{\sigma \mid v; t \longrightarrow \sigma \mid t}$$

(E-Obj)
$$\frac{y \text{ fresh} \qquad \overline{a = \mathit{identify}(d)} \qquad \overline{a} \text{ unique}}{\sigma \mid \mathsf{object}\,\{\,\overline{d}\;t\,\} \longrightarrow \sigma(y \mapsto \{\,\overline{[\mathsf{self}./a]d}\,\}) \mid [y/\mathsf{self}]\overline{[y./a]}t; y}$$

(E-Sfe)
$$\frac{}{\sigma \mid v \Uparrow b \longrightarrow \sigma \mid v}$$

(E-Upd)
$$\frac{\sigma(y) = \{\,\overline{d}, \mathsf{method}\ D\,\{\,t\,\}\,\}}{\sigma \mid y \leftarrow \mathsf{method}\ D\,\{\,t'\,\} \longrightarrow \sigma(y \mapsto \{\,\overline{d}, \mathsf{method}\ D\,\{\,t'\,\}\,\}) \mid y}$$

(E-Req)
$$\frac{\mathsf{method}\ m(\overline{x_i : T_i}) \to T\,\{\,t\,\} \in \sigma(y)}{\sigma \mid y.m(\overline{v_i}) \longrightarrow \sigma \mid [y/\mathsf{self}]\overline{[v_i/x_i]}t}$$

(E-Rsc)
$$\frac{}{\sigma \mid F[\Uparrow v] \Uparrow \{\,z \to t\,\} \longrightarrow \sigma \mid [v/z]t}$$

(E-Fst)
$$\frac{\overline{d} = \sigma(v) \qquad a \in \overline{\mathit{identify}(d)}}{\sigma \mid v \ni a\,\{\,z \to t\,\}\,b \longrightarrow \sigma \mid [v/z]t}$$

(E-Snd)
$$\frac{\overline{d} = \sigma(v) \qquad a \notin \overline{\mathit{identify}(d)}}{\sigma \mid v \ni a\,b\,\{\,z \to t\,\} \longrightarrow \sigma \mid [v/z]t}$$

$\boxed{\sigma \mid t \longmapsto \sigma \mid t}$

(E-Cng)
$$\frac{\sigma \mid t \longrightarrow \sigma' \mid t'}{\sigma \mid E[t] \longmapsto \sigma' \mid E[t']}$$

(E-Rse)
$$\frac{}{\sigma \mid G[\Uparrow v] \longmapsto \sigma \mid \Uparrow v}$$

Figure 4.3.1: Graceless reduction rules

In the computation relation, Rule E-Req processes qualified requests by looking up the corresponding method in the receiver reference y, resolving to the method body t with both self and the parameters $\overline{x_i}$ bound to the receiver y and the arguments $\overline{v_i}$ respectively. As discussed in §4.1.3, there is only a computation rule for qualified requests, and unqualified requests are transformed into qualified ones through substitution. Attempting to reduce an unqualified request fails because it indicates that there was no surrounding object that defined such a method.

Rule E-Obj reduces an object in the way described in §4.1.3, replacing the constructor with its own initialisation code (the term t in its body) sequenced to ultimately return the newly constructed reference of the object y, substituting self for y and qualifying all local methods in that body. The syntax $\sigma(y \mapsto \{\, \overline{d} \,\})$ maps y to the set of definitions $\overline{d}$ in the store σ, whether or not the key y already appeared in the store. In this reduction it is guaranteed to be a new key, because of the requirement of y fresh. Object constructors can only be reduced if they do not include duplicate method identifiers, otherwise the resulting object in the store would not have a sensible form.

Rule E-Seq throws away the head of a sequence when it is a value, so that the tail can begin evaluating. Rule E-Upd updates a method definition in an object $\sigma(y)$: the method must already exist in the object, and have exactly the same signature, so this effectively replaces the method body t with the updated t'. The result of an update is the reference y whose object was updated, for lack of a better value to return.

Rules E-Sfe and E-Rsc process rescue forms, the former by removing the rescue if its body did not raise anything, and the latter by handling a raise in the body with the accompanying block by binding the block parameter z to value that was raised v in the block body t. A rescue uses the context F to match any term whose next non-value sub-term to compute is a raise, so we do not need to manually raise the value directly into the body of the rescue before handling it. Because there is no computation rule for a raise, we have a guarantee that the body of the rescue cannot be reduced by any other rule.

Rules E-Fst and E-Snd handle branching on the outcome of a match, on the success or failure of the condition to the left or right block respectively. In either case, the corresponding block parameter z is bound to the matched value y in the

corresponding block body t. The choice is based on whether the definition identifier a appears in the object at y in the store: one or the other always applies.

A reducible Graceless program $\langle \sigma, t \rangle$ is reduced a single step to some program $\langle \sigma', t' \rangle$ with $\sigma \mid t \longmapsto \sigma' \mid t'$. We write $\sigma \mid t \longmapsto^* \sigma' \mid t'$ as the reflexive and transitive closure of reduction for any Graceless program σ and t, and $\sigma \mid t \longmapsto^* \sigma' \mid v$ for a complete execution. Not all programs are guaranteed to reduce to a value, even in the absence of a stuck state, since Graceless evaluation is not total.

4.4 Static Semantics

We now define a term typing judgment for Graceless programs, and prove that the judgment soundly prevents programs from getting stuck when executed by $\longmapsto^*$ (though it does not prevent the production of an uncaught raise). Typing takes place in a type environment Γ, which is a standard sequence of variable bindings $x : T$ (where x includes both self references and store references y). The type environment syntax captures all of the information about the local scope, and it is trivial to look up the type of a variable in any Γ, but we need to define a mechanism for also discovering what *methods* also appear in a type environment.

4.4.1 Signature Selection

The typing environment contains variable bindings $x : T$, not signatures, but signatures can appear in a local scope, referenced by unqualified requests. Selecting a signature from the environment is fairly simple, since signatures defined in any surrounding object are in local scope, and we have self as a unique identifier for each surrounding object. To select a signature from the surrounding scope, we need only search in the type of each self in the scope Γ, and select the first one that we find. Selecting either a variable binding or a signature from the typing environment is complicated by the fact that they can shadow one another.

The environment signature selection judgements $\Gamma \ni D$, selecting the signature D from the scope Γ, and $\Gamma \ni x : T$, selecting a variable binding $x : T$ from the scope Γ, are defined in Figure 4.4.1. For the former, rather than define the selection as an auxiliary function from a scope Γ *and* a method identifier a to a selected signature

$\boxed{\Gamma \ni D}$

$$\frac{D \in \overline{D_i}}{\Gamma, \mathsf{self} : \mathsf{type}\ \{\ \overline{D_i}\ \} \ni D}\ (\text{D-Hre}) \qquad \frac{x : T \not\ni \mathit{identify}(D) \qquad \Gamma \ni D}{\Gamma, x : T \ni D}\ (\text{D-Thr})$$

$\boxed{\Gamma \ni x : T}$

$$\frac{}{\Gamma, x : T \ni x : T}\ (\text{B-Hre}) \qquad \frac{x_2 : T_2 \not\ni \langle x_1, 0\rangle \qquad \Gamma \ni x_1 : T_1}{\Gamma, x_2 : T_2 \ni x_1 : T_1}\ (\text{B-Thr})$$

$\boxed{x : T \not\ni a}$

$$\frac{m \neq \mathsf{self} \qquad \langle m, n\rangle \notin \overline{\mathit{identify}(D_i)}}{\mathsf{self} : \mathsf{type}\ \{\ \overline{D_i}\ \} \not\ni \langle m, n\rangle}\ (\text{N-Slf}) \qquad \frac{x \neq \mathsf{self} \qquad a \neq \langle x, 0\rangle}{x : T \not\ni a}\ (\text{N-Var})$$

Figure 4.4.1: Environment signature selection

D, we simplify to a judgment that picks the signature D directly, and will never pick a signature if one with the same identifier appears 'closer' in Γ, using an auxiliary judgment $x : T \not\ni a$ that indicates no signature with the identifier a could be selected from the binding $x : T$. The variable binding selection works in a similar way.

Rule D-Hre selects the signature from the type of a self reference (which we can assume is a single structural type). Rule D-Thr skips over a binding in the environment if no signature with the same identifier appears in the binding. Similarly, Rule B-Hre immediately selects an equivalent binding, and Rule B-Thr skips over a binding if it does not match the selected variable name. The use of $x_2 : T_2 \not\ni \langle x_1, 0\rangle$ is not syntactically correct if x_1 is a concrete variable y: in this case we treat it as the simpler case of $x_1 \neq x_2$. Rule N-Slf confirms that no such identifier appears in the binding if the variable name is self, and Rule N-Var succeeds if the variable name is not self — so we do not care about its signatures — and that the variable binding itself does not satisfy the identifier.

Note that for an unqualified zero-argument request z, the type of the name z can only be selected as a signature or a variable binding, not both, since if both appear in the typing environment then the closer of the two will shadow the other one. For instance, in the environment $\mathsf{self} : \mathsf{type}\ \{\ z \longrightarrow T_1\ \}, z : T_2$, the Rule N-Var

does not permit Rule D-THR to proceed, so selecting the signature $z \to T_1$ cannot succeed, only the selection of the binding $z : T_2$ is allowed. Only one rule will be relevant in the typing judgment for typing unqualified zero-argument requests — either as a variable or as a real request — when the syntax of the term is otherwise ambiguous.

4.4.2 Term Typing

The term typing judgment takes the form $\Gamma \vdash t : T$, meaning that in the type environment Γ the term t has type T, and is defined in Figure 4.4.2. An accompanying judgment for definitions $\Gamma \vdash d : D$ assigns d the signature D. A program $\langle \sigma, t \rangle$ is typed by first determining an initial type environment with the judgment $\vdash \sigma : \Gamma$ — which ensures that the store σ is well-typed and binds each reference to the exact type of its corresponding object in the resulting environment Γ — and then typing the term in that environment.

Rule T-VAR types a variable (either concrete or abstract) with the type that it is assigned in the environment Γ. Rule T-SUB applies subsumption to the typing judgment, so we can choose to assign a super-type to a term instead of the exact type assigned by the relevant rule for the term's syntax, so long as the super-type is well-formed. Rule T-RSE assigns the $\bot$ type to a raise of any term t so long as t itself is well-typed. Combined with the subsumption rule this allows a raise to have any type, since it will never reduce to a value.

Rule T-SEQ ensures that both sides of a sequence are well-typed, and assigns the type of the second half of the sequence to the whole sequence, since the first half is thrown away once it is reduced to a value. Rule T-OBJ types an object by adding the type of the object as self into the typing environment and typing both the definitions and the term in the object in this extended context. The rule also ensures that the identifiers of the definitions are unique, to ensure that the resulting type is well-formed under the rules given in Figure 4.2.1. Note that the signatures $\overline{D}$ appear to be both inputs (to the type environment) and outputs (as the result of typing the definitions $\overline{d}$ and in the type of the overall object), but the signatures can be trivially calculated from the syntax of the definitions $\overline{d}$ before attempting to apply any judgment.

$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \Uparrow t : \bot}\ (\text{T-Rse}) \qquad \frac{\Gamma \vdash t : T_1 \qquad T_1 <: T_2 \qquad \vdash T_2}{\Gamma \vdash t : T_2}\ (\text{T-Sub})$$

$$\frac{\Gamma \vdash t_1 : T_1, t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2}\ (\text{T-Seq}) \qquad \frac{\Gamma, \text{self} : \text{type}\ \{\ \overline{D}\ \} \vdash \overline{d : D}, t : T \qquad \overline{\mathit{identify}(D)}\ \text{unique}}{\Gamma \vdash \text{object}\ \{\ \overline{d}\ t\ \} : \text{type}\ \{\ \overline{D}\ \}}\ (\text{T-Obj})$$

$$\frac{\Gamma \ni m(\overline{x_i : T_i}) \to T \qquad \Gamma \vdash \overline{t_i : T_i}}{\Gamma \vdash m(\overline{t_i}) : T}\ (\text{T-R/U}) \qquad \frac{\Gamma \vdash t : \text{type}\ \{\ m(\overline{x_i : T_i}) \to T\ \} \qquad \Gamma \vdash \overline{t_i : T_i}}{\Gamma \vdash t.m(\overline{t_i}) : T}\ (\text{T-R/Q})$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, z : \top \vdash t_2 : T_2}{\Gamma \vdash t_1 \Uparrow \{\ z \to t_2\ \} : T_1 \cup T_2}\ (\text{T-Rsc}) \qquad \frac{w : \text{type}\ \{\ \overline{D_i}\ \} \in \Gamma \qquad \Gamma \vdash d : D \qquad D \in \overline{D_i}}{\Gamma \vdash w \leftarrow d : \text{type}\ \{\ \overline{D_i}\ \}}\ (\text{T-Upd})$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, z_1 : T_1 \cap \mathit{ground}(a) \vdash t_2 : T_2 \qquad \Gamma, z_2 : T_1 - a \vdash t_3 : T_3}{\Gamma \vdash t_1 \ni a\ \{\ z_1 \to t_2\ \}\{\ z_2 \to t_3\ \} : T_2 \cup T_3}\ (\text{T-Mch})$$

$\boxed{\Gamma \vdash d : D}$

$$\frac{\vdash \overline{T_i}, T \qquad \Gamma, \overline{x_i \to T_i} \vdash t : T}{\Gamma \vdash \text{method}\ m(\overline{x_i : T_i}) \to T\ \{\ t\ \} : m(\overline{x_i : T_i}) \to T}\ (\text{T-Sig})$$

$\boxed{\vdash \sigma : \Gamma}$

$$\frac{\overline{y_i : \text{type}\ \{\ \overline{D_{ij}}\ \}} \qquad \overline{\text{self} : \text{type}\ \{\ \overline{D_{ij}}\ \} \vdash \overline{d_{ij} : D_{ij}}} \qquad \overline{\overline{\mathit{identify}(D_{ij})}\ \text{unique}}}{\vdash \overline{\{\ y_i \mapsto \{\ \overline{d_{ij}}\ \}\ \} : (y_i : \text{type}\ \{\ \overline{D_{ij}}\ \})}}\ (\text{T-Sto})$$

$\mathit{ground} : \text{Ident} \to \text{Type}$

$\mathit{ground}(\langle m, n \rangle) = \text{type}\ \{\ m(\overline{x_i : \bot}^{\,i \le n}) \to \top\ \}$

Figure 4.4.2: Typing judgements

Rule T-R/U types an unqualified request, selecting the relevant signature from the environment and ensuring that the arguments satisfy the parameter types. Rule T-R/Q types a qualified request by typing the receiver as a structural type containing the necessary signature. If the value of the receiver contains more than this signature, then the necessary type assignment for the receiver can be achieved using the subsumption rule; this also means that the parameter types $\overline{T_i}$ and return type T may be less specific than the annotations on the actual signature of the value of the receiver. If the arguments to the qualified request can be typed with the parameter types of the signature in the receiver, then the request is assigned the type of the return type of the signature.

Rule T-Rsc types a rescue by ensuring that the body of the rescue is well-typed, and that the body of the block is well-typed with the parameter in the type environment. Since any value can be raised, the type of the parameter in the environment is $\top$. Evaluating a rescue produces either the value of the body or the value of applying the block to a raised value, and the resulting type of a raise reflects these two possibilities by taking the union of each type.

Rule T-Upd types a method update, looking up the type of the receiver w in the environment Γ to ensure that the signature of the method that is the subject of the update matches exactly the one that is replacing it. The property is enforced by typing d with the signature D (there is no subsumption rule for the judgment $\Gamma \vdash d : D$, so D is guaranteed to be the exact signature of d), and then requiring that D appears directly in the structural type of w. The type of the update itself is the type of w, since w is the result of evaluating an update.

Rule T-Mch types the match construct, first by typing the matched object, and then typing the bodies of the two branches with the block parameters bound to to a slight variation of the type of the matched object for each of the branches. In the 'then' branch, we know that the matched object contains a definition identified by a, but we don't know the type annotations on the corresponding signature, so we intersect the type of the matched object with the *ground*-type of a. For any identifier $\langle m, n \rangle$, the ground-type has a corresponding signature (i.e. the name m and arity n) where the parameter types are the $\bot$ type and the return type is the $\top$ type. These type annotations reflect the contravariant and covariant nature nature of parameter and return types, respectively: we do not know what the method

expects, and nor do know what the method will return. Combined with an existing type under intersection, these type annotations may become more detailed.

As mentioned in §4.2.3, the addition of information to a structural type through intersection as in the 'then' branch of a match construct is less useful than it may first appear, because we gain only the information that was added, and no more. Consider the *List* type from §4.1.2 again:

$$\mathit{List} = \bigcup \mathsf{type}\ \{\},\ \mathsf{type}\ \{\ \mathsf{head} \rightarrow \mathsf{T},\ \mathsf{tail} \rightarrow \mathit{List}\ \}$$

It seems reasonable that we might use the match construct to determine whether an object y of type *List* is a node or the end of the list by testing if it has a head method.

$$\mathtt{y} \ni \langle \mathsf{head}, 0 \rangle\ \{\ \mathtt{z} \rightarrow \mathtt{t}\ \}\ \mathtt{b}$$

The type of z in the environment when typing t is *not* that of a node, since the result of applying the intersection specified by Rule T-Mch is:

$$\mathit{List} \cap \mathsf{type}\ \{\ \mathsf{head} \rightarrow \top\ \} = \bigcup \mathsf{type}\ \{\ \mathsf{head} \rightarrow \top\ \},\ \mathsf{type}\ \{\ \mathsf{head} \rightarrow \mathsf{T},\ \mathsf{tail} \rightarrow \mathit{List}\ \}$$

It is now well-typed to request head with z as the receiver, but the resulting return type is not T: type {} must appear in the return type's union. It is still not well-typed to request tail on z since it does not appear in all of the variants of the union.

The 'else' branch of a match gives the block parameter a type in the block's body by subtracting the identifier a from the type of the matched object using the signature subtraction defined in §4.2.3: since we know that a signature identified by a does not appear in the value of the matched term, we can remove all possibilities from the union type of the term that claim it would appear.

For the example above, the signature subtraction also does not win us much:

$$\mathtt{y} \ni \langle \mathsf{head}, 0 \rangle\ \mathtt{b}\ \{\ \mathtt{z} \rightarrow \mathtt{t}\ \}$$

The type of z in the environment when typing t is type {}, since we have removed the possibility of it being a node and are left with the remaining structural type in the union. Of course, we cannot do much with z as a result. If a structural type

S_1 in a union is a super-type of another structural type S_2 in the same union, then S_2 is effectively useless: in this example S_1 was empty, but even if it had signatures that also appeared in S_2 then neither intersecting the total type with a ground-type nor subtracting S_2 from the union gains any new information about the type of the matched object.

The matching construct is not useless under typing: given a type with variants in a union that do not subsume each other, we gain useful type information in the 'else' branch of a match. Assume the following judgment holds for some y in a typing environment Γ:

$$\Gamma \vdash y : \bigcup \mathsf{type}\,\{\, m_1 \to T_1 \,\}, \mathsf{type}\,\{\, m_2 \to T_2 \,\}$$

Neither of the requests $y.m_1$ or $y.m_2$ are well-typed: the only common structural super-type of both types in the union is $\mathsf{type}\,\{\}$, so we cannot provide the necessary typing required by Rule T-R/Q. Consider a test to see if a method m_1 appears in y:

$$y \ni \langle m_1, 0 \rangle \,\{\, z \to t_1 \,\}\,\{\, z \to t_2 \,\}$$

We gain information about y in both branches. As with the *List* type, in t_1 we know that m_1 is present in z, but the type of a request for it will be effectively equivalent to $\top$ because a method m_1 might appear in the variant that already contains m_2, and so we have no information about what the actual method returns.

For typing t_2, though, z has the type $\mathsf{type}\,\{\, m_2 \to T_2 \,\}$, since we can entirely eliminate the first variant in the union. The knowledge that m_1 is *not* present in the object guarantees that the first structural type in the union cannot apply to the object, which leaves only the remaining structural type.

The result is that, in order to fully discriminate on the variants in the union type of an object like y in the example above, we must account for the possibility that the object might contain any combination of the methods specified across the structural types, so long as it contains at least all of the methods in one of the variants. This can be achieved by nesting match constructs:

$$y \ni \langle m_1, 0 \rangle \,\{\, z \to z \ni \langle m_2, 0 \rangle \,\{\, z \to t_1 \,\}\,\{\, z \to t_2 \,\}\,\}\,\{\, z \to t_3 \,\}$$

When typing t_2 and t_3 we know the type of z is $\mathsf{type}\{ m_1 \to T_1 \}$ and $\mathsf{type}\{ m_2 \to T_2 \}$, respectively. The branch of t_1 corresponds to the case where y contains *both* m_1 and m_2, where z has the type:

$$\bigcup \mathsf{type}\{ m_1 \to T_1, m_2 \to \top \}, \mathsf{type}\{ m_1 \to \top, m_2 \to T_2 \}$$

As far as requesting a method on z is concerned, this type is equivalent to:

$$\mathsf{type}\{ m_1 \to \top, m_2 \to \top \}$$

Both m_1 and m_2 can be requested on z, but the result of either must be typed as $\top$. Since y might contain both of these methods, this case must be accounted for, though it can be trivially implemented by raising an error.

The judgment $\Gamma \vdash d : D$ types a definition with a signature, and by Rule T-Sig a method is always typed with exactly its own signature. The body of the method must satisfy the return type of the signature with the typing environment extended with the parameters and their type annotations.

Rule T-Sto generates an initial typing environment for typing a program. As with Rule T-Obj, the signatures of the resulting types are immediately generated from the signatures on the definitions in the store, and are used as the typing environment to check that each of the methods are well-typed, with self bound to the total type of each object. The resulting typing environment is the same one used to type the definitions, without the self binding.

4.4.3 Properties

We now prove that the type system of Graceless is sound with a standard proof of progress and preservation. When a Graceless program can no longer be reduced, the term is either a value or an unrescued raise. The proof proceeds through a number of lemmas, and proof summaries are presented below. First, we show that the bottom type is empty.

Lemma 15 (Emptiness of $\bot$)**.** *If* $\vdash \sigma : \Gamma$*, then* $\Gamma \vdash v : \bot$ *cannot hold.*

Proof. Assume that $\Gamma \vdash v : \bot$ does hold, case analysis on the proof.

(T-Var) This rule cannot apply, because the judgement $\vdash \sigma : \Gamma$ cannot assign $\bot$ to a variable.

(T-Sub) This rule must select a type T_1 such that $T_1 <: \bot$. By Lemma 9, $T_1 = \bot$, so the rule also contains a proof that $\Gamma \vdash \nu : \bot$.

The remaining rules do not apply to $\bot$. Since Rule T-Sub is defined inductively and is the only applicable rule, the fact that it must repeat infinitely forms a contradiction with the assumption that such a proof could exist. □

We also require a number of lemmas about the type of a store. Typing a store σ produces a type environment Γ. The variables bound in Γ should be exactly the domain of σ, which means each variable must be a concrete reference y.

Lemma 16 (Store typing domain equality)**.**

$$\frac{\vdash \sigma : \overline{(x : T)}}{\overline{x} = \mathit{dom}(\sigma)}$$

Proof. Immediate from the judgment $\vdash \sigma : \Gamma$ by Rule T-Sto. □

The type of a store always contains structural types as well.

Lemma 17 (Store typing is structural)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \ni y : T}{T = \mathsf{type}\ \{\ \overline{D}\ \}}$$

Proof. Immediate from the judgment $\vdash \sigma : \Gamma$ by Rule T-Sto. □

Furthermore, the type of a store includes the *exact* information about the signatures of the objects therein.

Lemma 18 (Environment lookup corresponds to store)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \ni y : \mathsf{type}\ \{\ \overline{D}\ \}}{\overline{d} = \sigma(y) \qquad \overline{\mathit{signature}(d) = D}}$$

Proof. By Rule T-Sto, the signatures assigned to the store are computed by Rule T-Sig, which assigns the exact signature of the definition, so any signature stored in the type of a reference y corresponds exactly to the definition found at y in σ. □

The subsumption rule in term typing is used to 'select' a signature from the type of a term, by using a structural type containing only the relevant signature. The following lemma demonstrates that if in a store σ, a reference y has such a type type { D }, then there is guaranteed to be a corresponding definition at $\sigma(y)$ with a signature that is compatible with D.

Lemma 19 (Canonicity of forms)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : \mathsf{type}\ \{\ D\ \}}{d \in \sigma(y) \qquad \mathit{signature}(d) <: D}$$

Proof. By analysis of the proof that $\Gamma \vdash y : \mathsf{type}\ \{\ D\ \}$, proceeding through an arbitrary number of applications of Rule T-Sub, terminating in Rule T-Var where $\Gamma \vdash y : \mathsf{type}\ \{\ \overline{D_i}\ \}$. By Lemma 18, $\overline{D_i}$ correspond exactly to the signatures of the definitions at $\sigma(y)$. The intervening subtyping premises of Rule T-Sub combine under Lemma 5 to form a proof that $\mathsf{type}\ \{\ \overline{D_i}\ \} <: \mathsf{type}\ \{\ D\ \}$, so it follows by Lemma 10 that one of the signatures D_i is compatible with D. □

Now we can prove the Progress lemma.

Lemma 20 (Typing implies progress)**.** *For any program* $\langle \sigma, t \rangle$*, if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : T$ *then either:*

- $\exists v.\ t = v$
- $\exists v.\ t = \Uparrow v$
- $\exists \sigma'\ t'.\ \sigma \mid t \longmapsto \sigma' \mid t'$

Proof. By induction on the derivation of the proof that $\Gamma \vdash t : T$, with a case analysis on the last step.

(T-Var) By Lemma 16, $t = y$, so immediate.

(T-Rse) $t = \Uparrow t'$, so immediate. Induction applies Rule E-Rse when this appears in the hole of a context F.

(T-Sub) Induction on the premise that $\Gamma \vdash t : T'$ for some T'.

(T-Seq) Induction on the premise that the head of the sequence is typed $\Gamma \vdash t_1 : T_1$. If t_1 is a value, then Rule E-Seq.

(T-Obj) Rule E-Obj, the requirement of uniqueness of method identifiers is immediate from the premise of the assumption.

(T-R/U) self $\notin dom(\sigma)$, so by Lemma 16 the premise $\Gamma \ni D$ forms a contradiction.

(T-R/Q) Induction on the typing of the subterms. If all subterms are values, then Rule E-Req, with the premise satisfied by Lemma 19 and Lemma 11.

(T-Rsc) Induction on the typing of the body. If the body is a value, then Rule E-Sfe. If the body contains a raise in the hole of a context F (the only location where this is not handled immediately by Rule E-Rse), then Rule E-Rsc.

(T-Upd) By Lemma 16, $w = y$. By Lemma 18, the signature of the definition d appears exactly in the store at y, so Rule E-Upd.

(T-Mch) By induction on the typing of the body. If the body is a reference y, then y appears in the domain of Γ by Lemma 16, so either Rule E-Fst or E-Snd apply.

This covers all cases. □

To begin proving the preservation lemma, we first need to show that substitution preserves typing, for both forms of substitution.

Lemma 21 (Value substitution preserves typing)**.**

$$\frac{\Gamma \vdash v : T_1 \qquad \Gamma_1, z : T_1, \Gamma_2 \vdash t : T_2 \qquad \Gamma_2 \not\ni \langle z, 0 \rangle}{\Gamma_1, \Gamma_2 \vdash [v/z]t : T_2}$$

Proof. By induction on the derivation of the proof that $\Gamma_1, z : T_1, \Gamma_2 \vdash t : T_2$, with a case analysis on the last step.

(T-Var) If $t = v$ then $T_1 = T_2$, so immediate from the proof that $\Gamma \vdash v : T_1$. Otherwise the variable has not been substituted and the proof remains unchanged.

The remaining rules all follow immediately from induction and applications of weakening; any binding that shadows z terminates the substitution. □

Lemma 22 (Qualifying substitution preserves typing)**.**

$$\frac{\Gamma \vdash t : T \qquad \Gamma \ni \mathsf{self} : \mathsf{type}\ \{\ \overline{D}\ \} \qquad a \in \overline{\mathit{identify}(D)}}{\Gamma \vdash [\mathsf{self}./a]t : T}$$

Proof. By mutual induction with Lemma 23 on the derivation of the proof that $\Gamma \vdash t : T$, with a case analysis on the last step.

(T-R/U) If $t = m(\overline{t_i})$ where $a = \langle m, |\overline{v}| \rangle$, then the corresponding D such that $a = \mathit{identify}(D)$ is a signature $m(\overline{x : T_i}) \to T$; the result of the substitution is $\mathsf{self}.m(\overline{t_i})$, which can be typed by Rule T-R/Q since self is still in the typing environment and contains the same signature that was the result of the lookup before, along with the existing proofs that $\overline{t_i}$ are typed T_i. Otherwise the request has not been qualified and the proof remains unchanged.

The remaining rules all follow immediately from induction and applications of weakening. □

Lemma 23 (Qualifying substitution preserves signatures)**.**

$$\frac{\Gamma \vdash d : D \qquad \Gamma \ni \mathsf{self} : \mathsf{type}\ \{\ \overline{D}\ \} \qquad a \in \overline{\mathit{identify}(D)}}{\Gamma \vdash [\mathsf{self}./a]d : D}$$

Proof. By mutual induction with Lemma 22 on the derivation of the proof that $\Gamma \vdash d : D$, with a case analysis on the last step.

(T-Sig) Rebuild the proof under Rule T-Sig using Lemma 22 and weakening on the typing of the body.

This covers all cases. □

In order to show that the result of a successful match preserves the type of the overall match, we need to first show that a well-typed store reference can have the ground-type of a method identifier added to its type if a method with that identifier actually appears in the store at the location of the reference.

Lemma 24 (Adding ground preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \overline{d} = \sigma(y) \qquad a \in \overline{\mathit{identify}(d)}}{\Gamma \vdash y : T \cap \mathit{ground}(a)}$$

Proof. By induction on the derivation of the proof that $\Gamma \vdash y : T$, with a case analysis on the last step.

(T-Var) T is a structural type type $\{ \overline{D_i} \}$ by Lemma 17. Since $a \in \overline{\mathit{identify}(d)}$, by Lemma 18 there is a D_i such that $\mathit{identify}(D_i) = a$. Intersecting the ground type of a signature with a structural type that contains that signature is an identity on the structural type, by Lemmas 2 and 1.

(T-Sub) By Lemma 13 the result of the intersection is a subtype of the original type, so one more application of Rule T-Sub.

The remaining cases do not apply to y. □

For the failure of a match, we will need to show the opposite; that a well-typed store reference can have a method identifier subtracted from its type if no such method with that identifier appears in the store at the location of the reference.

Lemma 25 (Subtraction preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \overline{d} = \sigma(y) \qquad a \notin \overline{\mathit{identify}(d)}}{\Gamma \vdash y : T - a}$$

Proof. By induction on the derivation of the proof that $\Gamma \vdash y : T$, with a case analysis on the last step.

(T-Var) T is a structural type type $\{ \overline{D_i} \}$ by Lemma 17. Since $a \notin \overline{\mathit{identify}(d)}$, by Lemma 18 there is no D_i such that $\mathit{identify}(D_i) = a$. By Lemma 3, $T - a = T$.

(T-Sub) By Lemma 14 the result of the subtraction is a subtype of the original type, so one more application of Rule T-Sub.

The remaining cases do not apply to y. □

With these in hand, we can prove the preservation lemma.

Lemma 26 (Reduction preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash t : T \qquad \sigma \mid t \longmapsto \sigma' \mid t'}{\vdash \sigma' : \Gamma' \qquad \Gamma' \vdash t' : T}$$

Proof. Induction on the derivation of the proof that $\sigma \mid t \longmapsto \sigma' \mid t'$, with a case analysis on the last step.

(E-Seq) Rule T-Seq.

(E-Obj) It follows from the inversion of Rule T-Obj (and Rule T-Sub) that the definitions $\overline{d}$ in the object constructor are typed $\Gamma, \mathsf{self} : T \vdash d : D$, and the object body t is typed $\Gamma, \mathsf{self} : T \vdash t : T'$. By Lemma 23, the definitions $\overline{[\mathsf{self}./a]d}$ in the store at the newly allocated reference y can be typed with the same signatures $\overline{D}$, so the reference can be typed $\Gamma \vdash y : T$ by Rule T-Var. By Lemmas 21 and 22, the resulting head of the sequence $[y/\mathsf{self}]\overline{[\mathsf{self}./a]}t$ can be typed $\Gamma \vdash [y/\mathsf{self}]\overline{[\mathsf{self}./a]}t : T'$, so the total resulting sequence has the type T by Rule T-Seq.

(E-Req) In the result $[y/\mathsf{self}]\overline{[v/z]}t$, the term t has come from the body of a definition with signature D in the store at the reference y, so Lemma 19 applied to the proof that y has the type $\mathsf{type}\ \{\ D\ \}$ means that, through inversion of Rule T-Var (where y has the type T_1) on the typing of y and Rule T-Sto on the proof that $\vdash \sigma : \Gamma$ (plus the guarantee that the identifiers of the definitions in an object in the store are unique), it must be the case that $\Gamma, \mathsf{self} : T_1, \overline{z : T_i'} \vdash t : T'$ where $T_i <: T_i'$ and $T' <: T$. Inversion of Rule T-R/Q (and Rule T-Sub) proves that $\Gamma \vdash \overline{v : T_i}$. Lemma 21 moves from the typing environment to substitution to prove that $\Gamma \vdash [y/\mathsf{self}]\overline{[v/z]}t : T'$, and Rule T-Sub applies to the proof that $T' <: T$ to prove that the result is typed T.

(E-Rse) Rule T-Sub applied to Rule T-Rse. The necessary subtyping is immediately satisfied by Lemma 6.

(E-Sfe) It follows from the inversion of Rule T-Rsc (and Rule T-Sub) that the resulting value v is typed $\Gamma \vdash v : T_1$, where the type of the rescue was $T_1 \cup T_2$. Rule T-Sub and $T_1 <: T_1 \cup T_2$ (by Lemma 12) combine to prove that $\Gamma \vdash v : T_1 \cup T_2$.

(E-Rsc) It follows from the inversion of Rule T-Rsc (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : \top \vdash t : T_2$, where the type of the rescue was $T_1 \cup T_2$. Lemma 21 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_2$ by applying Rule T-Sub to the proof that $\Gamma \vdash v : T_1$ and the trivial subtyping $T_1 <: \top$. Rule T-Sub and $T_2 <: T_1 \cup T_2$ (by Lemma 12) combine to prove that $\Gamma \vdash [v/z]t : T_1 \cup T_2$.

(E-Upd) Immediate from the inversion of Rule T-Upd (and Rule T-Sub) that $\Gamma \vdash w : T$.

(E-Fst) It follows from the inversion of Rule T-Mch (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : \mathit{ground}(a) \vdash t : T_2$, where the type of the match was $T_2 \cup T_3$. Lemma 21 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_2$ by applying Lemma 24 to the proof that $\Gamma \vdash v : T_1$. Rule T-Sub and $T_2 <: T_2 \cup T_3$ (by Lemma 12) combine to prove that $\Gamma \vdash [v/z]t : T_2 \cup T_3$.

(E-Snd) It follows from the inversion of Rule T-Mch (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : T_1 - a \vdash t : T_3$, where the type of the match was $T_2 \cup T_3$. Lemma 21 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_3$ by applying Lemma 25 to the proof that $\Gamma \vdash v : T_1$. Rule T-Sub and $T_3 <: T_2 \cup T_3$ (by Lemma 12) combine to prove that $\Gamma \vdash [v/z]t : T_2 \cup T_3$.

This covers all cases. □

The progress and preservation lemmas combine to give us type soundness.

Theorem 27 (Well-typed programs don't get stuck)**.** *For any Graceless program* $\langle \sigma, t \rangle$, *if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : T$, *then either:*

- $\exists v.\ t = v$, *so* $\Gamma \vdash v : T$

- $\exists v.\ \mathrm{t} = \Uparrow v$, *so* $\Gamma \vdash \Uparrow v : \mathsf{T}$
- $\exists \sigma'\ \Gamma'\ \mathrm{t}'.\ \sigma \mid \mathrm{t} \longmapsto \sigma' \mid \mathrm{t}'$, *with* $\vdash \sigma' : \Gamma'$ *and* $\Gamma' \vdash \mathrm{t}' : \mathsf{T}$

Proof. Immediate from Lemmas 20 and 26. □

5 Casts

The Graceless language we have presented so far is sufficient for expressing much of the semantics of Grace programs, but its static type system does not take into account the fact that Grace is not inherently typed, and that it is legitimate to run *parts* of a Grace program without type-checking enabled. Graceless cannot express the safe interaction of typed and untyped components, as a typed program can only minimally interact with objects it has no type information about: the type checker will not permit the interaction without being sure that it is safe.

We now extend Graceless with structural casts, expressed through the existing language of types and method reflection to describe assumptions about the type of an object behind a cast. The type system can then permit access to methods that are assumed to exist using a *coercion*, which ensures that a failed assumption safely results in a raise instead of the evaluation getting stuck. Higher-order *casts* then chaperone objects to maintain the remaining unchecked assumptions about the methods, expanding to a coercion when it becomes possible to check an assumption.

The necessity of interaction between typed and untyped components of the language is already apparent in the typing of the match construct. Because a match can only determine if an object contains a method with the right identifier, and does not consider the annotations on any such method, the type of any signature added by the *ground* function (from Figure 4.4.2) is difficult to use in the 'then' branch of a match. If the method requires any parameters then it cannot be requested, because all of its arguments must accept values from the uninhabited type $\bot$. Even if the method does not accept arguments, the return type of the method is $\top$, making it difficult to use the outcome except through the use of more matches.

Casts allow the 'then' branch of a match to check for the presence of a method,

then make assumptions about the inputs and outputs of that method. The match form in Grace proper behaves in the same way, and this is also the behaviour observed by Boyland (2014) in their analysis of Minigrace. Minigrace has never performed higher-order assumptions in a type-safe way.

5.1 Design

The semantics of the existing Graceless language already contains reflection mechanisms for examining the structure of an object; these can be used to implement most of the casting framework required to make assumptions about the type of an object. The language's match form allows the type of an object to be inspected, so the ability to discover whether a method is present is already available. Invalidated assumptions can be encoded using the existing raise form.

We use the term 'coercion' to describe an examination of an object's structure with match expressions in an attempt to give it a more specific type that it already has. Coercing a term t to a type that assumes it has a method $\langle m, 1\rangle$ can already be encoded as the following match expression:

$$t \ni \langle m, 1\rangle \{ z \to z \} \{ z \to \Uparrow z \}$$

If the value of t has the relevant method, then the match will reduce directly to the value unchanged, otherwise the value is raised to indicate that the coercion has failed. Since the type of a raise is $\bot$ and the type of a match is the union of the types of the two branches, Lemma 1 tells us that the type system will assign a type to the coercion that contains a method identified by $\langle m, 1\rangle$.

A match only examines the name and arity of the methods in the object, not the types annotating the parameters and return of the methods. Although the type assigned to the above coercion contains a signature with the identifier $\langle m, 1\rangle$, the match only guarantees that the method is present. The type checker must assign the signature $m(z : \bot) \to \top$ to the method (unless there was already information about this method in the type of t, in which case the coercion is useless because we already knew that the method was present).

Our design extends Graceless with a syntax for casts $t : S$, indicating the term

t is assumed to have the type expressed by the structural type S. A cast encodes the higher-order assumptions about an object's type, chaperoning a reference to the object and applying the relevant coercions when a method is requested that has assumptions that must be upheld.

Assuming that a method m in a term t accepts a Number and returns a Boolean is written:

$$t : \text{type}\ \{\ m(x : \text{Number}) \rightarrow \text{Boolean}\ \}$$

The cast only makes assumptions about the type annotations on the methods, and does *not* assume the existence of the method m. The term t must be known to contain a method $\langle m, 1 \rangle$ for the cast to be well-typed.

By combining the coercion of the existing match form and the new casts, any structural assumption can be applied to a term. The combination of the coercion and cast above is:

$$t \ni \langle m, 1 \rangle\ \{\ z \rightarrow z : \text{type}\ \{\ m(x : \text{Number}) \rightarrow \text{Boolean}\ \}\ \}\ \{\ z \rightarrow \Uparrow z\ \}$$

After evaluating t, the match inspects the resulting value to see if it contains a method $\langle m, 1 \rangle$, and attaches the higher-order assumptions about the inputs and outputs if the method is present. The cast only chaperones the reference of the object that is the body of the cast: other references to the object may exist that are not constrained by the assumptions in the cast.

5.1.1 Coercing Requests

Coercions are generated whenever a method is requested on a cast that makes assumptions about that method. A coercion generated by a request checks that the arguments satisfy the type annotations of the parameters on the method, and the result of the request is coerced to the return type in the cast. Consider a request on a cast that assumes that a method accepts anything and returns an object with a unary method named a:

$$(y : \text{type}\ \{\ m(z : \top) \rightarrow \text{type}\ \{\ a \rightarrow \top\ \}\ \}).m(t)$$

The object at y contains a method m that requires its input to contain a method b:

$$\sigma(y) = \{\ \mathsf{method}\ m(z : \mathsf{type}\ \{\ b(x : \bot) \rightarrow \top\ \}) \rightarrow \top\ \{\ \cdots\ \}\ \}$$

Requesting m on the cast removes the cast, but wraps the input in a coercion that tests for the presence of b, and wraps the remaining request in a coercion that tests for the presence of a. The request unfolds the cast to become:

$$\mathit{coerce}(y.m(\mathit{coerce}(t, \mathsf{type}\ \{\ b(x : \bot) \rightarrow \top\ \})), \mathsf{type}\ \{\ a \rightarrow \top\ \})$$

Each of the coercions becomes a match expression around their inputs.

$$y.m(t \ni \langle b, 0\rangle\ \{\ z \rightarrow z\ \}\ \{\ z \rightarrow \Uparrow z\ \}) \ni \langle a, 0\rangle\ \{\ z \rightarrow z\ \}\ \{\ z \rightarrow \Uparrow z\ \}$$

If the input does not match the type expected by the method, or the method returns a value that does not satisfy the assumption of the cast, then an error is raised.

There appears to be an interesting asymmetry in the cast expansion: while the result of the request is checked against the return type in the cast, the arguments of the request are checked against the parameter types of the method that actually appears in the object that is the receiver of the request, and the parameter types in the cast are ignored. An assumption about the return type of a method adds information to the expected type of the request, but an assumption on parameter types *forgets* information about the expected type of an input to the method. Without also including a *source* type with the target type in the cast, the information about what was originally expected as the type of an input is lost when a cast is applied.

Using the parameter types in the method rather than in the cast is part of a broader philosophy of treating the object that is the body of the cast as the ultimate truth about that object's type. The higher-order coercions of Henglein (1994) and casts of Siek and Taha (2006) and Wadler and Findler (2009) only ever consider the consistency of the source and target of a cast, never the actual type of the underlying body. The advantage of their approach is that type annotations outside of a cast are irrelevant to the execution of a program and can be erased before run-time, whereas all parameter type annotations must be preserved at run-time in our implementation (though this is already true of Grace programs, because they are

translated into contracts at run-time). The disadvantage of not examining the body of the cast when attempting to determine if the assumptions of a cast are upheld is that a cast that makes valid assumptions may still fail: we discuss this problem in §5.5.2.

The apparent asymmetry is resolved by considering that the parameter types in the cast are not wholly useless: they are still used by the type system to type check the inputs before run-time. The parameter types in the cast correspond to the return type on the *method that is ultimately called*, not the return type in the cast: in both cases we trust the type system to check the relevant terms under the assumption that the surrounding context is sensible. The reverse of this correspondence is that the return type in the cast corresponds to the parameters on the method, in that they both must be checked at run-time.

Retaining both the source and target of a cast — and reversing these types when distributing a cast over an input — instead of investigating the parameters on the actual method being called would also prevent casts from fulfilling their most useful function in Graceless: calling methods discovered by a match. Any method discovered by a match has inputs of type $\bot$ within the 'then' branch of the match, but the parameter types on the actual method may be less restrictive. If the input types are checked by reversing the direction of a cast, then casting the input from $\bot$ to some type T would require checking that the input (of type T) satisfies $\bot$, which would mean the methods are still unable to be called (since $\bot$ is empty).

Consider the following use of a match, checking if a method m appears in the value of a term t, and requesting it with some input x if it does.

$$t \ni \langle m, 1 \rangle \{ z \to z.m(x) \} \cdots$$

Either this match is not well-typed, or the input x has type $\bot$. In the latter case the request to m can never be evaluated because the input has no value: the argument will never reduce to a value, either diverging or resulting in a raise.

The application of a cast to the receiver z can be used to make the request well-typed, by assuming that the m method will actually accept x (of some type T):

$$t \ni \langle m, 1 \rangle \{ z \to (z : \text{type} \{ m(x : T) \to \top \}).m(x) \} \cdots$$

So long as x has type T, this is now well-typed, but if the cast is required to ensure that it is correct by reversing the assumption it made about the input, the request on the cast would be transformed into the following term (where the reference y is the value of t):

$$y.m(\Uparrow x)$$

The input becomes a raise because it is a degenerate case of a coercion: there is nothing to check, because the coercion is guaranteed to immediately fail. By checking the parameter types on the actual method being called, the immediate failure of any inputs with assumptions on their types only happens if the method truly does not accept any object by including a $\bot$ parameter.

5.2 Syntax

The syntax of casts on top of the existing Graceless definitions is presented in Figure 5.2.1. Although we have motivated the concept for the interaction of typed and untyped code in the vein of gradual typing (Siek and Taha 2006), the type syntax is not extended to include the unknown or dynamic type ? directly, and there is no mechanism to 'forget' the type of the term in the cast.

Unlike the modern literature for cast calculi in the context of gradual typing (Herman, Tomb, and Flanagan 2010), our coercions do not describe their assumptions on top of the entire type of the term in the cast, only a delta from the existing type. The resulting coercion language only describes down-casts, never up-casts. A term can be cast to any other type through a combination of casts and subsumption.

As discussed in the design, terms can now be wrapped in a cast with the syntax $t : S$. The syntax of values v now includes any cast whose body is a reference, as casts must be retained to continue tracking the assumptions on the type annotations of the signatures in the cast. The body of a cast value is only ever a reference, not other cast values, so adjacent casts are not considered a value and must be merged together.

New evaluation contexts have also been added for the purpose of evaluating the body of casts from outside-in, rather than inside-out. The context I has a hole that

Grammar

$$t ::= \cdots \mid t : S \qquad \textit{(Term)}$$

$$v ::= \cdots \mid y : S \qquad \textit{(Value)}$$

Evaluation contexts

$$G ::= \cdots \mid \Box : S \mid I[F] : S \qquad \textit{(Sub-context)}$$

$$H ::= \Box \mid E[I] \mid E[\Box \Uparrow b] \qquad \textit{(Direct cast-free context)}$$

$$I ::= \Box.m(\overline{t}) \mid v.m(\overline{v_i}, \Box, \overline{t}) \mid m(\overline{v}, \Box, \overline{t}) \mid \Uparrow \Box \mid \Box \ni a\ b_1\ b_2 \mid \Box; t \qquad \textit{(Sub-context)}$$

Figure 5.2.1: Graceless grammar extended with casts

is immediately surrounded by any other term that is not a rescue or a cast. There is no recursion in the context I, as a hole appears directly at each relevant sub-term, so the metavariable only matches exactly the surrounding term, and there is no further nesting beyond this term.

The context G has been extended to include any cast, without descending further if the body is a raise or another cast. The structure of G matches either a cast whose body is directly a hole, or whose body is a recursive application of the context F through the direct rescue- and cast-free context I. The context term $(\Box : S_1) : S_2$ is not bound by G, because the body of the outer cast is not bound by I; the context term $(\Uparrow (\Box : S_1)) : S_2$ *is* bound by G, because the intervening raise is bound by I and the inner cast is bound by recursion on F in place of the hole of I.

The evaluation context H is the direct cast-free context, which means that it has a hole in any term, like the context E, so long as the hole does not appear directly inside of a cast. A hole in H may appear inside of a cast, so long as it is separated from the cast by at least one other term: for instance, in the context term $\Box; t : S$, the hole appears inside of a cast but is separated from it by the surrounding sequence. Any context H is either just a hole or an application of the context E whose hole is filled by some other binding of either the context I or a rescue.

The use of evaluation contexts to avoid evaluating the body of a cast that is immediately nested inside another cast is similar to the implementation of coercions (Henglein 1994; Herman, Tomb, and Flanagan 2010; Siek, Vitousek, and Bharad-

waj 2012) or threesomes (Siek and Wadler 2010; Garcia 2013), though the inclusion of raise and rescue forms causes the complexity of the added contexts to increase.

5.2.1 Type Coercion

A cast must describe the actual type of the object that it is casting at a shallow level: the declarations that appear directly inside of a cast's type must describe real definitions in the object, though the type annotations on the signatures need not be correct. We have codified the generation of the analysis of an object's structure in the *coerce* metafunction, defined as a series of overloaded functions in Figure 5.2.2.

The form *coerce*(t, T) checks if t satisfies the shallow description of T, and casts it to the appropriate type: that is, given any term t *coerce* can generate a term of type T with the underlying value of t, or that raises the value of t if it does not satisfy the shallow description of T. The simple cases are when the input type T is $\bot$, in which the result is a raise of the term (note that the term itself will still be evaluated before the cast is 'checked' even in this case), and when T is $\top$, in which case no cast is applied because no assumptions have been made.

Checking single structural types is also fairly straightforward by sequencing matches inside of one another to test for the presence of all of the necessary definitions. The only complication is that any of the matches could fail along the way, so each one has a unique 'else' branch with the same contents. Consider the following coercion:

$$\mathit{coerce}(\mathrm{t}, \mathsf{type}\ \{\ \mathrm{m}_1 \longrightarrow \mathrm{T}_1, \mathrm{m}_2 \longrightarrow \mathrm{T}_2\ \})$$

The application of *coerce* generates a match:

$$\mathrm{t} \ni \langle \mathrm{m}_1, 0 \rangle\ \{\ \mathrm{z} \longrightarrow \mathrm{z} \ni \langle \mathrm{m}_2, 0 \rangle\ \mathrm{b}\ \{\ \mathrm{z} \longrightarrow \Uparrow \mathrm{z}\ \}\ \}\ \{\ \mathrm{z} \longrightarrow \Uparrow \mathrm{z}\ \}$$

Note the two distinct uses of raise, that both represent the same failure of a shallow structural type check. The block b represents the success of the shallow check, and so should cast the object to the full input type; *coerce* gives us b as:

$$\{\ \mathrm{z} \longrightarrow \mathrm{z} : \mathsf{type}\ \{\ \mathrm{m}_1 \longrightarrow \mathrm{T}_1, \mathrm{m}_2 \longrightarrow \mathrm{T}_2\ \}\ \}$$

$$
\begin{aligned}
&\mathit{coerce} : \textsc{Term} \times \textsc{Type} \to \textsc{Term} \\
&\mathit{coerce}(t, T) = \mathit{coerce}(t, \bot, T)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Type} \to \textsc{Term} \\
&\mathit{coerce}(t, T, \bot) \quad\quad = \mathit{cast}(t, T) \\
&\mathit{coerce}(t, T, (\bigcup S, \overline{S_i})) = \mathit{coerce}(t, T, S, \bigcup \overline{S_i}, S)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Type} \times \textsc{Struct} \to \textsc{Term} \\
&\mathit{coerce}(t, T_1, S, T_2, \mathsf{type}\ \{\}) \quad\quad = \mathit{coerce}(t, T_1 \cup \bigcup S, T_2) \\
&\mathit{coerce}(t, T_1, S, T_2, \mathsf{type}\ \{\ D, \overline{D_i}\ \}) = \mathit{coerce}(t, T_1, S, T_2, \mathsf{type}\ \{\ \overline{D_i}\ \}, D)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Decl} \to \textsc{Term} \\
&\mathit{coerce}(t, T_1, S_1, T_2, S_2, D) = \\
&\qquad t \ni \mathit{identify}(D)\ \{\ z \to \mathit{coerce}(z, T_1, S_1, T_2, S_2)\ \}\ \{\ z \to \mathit{coerce}(z, T_1, T_2)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cast} : \textsc{Term} \times \textsc{Type} \to \textsc{Term} \\
&\mathit{cast}(t, \bot) \quad = \Uparrow t \\
&\mathit{cast}(t, \top) \quad = t \\
&\mathit{cast}(t, \bigcup \overline{S}) = t : \mathit{shallow}(\overline{S})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{shallow} : \textsc{Seq}(\textsc{Struct}) \rightharpoonup \textsc{Struct} \\
&\mathit{shallow}(S) \quad\quad\quad\quad = S \\
&\mathit{shallow}((\mathsf{type}\ \{\ \overline{D_i}\ \}, \overline{S_j})) = \mathsf{type}\ \{\ \mathit{shallow}((\overline{D_i}), \mathit{shallow}(\overline{S_j}))\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{shallow} : \textsc{Seq}(\textsc{Decl}) \times \textsc{Seq}(\textsc{Decl}) \to \textsc{Seq}(\textsc{Decl}) \\
&\mathit{shallow}((D_1, \overline{D_i}), (\overline{D_j}, D_2, \overline{D_k})) = \mathit{shallow}(D_1, D_2), \mathit{shallow}((\overline{D_i}), (\overline{D_j}, \overline{D_k})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where}\ \mathit{identify}(D_1) = \mathit{identify}(D_2) \\
&\mathit{shallow}((D, \overline{D_i}), \quad \overline{D_j}) \quad\quad\quad\quad = D, \mathit{shallow}(\overline{D_i}, \overline{D_j}) \\
&\mathit{shallow}(\cdot, \quad\quad\ \overline{D_j}) \quad\quad\quad\quad = \overline{D_j}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{shallow} : \textsc{Decl} \times \textsc{Decl} \rightharpoonup \textsc{Decl} \\
&\mathit{shallow}(m(\overline{z : T_{i1}}) \to T_1, m(\overline{z : T_{i2}}) \to T_2) = m(\overline{z : T_{i1} \cap T_i 2}) \to T_1 \cup T_2
\end{aligned}
$$

Figure 5.2.2: Coercion generation metafunctions

The entire term either raises the value of t or returns its value cast to the given type.

Thanks to the presence of union types, the *coerce* function must consider every possible combination of shallow matching between structural types. Consider an application of *coerce* and the resulting match term:

$$\mathit{coerce}(t, (\mathsf{type}\ \{\ m_1 \to T_1, m_2 \to T_2\ \} \cup \mathsf{type}\ \{\ m_1 \to T_3, m_3 \to T_4\ \}))$$

$$t \ni \langle m_1, 0\rangle\ \{\ z \to z \ni \langle m_2, 0\rangle\ b_1\ b_2\ \}\ b_2$$

The block b_1 corresponds to the success of a shallow check for the first structural type in the union, and b_2 to its failure. Since the type is a union, the failure of the check for the first structural type should not result in a raise, but a test for the second structural type matches instead. The block b_2 is therefore:

$$\{\ z \to z \ni \langle m_1, 0\rangle\ \{\ z \to z \ni \langle m_3, 0\rangle\ b_3\ \{\ z \to \Uparrow z\ \}\ \}\ \{\ z \to \Uparrow z\ \}\ \}$$

The 'else' blocks in this nested match correspond to the failure of the shallow checks of both types, and so raise the input.

The block b_3 represents the success of a shallow check on the second structural type in the context of the failure of a check on the first, so it casts the object to just part of the type that succeeded.

$$\{\ z \to z : \mathsf{type}\ \{\ m_1 \to T_3, m_3 \to T_4\ \}\ \}$$

The cast in the body of the match has actually refined the input type to a sub-type, since if the program evaluation enters b_3 then we have discovered more information about the underlying object.

The value of b_1 is more complicated than b_2, because the success of the check for the first structural type *has not invalidated the presence of the second.* The block b_1 must still check the second structural type. The *coerce* function will generate the following b_1:

$$\{\ z \to z \ni \langle m_1, 0\rangle\ \{\ z \to z \ni \langle m_3, 0\rangle\ b_4\ b_5\ \}\ b_5\ \}$$

This time the failure of the shallow check happens in the context of the success of

the first structural type, so b_5 corresponds to a cast of z to that type:

$$\{\, z \to z : \mathsf{type}\, \{\, m_1 \to T_1, m_2 \to T_2 \,\} \,\}$$

As with the success of the second structural type in the context of the failure of checking the first, this cast refines the type of the input to a sub-type.

The block b_4 is only entered if *both* structural types satisfied a shallow check, in which case the object should be cast to the original input type, so *coerce* generates b_4 as:

$$\{\, z \to z : \mathsf{type}\, \{\, m_1 \to (T_1 \cup T_3), m_2 \to T_2, m_3 \to T_4 \,\} \,\}$$

The cast is generated by the metafunction *cast*, which generates a cast on a term given a union type where every type in the union has succeeded in a check against the term. If the union is empty, none of the checked types succeeded, and a raise is generated. If the union only contains an empty structural type then there are no added assumptions, so no cast is generated and the term is returned unchanged.

The metafunction *shallow* takes a list of structural types and collapses them in a similar fashion to the intersection operation, except that the use of union and intersection on the parameter and return types of the declarations are reversed. Casts must appear only after the relevant type has been determined, not after each method identifier is matched, to avoid otherwise invalidated casts from appearing in the 'else' branches of nested shallow coercions.

5.3 Dynamic Semantics

An extension to the dynamic semantics of Graceless to evaluate casts is presented in Figure 5.3.1. Since evaluation in a cast is already handled by Rule E-Cng with the extension of the evaluation context F, the only remaining forms that need to be reduced are those that rely on existing elimination rules where casts can now appear. These are the rules for reduction of requests, Rule E-Req, and for processing match constructs, Rules E-Fst and E-Snd. For the latter two, we simply define store lookup to be 'cast transparent', so that $\sigma(y : S) = \sigma(y)$; this allows the rules to eliminate the match as before. The cast value is still substituted into the body of the match, ensuring that any existing assumptions are upheld.

$\boxed{\sigma \mid t \longrightarrow \sigma \mid t}$

$$\frac{S \ni m(\overline{z : T_{i1}}) \to T_1 \qquad \mathsf{method}\ m(\overline{z : T_{i2j}}) \to T_2\ \{\, t \,\} \in \sigma(y)}{\sigma \mid (y : S).m(\overline{v_i}) \longrightarrow \sigma \mid \mathit{coerce}(y.m(\overline{\mathit{param}(v_i, T_{i2}, T_{i1})}), T_1)}\ (\text{E-Cst})$$

$\boxed{\sigma \mid t \longmapsto \sigma \mid t}$

$$\frac{}{\sigma \mid H[(t : S_1) : S_2] \longmapsto \sigma \mid H[t : S_1 \cap S_2]}\ (\text{E-Mrg})$$

$\boxed{S \ni D}$

$$\frac{D \in \overline{D_i}}{\mathsf{type}\ \{\, \overline{D_i} \,\} \ni D}\ (\text{F-Mem}) \qquad \frac{\langle m, |\overline{z}| \rangle \notin \overline{\mathit{identify}(D_i)}}{\mathsf{type}\ \{\, \overline{D_i} \,\} \ni m(\overline{z : \bot}) \to \top}\ (\text{F-Nin})$$

$\mathit{param} : \textsc{Term} \times \textsc{Type} \times \textsc{Type} \to \textsc{Term}$
$\mathit{param}(t, T_1, \bot) = t$
$\mathit{param}(t, T_1, T_2) = \mathit{coerce}(t, T_1)$

Figure 5.3.1: Reduction with casts

Rule E-Mrg reduces a cast directly containing another cast by merging the two casts together. The resulting structural type is the intersection of the two cast types, as the new cast must enforce the assumptions of both casts at once. The cast reduction occurs in an evaluation context H, so the rule applies to any two adjacent casts except if the outer cast appears directly in a cast as well. Casts are merged from outside-in, and the body of a cast is not evaluated until all of the directly surrounding casts are collapsed into a single cast.

Rule E-Cst handles reducing qualified requests where the receiver is a cast instead of just an object reference; this is necessary in the case that the cast makes assumptions about the type of the method being requested. In order to precisely retrieve these assumptions, a signature selection judgement for types $\mathsf{T} \ni \mathsf{D}$ is defined in the same figure, in a similar vein to the typing environment signature selection judgement $\Gamma \ni \mathsf{D}$.

Signature selection from a type ensures that the most precise information is gathered about the possible return type of the selected method. The selection will still succeed if a signature with the relevant identifier is not present anywhere in the type, but the resulting signature will be in its *ground* form: this saves us from having to define a separate rule in the reduction relation for handling the case where the cast does not address the method that is being requested.

The signature selection judgement is also precise on the types of the parameters, but these are irrelevant to processing a request on a cast. Because parameter types in a cast are ignored, the only types that end up mattering are the ones on the actual method in the object itself. Cast transparency in store lookup comes into play here as well, as the value in the cast is used to look up the actual method that will be requested, and source the appropriate parameter types from there.

Typically casts in a contravariant position such as parameter types on a method express a negative cast that needs to be reversed when flipped into a positive cast on the arguments of a request or call. If the type of a parameter is $\bot$ and this is cast to $\top$ (still a down-cast, since the polarity is reversed in a contravariant position), it would be necessary to remember that $\bot$ was the original type rather than that $\top$ is the outcome, so that reversing the parameter cast produces a new cast from the input of type $\top$ to $\bot$. Under subsumption, the parameter type on the underlying method might actually be *more permissive* than the parameter type, and casting

$$\boxed{\Gamma \vdash t : T}$$

$$\text{(T-Cst)}\quad \frac{\Gamma \vdash t : T \qquad \vdash S \qquad T <: \bigcup ground(S)}{\Gamma \vdash (t : S) : T \cap \bigcup S}$$

$$ground : \textsc{Struct} \to \textsc{Type}$$
$$ground(\mathsf{type}\ \{\ \overline{D}\ \}) = \bigcup \mathsf{type}\ \{\ \overline{ground(identify(D))}\ \}$$

Figure 5.4.1: Term typing with casts

the input to $\bot$ is not necessary to preserve the safety of the request: satisfying the parameter on the method is all that is required.

As discussed in §5.1.1, casting to the real parameter types instead of just reversing the assumption allows casts to change the parameter types of a method in the 'then' branch of a match to the desired assumption, without having to cast the input to $\bot$ when the method is requested. The parameter types of any method newly discovered by a match are always $\bot$ regardless of what actually appears on the method in the store, so just reversing the assumption would still leave the method unable to be requested: it would now be type-safe, but always evaluate to a raise.

5.4 Static Semantics

Given that we have only introduced a single new form of term, only a single rule is required to type this extension. The extended type system is defined in Figure 5.4.1. Since a cast is a delta of an the underlying term's type, Rule T-Cst first types the term in the cast, and then types the cast as the intersection of that term's type and the type in the cast (guaranteed to be a sub-type of both by Lemma 13). The type in the cast must also be well-formed.

The *ground* of a type is effectively the application of the *ground* function on method identifiers applied to all of the signatures of the type, so for instance the type $\mathsf{type}\ \{\ m_1(z : T_1) \to T_2\ \}$ has the ground type $\mathsf{type}\ \{\ m_1(z : \bot) \to \top\ \}$. The type of the term must subtype the *ground* of the cast type to enforce the property that the shallow description of the type in a cast always correctly describes the type of

the term in the same cast. For any well-typed request with a cast as the receiver, a corresponding method always appears in the resulting object (or the term in the cast never reduces to a value).

Note that the type of a cast can be the type $\bot$, in which case the *ground* type is also $\bot$. The term in the cast has no value by Lemma 15, and it is not necessary for this cast to translate to a raise to ensure that the outcome is uninhabited.

5.4.1 Properties

The cast extension has been relatively small thanks to the use of the existing features to perform shallow cast tests, so we only need a few extra lemmas to prove that type soundness is retained for the extended language. The first is a modified canonicity lemma: Lemma 19 still holds, but now that casts are also values we need to show that typing guarantees that a relevant method appears in the underlying object. The conclusions of this lemma are different to Lemma 19, as a method in an object behind a cast does not necessarily have a compatible signature. The only guarantee is that a method with the same identifier is present.

Lemma 28 (Canonicity of casts)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash v : \mathsf{type}\ \{\ \mathrm{D}\ \}}{\mathrm{d} \in \sigma(v) \qquad \mathit{identify}(\mathrm{d}) = \mathit{identify}(\mathrm{D})}$$

Proof. Immediate from the induction hypothesis applied to the inversion of Rule T-Cst, with the subtyping conclusion of Lemma 19 implying the equality of identifiers. □

This is sufficient to update the progress lemma.

Lemma 29 (Typing implies progress)**.** *For any program* $\langle \sigma, \mathrm{t} \rangle$*, if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *then either:*

- $\exists v.\ \mathrm{t} = v$
- $\exists v.\ \mathrm{t} = \Uparrow v$
- $\exists \sigma'\ \mathrm{t}'.\ \sigma \mid \mathrm{t} \longmapsto \sigma' \mid \mathrm{t}'$

Proof. Extend the existing case analysis to consider the new typing rule, as well as new reduction possibilities in old rules.

(T-R/Q) If the receiver is a cast, then Rule E-Req no longer applies. Signature selection from a type $T \ni D$ always succeeds thanks to Rule F-Nin, and Lemma 28 guarantees that a corresponding method appears in the store, so Rule E-Cst can be applied.

(T-Cst) Immediate from the induction hypothesis and either Rule E-Mrg if the body is a cast or Rule E-Cng for any other term, or the whole term is a value.

The remaining rules are unchanged. □

For preservation, we need two extra lemmas. The first is that the *coerce* function actually coerces the given term into the given type, so long as the term itself is typed.

Lemma 30 (Coercion implies typing)**.**

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash \mathit{coerce}(t, T_2) : T_2}$$

Proof. Case analysis of the input T_2:

$\bot$ The outcome is $\Uparrow t$, which has the type $\bot$ by Rule T-Rse and the existing proof that t is well-typed.

$\top$ The outcome is t, which has the type $\top$ by Rule T-Sub and the existing proof that t is well-typed.

else The outcome is $t \ni a\ \{\, z \longrightarrow t_1 \,\}\ \{\, z \longrightarrow t_2 \,\}$, so we can type this with Rule T-Mch if t_1 and t_2 are well-typed. Use of z is well-typed by Rule T-Var in the extended typing environment of the two blocks. The ground of cast signatures introduced in t_1 onto z are guaranteed to appear in the type of z by the surrounding match, and so can be typed by Rule T-Cst, with the remaining forms typed by induction. The typing of t_2 follows directly from induction.

This covers all cases. □

The second lemma proves that signature selection is as precise as possible about the return type of the selected signature; that is, any signature selected through the subtyping relation produces a return type that is a super-type of the result of the selection judgement.

Lemma 31 (Selection return subtypes lookup return)**.**

$$\frac{\mathsf{T} <: \mathsf{type}\,\{\, \mathfrak{m}(\overline{z_i : \mathsf{T}_i}) \to \mathsf{T}_1 \,\} \qquad \mathsf{T} \ni \mathfrak{m}(\overline{z_i : \mathsf{T}'_i}) \to \mathsf{T}_2}{\mathsf{T}_2 <: \mathsf{T}_1}$$

Proof. Mutual analysis of the two inputs and rebuilding the corresponding subtyping rules, ignoring Rule F-Nin thanks to Lemma 1. □

These are sufficient to show preservation.

Lemma 32 (Reduction preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash \mathsf{t} : \mathsf{T} \qquad \sigma \mid \mathsf{t} \longmapsto \sigma' \mid \mathsf{t}'}{\vdash \sigma' : \Gamma' \qquad \Gamma' \vdash \mathsf{t}' : \mathsf{T}}$$

Proof. Extend the existing case analysis to consider the new reduction rule.

(E-Mrg) Rule T-Cst, with the resulting obligation that $\mathsf{T} <: \mathit{ground}(\mathsf{S}_1 \cap \mathsf{S}_2)$, which is satisfied by the fact that no new signatures can be added by a cast and that ground signatures are trivial to subtype.

(E-Cst) The inversion of Rule T-Cst (and Rule T-Sub) on the original cast proves that the value v has a type that subtypes the *ground* of the cast type T, and the type of the cast itself is the intersection of the type of v with T. The application of Lemma 30 to the types of the arguments alongside the typing of v, whose relation with *ground* ensures that the relevant method appears in the type, rebuilds Rule T-R/Q for the request inside the resulting cast, and then an application of Rule T-Cst types the cast as the intersection of the return type of the inner request and the return type selected from T by Lemma 30. The type is the lowest type on the lattice that could have been selected by the old typing rules by Lemmas 13 and 31, so applying Rule T-Sub to this typing with Lemma 31 provides the necessary type.

The remaining rules are unchanged. □

As before, the progress and preservation lemmas combine to extend our proof of soundness to the cast language.

Theorem 33 (Well-typed programs don't get stuck)**.** *For any Graceless program* $\langle \sigma, t \rangle$*, if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : T$*, then either:*

- $\exists v.\ t = v$*, so* $\Gamma \vdash v : T$
- $\exists v.\ t = \Uparrow v$*, so* $\Gamma \vdash \Uparrow v : T$
- $\exists \sigma'\ \Gamma'\ t'.\ \sigma \mid t \longmapsto \sigma' \mid t'$*, with* $\vdash \sigma' : \Gamma'$ *and* $\Gamma' \vdash t' : T$

Proof. Immediate from Lemmas 29 and 32. □

5.5 Discussion

Directly adjacent casts are immediately merged together to avoid duplicating information; this behaviour is not necessary, but it avoids the potential for massive number of casts with overlapping information building up on a single reference. Merging casts is an interesting comparison point to casts in gradually typed languages, where the dynamic semantics rely entirely on merging casts to even detect if the assumptions of the cast are upheld (Siek and Taha 2006; Siek and Taha 2007). Many of the formal gradual languages perform early merging of casts to maintain efficiency (Herman, Tomb, and Flanagan 2010; Siek and Wadler 2010; Garcia 2013), but adding efficient merging to casts makes the application of blame tracking vastly more complicated than just tagging a cast with a blame label.

5.5.1 Blame

One aspect that is missing from the design for casts is a system for blame tracking. Consider a hypothetical design for blame: casts labelled with blame labels ℓ as in the cast calculi of Wadler and Findler (2009) or Cimini and Siek (2017). The

relevant blame label is attached to a raise generated by a coercion to indicate which cast was to blame for a run-time type error:

$$t \ni \langle m, 1 \rangle \{ z \to z :^{\ell} \mathsf{type} \{ m(x : T_1) \to T_2 \} \} \{ z \to \Uparrow^{\ell} z \}$$

When a cast expands into a coercion on a request, the blame label on the cast is distributed into the generated cast and raise forms.

A *safe* cast is one where the type in the cast is entirely subsumed by the already known information about the term in the cast, such that the down-cast is to the existing type of the term (a cast cannot lose type information, but it can express a cast that gains no new information). A safe cast in some typing environment Γ is defined as:

$$\frac{\Gamma \vdash t : \bigcup S}{t : S \text{ safe}}$$

The blame theorem for Graceless would then be that any raise cannot have a label from a safe cast.

Proposition 1 (Safe casts cannot be blamed)**.** *For a store σ and term t_1, if $\vdash \sigma : \Gamma$, $\Gamma \vdash t_1 : T$, all labels ℓ in t are unique, and $\sigma \mid t \longmapsto^* \sigma' \mid \Uparrow^{\ell_1} v$, then for any cast $t_2 :^{\ell_2} S$ in t_1, if $t_2 :^{\ell_2} S$ safe then $\ell_1 \neq \ell_2$.*

This result is not particularly interesting, because a safe cast is effectively useless anyway.

The merging of casts makes the design above untenable. Consider the following adjacent casts:

$$(t :^{\ell_1} m \to T_1) :^{\ell_2} m \to T_2$$

To merge these casts together, the types T_1 and T_2 must be intersected together, but the blame labels must be part of this intersection as well: if m is requested on this cast, the resulting coercion on the result of the request must know where to attribute blame if the result fails to satisfy the assumptions of either T_1 or T_2:

$$t : m \to T_1^{\ell_1} \cap T_2^{\ell_2}$$

The resulting form must also be ordered, as the cast blaming ℓ_1 is the closer as-

sumption to the body, so if both T_1 and T_2 fail then ℓ_1 must be blamed, not ℓ_2.

In order to preserve the appropriate blame label while also coalescing adjacent casts, the types in a cast must be labelled instead of the cast itself. A single cast must be capable of having multiple blame labels in it at once — as suggested by the syntax used in the intersection above — in order to preserve the necessary information to correctly assign blame. The ordering of the arguments to the intersection must also be retained, so the definition of a corresponding operator to the coercion composition operators $\circ$ of Henglein (1994) or $\fatsemi$ of Garcia (2013) would also need to be defined once the syntax of labelled types was established. Given the potential for an extreme amount of duplicated information that results if casts are not merged, we have opted to merge casts and leave blame tracking as future work.

5.5.2 Gradual Typing

Gradual typing is a large part of the motivation for these casts, but the extended Graceless language presented here is not gradual, and does not include a dynamic or unknown type in its static type system. The design of Graceless casts differs substantially from the casts found in most gradual typing literature, and in particular do not describe total source and target types for the cast, just the added structural assumptions.

Both casts and coercions in gradual typing explicitly describe the act of *forgetting* type information as well as making assumptions about the type of an object. The rules around consistency (or the syntax of coercions more directly) only allow information about the type to be forgotten by replacing a type with the dynamic type ?; in comparison, forgetting some but not all structural information through subsumption is always performed implicitly, just as in a statically typed language. Unlike other cast calculi, Graceless only uses casts to describe added assumptions, and always combine with the existing type of the body of the cast: information cannot be lost in a cast, only through subsumption.

Our philosophy that the body of a cast should be treated as the source of truth for checking a cast's assumptions avoids a problem with structural casts in object-oriented languages with gradual typing, where a cast that accurately describes the structure of its body can still fail. Consider the following term in the cast calculus

$\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ of Siek and Taha (2007):

$$\langle [\mathtt{l} : \cdots] \Leftarrow ? \rangle \langle ? \Leftarrow [] \rangle [\mathtt{l} = \cdots]$$

The type and implementation of the $\mathtt{l}$ method is unimportant, and so elided. The term is considered a *bad cast*, as the casts cannot be merged with a reduction step despite the fact that the term is well-typed. The merge reduction rule requires that the target type be a consistent super-type of the source, but in the given term the target is a sub-type, not a super-type.

Despite the cast's failure, the assumptions of the target type are satisfied by the object that is the body of the cast. The only reason for the cast's failure is that only the consistency of the types in the casts are considered, never the body of the cast itself. Because the source type of the cast is a strict super-type of the underlying object, information has been lost in the cast, and it is this loss of information that ultimately causes the cast to fail when it would otherwise be safe to proceed with a merge operation.

We can construct a program in the gradual source language that, following the cast insertion procedure, reduces to the given $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ term. The information loss is encoded using subsumption to forget the exact type of the object before casting it back to its original type. First, consider a metafunction *id* that takes a term and a type, and applies the term to an identity method:

$$id(\mathtt{t}, \mathtt{T}) = [\mathtt{m} = \mathtt{T}\ \varsigma(\mathtt{x} : \mathtt{T})\ \mathtt{x}].\mathtt{m}(\mathtt{t})$$

Any term generated by *id* is well-typed so long as the given type is a consistent super-type of the type of the term. The metafunction can be used to forget or assume information about the type of the term, either through subsumption or the application of a cast (generated by the cast insertion procedure).

An $\mathbf{Ob}^{?}_{<:}$ program that, following cast insertion, reduces to the bad cast above is the application of a method that accepts a value of unknown type and casts it to the ultimate target type of the cast, applied to the cast's body whose type is forgotten by subsumption:

$$[\mathtt{m} = [\mathtt{l} : \cdots]\ \varsigma(\mathtt{x} : ?)\ id(\mathtt{x}, [\mathtt{l} : \cdots])].\mathtt{m}(id([\mathtt{l} = \cdots], []))$$

The application of *id* inside of the object is used to add an assumption that the type of the otherwise unknown parameter x has the type $[l : \cdots]$, required by the return type of the surrounding method: this assumption will cause a cast to be inserted. The other application of *id* uses a simple super-type of the actual type of the object, so the type information is lost purely by subsumption and no cast is generated.

The $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ term generated by applying cast insertion to the $\mathbf{Ob}^{?}_{<:}$ term above is:

$$[m = [l : \cdots]\ \varsigma(x : ?)\ id(\langle [l : \cdots] \Leftarrow ? \rangle\, x, [l : \cdots])].m(\langle ? \Leftarrow [] \rangle\, id([l = \cdots], []))$$

The cast inside of the application of *id* is the result of the method application generated by *id*, whereas the cast in the argument of the call to m appears because the type of the parameter of m is $?$. The reason for the bad cast is that the source type of the latter cast is $[]$, since the more specific type of the body has been forgotten through subsumption.

The cast insertion procedure does not insert casts that only encode information loss from subsumption. The type system does not permit such a cast: the source and target types of a cast must be consistent, not consistent sub-types, so no subsumption can be present in the difference of the two types in a cast. Without interrogating the body, encoding subsumption into casts is necessary to prevent otherwise correct casts failing because of a loss of information, which would mean that any use of subsumption would cause a cast to appear in the run-time program.

A coercion in Graceless that correctly describes the shallow structure of its subject cannot fail, because the match form actually interrogates the underlying object. Encoding subsumption in casts is not necessary, as all of the relevant information is available directly in the value of the object itself, with the trade-off that parameter types must be retained in every method definition.

5.5.3 Gradual Guarantee

We have taken care in the design of the match and cast forms to apply the lessons of the gradual guarantee (Boyland 2014; Siek, Vitousek, Cimini, et al. 2015), even if Graceless itself is not a gradual language. Only performing shallow matching is desirable because performing a full examination of an object's structural type can be an inefficient operation, particularly in the presence of recursive types. Despite our

efforts, a gradual form of Graceless would still fail to uphold the dynamic portion of the gradual guarantee, but the structural match is not to blame.

Run-time type errors become a raise of the body of a failed cast, and since a raise can be rescued, a Graceless program can suffer a type error and recover. The guarantee requires that changing a type in otherwise equivalent programs should not affect its behaviour, except if the change now causes or prevents a type error. In practically every gradual calculus a type error is fatal to a program's execution, so the raise of such an error cannot be manipulated in any way.

If type errors can be rescued then it is possible to construct a program that violates the gradual guarantee. In Graceless, if we have a term t with an assumption on its method m, we can attempt to request m with an argument x. If x fails to satisfy the parameter type on m then a type error will be raised, but the error can be rescued and a different term evaluated instead.

$$\mathrm{t.m(x)} \Uparrow \{\, \mathrm{z} \rightarrow \mathsf{object}\ \{\}\,\}$$

The behaviour of this program depends entirely on the parameter type annotating m: for instance, changing the type between $\bot$ and $\top$ is guaranteed to produce different outcomes (the fresh object in the block, and the result of calling m, respectively).

The ability to inspect the higher-order types of an object is not specific to Graceless, and any language gains this ability if type errors can be rescued, including languages that are functional rather than object-oriented. The alternative is that type errors always fatally crash a program, which may be an alarming property for a practical language to have. The gradual guarantee is only acceptable because it is applied to core calculi with irrecoverable error states; updating the guarantee to apply to more practical languages may be necessary to preserve its relevance, but we leave this as future work.

6 Brand Typing

The encoding of a class as a method is sufficient for constructing objects, but structural types cannot express information about where an object was constructed. This is an intentional design decision: structural types correspond more closely to the fundamental nature of object-orientation as the passing of messages: the type of an object is the set of messages it is capable of responding to, and where it comes from is unimportant. In contrast, nominal types permit the programmer to explicitly encode their intent for relationships between objects in a way that is difficult to express with only structural types.

One of the key advantages of nominal typing over structural typing is that if we need to determine whether an object satisfies an interface at run-time, it is typically simpler to ask whether an object is from a particular class that implements that interface than it is to manually examine the object's structure. The disadvantage is that the object may well satisfy the required structure of the interface even if it was not sourced from the particular class we are using to perform this discrimination, but it is also possible under structural typing that an object implements an unrelated interface that happens to declare the same structure as the type. Few languages provide a middle-ground between these two design points, where the user of the language can choose to discriminate based on either the class or the interface (or both).

This chapter introduces *brand objects* to implement nominal typing for Grace objects and classes. Brand objects are implemented as annotations that tag an object constructor or method definition. Each brand has an associated guard object that matches at run-time those objects that have been appropriately annotated. A

Aspects of this chapter appeared in the ECOOP'15 paper Jones, Homer, and Noble 2015.

Grace dialect can then reason about these objects via a static, quasi-nominal type system. Brand typing is modelled using the existing features of Graceless and as a new well-formed relation for programs that can either extend the previous structural system or stand alone. We show that if a program is fully-typed then the combined well-formedness soundly prevents type errors. The nominality is built from scratch, without modifying the underlying language at all.

The contributions of this chapter are:

- A practical design of *brand objects* for nominal typing on top of Grace's existing structural type system.
- Applications of branding for various components of the language design.
- A formal model of brands as nominal types as an extension to Graceless.
- An implementation of brand objects and a static nominal type checker in Hopper, our prototype implementation of Grace.

This chapter presents a design for brand types, describes how they can be implemented in Grace and further used to replace a number of otherwise built-in concepts in the language, and, along with the extension to Graceless, presents how these features that were missing from the formal model can now be implemented.

6.1 Design

In order to support nominal typing, we have added brand objects to the Grace programming language as an extension to its existing structural typing mechanism. A brand object represents a unique marker that can be applied to an object and then subsequently detected, either statically or dynamically; the effects of these objects are similar to the branded types in Modula-3 (Nelson 1991). By separating the *permission* from the *guard*, a brand can act as a capability object. If the permission object is kept private to a class then it has exclusive permission to brand the objects that it constructs, while publicly exporting the guard object as the reified nominal type.

Brands applied to nominal typing are typically structured alongside a hierarchy of classes, encoding only nominal type structures and including the interface of objects constructed by the class in the type. In contrast, our brand objects have no associated class, and two objects branded with the same brand may have entirely distinct interfaces — we rely entirely on the existing structural type system to provide interface information. A brand type represents exactly those objects which have been branded by the underlying brand object, and no more. This is not a weakness of the design: by run-time discrimination on the guard objects, it *is* possible to determine more information about an object's interface when variants in a union type are eliminated.

We use three existing features of Grace in our implementation. Object annotations (Black, K. B. Bruce, and Noble 2016) — where a newly constructed object is annotated with some other object — are used to explicitly brand objects. A pattern object (Homer, Noble, et al. 2012) — which provides run-time pattern-matching facilities — acts as a brand's guard object, to test the presence of that brand on a given object. The dialect system (Homer, Jones, et al. 2014) — allowing the creation of a pluggable static type system — reasons about the patterns of brands bound to statically observable names as nominal types, and treats branded objects as inhabitants of these types.

6.1.1 Creating, Applying, and Using Brands

Consider a class hierarchy representing shapes, defining the concrete classes square and circle. In Grace, we might first define a Shape type:

```
let Shape = {
    at → Point
    area → Number
}
```

The Shape type describes the expected structure of a shape object. We could then define a class hierarchy which implements this interface. The top of the hierarchy is an abstract class that implements the common behaviour of all shapes. The shapeAt class builds an object which has a location, and leaves the area method unimplemented, so it is annotated as abstract with the keyword is:

```
class shapeAt(location : Point) → Shape is abstract {
    method at → Point { location }
    method area → Number { required }
}
```

We can then complete the implementation with concrete classes that inherit from the abstract one.

```
class squareAt(location : Point) withLength(length : Number) → Shape {
    inherit shapeAt(location)
    method area → Number is override { ... }
}

class circleAt(location : Point) withRadius(radius : Number) → Shape {
    inherit shapeAt(location)
    ...
    method area → Number { ... }
}
```

Note that the classes are annotated with return types, as Grace classes are distinct from types. Moreover, all of these classes have the *same* return type, because their instances all have the same interface.

We could explicitly declare types for the objects created by the squareAt and circleAt classes, by listing the signatures of the public methods in each class:

```
type Square = { at → Point; area → Number }
type Circle = { at → Point; area → Number }
```

These new types are identical to the Shape type defined above, and represent exactly the same set of objects. Structural types cannot distinguish between different objects with the same interface, either during static checking or at run-time.

Brands can be used to make finer distinctions between objects, distinctions that correspond to standard nominal types. In this design, brand objects are created by the brand method, which returns a new unique brand object. For example:

```
def aSquare = brand
```

This will create a new brand object named aSquare. We can use this object to brand other objects (e.g. those created by the square class) by annotating the class declaration with the brand:

```
class squareAt(location : Point)
        withLength(length : Number) → Shape is aSquare { ... }
```

The brand aSquare is not a type, and annotating the class with the brand is different from providing a return type, hence the appearance of both the aSquare brand and the Square type. Brand objects have a Type method that returns the guard object of the brand as a Grace pattern, matching the objects that have been branded. Grace's pattern objects are also used to reify types as objects, so guards have the same interface as types. This lets us define distinct Square and Circle types by combining the structural Shape type with the types of the respective brands via Grace's type intersection operator (∩).

```
type Square = Shape ∩ aSquare.Type

def aCircle = brand
type Circle = Shape ∩ aCircle.Type
```

Brands combined with structural interfaces produce Grace types that behave like nominal types. The Square and Circle above define different types, rather than aliases of the same structural type.

We can now declare that the square class returns an object of the Square type instead of the purely structural Shape type:

```
class squareAt(location : Point)
        withLength(length : Number) → Square is aSquare { ... }
```

The instance's structural type is the same as before, but it carries the added information that it is branded as aSquare, making it an instance of the Square type as well.

The combination of brand types with structural types follows the same type rules as other types, including subtyping. A branded object (with the appropriate brand) must be supplied where a branded object is expected:

```
def mySquare : Square = squareAt(10 @ 50) withLength(20)
```

A branded object may be used anywhere an unbranded object with the same structure is expected:

```
def myShape : Shape = mySquare
```

Critically, an unbranded object cannot be used where a branded object is expected:

```
// Error: not an instance of Square
def myCircle : Square = circleAt(10 @ 50) withRadius(20)
```

If these definitions of brands were repeated in another module (or even in the same module) then each definition will create a *different* unique identifier and so represent distinguishable, different brands (and pattern objects that guard only their associated declaration), regardless of the name the brand is bound to. The nominal aspect of the brand is its underlying object identity, and there is no requirement that a brand even be immediately bound to a name.

6.1.2 Brands vs. Brand Types

The distinction between a brand object like aSquare and its type aSquare.Type or Square is crucial. Branded objects can only be created with access to the underlying brand. An untrusted object can safely be given access to the guard object, as this does not allow that object to fraudulently brand other objects. This can be achieved by exposing the branded type to other code as a public constant and retaining the brand object only within the scope where the permission should be available, for instance as a confidential field in the surrounding object.

In Grace, a brand definition — like any other named definition — is a method in an object; the name of a type (branded or not) is simply a request to the object declaring the type, and so Grace's existing visibility mechanism suffices to protect brands. When we define a brand with a def, the definition is confidential by default so no further action is required. At the user's discretion the definition could also be made public. Care does have to taken with confidential fields in the presence of inheritance, since an inheriting object will also have access to the brand: to make a brand definition truly private, it can be defined in the closure surrounding the object that will export the type.

```
def aSquare = brand
```

```
object {
    type Square = aSquare.type
}
```

Access only flows in one direction: the brand object cannot be retrieved from the type, but access in the other direction is not limited, as the pattern object is available through the brand object with the Type method. The pattern object does not provide any privileged behaviour, so it makes sense to provide uni-directional access between the objects rather than not linking them at all, and returning a pair from the brand constructor instead.

The three branding utilities — the brand method, the use of brands as annotations on object literals and classes, and the unique types they introduce — are the only additions Grace's structural type system requires in order to support nominal types. Moreover, using Grace's patterns and dialects, they are all achieved using existing functionality, with no brand-specific modifications to the language's syntax or semantics.

6.1.3 Extending Brands

Inheriting from a branded object causes the inheriting object to have the same brands: this behaviour is necessary for inheritance to preserve subtyping (required by the Grace specification, Black, K. B. Bruce, and Noble 2016). Inheritance is the easiest mechanism for extending an existing brand, and provides a correspondence between class and (nominal) type, as in most nominally typed languages.

Consider if the shapeAt class above was also branded:

```
def aShape = brand
type Shape = aShape.Type ∩ type { ··· }
class shapeAt(location : Point) → Shape is abstract, aShape { ··· }
```

Now the whole shape hierarchy is branded, and the Shape type will only match objects created by the shape class, including those which inherit from it. The Square and Circle types remain subtypes of Shape and the square and circle classes inherit the aShape brand, just as if the classes were in a standard nominal typing hierarchy.

Brands need not conform to single-inheritance class hierarchies, even if the

class inheritance system does. Because brands are not inherently associated with an interface and access to the brand object is all that is required to build an object which satisfies the brand type, any object can inhabit multiple brand types without multiple-inheritance by simply being branded multiple times. This is conceptually similar to a class implementing multiple interfaces in Java or C#, providing a typing relationship without method reuse.

Brand objects also support the + operator, which creates a 'sub-brand' from two existing brands. Using this new brand is exactly the same as using the two parts together: branding an object with a composite brand causes the object to be branded with both of its component parts, and so the object is matched by both constituent brands' Types. Combining a brand with a new, anonymous brand, creates a unique sub-brand of the extended one. This behaviour is included in the brand interface as the extend method.

6.2 Applications

Brands fulfil a number of different use cases: not only can they be used to simulate classical nominal types, the dynamic semantics provide the behaviour of object capabilities (Miller 2006), and the combination of the static brand semantics with Grace's existing type system permits the encoding of other typing disciplines as well. This section presents applications of brands within the existing language implementation, replacing ad-hoc implementations with the branding mechanism.

6.2.1 Abstract Syntax Tree

An abstract syntax tree (AST) may contain many nodes with the same structure, but which must nevertheless be distinguished. This is a particularly important problem for Grace, as dialect check methods operate over the AST of the modules that they check. Nodes for variable and constant definitions will have a name, a value, and a type, but it is important that neither be mistaken for the other when they must be considered distinct. While the subtyping structure of an AST node is likely to be 'flat', brands allow overlaying distinguishing features on a range of otherwise-identical types.

We draw out two cases in particular from the AST of Grace source code, reflecting issues we have had ourselves in implementing the language. The var and def (variable and constant definition) nodes have the same fundamental shape, while a class node has a superset of the methods of an object node. Before brands, AST nodes were 'stringly-typed', using a kind string field with the name of the node type, but this was an ad-hoc solution that sat outside of the type system.

We can combine brands and types to avoid both of these issues: each kind of node now has both a structural interface and one or more nominal brands. Once we have created the relevant brands, the types can be constructed as:

```
// The common interface of both var and def nodes.
type DeclNode = Node ∩ type {
    name → String
    value → Expression
    typeAnnotation → Expression
}

type VarNode = aVarNode.Type ∩ DeclNode
type DefNode = aDefNode.Type ∩ DeclNode
```

The DeclNode type is purely structural, and before brands this was the *only* type that applied to each of our nodes (other than its super-types). VarNode, however, combines the structural type with the pattern of the aVarNode brand: to belong to the VarNode type, an object must have both the structural type and be branded as aVarNode.

```
class varNode(...) → VarNode is aVarNode { ... }

match(varNode(...))
    case { d : DefNode →
        print("A def!")
    } case { v : VarNode →
        print("A var!")
    }
```

Prior to brands, just as in our shapes example from earlier, the VarNode statement would 'fall into' the DefNode branch (Boyland 2014), because the structural

type matches, and the DefNode brand appears first. Similarly, a DefNode could be passed to a method expecting a VarNode node without error. With brands, the branch for DefNode does not match against VarNode objects and the correct branch is given an opportunity to match, while both static and dynamic type checks will behave as desired.

Alternatively, the node patterns like DefNode and VarNode could be defined as regular objects, rather than structural types, matching on the string value of a node's kind field. This would produce the right dynamic behaviour, but the code could no longer be checked by the standard type checking dialects. A custom dialect that treated these custom pattern objects as types could also solve this problem, but the branding dialect subsumes this solution anyway.

6.2.2 Dialects

Dialects can be defined by expressing the checking as a series of rule blocks (Homer, Jones, et al. 2014). Rule blocks specify which nodes they apply to by typing their input, but this presents a problem to the type checker: if the input is stringly-typed, the type checker cannot determine what the type means and so cannot check the body of the rule. Misuses such as the spelling error below will not be caught until run-time, despite the rule being annotated with what appear to be types.

```
rule { vn : VarNode →
    if (vn.vallue.isEmpty) then {
        vn.raise("All vars must have initial assignments")
    }
}
```

The extended reasoning of the branding allows the type checker to understand the combination of structural and nominal type.

Using structural types is not sufficient to express the necessary run-time matching to discriminate different kinds of nodes. When one type is a structural super-type of another, the super-type will match instances of both types. In the case of class and object AST nodes, the type of object nodes subsumes the type of class nodes, so using the structural ObjectNode type as the pattern in a rule will cause

class nodes to be erroneously matched to that rule as well. Brands can resolve this problem:

```
type ObjectNode = anObjectNode.Type ∩ type {
    body → List⟦Node⟧
}

type ClassNode = aClassNode.Type ∩ type {
    body → List⟦Node⟧
    name → String
}

class classNode(···) → ClassNode is aClassNode { ··· }
```

Objects created by a request to classNode will not be considered to belong to the type ObjectNode, notwithstanding that they possess all of the methods of object nodes. As discussed above, before brands these nodes were distinguished by string fields found in all nodes, outside of the type system. Using fields in this way is clearly sub-optimal, particularly as it sits outside the protection of the type system. Now the structural information is still encoded in the types, but the run-time matching behaves as required.

6.2.3 Exceptions

Representations of run-time errors encode a degree of hierarchy, and must be both created and caught within this hierarchy. For example, a FileNotFoundError may be a specialisation of IOError, which is itself a RuntimeError. An exception handler must be able to declare it wishes to trap all IOErrors, including specialisations. In a nominal language such as Java this behaviour maps naturally onto nominal class inheritance, with a handler for one exception type implicitly trapping all its subtypes by subsumption. In a structurally-typed language this relationship does not exist innately and must be created.

Grace's explicit exception hierarchy leverages the pattern-matching system for handlers. An *exception kind* is an object representing a kind of exception, and includes two methods. The refine method creates a new exception kind as a child of the receiver. The raise method creates an exception object, which is propagated

up the stack until a handler is reached. All exception kind objects are patterns, matching any exception packet derived from itself or its refined descendants.

```
def FileNotFoundError = IOError.refine("File not found")

try {
    if (!exists(path)) then {
        FileNotFoundError.raise("{path} does not exist")
    }
} catch { e : IOError →
    print("An IO error occurred: {e}")
}
```

The catch block above will trap the exception raised in the try block because the exception kind FileNotFound was refined from IOError.

This system is reminiscent of brands and can be placed on firmer footing through their use. An ExceptionKind's match method delegates to the Type object of a brand, and its raise method creates an appropriately-branded exception packet. The structure of the exception kind hierarchy looks like the following:

```
class exceptionKind(name : String)
        branded(aKind : Brand) → ExceptionKind {
    method refine(name : String) → ExceptionKind {
        exceptionKind(name) branded(aKind.extend)
    }

    method raise(message' : String) → None {
        object is aKind {
            inherit exception
            def message is public = message'
        }.raise
    }

    method match(obj : Object) → MatchResult {
        aKind.Type.match(obj)
    }
}
```

The root of the hierarchy is then created with a fresh brand.

```
def Exception = exceptionKind("Exception") branded(brand)
```

In this way brands provide a well-founded structure for an existing sui generis construct of the language. An exceptional behaviour has been replaced with a consistent general-purpose approach that can be applied in user code elsewhere.

6.2.4 Singleton Types and Variants

A singleton type is a type with only a single element, which may or may not be trivial. Singleton types are one way of adding nominal types into a structural language, (we discuss this approach in §6.4) but we find it more advantageous to go in the other direction: to use brands as the means to add singleton types to a language without them. Our sentinel value done is defined as an empty object, then its structural type is type {}, which is inhabited by every object. If Done is to be a proper unit type, with done as its only inhabitant, then we can define:

```
def theDone is confidential = brand
type Done = theDone.Type

def done is public = object is theDone {}
```

As theDone is not publicly available, other modules cannot brand other objects with it, and so done will always be the only inhabitant proper of Done.

Similarly, an empty type can also be constructed by taking the pattern of an anonymous brand, ensuring that no object can ever be branded by it and, by extension, ever be an instance of the resulting type.

```
type None = brand.Type
```

A brand need not be bound to a name to take its Type. Note that the type system does not treat this type as bottom — it is not a subtype of every other type — because only the code flow is responsible for it being empty, and it does not seem worth adding a special case for this syntax given that other brands could well go unused as well, and Grace already has a bottom type.

We can generalise these type forms to any type with any set of *variants*, or tagged sums: each of the variants can be represented by a brand, and the over-

all type is the union of all the brand types (intersected with the relevant structural types). As an example, Grace's Boolean type is just the interface of the true and false objects, and as such the Boolean type is inhabited by any object that satisfies the interface, not just true and false. This can be problematic if we need guarantees about the behaviour of a program using an object of type Boolean. Consider the following expression:

```
t1.ifTrue({ t2 }) ifFalse({ t3 })
```

If the type of t_1 is Boolean, then we might expect that either t_2 or t_3 will be evaluated, but the reality is that we have no such guarantee: t_1 could also be an object whose ifTrue() ifFalse() method executes both blocks, or neither, or multiple times, or stores the blocks for execution in the future.

With brands, we can define a type for booleans that truly only contains the relevant objects:

```
def theTrue = brand
def theFalse = brand

type True = theTrue.Type
type False = theFalse.Type

theTrue.annotateObject(true)
theFalse.annotateObject(false)

type RealBoolean = Boolean ∩ (True ∪ False)
```

Not only do we now know the entire potential of a request to ifTrue() ifFalse() on an object of type RealBoolean, we can also write code involving booleans in a functional style:

```
match (t1)
    case { x : True → t2 }
    case { x : False → t3 }
```

The values true and false already match themselves as patterns so we can already write matching expressions like this, but if t_1 is only typed structurally with Boolean then the type system would not be able to guarantee that using a match like this does not 'fall-off' the bottom when neither case matches. With the brand types

True and False we know that we have covered all possible cases when t_1 is of type RealBoolean.

Each of the variants need not contain a single object, nor do the variants need to have the same interface: the type of a linked list can be described with two brands aNode and theEnd, where Node is inhabited by all of the constructed node objects.

```
type Node = aNode.Type ∩ type { value → E; next → List }
type List = Node ∪ End

class nodeWith(value : E) next(next : List) → Node is aNode { ··· }
```

Note that the return type of nodeWith() next() is not List but Node, which is a subtype of List.

Combining brands with both intersection and union types means that we can encode any algebraic data type and perform case analysis on the resulting constructors as in functional languages such as ML or Haskell. We also have individual types for each of the variants, so we can express more precise combinations of the variants that those languages permit: this is how typing a match expression works, because we can subtract the type on each case from the union type of the object being matched as we examine each branch of the match, and guarantee that we have matched all of the possibilities if the remaining type at the end is ⊥.

6.3 Branded Graceless

We now proceed to extend the Graceless language with brands. As with Graceless, this extension allows the features of brands to be encoded through a translation, rather than directly encoding the design presented thus far. We discuss these distinctions and present the relevant translations as they come up.

We do not include casts in this model, as the semantics require that abstract variables will always be substituted with concrete variables to remain syntactically valid. Given the existing cast transparency property, integrating casts should not be too difficult, but we leave this as future work.

Grammar

$$
\begin{aligned}
D &::= \cdots \mid \beta(x) \mid \eta(x) \mid \tau(x) \mid x \\
a &::= \cdots \mid \beta(x) \mid \eta(x) \mid \tau(x) \mid x \\
d &::= \cdots \mid \beta(x) \mid \eta(x) \mid \tau(x)
\end{aligned}
$$

Figure 6.3.1: Extended grammar for Branded Graceless

6.3.1 Syntax

The grammar for Branded Graceless is defined in Figure 6.3.1. The extension adds a number of new object definitions, as well as corresponding signatures. Variables feature in all of these additions, and variables may now also be used as declarations even though they cannot appear directly as a definition in an object. One of the key differences between Branded Graceless and the earlier design of brands is that *any* object can be used as a brand, not just those created by the brand method, so long as it is abstracted behind a variable.

The form $\beta(x)$ *brands* an object with the brand x, which encodes the use of the annotation list with the is keyword. The form $\eta(x)$ *extends* the brand x, so that if the extending object is used to brand another object (with the β form), the object is also branded with x. The form $\tau(x)$ indicates that an object is a *type* of the brand x: this is the purpose of allowing regular variables to appear as signatures in types, as using an object with the $\tau(x)$ signature adds those objects that have been branded by x to the type. The τ form allows Branded Graceless to separate the brand from the type, as in the conceptual design.

This extended grammar diverges fairly significantly from the design for brands presented up until now, and reflects much of the reasoning that is otherwise internal to the type-checker's implementation. Where the user of brands sees the brand method, the extension methods extend and +, and the Brand type, the formal model provides the necessary foundation for building these forms instead of providing them directly. The static semantics can then reason about this foundation instead. In particular, when the type-checker sees a bound name with the type Brand, it actually allocates a unique type to the name so that it can distinguish

between uses of different brands: this is what the $\eta(x)$ form is for.

One purely syntactic difference is that branding an object includes the brand in the constructor, rather than annotating it: object is $\overline{x}\,\{\,\overline{d}\;t\,\}$ is written object $\{\,\overline{\beta(x)}\;\overline{d}\;t\,\}$. The use of a brand is included in the metavariable d instead of adding the annotation syntax to object constructors because it is easier to extend our existing treatment of method definitions in objects than to add extra syntax to the existing terms (and add new judgements to deal with them too).

The form $\eta(x)$ declares a brand that extends another brand x. As with applying a brand, this form appears in the set of object definitions d such as in the constructor object $\{\,\eta(x)\;t\,\}$. If an object is constructed with one of these definitions in it, and it is used to brand other objects, it will also add the brand of x to it. Such an object uses its own identity as a brand as normal.

Adding the type of a brand to an object with $\tau(x)$ allows the calculus to simulate the pattern object returned by a brand's Type method: the appearance of $\tau(x_1)$ in the definitions of an object x_2 means that using x_2 as an identifier in a match $t \ni x_2\ b_1\ b_2$ will cause the match to enter the b_1 branch if t is branded with x_1, and the b_2 branch if not. The use of $\tau(x)$ in a signature of a type means that the type guarantees any inhabiting object contains the branding definition $\beta(x)$.

The appearance of these forms in the definition of signatures D rather than the top-level types T or structural types S makes sense under consideration that we need to be able to express the result of intersecting the type x with some structural type type $\{\ \cdots\ \}$: the result is type $\{\,x\ \cdots\ \}$. Even though we are using the brands as nominal types, these new definitions still correspond to structural information: that an object satisfies a particular nominal type is just part of its structure, rather than a fundamental component of its construction.

The use of any of these forms requires that any variable they use actually appear in scope (and is not shadowed by an intervening method). Note that since variables can now appear in structural types, each appearance of the type form binds an implicit self reference in the same way that the object construct does. In the following code listings, we use the outer keyword to refer to a surrounding self reference — this is purely for the purposes of textual representation and is a regular self reference in the calculus.

Using these constructs, we can define the Brand type as:

```
Brand = type {
    Type → type { τ(outer) }
    extend → Brand ∩ type { η(outer) }
    +(x : type{ η }) → Brand ∩ type { η(outer) η(x) }
}
```

In this type, the references to outer refer to the self reference bound by the outermost Brand. This variable refers to the particular *value* of the brand inhabiting this Brand type, so Brand actually describes a *family* of types indexed by an object reference, where each object's type depends on its own value. While two variables x_1 and x_2 might both share the Brand type, when requesting the extend method on both of them they each produce an object of distinct type, since the return types of the two methods each depend on the values of x_1 and x_2 respectively.

The brand method is more complicated than the encoding of the Brand type, since we need to be able to implement the extend and + methods recursively. First we define a variant of the method that accepts a brand as an argument, and builds a brand that extends the argument. Since every object can act as a brand, the argument has the type ⊤. First we present the signature of the method:

```
method brand(a : ⊤) → Brand ∩ type { η(a) }
```

Note that return type of the method intersects the structural type Brand with the nominal information that the resulting brand also extends the argument a. The type system for Branded Graceless encodes a limited form of dependent typing, where the return type of a signature can depend on the values of the parameters (parameter types may also depend on the values of earlier parameters). When requesting this brand method, the type system knows that the resulting brand extends the argument, because this is encoded directly into the return type.

We now consider the body of the object returned by the method. Firstly, the object extends the argument a, as required by the return type.

```
η(a)
```

Even though a has the type ⊤, every object acts as a brand, so it is always safe to extend any variable.

Next, the Type method builds the simplest object that satisfies the return type required by the Brand type of the surrounding object.

```
method Type → type { τ(outer) } {
    object { τ(outer) }
}
```

This object isn't a type by itself, but it can be placed inside of a type as a signature and combined with other types using the ∩ and ∪ combinators.

The extend method is implemented simply in terms of the surrounding brand method, using it to build a fresh brand that extends the receiver of the request to extend.

```
method extend → Brand ∩ type { η(outer) } {
    brand(self)
}
```

The surrounding object is referred to by two different variables in this representation, since the use of outer appears behind another binding of self, whereas the request of the brand method does not. In the formal syntax these two variables are not distinct.

Finally, there is the + method. As required by the Brand type, this takes an argument and produces a brand that extends both the receiver and the argument. In order to produce this type, the argument to the method must be a variable, but the argument is actually an object constructor that must be evaluated before it becomes a concrete variable, so the use of the brand method is wrapped in a match construct that is guaranteed to succeed, just to bind the object to a variable.

```
method +(b : ⊤) → Brand ∩ type { η(outer) η(b) } {
    object { η(outer) η(b) } ∋ η(outer) { c → brand(c) } { c → c }
}
```

The 'else' case of the match is absurd (c has type ⊥), so we can just immediately return the parameter of the block and it will take whatever type we need by subsumption.

The brand method presented in the conceptual design can now be defined by overloading the previous definition, this time taking no parameters. The body is exactly the same, except the resulting object does not extend any other brand; this is presented in Figure 6.3.2.

```
method brand → Brand {
  object {
    method Type → type { τ(outer) } { object { τ(outer) } }
    method extend → Brand ∩ type { η(outer) } { brand(self) }
    method +(x : ⊤) → Brand ∩ type { η(outer) η(x) } {
      object { η(outer) η(b) } ∋ η(outer) { c → brand(c) } { c → c }
    }
  }
}
```

Figure 6.3.2: Implementation of brand method in Branded Graceless

The use of the match construct in the definition of the + methods deserves some further explanation. Brand expressions can only be variables x, to avoid any computation occurring in types. Reduction can change types using substitution, but general terms cannot appear in a type. We *can* encode general terms as types by binding them to a parameter first, so for instance the expression object is t_1 { t_2 } can be encoded as:

t_1 ∋ α { z → object { β(z) t_2 } } { z → z }

The form α should be any identifier that definitely appears in the value of t_1, to ensure that the second block is absurd. If this is not possible, a rescue with an immediate raise will also suffice:

(⇑ t_1) ⇈ { z → object { β(z) t_2 } }

All existing type information about z is lost in this encoding. An immediate method request on an object constructor is another alternative:

```
object {
    method apply(z : T1) → T2 { object { β(z) t2} }
}.apply(t1)
```

The types need to be repeated explicitly on the method defined inside of the object constructor in this approach.

As variables are indistinguishable from unqualified zero-argument requests, a syntactically valid use of a brand might reduce to a syntactically invalid state if x

ends up qualified: the typing judgement presented in §6.3.4 ensures that any variable used as a brand is always a parameter reference instead of a method call.

Putting these forms together, we can build an example of a nominal class in Branded Graceless presented in Figure 6.3.3, a class dog that constructs objects of the nominal type Dog, using the brand aDog. This utilises the separation of a brand from its type to ensure that the client of the class can use the nominal type of the class without allowing the client to brand its own objects with the brand of the class.

The outermost object has two (overloaded) apply methods. The first method is the client of the class, and the second constructs the dog class and passes it to the client. In order for the client to be able to type the class object using a nominal type, it must receive the nominal type separately from the class itself, after which the type of the class can depend directly on the value of the type. This is expressed in the signature of the client method: first the Dog object is received as a parameter, and then it is applied as a nominal type to the return type of the new method in the second dog parameter. Because of this arrangement, the class must be responsible for passing itself (and its nominal type) to the client. The return type of the new method is not Dog, but a structural type that includes Dog as a signature.

In this example, the client constructs a new Dog and then promptly forgets whether it was a Dog or a Cat (presumably with a meow method) by passing it to a method that accepts either. This method checks to see if the argument is a Dog, barking if it is, and meowing if not. The body of the method is ill-typed, because the client has forgotten to consider the fact that there might be objects that are *both* a Dog and a Cat: the presence of the brand aDog in the object does not imply the appearance of the method bark, since the Cat could also be branded aDog. In contrast, the 'else' branch of the match is well-typed, because the absence of aDog implies Cat, through the negation of one side of the union type applied to a.

Structural information about an object such as the presence of the bark method *can* be recovered by matching on a brand type, but only through signature subtraction in the 'else' branch of a match construct. The Dog branch just needs to include another match against some sort of discriminator for the Cat type (like another brand type) to perform a (now type-safe) bark in the new 'else' case, and the type system can be trivially satisfied in the new 'then' case by raising an error

```
object {
    // The client code.
    method client(Dog : ⊤, dog : type {
        new ⟶ type { Dog; bark ⟶ ⊤ }
    }) ⟶ ⊤ {
        object {
            method makeNoise(a : type { Dog; bark ⟶ ⊤ } ∪ Cat) ⟶ ⊤ {
                // Barking remains ill–typed.
                a ∋ Dog { d ⟶ d.bark } { c ⟶ c.meow }
            }
        }.makeNoise(dog.new)
    }

    method apply(aDog : Brand) ⟶ ⊤ {
        object {
            // The constructor of this dog class.
            method new ⟶ type { β(aDog); bark ⟶ ⊤ } {
                object {
                    β(aDog)
                    method bark ⟶ ⊤ { ··· }
                }
            }

            // Responsible for passing the type and class to the client.
            method runClient(Type : type { τ(aDog) }) ⟶ ⊤ {
                client(Type, self)
            }
        }.runClient(aDog.Type)
    }
}.apply(brand)
```

Figure 6.3.3: Example of a nominally-typed class and client

$\boxed{\Gamma \vdash D}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash \beta(x)}\ (\text{W-Bnd}) \qquad \frac{\Gamma \ni x : T}{\Gamma \vdash \eta(x)}\ (\text{W-Ext}) \qquad \frac{\Gamma \ni x : T}{\Gamma \vdash \tau(x)}\ (\text{W-Typ}) \qquad \frac{\Gamma \ni x : T}{\Gamma \vdash x}\ (\text{W-Var})$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, z : T_1 \vdash m(\overline{z_i : T_i}) \to T_2}{\Gamma \vdash m(z : T_1, \overline{z_i : T_i}) \to T_2}\ (\text{W-Dep})$$

$$\begin{aligned} &\mathit{identify} : (\textsc{Decl} \cup \textsc{Def}) \to \textsc{Ident} \\ &\mathit{identify}(\beta(x)) = \beta(x) \\ &\mathit{identify}(\eta(x)) = \eta(x) \\ &\mathit{identify}(\tau(x)) = \tau(x) \\ &\mathit{identify}(x) \quad = x \end{aligned}$$

Figure 6.3.4: Well-formedness for Branded Graceless types

complaining that an object cannot be both a Dog and Cat at the same time.

6.3.2 Types

Due to the presence of variables in types, the well-formedness relation needs to be updated to include the requirement that the variables actually appear in scope. Nothing beyond this is required, since every object can act as both a brand and a type (signature): the well-formedness relation does not have to type the variables to check they are being used correctly, as any use of a variable in any signature form is always valid so long as the variable actually appears in scope.

The new rules for the well-formedness judgement are defined in Figure 6.3.4. The judgement has the new form $\Gamma \vdash T$ and overloaded variants for structural types and signatures; only the new rules for signatures are defined here. For the existing rules, the input Γ is threaded through unchanged into all recursive uses of the judgement. As discussed, all of the rules simply check that any used variable can be selected from Γ, using the selection $\Gamma \ni x : T$ defined in §4.4.1. The variable must be typed by a variable binding, not as a signature, and the selection operator ensures that if a local method shadows the variables then it is no longer usable in a signature.

One major caveat in the new well-formedness rules is the way that definitions appear in an object. While we can require in the typing rules that the definitions in the body of an object constructor have unique identifiers, this property is not necessarily preserved by reduction. Consider the perfectly valid term object $\{\, \beta(y)\ \beta(z) \,\}$: there is no reason that the variable z cannot refer to the reference y, so a particular substitution could reduce this object to an ill-formed state:

$$[y/z]\text{object}\ \{\, \beta(y)\ \beta(z) \,\} = \text{object}\ \{\, \beta(y)\ \beta(y) \,\}$$

The identifiers of this object's definitions are not unique, so its type is not well-formed. This isn't actually a problem for the typing, and we could update well-formedness to only consider the identifiers of method definitions and not of the various brand forms, but instead we define substitution to remove any duplicate definitions it would otherwise create in an object, so:

$$[y/z]\text{object}\ \{\, \beta(y)\ \beta(z) \,\} = \text{object}\ \{\, \beta(y) \,\}$$

The same applies to the intersection operator: if a brand form appears in both arguments to an intersection between structural types, it only appears once in the result.

Rule W-Dep handles the types in a signature depending on one of the signature's parameters. The binding of the head of a parameter list is added to the environment if its own type is well-formed, so that parameter may appear in any of the remaining parameter types and the return type. When there are no remaining parameters that need to be added to the environment, the regular Rule W-Sig suffices to finish the judgement.

Extending the type combinators to handle the new forms is straightforward: union is unchanged, since that combinator never examines a type beyond the top-level structure of its union, and intersection just needs to accept the intersection of the new declaration forms when their identifiers are equal. This is trivial, since the new forms only have equal identifiers when the forms themselves are equal. This is defined explicitly in Figure 6.3.5.

Subtyping also needs to be updated. As with the well-formedness relation, the

$$\cap : \textsc{Decl} \times \textsc{Decl} \rightharpoonup \textsc{Decl}$$

$$\begin{array}{lcll}
\overline{m(x_i : T_{i1})} \to T_1 & \cap & \overline{m(x_i : T_{i2})} \to T_2 & = \overline{m(x_i : (T_{i1} \cup T_{i2}))} \to (T_1 \cap T_2) \\
\beta(x) & \cap & \beta(x) & = \beta(x) \\
\eta(x) & \cap & \eta(x) & = \eta(x) \\
\tau(x) & \cap & \tau(x) & = \tau(x) \\
x & \cap & x & = x
\end{array}$$

Figure 6.3.5: Extended declaration intersection for Branded Graceless

presence of variables in types means the judgement must be parameterised by a typing environment Γ. Subtyping the use of brands in β and τ forms requires delegating to the extends relationship between brands with the form η, and subtyping η and plain variables x requires a mutual recursion with a typing judgement in order to type the variables. This restricted typing judgement $\Gamma \vdash_! x : T$ effectively reduces the existing typing rules to just Rules T-Var and T-Sub, but we redefine this restricted relation in order to maintain a separation from the actual typing judgement defined in §6.3.4.

Rule S-Bnd delegates to subtyping on the corresponding η forms, as mentioned. An object branded by x_1 ($\beta(x_1)$) can also be considered as branded by x_2 ($\beta(x_2)$) if the object at x_1 contains a definition that indicates the object extends the brand x_2 ($\eta(x_2)$ or $\eta(x')$ where x' also extends x_2). Similarly, an object that acts as the type of x_1 also acts as the type of x_2 is x_1 is a brand that extends x_2.

Rule S-Ext implements subtyping between brand extension forms. If typing the sub-form $\eta(x_1)$ using the restricted typing judgement $\Gamma \vdash_! x : T$ includes the super-form $\eta(x_2)$, then an object that extends the brand x_1 also extends x_2. Rule S-Var follows a similar logic to have a branding declaration $\beta(x_1)$ subtype a nominal type x_2 if x_2 is a type for a brand that x_1 extends.

Rules S-Typ and S-Rfl encode reflexivity for τ forms and variables, so long as the type is well-formed. The τ forms are invariant under subtyping, since otherwise no brand could otherwise safely inhabit the resulting type: if they respected subtyping of brands, then there would be no guarantee that a particular brand satisfied the requirement of the type because the actual definition $\tau(x_1)$ could be more specific than the signature.

$\boxed{\Gamma \vdash T <: T}$

$$\text{(S-Uni)}\quad \frac{\forall S_i.\ \exists S_j.\ \Gamma, \mathsf{self} : \bigcup \overline{S_i} \vdash S_i <: S_j}{\Gamma \vdash \bigcup \overline{S_i} <: \bigcup \overline{S_j}}$$

$\boxed{\Gamma \vdash D <: D}$

$$\text{(S-Bnd)}\quad \frac{\Gamma \vdash \eta(x_1) <: \eta(x_2)}{\Gamma \vdash \beta(x_1) <: \beta(x_2)} \qquad \text{(S-Ext)}\quad \frac{\Gamma \vdash_! x_1 : \mathsf{type}\,\{\,\eta(x_2)\,\}}{\Gamma \vdash \eta(x_1) <: \eta(x_2)} \qquad \text{(S-Typ)}\quad \frac{\Gamma \ni x : T}{\Gamma \vdash \tau(x) <: \tau(x)}$$

$$\text{(S-Rfl)}\quad \frac{\Gamma \ni x : T}{\Gamma \vdash x <: x} \qquad \text{(S-Var)}\quad \frac{\Gamma \vdash_! x_2 : \mathsf{type}\,\{\,\tau(x_3)\,\} \qquad \Gamma \vdash \eta(x_1) <: \eta(x_3)}{\Gamma \vdash \beta(x_1) <: x_2}$$

$$\text{(S-Dep)}\quad \frac{\Gamma \vdash T_3 <: T_1 \qquad \Gamma, z : T_3 \vdash m(\overline{z : T_{i1}}) \to T_2 <: m(\overline{z : T_{i2}}) \to T_4}{\Gamma \vdash m(z : T_1, \overline{z : T_{i1}}) \to T_2 <: m(z : T_3, \overline{z : T_{i2}}) \to T_4}$$

$\boxed{\Gamma \vdash_! t : T}$

$$\text{(V-Var)}\quad \frac{\Gamma \ni x : T}{\Gamma \vdash_! x : T \cap \mathsf{type}\,\{\,\eta(\mathsf{self})\,\}} \qquad \text{(V-Slf)}\quad \frac{\Gamma \vdash_! x : T}{\Gamma \vdash_! x : [x/\mathsf{self}]T}$$

$$\text{(V-Sub)}\quad \frac{\Gamma \vdash_! t : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash_! t : T_2}$$

Figure 6.3.6: Subtyping extended with brands

Like its well-formedness counterpart, Rule S-Dep extends the existing Rule S-Sig to add a parameter from the signature to the typing environment, since it might appear in types of the following parameters and return types. As usual, the parameter names in signatures are not significant between different signatures, so we assume that the names are the same in both signatures. Rule S-Dep only processes the parameter at the head of the lists, and then requires that the same signatures with the tail of the parameter lists are also compatible. When there are no remaining parameters that are depended on by the types (including simply no parameters at all), the regular Rule S-Sig can finish the subtyping.

Some of the lemmas from Chapter 4 need extending to prove that they remain true for this extended definition.

Lemma 34 (Subtyping is reflexive)**.**

$$\frac{\Gamma \vdash T}{\Gamma \vdash T <: T}$$

Proof. Coinduction over the derivation of $\Gamma \vdash T$. The new forms $\tau(x)$ and x are immediate from their corresponding rules because their subtyping is invariant. The two interesting cases are:

$\eta(x)$ Rule V-Var adds the signature $\eta(x)$ to the type of x, so x can trivially be given the type $\mathsf{type}\ \{\ \eta(x)\ \}$ in Rule S-Ext.

$\beta(x)$ Immediate from the previous case by Rule S-Bnd.

The remaining cases trivially follow from the coinduction hypothesis and the proof that $\Gamma \vdash D$. □

Transitivity is less obvious, because it is encoded behind the auxiliary $\Gamma \vdash_! x : T$ judgement and its subsumption rule. First we need to show that subtyping is preserved in a narrower environment, to deal with dependent signatures.

Lemma 35 (Subtype environment narrowing)**.**

$$\frac{\Gamma_1, z : T, \Gamma_2 \vdash T_1 <: T_2 \qquad \Gamma_1 \vdash T' <: T}{\Gamma_1, z : T', \Gamma_2 \vdash T_1 <: T_2}$$

Proof. Coinduction on the derivation of $\Gamma_1, z : \mathsf{T}, \Gamma_2 \vdash \mathsf{T}_1 <: \mathsf{T}_2$, with a case analysis on the last step.

(S-Ext) If the type of x_1 has changed, then the original rule with $\eta(x_2)$ can be recovered under Rule V-Sub. If the type of x_2 has changed, then this does not change the type of x_1: either $\eta(x_2)$ is still present in the type of x_1 in Γ or some other variable that transitively references x_2.

(S-Typ) A change in the type of x does not affect its presence in the environment.

(S-Rfl) A change in the type of x does not affect its presence in the environment.

(S-Var) If the type of x_1 has changed, then Rule S-Ext still applies by induction. If the type of x_2 has changed, then the original rule with $\tau(x_3)$ can be recovered under Rule V-Sub.

The remaining cases follow immediately from the coinduction hypothesis. □

This is used to show transitivity.

Lemma 36 (Subtyping is transitive)**.**

$$\frac{\Gamma \vdash \mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3 \qquad \Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3}$$

Proof. Mutual coinduction on the derivations of $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$ and $\Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3$, with a case analysis on the last step of the latter.

(S-Ext) x_1 has the type $\eta(x_2)$, and x_2 has the type $\eta(x_3)$. The goal $\Gamma \vdash \eta(x_1) <: \eta(x_3)$ can be shown by Rule S-Ext with a proof that $\Gamma \vdash_! x_1 : \eta(x_3)$, through a combination of the existing proof that $\Gamma \vdash_! x_1 : \eta(x_2)$ and Rule V-Sub with the existing proof that $\Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3$.

(S-Var) $\Gamma \vdash \eta(x_1) <: \eta(x_2)$ by the inversion of Rule S-Bnd on the former proof, so Rule S-Var applies on the coinduction hypothesis that the two subtype proofs combine to prove that $\Gamma \vdash \eta(x_1) <: \eta(x_3)$.

(S-Dep) Immediate from induction and Lemma 35.

The remaining cases trivially follow from the coinduction hypothesis and the proof that $\Gamma \vdash D_1, D_2, D_3$. □

Some of the lemmas about declarations from §4.4.3 still hold for signatures, but not for the updated declarations. One such lemma is that subtyping between declarations implies that their identifiers are equal: this is clearly no longer the case, since Rule S-VAR subtypes $\beta(x)$ with x. As such, we redefine the lemma for signatures only.

Lemma 37 (Signature subtype implies identifier equality)**.**

$$\frac{m_1(\overline{z_{1i} : T_{1i}}) \to T_1 <: m_2(\overline{z_{2j} : T_{2j}}) \to T_2}{\mathit{identify}(m_1(\overline{z_{1i} : T_{1i}}) \to T_1) = \mathit{identify}(m_2(\overline{z_{2j} : T_{2j}}) \to T_2)}$$

Proof. Immediate from Rule S-SIG. □

Since the definition of intersection has also been extended, it is worth reiterating the lemma that it always produces a sub-type of both of its inputs.

Lemma 38 (Intersection produces sub-type)**.**

$$\frac{T_1 \cap T_2 = T_3}{T_3 <: T_1 \qquad T_3 <: T_2}$$

Proof. The proof remains the same as in Lemma 13, since the new forms only appear in the result if they appeared in the corresponding structural part of either input: if one of these forms appears in the super-type, and so must be satisfied in the sub-type by Rule S-STR, then an equal form is guaranteed to appear in the corresponding structural type of the sub-type by the definition of the $\cap$ operation. Equal forms are always subtypes by Lemma 34. □

6.3.3 Dynamic Semantics

While most of the existing dynamic semantics of Graceless are sufficient to describe the behaviour of Branded Graceless programs, the matching mechanism needs to be updated to account for the several different new identifier forms. The updated reduction rules are presented in Figure 6.3.7, changing Rules E-FST and E-SND to use a new identifier lookup judgement $\sigma \mid y \ni a$ instead of just looking up the

$\boxed{\sigma \mid t \longrightarrow \sigma \mid t}$

$$\text{(E-Fst)}\ \frac{\sigma \mid y \ni a}{\sigma \mid y \ni a\,\{\,z \to t\,\}\,b \longrightarrow \sigma \mid [y/z]t} \qquad \text{(E-Snd)}\ \frac{\sigma \mid y \not\ni a}{\sigma \mid y \ni a\,b\,\{\,z \to t\,\} \longrightarrow \sigma \mid [y/z]t}$$

$\boxed{\sigma \mid y \ni a}$

$$\text{(L-Mem)}\ \frac{\sigma(y) = \overline{d} \qquad a \in \overline{\mathit{identify}(d)}}{\sigma \mid y \ni a} \qquad \text{(L-Idn)}\ \frac{}{\sigma \mid y \ni \eta(y)} \qquad \text{(L-Ext)}\ \frac{\eta(y_2) \in \sigma(y_1) \qquad \sigma \mid y_2 \ni \eta(y_3)}{\sigma \mid y_1 \ni \eta(y_3)}$$

$$\text{(L-Bnd)}\ \frac{\beta(y_2) \in \sigma(y_1) \qquad \sigma \mid y_2 \ni \eta(y_3)}{\sigma \mid y_1 \ni \beta(y_3)} \qquad \text{(L-Var)}\ \frac{\tau(y_3) \in \sigma(y_2) \qquad \sigma \mid y_1 \ni \beta(y_3)}{\sigma \mid y_1 \ni y_2}$$

Figure 6.3.7: Extended reduction rules for brands

identifier in the store at y. The application $\sigma \mid y \not\ni a$ indicates the judgement does not hold, rather than applying a separate judgement. Rule L-Mem of this new judgement encodes the original behaviour of direct lookup in the store.

Rule L-Idn encodes the concept that every object is its own brand: the reference y automatically satisfies a lookup for an object that brands an object with y, since it does so trivially without needing to contain any definitions. Rule L-Ext and Rule L-Bnd encode the transitive nature of brands through extension forms η: a definition with argument y_3 appears in the object at y_1 if there is a definition of the same form at y_1 with an argument y_2, and y_2 extends y_3 as a brand. This is why all three of the rules delegate to searching for η forms. Subsequent searches for an η form can recurse on Rule L-Ext, ultimately terminating in either Rule L-Mem or Rule L-Idn.

Rule L-Var handles the case where the identifier is a plain variable, which indicates the use of a type object. In order to satisfy this type, the matched object y_1 must be branded by one of the variable arguments in a τ forms in the matching object y_2. Rule L-Var delegates to a search for branding forms β for a variable that appeared in one of the τ forms, which will in turn delegate to searches for η forms

if the brands do not appear immediately in the object.

As an algorithm, this judgement (and its negation) correspond to a nominal type check in a multiple-inheritance language: if the object does not immediately satisfy the given type, it is necessary to check if the object satisfies some extending type instead, by searching upwards from every nominal type the object is declared to inhabit. The negation fails if a complete search is exhausted without finding the original type. The judgement permits a richer analysis than a typical 'instance-of' check, since a program can also examine if an object is the type itself, or the brand that adds the type to other objects.

The resulting behaviour is as expected: first of all, if a brand x is in scope, the presence of the brand in a term t can be checked for directly with the matching construct $t \ni \beta(x)\ b_1\ b_2$. If the value of t is branded $\beta(x)$ then the control flow will enter b_1 by Rule E-Fst and L-Mem. If the value of t is not branded $\beta(x)$, but is branded $\beta(x')$ where x' extends x with $\eta(x)$, then the match still enters b_1, this time through Rule L-Bnd on top of the other rules. Any number of intervening η forms can separate x' and x, with the path connected by Rule L-Ext.

If z corresponds to a variable with the form $\tau(x)$, it can also be used to check for the presence of a branding with x in the same way as $\beta(x)$ does, but x itself need not be in scope, as in $t \ni z\ b_1\ b_2$. If $t \ni \beta(x)\ b_1\ b_2$ enters b_1, then so does matching on z, assuming $\tau(x)$ is the only $\tau\alpha u$ form that appears in z (extra τ forms correspond to a series of successful matches on β forms). The derivation of the lookup judgement is the same, except that Rule L-Var handles the initial lookup for z by invoking the lookup on $\beta(x)$ as described above.

Handling a match on η and τ forms are not strictly necessary to encode the conceptual design, which permits matching on the Brand and Pattern types as usual but provides no mechanism for asking if a brand or type entirely subsumes another. Such a mechanism could be added if it turned out to be useful: for instance, the Brand type could include an extends method that takes another brand and returns a boolean indicating if branding an object with the receiver also brands that object with the argument.

6.3.4 Static Semantics

The extended typing rules for Branded Graceless are presented in Figure 6.3.8. Along with the additional rules for the new syntactic forms, the rules for typing requests need to be updated to handle types in signatures that depend on parameters. Rules T-R/U and T-R/Q have been updated to delegate the typing of the parameters to the new judgement $\Gamma \vdash (\overline{t}) \rightharpoonup (\overline{z : T}) : T \rightharpoonup T$, which works to match the argument terms $\overline{t}$ to the parameter bindings $\overline{z : T}$ and introduce them into the remaining types when possible. The resulting type of these requests is generated by this judgement.

Both Rule M-Var and Rule M-Trm take the head of both the term list and parameter binding list, and check if the term matches the parameter type as the typing judgement did in plain Graceless. Both rules also continue the match between the tail of the two lists. Rule M-End terminates the matching when both lists are empty, producing the included return type. If none of the argument terms are variables, then the result is exactly as under Graceless, just with the judgement manually stepping through the arguments and parameters.

The difference is in Rule M-Var, which applies if the next argument term is a variable. The corresponding parameter for this argument is substituted into both the remaining parameter types and the return type. Since Rule M-End requires that the final type is well-formed, all of the parameters that the return type depended on in the signature *must* be replaced by argument terms, since those parameters are no longer in scope outside of the signature, so the use of Rules M-Trm and M-Var is not ambiguous.

The new Rule T-Dep corresponds to Rule S-Dep in the subtyping judgement, typing a method by adding the parameter bindings into the environment before typing the body of the method t. The remaining rules trivially type the new definition forms so long as their variable arguments are bound in the typing environment. While the rest of the Graceless typing rules are unchanged, the *ground* function is trivially extended to be the identity function on the new signature forms in order to ensure that typing a match is still sensible.

We examine components of a typing judgement for the example presented in Figure 6.3.3, by presenting the derivations of the judgement. The derivation for typ-

$\boxed{\Gamma \vdash t : T}$

$$\text{(T-Var)}\ \frac{\Gamma \ni x : T}{\Gamma \vdash x : T \cap \mathsf{type}\,\{\,\eta(\mathsf{self})\,\}} \qquad \text{(T-Slf)}\ \frac{\Gamma \vdash x : T}{\Gamma \vdash x : [x/\mathsf{self}]T}$$

$$\text{(T-R/U)}\ \frac{\Gamma \ni m(\overline{x : T_i}) \to T \qquad \Gamma \vdash (\overline{t_i}) \rightharpoonup (\overline{x : T_i}) : T \rightharpoonup T'}{\Gamma \vdash m(\overline{t_i}) : T'}$$

$$\text{(T-R/Q)}\ \frac{\Gamma \vdash t : \mathsf{type}\,\{\,m(\overline{z : T_i}) \to T\,\} \qquad \Gamma \vdash (\overline{t, t_i}) \rightharpoonup (\mathsf{self} : \top, \overline{z : T_i}) : T \rightharpoonup T'}{\Gamma \vdash t.m(\overline{t_i}) : T'}$$

$\boxed{\Gamma \vdash (\overline{t}) \rightharpoonup (\overline{z : T}) : T \rightharpoonup T}$

$$\text{(M-Var)}\ \frac{\Gamma \vdash_! x : T_1 \qquad \Gamma \vdash (\overline{t_i}) \rightharpoonup (\overline{z_i : [x/z]T_i}) : [x/z]T_2 \rightharpoonup T_2'}{\Gamma \vdash (x, \overline{t_i}) \rightharpoonup (z : T_1, \overline{z_i : T_i}) : T_2 \rightharpoonup T_2'}$$

$$\text{(M-Trm)}\ \frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash (\overline{t_i}) \rightharpoonup (\overline{z_i : T_i}) : T_2 \rightharpoonup T_2'}{\Gamma \vdash (t, \overline{t_i}) \rightharpoonup (z : T_1, \overline{z_i : T_i}) : T_2 \rightharpoonup T_2'} \qquad \text{(M-End)}\ \frac{\Gamma \vdash T}{\Gamma \vdash () \rightharpoonup () : T \rightharpoonup T}$$

$\boxed{\Gamma \vdash d : D}$

$$\text{(T-Dep)}\ \frac{\Gamma \vdash T_1 \qquad \Gamma, z : T_1 \vdash \mathsf{method}\ m(\overline{z_i : T_i}) \to T_2\,\{\,t\,\} : m(\overline{z_i : T_i}) \to T_2}{\Gamma \vdash \mathsf{method}\ m(z : T_1, \overline{z_i : T_i}) \to T_2\,\{\,t\,\} : m(z : T_1, \overline{z_i : T_i}) \to T_2}$$

$$\text{(T-Bnd)}\ \frac{\Gamma \ni x : T}{\Gamma \vdash \beta(x) : \beta(x)} \qquad \text{(T-Ext)}\ \frac{\Gamma \ni x : T}{\Gamma \vdash \eta(x) : \eta(x)} \qquad \text{(T-Typ)}\ \frac{\Gamma \ni x : T}{\Gamma \vdash \tau(x) : \tau(x)}$$

$$\mathit{ground} : \textsc{Ident} \to \textsc{Type}$$
$$\mathit{ground}(\langle m, n\rangle) = \mathsf{type}\,\{\,m(\overline{x_i : \bot}^{\,i \le n}) \to \top\,\}$$
$$\mathit{ground}(D) = \mathsf{type}\,\{\,D\,\}$$

Figure 6.3.8: Typing extended with brands

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \ni \mathsf{aDog} : \mathsf{Brand}}}{\Gamma, \mathsf{self} : \mathsf{type}\,\{\,\beta(\mathsf{aDog})\,\} \vdash \beta(\mathsf{aDog}) : \beta(\mathsf{aDog})}\,(\text{T-Bnd})}{\Gamma \vdash \mathsf{object}\,\{\,\beta(\mathsf{aDog})\,\} : \mathsf{type}\,\{\,\beta(\mathsf{aDog})\,\}}\,(\text{T-Obj})$$

Figure 6.3.9: Derivation for typing a new dog object

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash_! \mathsf{Type} : \mathsf{type}\,\{\,\tau(\mathsf{aDog})\,\}}\,(\text{S-Var})}{\Gamma \vdash \beta(\mathsf{aDog}) <: \mathsf{Type}}}{\dfrac{\vdots}{\Gamma \vdash \mathsf{self} : [\mathsf{Type}/\mathsf{Dog}]\mathsf{type}\,\{\cdots\}}\,(\text{T-Var})} \qquad \Gamma \vdash () \rightharpoonup () : \top \rightharpoonup \top}{\Gamma \vdash (\mathsf{self}) \rightharpoonup (\mathsf{dog} : [\mathsf{Type}/\mathsf{Dog}]\mathsf{type}\,\{\cdots\}) : \top \rightharpoonup \top}\,(\text{M-Trm})}{\Gamma \vdash (\mathsf{Type}, \mathsf{self}) \rightharpoonup (\mathsf{Dog} : \top, \mathsf{dog} : \mathsf{type}\,\{\cdots\}) : \top \rightharpoonup \top}\,(\text{M-Var})}{\Gamma \vdash \mathsf{client}(\mathsf{Type}, \mathsf{self}) : \top}\,(\text{T-R/U})$$

Figure 6.3.10: Derivation for typing the client request

ing the body of the new method in the definition of the dog object is presented in Figure 6.3.9, ignoring the structural components of the object and type (the bark method). The object must satisfy the type type { β(aDog) }: it does so immediately, because it contains the same definition, and aDog appears in the typing environment. That it has the type Brand is irrelevant, as any type will satisfy the rule.

Of more interest is the use of the client method, whose second parameter has a type that depends on the first. A simplified derivation of its typing is presented in Figure 6.3.10, which proceeds by typing the request with Rule T-R/U, that in turn delegates to the term and parameter matching judgement.

Since the type of the dog parameter depends on the value of the Dog parameter, the first argument is matched with Rule M-Var. The premise that Type satisfies the type $\top$ is omitted, since it is trivial. The key point here is that, when reapplying the judgement to also match the tail of the parameter list, the name Dog is replaced with Type in the remaining parameter type (the same happens to the return type, but there is no use of Dog in $\top$).

The tail is matched with Rule M-Trm: since self is a variable, either Rule M-

VAR or Rule M-TRM could apply, but since there is no use of self in the return type of the signature the simple Rule M-TRM is the easier option. The matching is terminated by Rule M-END, and the only remaining premise is that self can be typed with the result of the previous substitution. Such a proof proceeds through the structure of the type until it needs to prove that $\beta(\mathsf{aDog})$ is compatible with Type, in order to show that the new method of self is compatible with the requirements of the type annotation on the dog parameter. This succeeds by Rule S-VAR, since Type contains a τ form for the aDog brand (the second premise, which is trivially fulfilled by Lemma 34). If the substitution of Dog for Type had not been applied, this subtyping judgement would have been nonsensical, since Dog does not even appear in the typing environment.

6.3.5 Properties

Branded Graceless does not introduce any new top-level terms to the language: the major complication to the type system is that the added type forms can include variables from the typing environment. The primary threat to the type soundness of Branded Graceless as an extension to the type-safe Graceless language is the behaviour of dependent signatures and other dependencies on variable values. Execution cannot get 'stuck' on any of the new definitions, so progress is not really a problem.

The existing typing of the matching construct — which is not changed by the extension except to trivially include the new forms into the *ground* function — and its use of subtraction in typing the 'else' branch can also pose a problem. Since the subtraction removes any structural variants that contain the failed match's identifier from the type of the matched object, it is important that the judgement $\sigma \mid \mathfrak{y} \ni \mathfrak{a}$ correctly discriminates on the identifier $\mathfrak{a}$. If not, then the subtraction may remove type variants from the union that are still valid, in the worse case assigning the block parameter of the 'else' branch the type $\bot$ when there is still a possibility of it being inhabited.

Both of these threats are potential problems with preservation, not progress, so we prove the simpler case of progress first. The relevant changes are the slightly modified variable typing, new typing rules for requests, and the new reduction

rules for matching. Firstly, the canonicity of forms lemma no longer holds, because every variable x can be given the type $\mathsf{type}\,\{\,\eta(x)\,\}$ even if no such definition appears in the corresponding object. Progress only requires canonicity for signatures, not all declarations, so the lemma is updated to only apply for signatures.

Lemma 39 (Canonicity of forms (signatures))**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : \mathsf{type}\,\{\,m(\overline{x : T_i}) \to T\,\}}{d \in \sigma(y) \qquad \mathit{signature}(d) <: m(\overline{x : T_i}) \to T}$$

Proof. As for Lemma 19, using the updated transitivity lemma (Lemma 36). □

Progress follows from this result.

Lemma 40 (Typing implies progress)**.** *For any program* $\langle \sigma, t \rangle$*, if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : T$ *then either:*

- $\exists v.\ t = v$
- $\exists v.\ t = \Uparrow v$
- $\exists \sigma'\ t'.\ \sigma \mid t \longmapsto \sigma \mid t'$

Proof. By induction on the derivation of the proof that $\Gamma \vdash t : T$, with a case analysis on the last step.

(T-Mch) The choice between Rules E-Fst and E-Snd still only depends on the success of a judgement or its negation, so either one must apply to any match that is ready to be eliminated.

(T-R/U) As in Lemma 20, this still forms a contradiction.

(T-R/Q) As in Lemma 20, with the two sub-lemmas updated to Lemma 39 and 37.

The remaining cases are unchanged from Lemma 20. □

The updated preservation proof is more detailed, mostly thanks to the presence of variables in the typing environment. All of the preservation sub-lemmas need updating, and first we need to show that substitution does not affect typing.

Since subtyping now occurs in the context of a typing environment, we also need to show that a value substitution into a *type* preserves the existence of subtyping relations, so that when the arguments of a request are substituted into the body of a method, dependent substitutions into their own types do not affect the subtyping relation between the types of the arguments and the parameter type annotations.

Lemma 41 (Value substitution preserves subtyping)**.**

$$\frac{\Gamma_1 \vdash_! v : T \qquad \Gamma_1, z : T, \Gamma_2 \vdash T_1 <: T_2 \qquad \Gamma_2 \not\ni \langle z, 0 \rangle}{\Gamma_1, [v/z]\Gamma_2 \vdash [v/z]T_1 <: [v/z]T_2}$$

Proof. Coinduction on the derivation that $\Gamma \vdash T_1 <: T_2$, with a case analysis on the last step.

(S-Ext) If x_1 is substituted, then immediate from $\Gamma \vdash_! v : T$. If x_2 is substituted, then it follows that the type of x_1 has changed in $[v/z]\Gamma_2$, since the environment cannot forward-reference variables.

(S-Typ) Immediate from the inversion of $\Gamma \vdash_! v : T$.

(S-Rfl) Immediate from the inversion of $\Gamma \vdash_! v : T$.

(S-Var) If x_1 is substituted, then immediate from the coinduction hypothesis. If x_2 is substituted, then immediate from $\Gamma \vdash_! v : T$.

The remaining cases follow immediately from the coinduction hypothesis. □

Then we have an updated preservation lemma for value substitution, where the substitution proceeds into the type as well (since the variable might also appear there).

Lemma 42 (Value substitution preserves typing)**.**

$$\frac{\Gamma_1 \vdash v : T_1 \qquad \Gamma_1, z : T_1, \Gamma_2 \vdash t : T_2 \qquad \Gamma_2 \not\ni \langle z, 0 \rangle}{\Gamma_1, [v/z]\Gamma_2 \vdash [v/z]t : [v/z]T_2}$$

Proof. The proof is mostly the same as in Lemma 21, though with Lemma 41 necessary in any applications of subsumption. Where z appears in T_2, the relevant typing rules are preserved by the inversion of $\Gamma \vdash v : T_1$ standing in for $z : T_1$ from the environment. □

The variable typing judgement is a subset of the full typing judgement: if we can demonstrate that $\Gamma \vdash_! t : T$, then we can prove that $\Gamma \vdash t : T$.

Lemma 43 (Typing subsumes variable typing)**.**

$$\frac{\Gamma \vdash_! t : T}{\Gamma \vdash t : T}$$

Proof. Immediate translation between the equivalent rules. □

The modifications to the *ground* auxiliary function and the addition of the $\sigma \mid y \ni a$ judgement mean that we also need to update the lemma that adding ground preserves the typing of a value.

Lemma 44 (Adding ground preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \sigma \mid y \ni a}{\Gamma \vdash y : T \cap \mathit{ground}(a)}$$

Proof. Induction on the derivation that $\sigma \mid y \ni a$, with a case analysis on the last step.

(L-Mem) Rule V-Var is implied by the proof that $\vdash \sigma : \Gamma$.

(L-Idn) Immediate from Rule V-Var.

(L-Ext) We know that $\eta(y_2) \in \sigma(y)$, and by induction $\Gamma \vdash_! y_2 : \mathsf{type}\,\{\,\eta(y_3)\,\}$. We can construct a type for y that contains $\eta(y_2)$, then add $\eta(y_3)$ with Rules V-Sub and S-Ext.

(L-Bnd) We know that $\beta(y_2) \in \sigma(y)$, and by induction $\Gamma \vdash_! y_2 : \mathsf{type}\,\{\,\eta(y_3)\,\}$. We can construct a type for y that contains $\beta(y_2)$, then add $\beta(y_3)$ with Rules V-Sub, S-Bnd, and S-Ext.

(L-Var) We know that $\tau(y_3) \in \sigma(y_2)$, and by induction $\Gamma \vdash_! y_1 : \mathsf{type}\,\{\,\beta(y_3)\,\}$. From this we can show $\Gamma \vdash_! y_2 : \mathsf{type}\,\{\,\tau(y_3)\,\}$, and Lemma 34 gives us $\Gamma \vdash \eta(y_3) <: \eta(y_3)$ to build Rule S-Var into Rule V-Sub.

This analysis proves $\Gamma \vdash_! y : T \cap \mathit{ground}(a)$ for the purposes of induction; the goal follows from Lemma 43. □

While subtraction was not modified, the existing lemma that a subtraction also preserves typing *is* affected by the new judgement $\sigma \mid y \ni a$ replacing the straightforward lookup.

Lemma 45 (Subtraction preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \sigma \mid y \not\ni a}{\Gamma \vdash y : T - a}$$

Proof. We show that any structural type subtracted by a is not a relevant super-type of y, by case analysis of a with a view to prove that there is no subtyping rule that applies to the relevant declaration.

$\langle m, n \rangle$ Unchanged from Lemma 25.

$\eta(y_2)$ The form itself cannot appear, nor any sub-declaration $\eta(y_3)$ that would permit Rule S-Ext to apply.

$\beta(y_2)$ The form itself cannot appear, nor any sub-declaration $\beta(y_3)$ that would permit Rule S-Bnd through Rule S-Ext to apply.

$\tau(y_2)$ The form itself cannot appear, so Rule S-Typ cannot apply.

y_2 The form itself cannot appear, so Rule S-Rfl cannot apply, nor can any sub-declaration $\beta(y_3)$ appear that would permit Rule S-Var to apply.

Since the structural type cannot be a super-type of the true type of y, some other member of the union must be providing the proof that $\Gamma \vdash y : T$, so the subtraction may proceed without affecting the typing. □

With these lemmas in hand, we can proceed to a proof of preservation.

Lemma 46 (Reduction preserves typing)**.**

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash t : T \qquad \sigma \mid t \longmapsto \sigma' \mid t'}{\vdash \sigma' : \Gamma' \qquad \Gamma' \vdash t' : T}$$

Proof. Induction on the derivation of the proof that $\sigma \mid t \longmapsto \sigma' \mid t'$, with a case analysis on the last step.

(E-Obj) By trivial extension of the existing proof with the typing rules for the new declaration forms. The addition of the form $\eta(y)$ for the newly allocated reference y does not affect the proof, by Lemma 38 and Rule T-Sub.

(E-Req) On top of the changes to Lemma 21, the old proof is affected by the fact that the inversion of Rule T-R/Q no longer proves that $\Gamma \vdash \overline{v : T_i}$, as each of the values v might have its type modified by the substitution of some of the earlier values. Lemma 41 converts this inversion of the $\Gamma \vdash (\overline{t}) \rightharpoonup (\overline{z : T}) : T \rightharpoonup T$ judgement to prove that the types of the values v still subtype the types in the signature, and can therefore apply to Lemma 42 when they are substituted both into the body of the method and its return type.

(E-Fst) As before, with the new Lemma 44.

(E-Snd) As before, with the new Lemma 45.

The remaining cases are unchanged from Lemma 26. □

As with Graceless, the progress and preservation lemmas combine to give us type soundness.

Theorem 47 (Well-typed programs don't get stuck)**.** *For any Branded Graceless program* $\langle \sigma, t \rangle$*, if* $\vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : T$*, then either:*

- $\exists v.\ t = v$*, so* $\Gamma \vdash v : T$
- $\exists v.\ t = \Uparrow v$*, so* $\Gamma \vdash \Uparrow v : T$
- $\exists \sigma'\ t'.\ \sigma \mid t \longmapsto \sigma \mid t'$*, with* $\vdash \sigma' : \Gamma'$ *and* $\Gamma' \vdash t' : T$

Proof. Immediate from Lemmas 40 and 46. □

6.4 Discussion

Another option to support nominal types in any structural system is by the addition of unique method names into the type, and include empty implementations of these methods into the objects which are expected to fulfil this type. While the 'phantom method' approach works in theory, it is difficult to implement in a way that preserves the necessary encapsulation goals of nominal typing. If the methods are provided manually then the developer must provide method names that will not be used anywhere else in the program, and the encapsulation is trivially bypassed by adding the appropriate methods to other, external objects. If the methods are produced automatically, then either the branding mechanism cannot be private because the name generation is globally accessible, or the automatic names must be indexed by something (presumably the module in which the branding appears), in which case brands cannot be shared because no other module may perform the same branding.

A unique singleton type as the return type of a specific signature name reserved for marking a brand in a type is another approach (intersecting singleton types together when there are multiple brands forming the type), but it suffers from the same problem in that the brand and type cannot be exposed separately: the singleton type must be exposed for the type of the signature to make sense, but if it is exposed it can trivially be applied to other signatures. Neither mechanism can simultaneously service both public construction and private implementation. Our brands, in contrast, provide a fine-grained mechanism for providing access to the branding mechanism components.

Branding provides a partial solution to the invalidation of the gradual guarantee (Siek, Vitousek, Cimini, et al. 2015) of structural type tests in gradually typed code (Boyland 2014), as testing an object against a brand pattern at run-time is always definitive, and is not affected by type annotations. As an extension, branding is not pervasive among objects, and so using brand patterns is only applicable to objects which have been explicitly branded. Removing reified types from the language (another proposed solution) while retaining the existing object instance rules would remove the consistency of brand patterns (as both static entities and run-time objects) alongside structural types.

Compared to standard nominal class declarations, the branding mechanism is necessarily verbose, requiring a manual separation of the brand from its type (mirroring the separation between classes and structural types in Grace). This verbosity is mostly a product of the fact that brands have been implemented without modifying the language syntax or semantics, but it also serves a purpose in demonstrating that it is not the natural mechanism for typing in Grace: structural typing is sufficient for most purposes, and it is only special cases (as seen in §6.2) where branding should apply. It is conceivable that a more terse mechanism for direct class/type declarations could exist:

```
class Shape is nominal { ... }
```

Adding nominal classes directly defies Grace's design goal of maintaining a separation of type and implementation (Black, K. B. Bruce, Homer, and Noble 2012).

Certainly the encoding of algebraic data types that we presented in §6.2.4 is very verbose. Consider the definition of a common type in the Haskell language:

```
data NumList = Cons Number NumList | Null
```

Compare this to its translation into brands in Grace:

```
def aCons = brand
def theNull = brand

type Cons = aCons.Type ∩ type { value → Number; next → NumList }
type Null = theNull.Type

type NumList = Cons ∪ Null

def null = object is theNull {}

class cons(x : Number) onto(xs : NumList) → Cons is aCons {
    def value is public = x
    def next is public = xs
}
```

This verbosity is more an indication of the unsuitability of this coding style to the Grace language than it is a mark against the design of brands. Data structure design like this is not considered good object-oriented design, and branching on potential variants of data in Grace (and other pure object-oriented languages like Smalltalk)

is often achieved by passing multiple blocks to methods, such as with the Boolean's ifTrue() ifFalse() method. Brands only need enter the equation if the user needs some guarantee about the behaviour of requesting such a method.

6.4.1 Comparison to Related Work

There are three recently developed formal languages that our model should be compared with. The first is the work in Jones, Homer, and Noble (2015) that this chapter is derived from. That paper presented the same design for brand objects as we have presented here, but a very different and much less powerful formal model of the brand objects, named Branded Tinygrace.

Rather than using the identity of an object as a type (since the model did not feature a store), all brands are uniquely generated by a preprocessing step run before the beginning of the program execution. Brands are bound to names by a set of recursive type bindings, and the preprocessing replaces every occurrence of the brand constructor with a unique β identifier, before substituting the bindings into the program's top-level term. The result is a significantly less expressive language than Branded Graceless, because β brands cannot be generated dynamically in a method.

Branded Tinygrace *is* sufficient to encode the basic idea of nominal types in the sense of a language like Java, where classes are declarative and therefore the entire nominal type hierarchy is constructed before the program's execution begins. It cannot encode the more free-form kind of nominality in more dynamic languages (including Grace itself), and in particular branding the objects from an 'inner class' in Branded Tinygrace all have the same type, even across what are really different instances of classes with the same definition.

Consider the following Grace program:

```
class newFactory(x : Brand) {
    object {
        class new is x {}
    }
}
```

The outcome of requesting newFactory twice are two distinct objects, and passing different brands to these requests mean that the classes produce objects with completely disparate nominal types. Branded Tinygrace cannot encode this behaviour: the best it can do is have a single brand shared between all of the objects constructed by any newFactory object, effectively this:

```
def x = brand
class newFactory {
    object {
        class new is x {}
    }
}
```

Branded Graceless, in comparison, can fully encode this behaviour, and just reuses object identity to implement the uniqueness of separate brands, which more closely follows the conceptual design presented in §6.1.

The other distinction is that all brands are always in scope in Branded Tinygrace, because they are all declared at the top-level of the program. This means that Branded Tinygrace cannot encode the necessary security property of the brands by hiding the declarations in a different scope while exporting the type. By contrast, Branded Graceless can use the τ declaration form to hide the declaration of a brand from the use of its type, as seen in Figure 6.3.10.

Branded Graceless still cannot encode the entirety of the conceptual design, despite extending much of the functionality of Branded Tinygrace: the new model cannot encode subtyping relationships between types when the brand is not in scope. If the client wants an Animal type as its first argument, and then Dog as a subtype of Animal, there is no mechanism for expressing this in the client without the presence of the brand as well. With the brands anAnimal and aDog in scope (where aDog η(anAnimal)), the variables-as-types are not necessary, because the forms β(anAnimal) and β(aDog) precisely describe the necessary type.

There is no syntax for expressing that when a variable is used as a type, it extends some other variable — or extends the type of some brand, for that matter. Variables can only precisely encode the type of one or more brands with a τ form. This means that a client cannot reason about nominal subtyping without also hav-

ing access to the underlying brand object, which also gives it permission to brand its own objects. While Branded Graceless *can* encode Java-like class nominal subtyping structures and programs that reason about them, it *cannot* encode programs that simultaneously describe such structures while preventing misuse of the brand object in the client.

The solution to this problem is to include type members, a feature that should actually already appear in Graceless in order to fully encode Grace programs. The τ form and using variables as types is effectively a stand-in for fully-fledged type member support, and type members would also allow the encoding of subtyping relationships outside of the scope containing the brand. The input to a client accepting both Animal and Dog types could be expressed in the Dependent Object Calculus (sugaring zero-parameter methods to use Grace syntax) as:

$$\{\, z \Rightarrow \mathsf{Animal} : \bot..\top \wedge \mathsf{Dog} : \bot..(z.\mathsf{Animal} \wedge \mathsf{bark} \rightarrow \top) \wedge \mathsf{dog} \rightarrow z.\mathsf{Dog} \,\}$$

This type does not reason about brands at all, but it is inhabited by objects encoding the Types of anAnimal and aDog, along with the dog class. Working out how to express a type like z.Dog while also allowing a run-time match against it remains as future work.

DOT can also express a form of pseudo-nominality in its types, as seen in the client signature above: even if the Type members of the Animal and Dog parameters are entirely structural, within in the body of the client method they must be treated as nominal because there is not enough information in the bounds to decide if any given object satisfies the type. The only objects that can be said to satisfy the type Dog.Type in the body of the client method are those created by the new method of the dog object, as desired by the design of the nominal system.

The missing feature of DOT to fully express a nominal, class-based type system is some form of instanceof check, available in Branded Graceless through the match construct. Such a feature is not strictly necessary — and arguably it is not good object-oriented design to rely on a discrimination of an object's type rather than delegating that discrimination to the object itself through a normal method request — but without it, a language cannot express the full set of program designs currently available in popular object-oriented languages. Developing the appropriate metatheory for small-step semantics in the presence of type members has

been a large, long-term problem, and only recently solved (Rompf and Amin 2016; Amin 2016).

Branded Graceless also bears similarities to the Tagged Objects theory of Lee et al. (2015), introduced in §2.5.2. While Branded Graceless and Tagged Objects seem to broadly achieve the same outcome, neither the goals nor the implementation of either are the same, and when investigated in detail the two are not nearly as similar as they may have seemed. Even ignoring differences between the cores of the two languages that are unrelated to branding or tagging, there are a number of differences between Branded Graceless and the Tagged Objects language.

One of the key differences between the two languages is that tags wrap around the value they tag, so that a tagged value cannot be transparently used as the underlying value without first extracting the value from the tag: the type of new($x; v$) is tagged x, and the type of v is irrelevant so long as it satisfies the corresponding type τ of the tag x. In contrast, branding an object in Branded Graceless with $\beta(x)$ occurs *inside* the object declaration, so that an object is inherently branded by x and may still be used normally without having to apply an extract function to unwrap the underlying value.

One of the key outcomes of this is that an object can satisfy the types of multiple brands at once, whereas in the Tagged Objects language a value wrapped in a tag *only* inhabits the type of the tag, and wrapping it in a second tag forgets the type of the first. The ability to inhabit the types of multiple brands at once does not actually require the ability to write multiple β forms in an object, as it suffices to declare a new brand object that extends multiple different brands at once, and then brand the object with that. Extending the subtag function to take multiple inputs (along with updating the corresponding judgements) would allow Tagged Objects to express the nominal subtyping relationships in languages with multiple inheritance, including implementing multiple interfaces in languages such as Java.

Branding objects after they were constructed was part of the conceptual design of §6.1 using the annotateObject method on the brand. This is more a happy accident of the design's implementation more than it is a crucial part of the design, though there is no reason a construct for adding a new β form to an existing object could not also be included in Branded Graceless, given that the resulting type of the object in the store would always be a subtype of its previous type. Such a

construct raises some interesting problems with the type of an object changing as the result of an imperative update, reminiscent of typestate (Aldrich et al. 2009) (though simpler, since the resulting type is always a subtype of the original), but the implementation of the brand type-checker in Hopper does not reason about this behaviour anyway, and branding an object after it is constructed is only useful for run-time matching.

Since the type of a tag depends directly on being able to name the tag (as in tagged χ), the Tagged Objects language is not able to express the separation of brand object and brand type that exists in Branded Graceless. As discussed in §2.5.2, this means that client code is free to tag its own objects, so a tag type provides no guarantee that an object was actually constructed by a given class. A tag's corresponding type τ enforces that an object implements a particular interface, but tags are *not* object capabilities; they cannot be used to enforce the same security properties as the seals of Morris (1973) that inspired the brand pairs of Miller (2006), the trademark design for JavaScript of Horwat and Miller (2011), and the brand design presented here, since the guard of a tag (and the run-time discriminator used in a match) is the same as the permission object itself. Like Branded Graceless, type members along with some mechanism to expose the run-time matching component of a tag without exposing the tag itself would allow tags to work this way.

The match construct also differs significantly between Branded Graceless and the Tagged Objects language. In the Tagged Objects language, the match can only discriminate against tags, and only if the input shares a common super-tag with the tag used for the match. In contrast, brands may appear in any object in Branded Graceless, so any object may be inspected for any brand. The 'else' branch of the Tagged Objects match does not bind the matched object to a variable, since no extra information has been learned in that branch: since the type of the input is just some tagged form then discovering that the tag on the value is *not* the one in the match doesn't provide any useful information. In Branded Graceless the 'else' branch is actually more useful than the 'then' branch.

The main advantage of the Tagged Objects approach to matching stems from the corresponding τ type of the matched brand, such that the match only needs to discriminate on the brand to work out what the type of the underlying wrapped value is. This is why it is useful for the 'then' branch to bind the input to a variable

whose type is tagged with the matched tag: the type of an application of extract on that variable may now be more precise, as it was in Figure 2.5.2. Branded Graceless works in the opposite fashion, in that discovering the presence of a brand gives no extra information beyond that the brand is there because there is no type information about any branded object associated with the brand itself, but discovering the *absence* of a brand can collapse a larger type down into one that is more precise.

The optional int example is actually *easier* to encode in Branded Graceless — despite the fact that matching against a Some brand type would not actually provide access to the structural type exposing the int — because all of the possibilities do not actually need to be branded, so a Some type is not necessary.

```
method incOpt(None : ⊤, x : Number ∪ type { None }) → Number {
    x ∋ None { y → −1 } { y → z + 1 }
}
```

This method can be requested with minimal fuss.

```
method apply(theNone : ⊤) → Number { incOpt(object { τ(theNone) }, 5) }

apply(object {})
```

The trouble is that, because an object can be branded with arbitrarily many brands, a single match is often not enough. If matching between two possible brands in an object, and the two variants that contain each brand in the object's type both contain structural information that needs to be accessed in the branches, then a Branded Graceless program needs to account for *three* cases, since a second match inside the 'then' branch of the first can also succeed.

Consider the following method, which takes two brand types A and B, each of which appears in one of two possible types for the third input, and then matches against the brand types in order to request the associated methods to return a value of some type T.

```
method oneOf(A : ⊤, B : ⊤, x : type { A; a → T } ∪ type { B; b → T }) → T {
    x ∋ A { y → y ∋ B { z → ⇑ x } { z → z.a } } { y → z.b }
}
```

In the 'else' branch of both matches, the execution has successfully determined which of the two variants in the union type of x applies based on discrimination

against the brand types, but in the 'then' branch of the inner match, x is branded by *both* the underlying brands objects of the types A and B, a perfectly valid possibility given only the information in the types. Having more than two variants only exacerbates this problem.

Associating structural information with tags allows the Tagged Objects language to not have to reason about the existing type of the matched object. Given the existence of reified types in Grace, there is no reason such an extension could not be added to the existing design with an overloading of the brand method to take a type object, so that brand(type { $\overline{\mathsf{D}}$ }) produces a brand that requires any branded object to have declarations $\overline{\mathsf{D}}$. The existing brand method would then be equivalent to brand(type {}). Alternatively, using some sort of *negative* type would also suffice, such that the type of x above could be expressed as:

$$\mathsf{type}\ \{\ \mathsf{A};\ \neg\mathsf{B};\ \mathsf{a} \rightarrow \mathsf{T}\ \} \cup \mathsf{type}\ \{\ \mathsf{B};\ \neg\mathsf{A};\ \mathsf{b} \rightarrow \mathsf{T}\ \}$$

Intersecting either type { A } or type { B } into the type yields one of the two variants because the other conflicts with the intersected type.

6.5 Implementation

We have implemented brands in Grace on top of Hopper, our existing prototype interpreter for the language (Jones 2016). The core implementation is a single Grace module, extending an existing Hopper implementation of a structural type checker as a dialect (Homer, Jones, et al. 2014). All of the functionality specific to brands is implemented using existing features of Grace, provided in a dialect with the necessary definitions. Static checking of brand usage is restricted to just those modules using the branding dialect, but the dynamic behaviour of the brands will work as expected in other modules.

Hopper allows arbitrary expressions to appear in any annotation list, requiring only that the type of the value satisfy a protocol specific to the syntactic object that it annotates. This protocol is implemented for brand objects so that any annotated object literal or class is added to a set of objects stored in the brand itself. This way there does not need to be any special functionality in an object to record that it has

been branded, because the implementation of the brand itself can handle all of this information. A brand implements this storage as a *weak* set, as the only necessary operation is a membership test, and this prevents a brand from keeping an object alive when there are no remaining strong references to it.

Because the annotation processing is just handled by a method request, brands need not be applied only at an object's creation, so any existing object can also be passed to the brand through the same protocol. No special access to an object is required in order to brand it, since no modification is made to the object when it is branded. Access to the identity of the brand itself is all that is necessary to brand an object.

Grace's Pattern type (Homer, Noble, et al. 2012) is used as the interface of the objects returned by a brand's Type method. By inheriting from a standard abstract class that defines the basic implementation of patterns, the objects need only provide a concrete match method, which uses the membership test to discover whether the relevant object is in the brand's weak set. Because the Type object is defined in the brand itself, it has access to the brand's internal of branded objects without the set having to be exported from the brand.

Internally, most of the brand object functionality is implemented in a preBrand class. This class is used to build the 'pre-brand' object aBrand, which will be used to brand all other brand types. This pre-brand is local to the dialect, but the dialect makes the public type Brand available for use outside of the dialect, the value of which is aBrand's pattern object combined with the interface of brands.

```
let Brand = aBrand.Type ∩ ObjectAnnotation ∩ type {
    Type → Pattern
    extend → Brand
    +(other : Brand) → Brand
}
```

aBrand has the same implementation as other brands, but an object cannot appear in its own annotation list and so aBrand does not satisfy the Brand type. Attempting to do so would look like this:

```
def aBrand = object is aBrand { inherit preBrand }
```

This would result in an uninitialised field error, since object cannot yet have been assigned to the aBrand field at the point when the annotation list is evaluated.

All other brands inherit from the same class that created aBrand, with the sole addition of being branded by aBrand, causing them to satisfy the Brand type.

```
class brand → Brand is aBrand { inherit preBrand }
```

Brands need to be branded themselves so that when we perform static reasoning about brands, we can rely on the Brand type to guarantee that the implementation of the object behaves as we expect. The definitions of the brand method and the Brand are included with the rest of the standard dialect definitions (the 'prelude') to provide a sensible set of default methods to any module that uses the dialect alongside the branding mechanisms.

With these definitions in hand, brands provide the appropriate run-time behaviour of nominal types: objects can be branded, method parameters can be guarded by brand types, and execution can branch based on the presence of a brand with the match construct. At run-time, matching against a brand's object identity provides no dynamic information about a brand's name, so that if a dynamic type error occurs involving a brand it cannot report the name of the brand that failed to match. Reporting type names is a standard problem in structural type systems as types do not naturally have names (Malayeri and Aldrich 2008). This is mostly solved by including the source location of the assertion that failed, where a name for the type that raised the error should be available if the type itself was referred to by name at that location.

On top of this run-time behaviour, the dialect extends the existing structural type checker's check method by including extra understanding of the Brand type and the result of requesting brand. Even though the type of the brand method only indicates that it returns a Brand, each creation of a brand object is considered distinct by the type system, and this identity is tracked within the scope of its creation as well as when it is exported by a def declaration (with some caveats explained below). Additionally, method parameters with the type Brand are also reasoned about statically as though they had been created at the beginning of the method. The existing structural rules are still enforced, including when structural types are paired with brand types. We formalise this combination of static typing in §6.3.

In order to ensure the type system is sound, it is important that the static identities that we assign to brand definitions always correspond to a single object identity:

```
method make {
    def aThing = brand
    ...
}
```

Each time this method is called, the request to brand will construct a new, unique brand. This is irrelevant to the static type checker, which is reasoning *for all* calls to the method. If a brand is bound to a 'statically-known' immutable definition, then we have a guarantee that the identity returned from referencing that definition will always be the same within the scope that surrounds that definition. It's important that we understand what a statically-known definition is, since the presence of shadowing and inheritance makes this more complicated.

6.5.1 Statically-Known Definitions

All method parameters are local definitions, but not all defs are. When a definition appears directly in the body of an object, referring to the name of that definition is a request to the corresponding accessor method on the object, rather than a direct reference to the definition's value. In the presence of inheritance, there is no guarantee that a reference to such a definition actually refers to the value defined. As discussed in Chapter 4, the following code can even result in an error reporting that aThing is uninitialised:

```
object {
    def aThing = brand
    object is aThing { ··· }
}
```

If this object expression appears as the final statement of a method body, then it can be inherited, and the inheriting object can override the definition of aThing with any method, providing no guarantee that the resulting object is the same every time the definition is referenced, or even that the result is a brand.

One potential solution to the problem of overriding otherwise constant definitions is to statically prevent any override that would invalidate the brand reasoning in the branding dialect. The dialect already knows that it needs to reason about such definitions under the assumption that the definition is immutable once assigned, so it makes sense that it could just prevent the override of any such brand with any method that is not itself a constant reference to a brand. This solution fails to take into account the fact that dialect checking is per-module, and so even if this was enforced it does not provide a static guarantee that any part of the program does not invalidate this constraint. This *could* be simulated as a gradual constraint, so any attempt to perform an invalid override that was not checked by the dialect results in a run-time error, but a simpler solution is to adapt the reasoning to make no assumptions about fields that could be overridden.

Any time a brand is to be created in the body of a potentially inherited object, then the brand can be created in the closure of the surrounding method instead:

```
def aThing = brand

object {
    object is aThing { ··· }
}
```

Since this definition appears in a method body instead of an object body, it is guaranteed to bind the result of the request of the brand method. The dialect can safely reason statically about it within the method scope. If the returned object wants to make the aThing brand publicly available, it can still declare a public definition in its body, so long as it has a different name than the private definition: if the name of the public definition is important, the name of the private definition can always be changed, since that only has significance in the surrounding scope.

Shadowing definitions in the presence of inheritance is also a potential problem, since method lookup proceeds up inheritance chains before proceeding into outer scopes. The dialect needs to be sure that a local reference will actually resolve to the brand definition it believes it will. We address the problem of ensuring correct resolution of local references in the presence of inheritance in Part III; given that you need to know the exact type of an object in order to inherit from it, it is not hard to guarantee that you can build an accurate model of local method requests.

6.5.2 Type Evaluation

The story of the implementation is more complicated than the story we have presented so far, because the Grace specification does not provide a definitive explanation of what terms can appear in a type definition (Black, K. B. Bruce, and Noble 2016). It states that, "Types, including parameterised types, may be named in type declarations." The examples that follow in the specification include type literals, applications of the standard type combinators (which are defined as types by the spec), and references to other type definitions (including applications of type arguments to parameterised definitions). Whether other terms that (might) also produce types can be used as type definitions is not defined, nor what kind of behaviour can be expected if such a term is used in a definitions.

Because types are reified into objects at run-time (and type definitions create corresponding accessor methods when they appear in an object body), the term in a type definition must be evaluated at some point. With the type forms presented as examples in the spec this evaluation could happen statically, with the resulting values substituted in for definitions at the relevant moments at run-time, but if more general terms are permitted to appear in a type definition then their evaluation must occur in the same scope as the definition, since they might refer to other definitions surrounding them. The implementation presented here relies on the ability to declare types with arbitrary terms, since a request to a brand's Type method is not a 'type' per the Grace spec — it just shares the interface of a reified type object. Since the implementation is in Hopper, which is already a non-compliant implementation of Grace, we can skirt around this issue somewhat by just defining the behaviour as whatever we need, but it does mean the evaluation semantics of type definitions need more investigation.

Type definitions cannot be overridden (Black, K. B. Bruce, and Noble 2016), and it makes sense that we would restrict the values that are assigned to type definitions to at least have the same interface as reified types. To that end, we might define type definitions as a form of def whose accessor method cannot be overridden:

```
def Thing : Pattern = ···
```

Given the evaluation order of object bodies, such an encoding could produce unexpected behaviour if a method is annotated with the following type definition.

```
method makeThing → Thing { ··· }

type Thing = ···
```

It's not clear what the behaviour of this code should be: does the Thing type need to be defined before the instantiation of the object that contains the method makeThing? Even if the type annotations on a method are evaluated lazily, it is still possible that the method could be requested in the object's initialisation code before the definition Thing is assigned a value.

It seems more reasonable that type definitions terms are evaluated *before* any object initialisation code that appears alongside them, and do not truly appear in the scope directly surrounding them, in the same way that inherit clauses appear inside the braces of an object constructor but may not refer to any definitions in that scope (including self) since the value of those definitions relies on the result of the inheritance. This could result in code that executes out-of-order even for a linear execution sequence (a problem that an inherit clause does not have, since it must appear at the top of an object constructor), but perhaps it is sufficient to recommend (and perhaps enforce in a dialect) that terms whose evaluation has side-effects should not appear in type definitions.

This design of type definitions affects the design for brands presented above, since the declarations of brand types often refer to a field definition in the same scope. The following object constructor is not valid:

```
object {
    def aThing = brand
    type Thing = aThing.Type
}
```

The evaluation of aThing.Type occurs *before* the evaluation of brand, and in fact aThing.Type is evaluated in a scope that does not even contain aThing. We can resolve this problem with the same solution used to avoid overriding definitions: define the brand outside of the object, and then export the type inside the object.

```
def aThing = brand
```

```
object { type Thing = aThing.Type }
```

Defining a brand in the scope outside the type definition does not work in cases where a definition cannot syntactically be written in the scope surrounding the type definition. Modules present such a case, since the top-level code of any file is the body of an object. This can be offset somewhat by defining any top-level brand objects in other modules and then importing them (or defining a dialect that extends the branding dialect with the brand object definitions) but given that modules are likely to be the most common definition site of types, this doesn't seem like a realistic solution.

Ultimately we have implemented a compromise between these two points in the design space for Hopper, evaluating a type definition immediately if it is requested before its appearance in the initialisation sequence, and otherwise evaluating it in the normal sequence. Type annotations on methods and other definitions are evaluated lazily, so type names can be used as annotations on definitions immediately alongside their definition. This design was a solution for the separate problem of run-time evaluation for reified types with mutual dependencies (that is, recursive types), similar to the lazy evaluation of sub-terms in coinductively defined types. Even type definitions that are defined to be valid in the Grace specification may refer to type names declared *after* the current one:

```
type A = { a → B }
type B = { b → A }
```

The ability to perform some sort of reference to uninitialised type definitions is already necessary when the implementation does not evaluate the types statically (which Hopper, as an interpreter, does not).

Of course, since a dialect is free to reason about whatever it chooses, we could also forgo type definitions using the type syntax entirely, and rely solely on def:

```
def aThing = brand
def Thing is public = aThing.Type
```

We no longer need to rely on undefined behaviour to implement the design, but we have less guarantees as a result, as def fields can be overridden whereas type definitions cannot.

Part III

Inheritance

7 Inheritance Semantics

The story for inheritance in Grace was supposed to be simple: inheriting from an objects reuses its implementation, extending and modifying it as desired by the inheriting object (Black, K. B. Bruce, Homer, and Noble 2012). A class is just a method that constructs and returns a fresh object, so inheriting from a class is as simple as inheriting from the result of requesting the method. This story turned out to be insufficient to capture the true complexity of the desired semantics, particularly with respect to the use of self reference during an object's initialisation. As a result, the design of Grace's inheritance has evolved from object delegation to a special semantics that attempts to simulate the behaviour of class in Java and C# (Noble et al. 2017).

This part investigates the challenge of implementing inheritance in a classless language by considering a number of different semantic models and comparing the outcomes of each. Driving the investigation is a series of modification to Graceless to express each semantics. All of these formal models are individually implemented in PLT Redex, making it easy to enumerate the individual steps of any given program and interpret how that program differs under each of the semantic interpretations.

7.1 On Inheritance

The problem we address in this part focuses more acutely on the tension between the conceptual simplicity of objects and the practical utility of classes than the previous part. Can a language be based conceptually on 'objects first' and include a

Aspects of this part appeared in the ECOOP'16 paper Jones, Homer, Noble, and K. Bruce 2016.

```
method graphic(canvas) {
    object {
        // An abstract method.
        method image { required }
        // Using an abstract method.
        method draw { canvas.render(image) }
        // An uninitialised field.
        var name
        // A use of self during construction.
        canvas.register(self)
        // A local method request.
        draw
    }
}
```

Figure 7.1.1: The graphic example class

relatively familiar notion of inheritance? We present models of several different inheritance mechanisms as modifications to the existing Graceless syntax and semantics.

The first set of models — forwarding, delegation, and concatenation — correspond to standard inheritance practice in many classless languages. The second set — 'merged identity' and 'uniform identity' — are novel constructs of behaviour in a classless language, but are intended to introduce classical behaviour and attempt to simulate the inheritance behaviour of C++ and Java classes respectively. The remaining models introduce several different languages with multiple object inheritance, using different resolution techniques for conflicts between methods from multiple inherited objects.

We evaluate the trade-offs between power and complexity of these models, particularly in their object initialisation semantics, and compare them to the behaviour of other languages, demonstrating that the typical class initialisation semantics are fundamentally at odds with prototypical object inheritance. We have also implemented all of the models in PLT Redex (Felleisen, Findler, and Flatt 2009), making the models executable and allowing direct comparison of the differences in execution for any program.

```
def amelia = object {
    inherit graphic(canvas)
    // An overriding method.
    method image { images.amelia }
    // An assignment to an inherited field.
    name := "Amelia"
}
```

Figure 7.1.2: The amelia example object

Consider the example in Figure 7.1.1, where the graphic method constructs a fresh object with some methods and a mutable (var) field, and runs some initialisation code including a local call to the draw method before returning. The required form is used to represent the body of an abstract method, implying that an implementation of the method's body is required in some inheriting definition. A required method evaluates to an exception if invoked.

If we can inherit from objects created by the graphic method, what is the resulting program behaviour? Results vary for different interpretations of inheritance, which we examine in depth as we present each semantics. We can inherit from the objects this method creates, or assign special semantics to methods which directly construct fresh objects like this and inherit from the method itself. Because of the presence of initialisation code in the class, these interpretations have different — and potentially unexpected — behaviours.

In its most basic form, inheritance permits reuse by providing a mechanism for an object to defer part of its implementation to an already implemented part of the program, but the reality is that there is much more to consider: the value of self references in method bodies and during initialisation, intended and accidental method overriding, whether objects are composed of several object identities in an inheritance chain or a single unified identity, the meaning of method requests which are not qualified with a receiver, and so on.

Consider the object amelia in Figure 7.1.2, that inherits from a call to the graphic method. We can draw different conclusions about the state of our program after amelia is constructed, depending on which inheritance semantics is in play. We group these into a number of relevant concerns:

- **Registration**. Is the identity of a super-object stored during initialisation, either explicitly or through lexical capture, the same as the final object? This is clearly the intention of the call to register in graphic's initialisation.

- **Down-calls**. Can a method in a super-object call down into a method in a lower object? Can it do so during initialisation? The implementation of the draw method relies on a down-call to the image method.

- **Action at a Distance**. Can operations on an object implicitly affect another object? If the registered graphic object is different to amelia, what is the value of its name field after amelia is initialised?

- **Stability**. Is the implementation of methods in an object the same throughout its lifetime? Which image method will be invoked by the request to draw at the end of graphic? Can the structural type of an object change after it has been constructed?

- **Preëxistence**. Can an object inherit from any other object it has a reference to? Does amelia have to inherit a call to the graphic method, or will a preëxisting object suffice?

- **Multiplicity**. Can an object inherit from multiple other objects? If amelia also wished to have the behaviour of another object, can a second inherit clause be added? If so, how are multiple methods with the same name resolved, and where are fields located?

Each of these concerns is explained in more detail as they become relevant to the model we are discussing.

We are also interested in the general semantics of the inheritance systems, such as what order the parts of initialisation execute in, what visibility and security concerns arise, and how local method requests are resolved. We also point out particular curiosities, such as the absence of overloaded methods from super references and inheriting from definitions already inherited from some other parent, but these features are specific to individual models.

These are the concerns that we will judge in the following object inheritance models. While our graphic example clearly assumes some of these features in its

implementation, these features are not universally desirable, and each provides certain abilities or guarantees at some cost. Our intention is to use these concerns to provide an accurate description of the trade-offs involved with each inheritance model. Some of our models of inheritance attempt to interpret graphic as a class, but only in the sense that it is a factory that guarantees fresh graphic objects. We also compare the models to existing languages that use each form of inheritance, particularly JavaScript, which is capable of implementing all of the models.

8 Object Inheritance

First we consider forms of inheritance that typically exist in classless languages, expressing reuse directly between objects at run-time rather than between declarative classes. Starting with Graceless, each of the models builds upon the previous one. The models encode three forms of object inheritance: forwarding, as used in E (Miller 2006); delegation, as found in JavaScript, Lua, and Self (ECMAScript Project 2016; Ierusalimschy, Figueiredo, and Filho 2007; Chambers et al. 1991); and concatenation, as in Kevo (Taivalsaari 1995) and numerous libraries and idioms for languages with open objects.

In our earlier example, amelia inherits from the call to the graphic method by evaluating the call normally, and then extending the behaviour of the resulting object in a new object. A possible resulting structure is visualised in Figure 8.0.1. We use a custom visual syntax for objects to more easily communicate the relevant information.

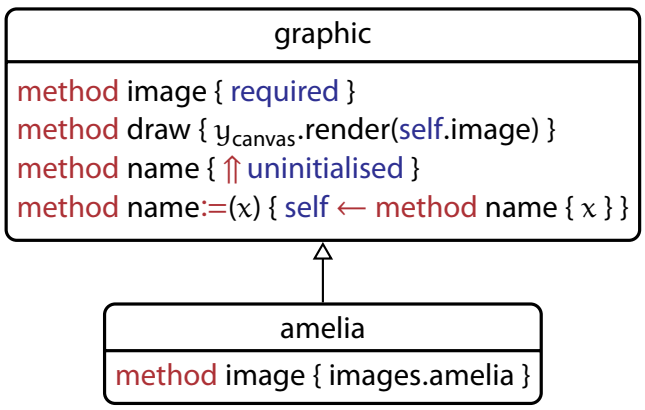

Figure 8.0.1: Example visualisation of object inheritance

An object is visualised as a rectangle with rounded corners, containing a name (purely for our reference) and the methods that make up the object's implementation. Conceptual inheritance relations between objects are represented by an arrow with a white triangle, pointing from the inheritor to inheritee. We also omit type annotations on the implementations. Objects are referred to by their name as a subscript on any concrete variable y, so y_{canvas} is a reference to the canvas object used by amelia in the inherit clause. We also use the form required as described above, and the form ⇑ uninitialised to represent the body of an uninitialised field accessor.

Under the object relationship visualised in Figure 8.0.1, the amelia object can directly handle the image method, and a request to any other method is 'passed on' to the graphic object. The specific behaviour of this passing on (and, in some cases, *which* graphic object the request to passed to) is what gives rise to the different models that we present here.

The extended syntax for object inheritance is given in Figure 8.0.2, extending the final Graceless syntax defined in Figure 5.2.1 in §5.2. The extension introduces two new components of user-facing syntax: the bodies of object expressions may now begin with an inherit t clause, and requests can now be qualified by the special variable super. We use the identifying subscripts $_\uparrow$ and $_\downarrow$ in many of our judgment definitions to help indicate whether a form relates to an inherited ($_\uparrow$) or inheriting ($_\downarrow$) object: these subscripts form part of the name, and do not change the semantics of the metavariable that they are attached to.

As methods are locally scoped within the body of an object expression, it's important to understand how inheriting methods affects this scope. An inherit clause inherit t introduces the methods from the super-object into the local scope of the inheriting object using an 'up, then out' rule (as opposed to the 'out then up' rule of Bracha (2016) in Newspeak): inherited definitions take precedence over those introduced in a surrounding scope, as if they were defined directly in the inheriting object.

Because the inherit clause contains an arbitrary expression, in general we cannot know what method identifiers the clause will introduce. To counter this, substitutions are delayed by an inherit clause in an object expression: while the substitution will transform the expression in the inherit clause itself, it *will not* proceed

Grammar

i	::=	ϵ | inherit t	*(Inherit clause)*
t	::=	$\cdots$ | object { i $\overline{s}$ $\overline{d}$ t } | required	*(Term)*
r	::=	$\cdots$ | super | (v as x)	*(Receiver)*
s	::=	$\cdots$ | (v as self)/super | y/self←	*(Substitution)*

Evaluation contexts

G	::=	$\cdots$ | (v as x).m($\overline{v_i}$, F, $\overline{t}$)	*(Sub-context)*

Figure 8.0.2: Graceless grammar extended for object inheritance

into the body of the object expression, and gets 'stuck' on the clause instead, hence the extra $\overline{s}$ syntax after an inherit clause. Once the expression in an inherit clause is resolved to an object reference, the substitution can proceed into the body of the surrounding object expression, where it may be removed by shadowing. The form s is a stuck substitution, and [s] continues its application.

Although any inherited object exists in its own right, making a request to the form super is not the same as making a direct request to the same inherited object, as a request to super binds the value of self in the super-method to the *inheriting* object. At run-time, the variable super is substituted for a special 'up-call' receiver (y as self), which indicates the method will be dispatched to the super-object at location y, but with self bound in the body of that method to the eventual value of self at the site of the request. A substitution for self will transform the receiver into ($y_\uparrow$ as $y_\downarrow$) before the request is evaluated. A super reference or up-call form by itself is not a valid term.

The substitution y/self← is a special substitution that only applies when self is the direct target of a method update. For instance, the result of applying the substitution [y/self←](self←(method m→T{t})) is the term y←(method m→T{t}), but [y/self←]self is still self. Receivers of method updates are still substituted as normal by the other forms of substitution.

Returning to our example, in the extended Graceless language we have presented there is no corresponding implementation for the connection between amelia

and graphic, indicated by an open triangle arrow in the visualisation of Figure 8.0.1. We have not modified the store to include the presence of an optional 'super-reference' for each object; such an extension would require extra reduction rules to deal with requests which the object cannot directly handle and must be delegated to the super-reference.

Fortunately, the language as presented is already capable of expressing this behaviour directly in the implementation of the inherit clause alone, by inserting methods into the inheriting object that just perform a super-call to their corresponding method. Reduction rules for object inheritance are defined in Figure 8.0.3, extending the computation judgement from Figures 4.3.1 and 5.3.1.

Including an inherit clause inherit t in an object evaluates the term t to an object reference. The evaluation context G in Figure 8.0.2 has not been extended to include a term in an inherit clause, to avoid issues in the future with constraints on what can be evaluated in such a clause. Evaluation in an inherit clause is instead handled by Rule E-I/C, which is modified by future models to ensure the context behaves as desired.

Rule E-R/S applies an up-call like a regular request, but looks up the method in the super object $y_{\uparrow}$ while retaining the original value of self, $y_{\downarrow}$. This means that the body of the method at the $y_{\uparrow}$ will be run, but any request to a method on self in that body will return to the inheriting object $y_{\downarrow}$. Inheriting from a cast is also legal, and Rule E-R/C ensures that any cast assumptions are distributed across a request when the receiver is an up-call to a cast.

Rule E-Inh transforms an object expression with an inherit clause into one without, adding a super-call method for each method in the inherited into the surrounding object expression. Overridden methods are removed with the *extend* auxiliary function, which also handles the generation of the super-call method body with the *delegate* function. The rule also applies the delayed substitutions to the object, after substituting super for an up-call receiver to the inherited object.

To better understand the way that the inheritance is encoded into Graceless, the encoding of the two objects from Figure 8.0.1 is visualised in Figure 8.0.4 (ignoring that the evaluation of the request to graphic will actually get stuck in its own initialisation). Since there is no direct encoding of the inheritance relationship between amelia and graphic, they are visualised here as unconnected objects,

$\boxed{\sigma \mid t \longrightarrow \sigma \mid o}$

(E-Inh)

$$\frac{\sigma(v_\uparrow) = \{\ \overline{d_\uparrow}\ \} \qquad \overline{d'_\uparrow} = \mathit{extend}(\overline{d}, \overline{d_\uparrow})}{\sigma \mid \mathsf{object}\ \{\ \mathsf{inherit}\ v_\uparrow\ \overline{s}\ \overline{d}\ t\ \} \longrightarrow \sigma \mid \overline{[s]}\mathsf{object}\ \{\ [(v_\uparrow\ \mathsf{as}\ \mathsf{self})/\mathsf{super}](\overline{d'_\uparrow}\ \overline{d}\ t)\ \}}$$

(E-R/S)

$$\frac{\sigma(y_\uparrow) \ni \mathsf{method}\ m(\overline{x : S}) \to U\ \{\ t\ \}}{\sigma \mid (y_\uparrow\ \mathsf{as}\ y_\downarrow).m(\overline{v}) \longrightarrow \sigma \mid [y_\downarrow/\mathsf{self}]\overline{[v/x]}t}$$

(E-R/C)

$$\frac{S \ni m(\overline{z : T_{i1}}) \to T_1 \qquad \mathsf{method}\ m(\overline{z : T_{i2j}}) \to T_2\ \{\ t\ \} \in \sigma(v)}{\sigma \mid (v : S\ \mathsf{as}\ y).m(\overline{v_i}) \longrightarrow \sigma \mid \mathit{coerce}((v\ \mathsf{as}\ y).m(\overline{\mathit{param}(v_i, T_{i2}, T_{i1})}), T_1)}$$

(E-I/C)

$$\frac{\sigma \mid t_\uparrow \longrightarrow \sigma' \mid t'_\uparrow}{\sigma \mid \mathsf{object}\ \{\ \mathsf{inherit}\ t_\uparrow\ \overline{s}\ \overline{d}\ t\ \} \longrightarrow \sigma' \mid \mathsf{object}\ \{\ \mathsf{inherit}\ t'_\uparrow\ \overline{s}\ \overline{d}\ t\ \}}$$

$$\mathit{extend} : \textsc{Seq}(\textsc{Def}) \times \textsc{Seq}(\textsc{Def}) \to \textsc{Seq}(\textsc{Def})$$

$$\mathit{extend}(\overline{d_i}, \cdot) = \overline{d_i}$$

$$\mathit{extend}(\overline{d_i}, (d, \overline{d_j})) = \begin{cases} \mathit{identify}(d) \in \overline{\mathit{identify}(d_i)} & \mathit{extend}(\overline{d_i}, \overline{d_j}) \\ \mathit{identify}(d) \notin \overline{\mathit{identify}(d_i)} & \mathit{delegate}(d), \mathit{extend}(\overline{d_i}, \overline{d_j}) \end{cases}$$

$$\mathit{delegate} : \textsc{Def} \to \textsc{Def}$$

$$\mathit{delegate}(\mathsf{method}\ D\ \{\ \mathsf{required}\ \}) = \mathsf{method}\ D\ \{\ \mathsf{required}\ \}$$

$$\mathit{delegate}(\mathsf{method}\ m(\overline{x : T_i}) \to T\ \{\ t\ \}) = \mathsf{method}\ m(\overline{x : T_i}) \to T\ \{\ \mathsf{super}.m(\overline{x})\ \}$$

Figure 8.0.3: Inheritance extended reduction

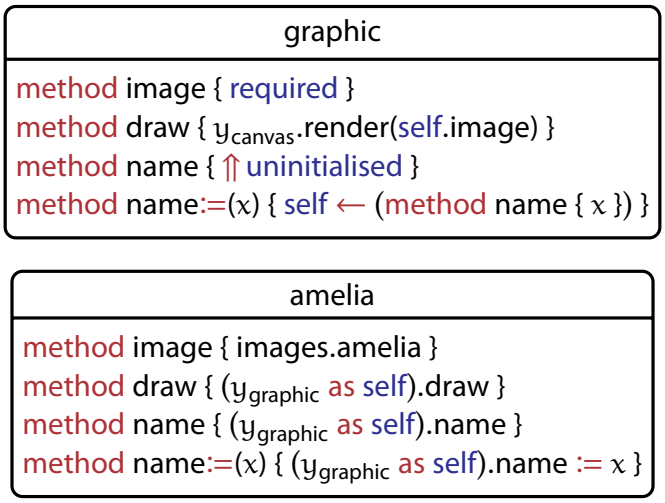

Figure 8.0.4: Visualisation of Graceless object inheritance

but amelia references graphic as its 'super-object' in the methods generated by the *delegate* function.

Now that we have the base of a model for object inheritance, we turn to examining the different semantic models of object inheritance by modifying this extended Graceless language.

8.1 Forwarding

The forwarding model of object inheritance is by far the simplest mechanism: if an object does not understand a request, it forwards the request unchanged to its super-object. The super-method receives the same arguments and self binding. In our example, if amelia receives a request for the draw method, which is not implemented directly inside of amelia, the request is passed on to the graphic super-object instead *as a normal request*. If we return to the conceptual linking between objects in an inheritance relationship, the objects and an example request are visualised in Figure 8.1.1, with a $\rightsquigarrow$ arrow describing an incoming request to the labelled method.

The structure of the objects is the same as before, but now we have evaluated amelia's initialisation code under forwarding. The assignment to name has updated the method in graphic, and the image field is initialised in amelia. Note also

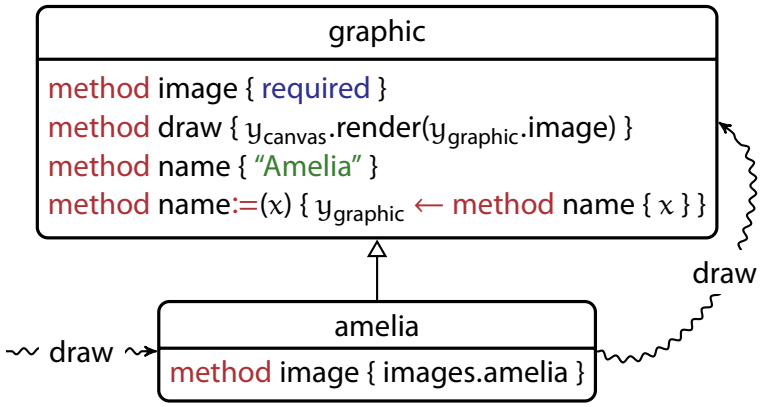

Figure 8.1.1: Visualisation of a forwarded request

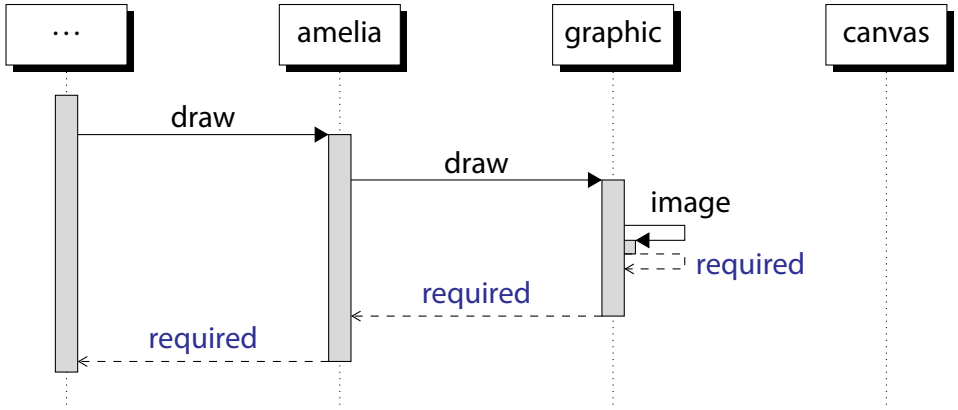

Figure 8.1.2: Sequence diagram of a draw request under forwarding

the absence of self: every use of it has been replaced with a direct reference to the object which defined the method.

A request to any method other than image will not be understood by amelia directly, and will be forwarded to the graphic object. The value of self in the resulting invocation is the identity of the super-object: this was the case for the assignment to name, so the value "Amelia" has been assigned to the field in the super-object. This is also the case for the incoming draw request, so graphic's draw method crashes, complaining that the object has not implemented its image method. The local request to image in the draw method has been resolved to the graphic object and not passed back down to amelia. The latter's behaviour is visualised as a sequence diagram between the relevant objects in Figure 8.1.2.

The modification to the existing dynamic semantics of Figure 8.0.3 to implement this object structure and behaviour is provided in Figure 8.1.3. The modification makes one subtle change to the Rule E-Obj, by *early-binding* the value of self when an object is created in all of its methods. The result of this change is that the binding of self in requests no longer achieves anything, because self has already been bound to the object that the method or field originally appeared in. Any forwarded request behaves as though the original object had received the request directly. Note that this applies to super requests as well, so that up-calls essentially ignore their second reference and the request is equivalent to just calling the method directly on the inherited object.

This early-binding of self is what gives us the correct forwarding semantics: even though amelia is performing super-calls to the graphic object, there are no remaining uses of self in the methods to allow down-calls to occur. This applies to the super-calls generated by the *delegate* auxiliary function, which means that the inherited methods still appear to be performing super-calls as normal as in Figure 8.1.4, but any method in the super-object that they call will never refer to self, having already substituted it away when the object was constructed.

The value of self is always the object a method was defined in, so down-calls are not possible (other than by manually referring to the inherited object in the super-object); similarly, the value of self during initialisation is the distinct identity of the super-object, so registration cannot occur then. In this model, an object can only inherit from another before it has run any of its own initialisation, so every object is guaranteed to have a stable structure, and one object may be inherited many times. In Chapter 10, we explain how forwarding could be extended with multiple inheritance.

(E-Obj)

$$\frac{y \text{ fresh} \qquad \overline{a = \mathit{identify}(d)} \qquad \overline{a} \text{ unique}}{\sigma \mid \mathsf{object}\,\{\,\overline{d}\ t\,\} \longrightarrow \sigma(y \mapsto \{\,[y/\mathsf{self}]\overline{[\mathsf{self}/a]d}\,\}) \mid [y/\mathsf{self}]\overline{[\mathsf{self}/a]}t; y}$$

Figure 8.1.3: Forwarding reduction

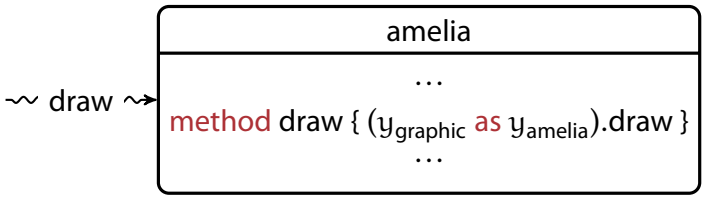

Figure 8.1.4: Visualisation of Graceless forwarding inheritance

8.1.1 In Other Languages

In the E language, there is no explicit self reference, as all object definitions are explicitly named (Miller 2006). The authors of E refer to the language's inheritance mechanism as "delegation", but in the absence of self references the behaviour aligns with what we have called forwarding.

```
def graphic { to draw() { canvas.render(graphic.image()) } }
def amelia extends graphic { to image() { images.cat() } }
```

Even though amelia defines a method image, the call in the draw method uses the graphic object as the receiver of its call to image, so calling amelia.graphic() produces an error that graphic.image() is not defined. Methods cannot be requested on the local scope, so the receiver must always be explicit. In order to achieve down-calls, the inheriting object must explicitly be passed to the super-object, which is a standard pattern for simulating class behaviour in E, and we consider this in Chapter 9.

8.2 Delegation

Delegation is a design for object inheritance that aimed to be at least as powerful as inheritance, if not more so since any existing object can be inherited from at any time (Lieberman 1986; Ungar and Smith 1991; Stein, Lieberman, and Ungar 1989). A self request in a method called under delegation goes *back to the original object*, which distinguishes it from forwarding where a self request to a delegatee will be handled only by that delegatee. This allows delegation to support down-calls.

The visualisation of a request to amelia under delegation is presented in Figure 8.2.1. The structure of the objects is the same, but the method implementations

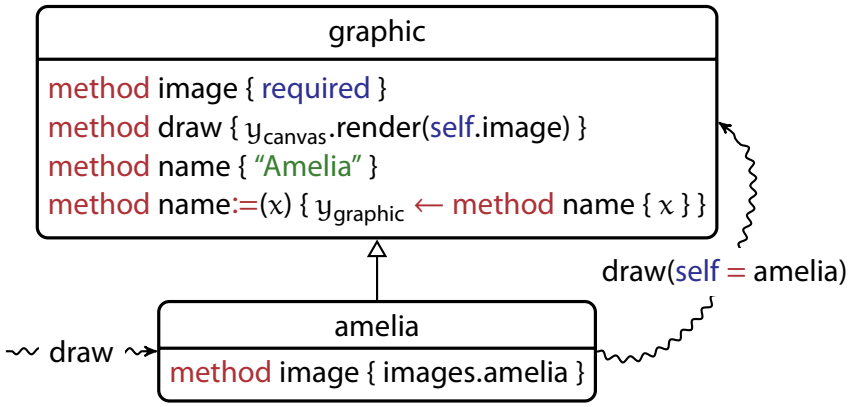

Figure 8.2.1: Visualisation of a delegated request

are slightly different, and the delegation of draw is not an ordinary request. The value of self is no longer early bound in method bodies. The behaviour of the draw delegation is actually exactly the same as under forwarding, but the early binding meant that its behaviour was not distinguishable from an ordinary request. Also note that like forwarding, the assignment to name in amelia updates the method in the super-object: self has been early-bound in the setter of the name field.

Under delegation, the self-request of image in the body of draw does not invoke the abstract method in the same object, but instead returns to amelia who can handle such a request directly, producing the result of images.amelia (which we refer to as y_{image}). This down-call behaviour is visualised by a sequence diagram in Figure 8.2.2. In the previous example, it was not safe to request the draw method on amelia under forwarding, as the implementation of draw expects to be able to see an overridden implementation of the image method. Now this is able to work as expected: draw executes with self as amelia, so the local request to image calls down into amelia and successfully retrieves the image of Amelia.

The modification to the dynamic semantics of forwarding to implement delegation is provided in Figure 8.2.3. As in the forwarding model, the modification early-binds the value of self in the Rule E-Obj, but this time *only as the receiver of an update*. The value of self remains late-bound in the bodies of methods to the receiver of a method request, allowing super-objects to perform down-calls to methods implemented in a sub-object. Fields are shared between the original object and any inheriting objects, and, as under forwarding, mutation of any field

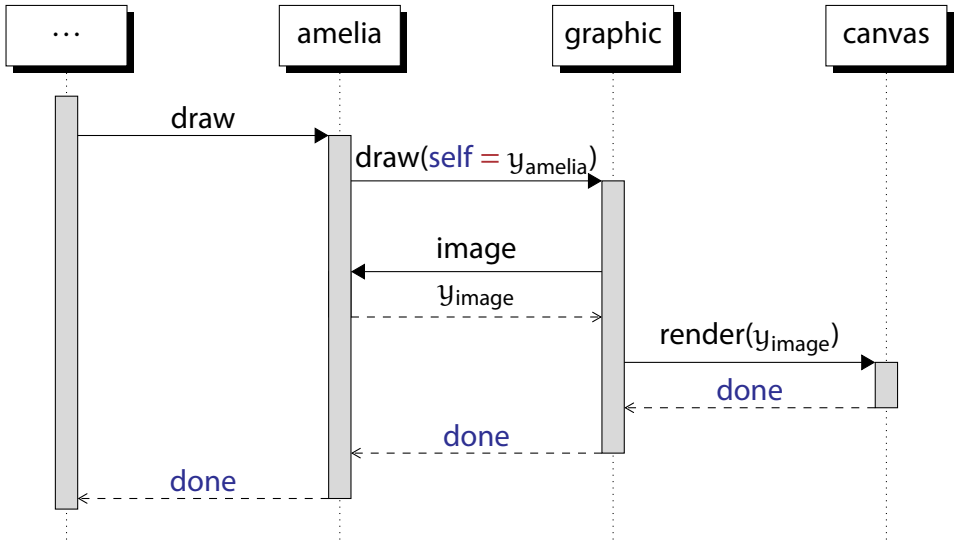

Figure 8.2.2: Sequence diagram of a draw request under delegation

which originated in a super-object is reflected in all of its heirs.

Unlike under forwarding, the self reference in a super-call is not early bound, so super-calls can be chained together, with self always bound by the original receiver of the request. The difference in the method implementation from the forwarding model is partly visualised in Figure 8.2.4, where the *delegate* function has generated a draw method that up-calls to the definition in $y_{graphic}$, and E-Obj has *not* early-bound the value of self in this up-call. This provides the behaviour that the delegating call will bind self to y_{amelia} in the body of the implemented draw method and request image on y_{amelia} instead of the $y_{graphic}$ reference.

Delegation makes no requirement of freshness, which combined with down-calls produces a further complication to information hiding in a language with confidential methods that can only be called on self. These methods may provide access to secret data or capabilities, and the ability to access them from arbitrary

(E-Obj)

$$\frac{y \text{ fresh} \qquad \overline{a = \mathit{identify}(d)} \qquad \overline{a} \text{ unique}}{\sigma \mid \mathsf{object}\,\{\,\overline{d}\;t\,\} \longrightarrow \sigma(y \mapsto \{\,[y/\mathsf{self}\leftarrow]\overline{[\mathsf{self}/a]d}\,\}) \mid [y/\mathsf{self}]\overline{[\mathsf{self}/a]}t; y}$$

Figure 8.2.3: Delegation reduction

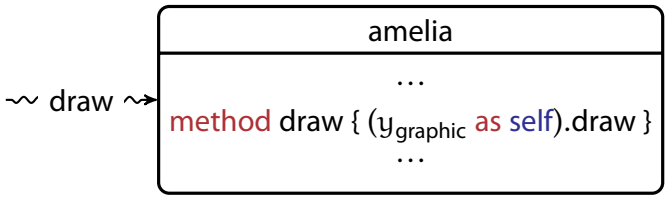

Figure 8.2.4: Visualisation of Graceless delegation inheritance

code could be a security concern. Delegation to preëxisting objects opens the way for the 'vampire' problem: any object to which a reference has been obtained can be taken over and fully controlled from the outside, merely by defining a new child object. If access to confidential methods is instead *not* provided to delegators, simulating common classical patterns becomes difficult or impossible, and an odd asymmetry is introduced: the delegator can override an inherited method, changing its behaviour, but has no access to use the overridden method in its own implementation.

Unlike forwarding, delegation permits down-calls after the object has been constructed, but *not during initialisation* since the inheritance relation has not actually been established while the super-object is being initialised. Objects under delegation cannot perform registration during initialisation, as a captured self reference in a super-object refers to that super-object, which may have no other references and will not have access to any overriding definitions from the child. These two limitations of object initialisation are the major barriers to simulating the typical behaviour of class-based inheritance under delegation.

Object structure and behaviour is not stable during construction, as new methods may appear on self and existing methods may have different implementations depending on the stage of inheritance. Like forwarding, delegation permits inheritance from a preëxisting object; if this is allowed, stability does not exist after construction either. Delegation, like forwarding, can be easily extended to multiple inheritance by introducing multiple inherit clauses and some resolution of multiply-defined methods.

Including the inherit clause in the object constructor ensures that objects can delegate either to a preëxisting prototype object or construct a new object with custom arguments and delegate to that, as an equivalent to calling a super-constructor

in a class system. This distinguishes it from the prototypical inheritance in JavaScript, which requires that the prototype property of the constructor be set before any inheriting object is constructed. In Self, object expressions can set their parents as a fresh object constructor call:

```
( | parent* = factory new. | )
```

This has the same semantics in terms of self binding in the super-constructor as presented in our model of delegation.

One of the distinguishing features of the particular form of delegation that we have presented here is that an object shares fields with its parent. Requesting the name:= method on the graphic object is visualised in a sequence diagram in Figure 8.2.5. The change to the super-object graphic is reflected both in graphic and in amelia: not unsurprising, since name is inherited from graphic by amelia.

Requesting the same setter method on amelia has the same effect, since the field is equally shared between amelia and graphic. This behaviour is visualised in the sequence diagram of Figure 8.2.6. Note that this means that *any other object* inheriting from graphic will also be affected by a request to a setter method in amelia: this is the crux of the Action at a Distance concern.

8.2.1 Receiver mutation

Removing the early-binding of self in an object entirely provides a different kind of delegation, similar to the one found in JavaScript. If self is bound to the original receiver of the request in method updates as well, then it is this original receiver that is mutated instead of the object that the method update appeared inside of. Inheriting objects share the fields of their super-object until the setter is requested on the lower object, at which point the lower object has its own field. This form of delegation does *not* have action at a distance through field setters.

In our running example, requesting the name method on either graphic or amelia produces the same result: amelia just goes through one layer of indirection for the super-call. The two objects (and any other object inheriting from graphic) share the field value: the behaviour is exactly the same as in Figure 8.2.5. As soon as the name:= method is called directly on amelia, though, amelia's name method

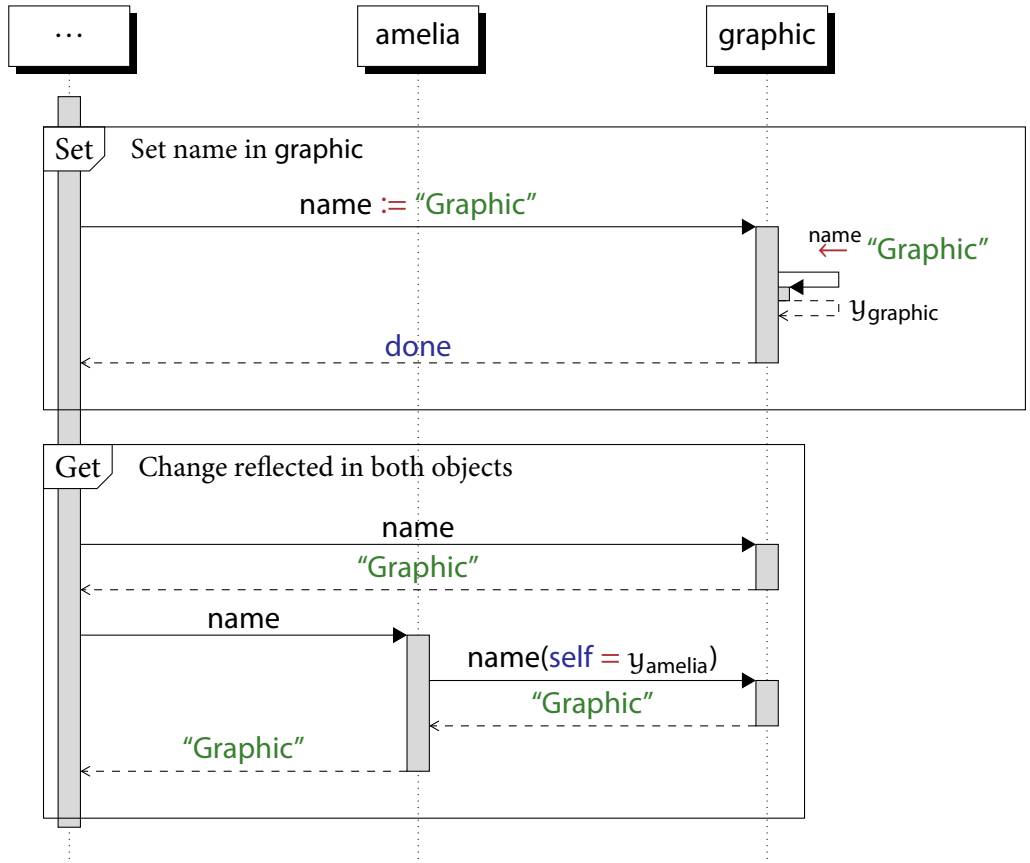

Figure 8.2.5: Field assignment to graphic under delegation

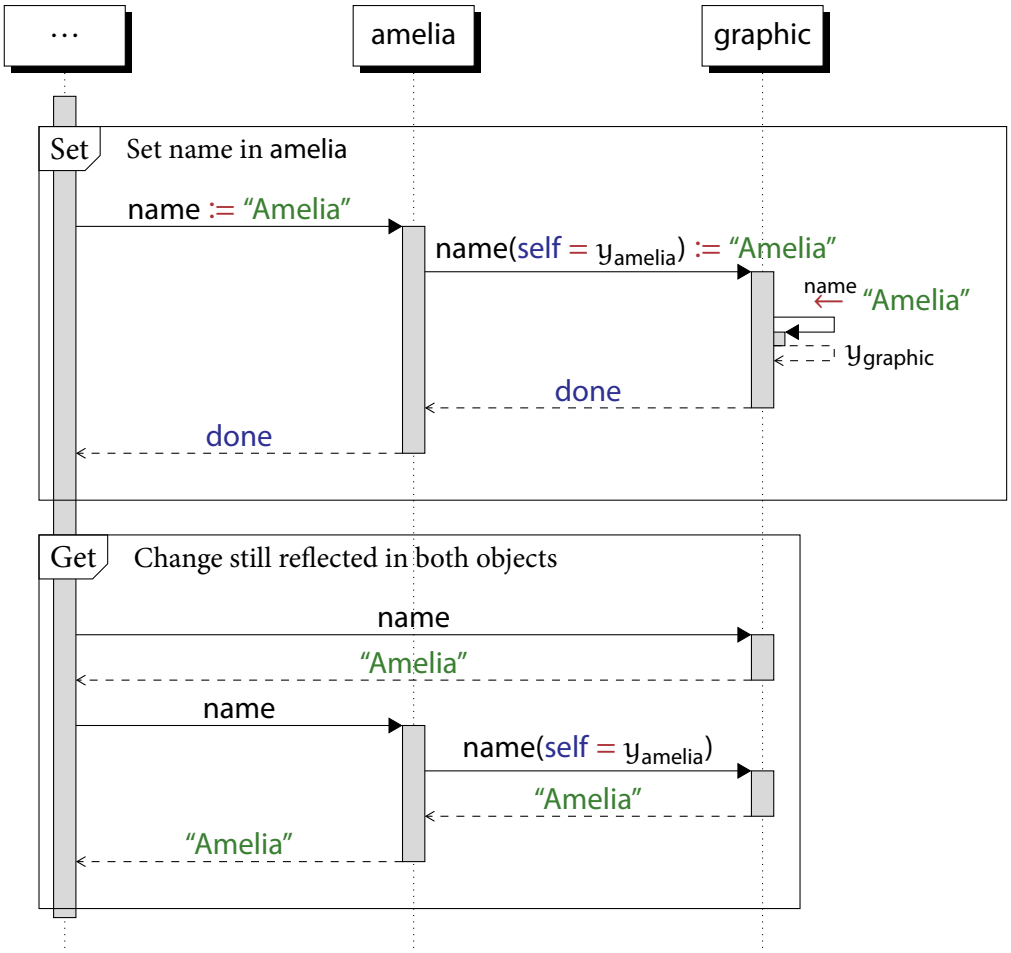

Figure 8.2.6: Field assignment to amelia under delegation

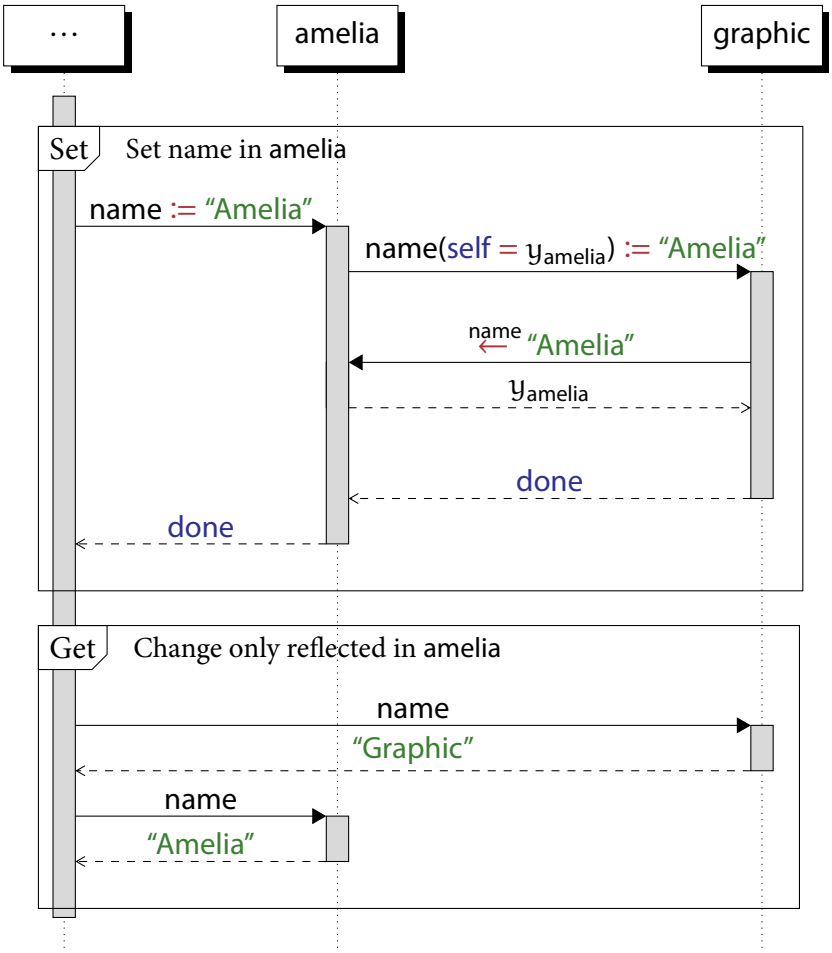

Figure 8.2.7: Field assignment to amelia under receiver mutation

is replaced by the assigned value, and will no longer look up the method in the super-object.

The reason this happens is that, while amelia still applies a super-calls to the name:= method, thanks to the behaviour of Rule E-R/S the binding of self in the term self ← method name { x } points back to the object that the request was originally made to: amelia. Field assignments always affect the original object, so there is no action at a distance. This sequence of events is visualised in a sequence diagram in Figure 8.2.7.

This behaviour is also what happens in JavaScript: fields are shared between objects until an inheriting object is assigned to, at which point the assigned field is

unique because it is inserted directly into the object itself.

```
const amelia = Object.create(graphic(canvas))

graphic.name = "Graphic"
amelia.name // ⟹ "Graphic"

amelia.name = "Amelia"
graphic.name // ⟹ "Graphic"
amelia.name // ⟹ "Amelia"
```

This is crucial to the language (or at least was before property descriptors; we can now completely initialise and seal amelia in the call to Object.create in modern JavaScript), because properties are not declarative and the methods in any object are set up imperatively by assigning functions into fields. The model presented here differs in that it will not update a method definition if there is not an already existing method with the same identity, so it is more accurate to say it behaves like JavaScript where every object is sealed, which prevents new properties from being added.

This behaviour is much more difficult to reason about, because field setters are no longer conclusively tied to their corresponding getter. If amelia had chosen to override the name method, calling the name:= method would ignore the fact that the newly declared method is not strictly a getter method (there is no distinguishing feature that marks it as such), and wipe this overriding definition for the assignment value.

8.3 Concatenation

Concatenation is an alternative design for object inheritance that aimed to have the power of inheritance without the drawbacks of delegation (Taivalsaari 1997; Taivalsaari 1995). Under concatenation, one object inherits from another by (conceptually) taking a shallow copy of the methods and fields of its parent into itself, and then appending local overriding definitions. Concatenation supports downcalls, but unlike delegation does not allow subsequent changes in either the parent or the child to affect each other.

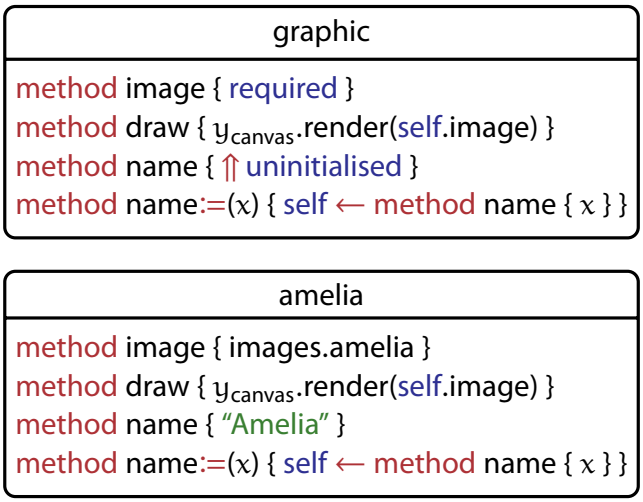

Figure 8.3.1: Visualisation of objects under concatenation inheritance

In the conceptual rendering of our example, there is no longer any connection between the graphic and amelia objects. The methods in the graphic object are concatenated into the definition of amelia (excluding overrides) and amelia is initialised as a single object. This is visualised in Figure 8.3.1. The difference in behaviour is immediately apparent in the different values of name. The assignment to name in the initialisation of amelia only affects amelia, and the change is not reflected in the graphic object or any other inheritor.

An incoming request to draw requires no special behaviour, as the draw method is now defined directly in amelia. Figure 8.3.2 visualises this final behaviour, which never refers to graphic at all. The objects in an inheriting relationship under concatenation are not as divorced as they first appear, though. Consider the fact that amelia is able to directly request render on the canvas reference: the y_{canvas} reference actually appears in the scope of the graphic method. If there are mutable references in the scope of an inherited object, then these can become shared mutable state between that object and any inheritor.

Our implementation of concatenation semantics is a bit more complicated than just copying the methods from the inherited object, mostly to deal with super-calls and overridden methods while avoiding the strange behaviour of the base model. The updated reduction judgement is defined in Figure 8.3.3 as an extension to the previous delegation semantics. This modification changes the Rule E-Inh to clone

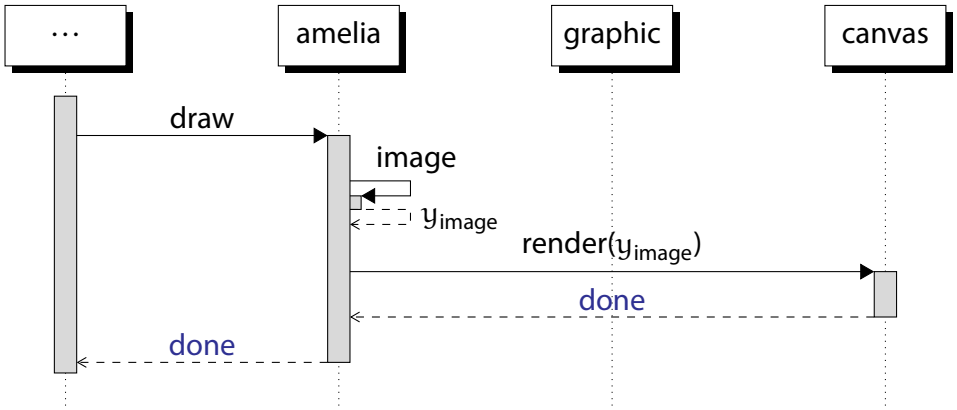

Figure 8.3.2: Sequence diagram of a draw request under concatenation

the inherited object, and substitute the clone as the super-object of the inheritor instead. These semantics are exactly delegation where inherit clauses clone the result of evaluating their term before applying the inheritance relation. An auxiliary definition *recast* also ensures that the clone reference is cast in the same way as the original inherited reference (including not cast at all).

The existing late-binding of self in methods is sufficient to provide the desired behaviour: any inherited method executes in the context of the inheriting object, and any request to an inherited field accessor will access the copied field in the inheriting object as well. The only relationship the inheriting object has with its super-object is explicit up-calls, but it is impossible to access or modify the state of the super-object without explicitly referring to it through an existing reference.

$$\text{(E-Inh)}\quad \frac{\sigma(v_\uparrow) = \{\ \overline{d_\uparrow}\ \} \qquad \overline{d'_\uparrow} = \mathit{extend}(\overline{d}, \overline{d_\uparrow}) \qquad y_\uparrow \text{ fresh}}{\sigma \mid \text{object}\{\ \text{inherit } v_\uparrow\ \overline{s}\ \overline{d}\ t\ \} \longrightarrow \sigma(y_\uparrow \mapsto \{\ \overline{d_\uparrow}\ \}) \mid \overline{[s]}\text{object}\{\ \overline{d'_\uparrow}\ [(\mathit{recast}(v_\uparrow, y_\uparrow) \text{ as self})/\text{super}](\overline{d}\ t)\ \}}$$

$$\begin{aligned} &\mathit{recast} : \textsc{Term} \times \textsc{Term} \rightharpoonup \textsc{Term} \\ &\mathit{recast}(v : S,\ y) = \mathit{recast}(v, y) : S \\ &\mathit{recast}(v,\quad y) = y \end{aligned}$$

Figure 8.3.3: Concatenation reduction

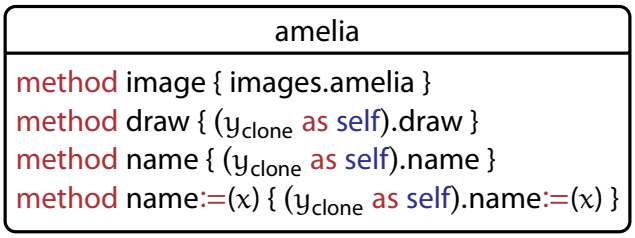

Figure 8.3.4: Visualisation of Graceless concatenation inheritance

The concatenation model builds on delegation because it is the same as delegation except that inheriting from an object clones the object first.

The cloned super-object is effectively invisible outside of the inheritance implementation, so it *appears* as though the method is copied directly into amelia, but the reality is that the incoming request to draw is just sent to the cloned super-object as under delegation. Field values are still shared between the inherited clone and inheriting object, but because no other part of the program has a reference to the clone this sharing cannot be observed, and it appears as though the inheriting object has its own personal set of fields copied out of the inherited object. The implementation of the amelia object is visualised in Figure 8.3.4, with y_{clone} as the reference to the clone of graphic that resulted from the inheritance of amelia.

Concatenation also makes no requirement of freshness. Concatenation with preëxisting objects does not quite permit the 'vampirism' of delegation, but does allow 'mind reading': any confidential state in an object can be read simply by inheriting from it, but the existing object cannot be manipulated by the child. Unlike the two previous models, mutations to inherited fields do not cause action at a distance, as the mutation will always affect a field in the actual receiver of the request (even for super-calls). With lexical scoping, inheriting objects are also not as independent as they seem: methods exist in the same scope in both inheritor and inheritee, and any lexically captured state is shared between the two. Since the clone is shallow, mutable state within objects in copied fields is also shared between objects.

Like delegation, concatenation permits down-calls after the object has been constructed, but not during construction. Concatenation does not allow registration, as a captured self reference in a super-object refers to the parent. Object struc-

ture and behaviour is not stable during construction; as for delegation, if inheritance from preëxisting objects is allowed then stability does not exist afterwards either. Concatenation can be straightforwardly extended to multiple inheritance by inserting the contents of each parent into the child, with some resolution of multiply-defined methods.

8.3.1 In Other Languages

Objects with mutable structure can trivially implement concatenation, by manually copying the structure of the inherited object into the inheriting one. JavaScript complicates this story with innumerable properties, implicit field accessors, and existing delegation relationships, but for the most part this is a valid implementation of concatenation:

```
for (let name in inheritee) { inheritor[name] = inheritee[name] }
```

It is also possible to use mutable object structure to implement either forwarding or delegation, by assigning methods (or field accessors) to the inheriting object that directly forward or delegate to the inherited object, as in our models. In a JavaScript constructor:

```
const self = this
this.foo = function () { self.bar() }
```

If the `foo` method is called on a sub-object, the call to `bar` is guaranteed to not perform a down-call, because `self` is bound directly to the original object.

JavaScript's built-in object inheritance otherwise works the same way as Graceless delegation, but field assignments are directly available in the language instead of only through accessor methods. The result is that an object shares each field of its inherited object, but when a field is assigned to it, the field is unique to that object, shadowing the inherited one. In any language where the objects have mutable structure it is necessary to retain a parent reference in order to implement delegation, in order to accurately defer to the current implementation of a parent object. Implementing more complicated forms of inheritance in JavaScript, such as with traits or mixins, tends to involve combining the built-in delegation alongside manual concatenation.

9 Emulating Classes

The inheritance models in the previous section represent the three foundational strands of purely object-based inheritance. Class-based languages tend to provide different semantics for inheritance, and programmers and language designers may wish to use those semantics or expect classless languages to behave in a similar way.

All of the object inheritance semantic models above have the same registration behaviour during construction: references to self in a inherited object's initialisation code are the identity of the parent object, because the object must be constructed and initialised before the inheritance can actually occur. When the graphic object's initialisation registers itself with the canvas in canvas.register(self), the object that is registered with the canvas is the graphic object, not amelia. This will lead to problems if the canvas intends to draw objects in its registry: the draw method of a graphic object is guaranteed to fail, because it will always request the unimplemented image method.

A sequence diagram of the behaviour of our running example (ignoring the down-call during initialisation), and a subsequent attempt by the canvas to draw the contents of its registry, is provided in Figure 9.0.1. The argument to the register method request is y_{graphic}, not y_{amelia}, so when the canvas is asked to update, it attempts to draw the incomplete graphic object instead of the fully implemented amelia and produces an error.

It is possible to construct object-based models that emulate classical semantics. In some languages with very flexible semantics, such as JavaScript and Lua, libraries exist to provide 'classes' as a second-class construct by mutating objects or leveraging specially-constructed objects with the existing inheritance systems (Lua-Users 2014). In this section we introduce two models that approximate the inheritance behaviour of C++ (§9.2) and Java (§9.3) respectively in an object-based system. They

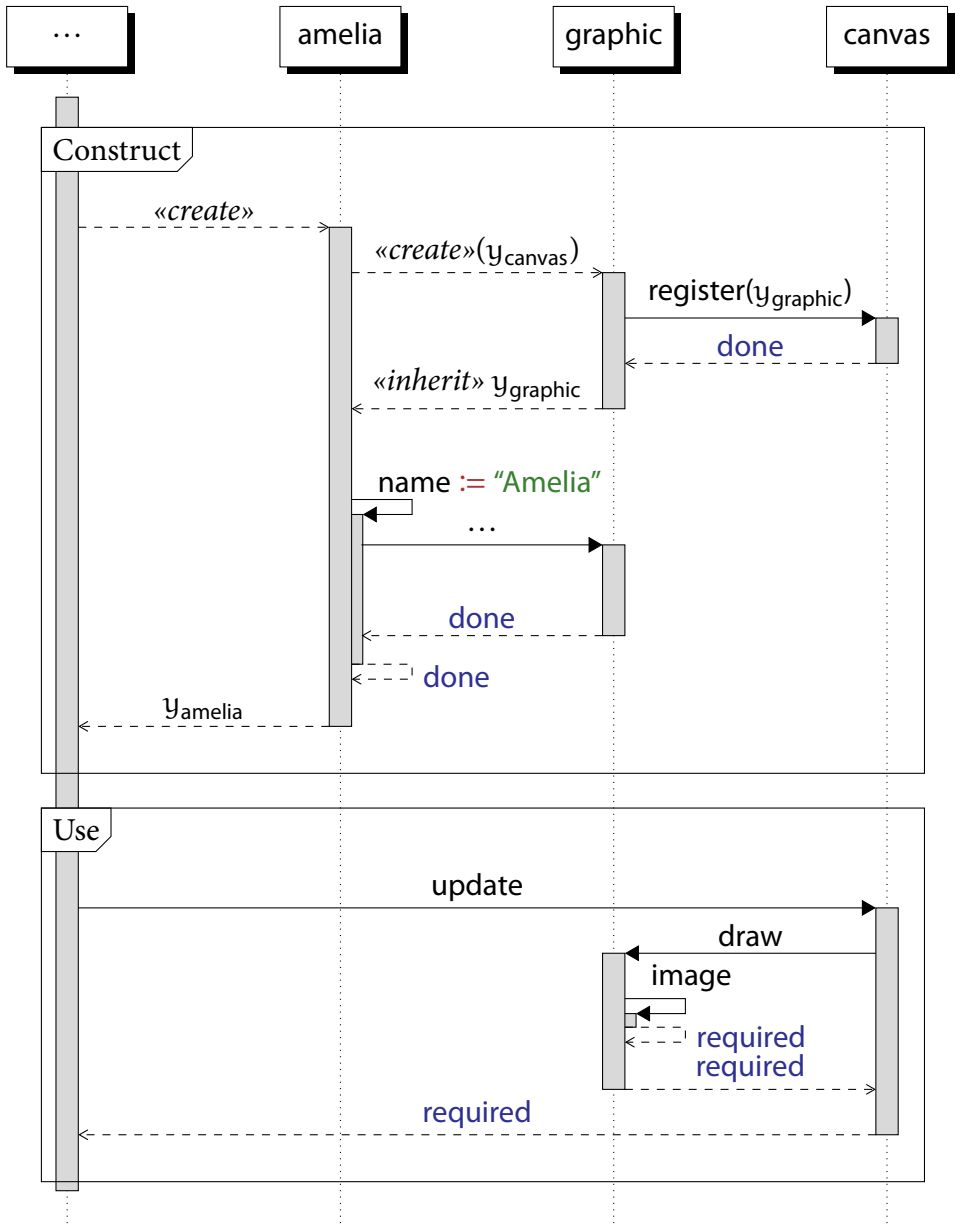

Figure 9.0.1: Sequence diagram for registration during construction

remain purely object-based, but trade off some of the flexibilities of the object inheritance models for the classical functionalities they provide, including registration, but also the behaviour of down-calls, and method definition stability.

9.1 Object Freshness

The key property that is enforced in the following models is *freshness*, requiring that only newly constructed objects may be inherited from. This is a syntactic property that means that terms in an inherit clause must reduce to an object constructor during their evaluation. More concretely, this means that preëxisting objects cannot be inherited from, as every inherited object must be constructed at the point of inheritance. Inheriting from the result of a request for graphic in amelia satisfies this condition, since the inherited object is constructed by the method. Constructing the object beforehand and then inheriting from it is not permitted:

```
def parent = graphic(canvas)
def amelia = object { inherit parent; ··· }
```

A fresh object expression is either an object constructor or a request to a method that *tail-returns* an object constructor. A method tail-returns an expression when it is returned as the final statement of the method, with no intervening explicit returns elsewhere in the body of the method, ensuring that if an object is returned it must have been the result of evaluating this final expression. The requirement that a method must tail-return an object constructor is stricter than necessary, since a method that tail-returns a request to another fresh object expression is also guaranteed to return a fresh reference. Explicitly requiring an object constructor is easier to implement, and though this means that a request to a method that tail-returns a request that is otherwise a fresh object expression cannot be inherited, it can always be modified to be inheritable using Homer's Device[1]: a method method m { t } where t is a fresh expression can instead be implemented as method m { object { inherit t } } to ensure that requests to it can be inherited from.

Rather than mutating preëxisting objects, inheritance must now occur from a

[1]Named for our colleague, Michael Homer.

$$\text{(E-I/C)}\quad \frac{\sigma \mid t \longrightarrow \sigma' \mid t' \qquad t = y.m(\overline{v}) \implies t' = (\overline{t};\ \mathsf{object}\ \{\cdots\})}{\sigma \mid \mathsf{object}\ \{\ \mathsf{inherit}\ t\ \overline{s}\ \overline{d_{\downarrow}}\ t_{\downarrow}\ \} \longrightarrow \sigma' \mid \mathsf{object}\ \{\ \mathsf{inherit}\ t'\ \overline{s}\ \overline{d_{\downarrow}}\ t_{\downarrow}\ \}}$$

Figure 9.1.1: Fresh inheritance reduction

fresh object, newly created and immediately returned from a method call. Without the requirement of freshness, inheriting objects can directly steal the identity of any other existing object, inserting its own definitions so that any existing reference is modified to work as the thief sees fit. The resulting 'body-snatchers' problem is substantially worse than delegation's vampirism, which can only control objects internally. The freshness requirement also prevents inheritance from a cast, since any intervening cast would syntactically interfere with the tail-return of the object that it wraps.

The models that make use of freshness use as their base a further extension to Graceless inheritance, provided in Figure 9.1.1, which enforces that the inherited object is a fresh reference. This extension modifies Rule E-I/C by adding a predicate so that method requests in an inherit clause may only be computed if they syntactically return an object expression. We abuse notation slightly here, given that sequences are defined in the grammar as pairs: the equality we draw here is insisting that the rightmost term in *any series* of sequences is an object constructor. This modification does not affect terms that are not requests in an inherit clause, and still permits an object constructor to be resolved to a reference before processing the inherit clause itself.

The first model, *merged* identity, builds directly on this new model, and *uniform* identity further modifies the context rule to also prevent an object expression from being reduced when it appears in an inherit clause. The requirement of syntactic freshness is not strictly required for merged identity, but it prevents potentially dangerous manipulation of an object's definitions from outside of an object's creator. Both models ensure that the identity of the resulting object is a consistent, single value, and they both use the new restriction applied to inherit clauses by the modified context rule.

9.2 Merged Identity

The goal of the merged identity model is to ensure that object registration in a constructor stores the (eventual) identity of the inheriting object, without changing the semantics of an inherited request. An inheriting object takes over the identity of its parent, 'becoming' that object but putting in place all of its own method definitions. The semantics of the model is effectively the reverse of concatenation: the definitions are copied up into the inherited object, rather than down into the inheriting one. The parent object is constructed and initialised before the child (as with the previous models), but is mutated when the inheritance relation is established, and no new reference is allocated for the child object, since this is subsumed by the reference to the parent.

Merged identity models the core of the C++ inheritance behaviour: the apparent type of the object changes during construction, as each layer of inheritance is processed. After object construction is complete, down-calls resolve to their final overridden method, but until then they obtain the most recent definition from at that point in the construction process. The stability of an object's definitions is guaranteed after construction by the requirement of freshness. Overridden methods from the parent remain accessible through super, and will always execute with the final identity of the object.

Returning to the example of amelia, the registration of self in the body of the graphic constructor under merged identity will now correctly store the value of amelia when initialising the super-object. Although the object is not yet the final version of amelia when it is registered (since the inherit clause in amelia's constructor has not finished evaluating its term yet), amelia *becomes* the object that was registered once it has finished initialising, merging the two identities together. The initialisation of the graphic object will still fail at the local request of draw, because amelia's overriding of the abstract method image has not yet been merged into the object at that point in the object's initialisation. Unlike in C++, where this is regarded as an unreliable behaviour to be avoided, under merged identity this effect is fundamental to the semantics.

Consider the conceptual visualisation of the object's construction in the middle of the graphic object's initialisation presented in Figure 9.2.1. The graphic object

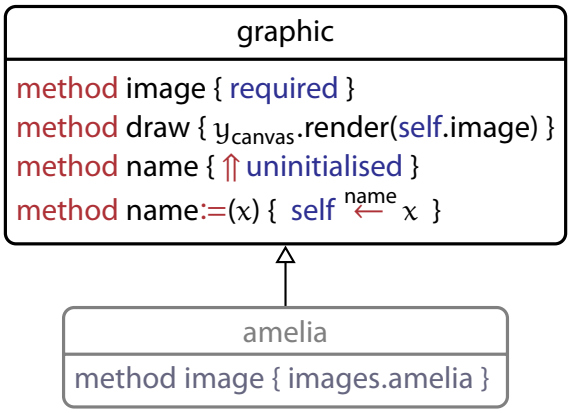

Figure 9.2.1: Conceptual visualisation of merged identity initialisation

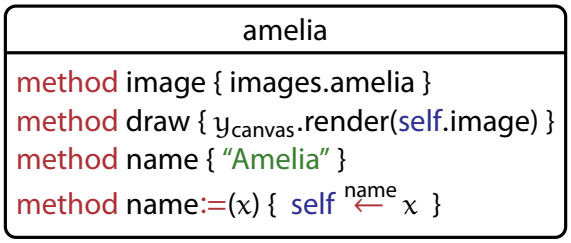

Figure 9.2.2: Visualisation of amelia under merged identity

in the visualisation is fully constructed, and is in the middle of running its initialisation code. The representation of amelia is greyed-out to represent that, although the code path has entered into amelia's object constructor (since this is where the inheriting request that generated the graphic object began), the actual amelia object has not been constructed, and does not have its own identity.

After the graphic object finishes its initialisation, amelia completes the inheritance relationship by inserting and replacing method definitions as under concatenation, but into the existing graphic object instead of into a freshly allocated object. The final object is visualised in Figure 9.2.2, with the conceptual amelia constructor gone without actually constructing a separate object. As under concatenation, there is no reified inheritance relationship between objects once they are constructed, but unlike concatenation there is only a single object at the end of the inheritance process.

An updated form of the sequence diagram from Figure 9.0.1 is provided in Figure 9.2.3. The amelia object no longer exists at any point in the program, so

it does not appear in the diagram. Once the graphic object is finished initialising, it is extended by the amelia constructor, which modifies the object's method definitions and runs the initialisation of the amelia constructor from within the existing graphic object (indicated by the *«extend»* special form). The argument to the register request is still the identity of the graphic object γ_{graphic}, which was the source of the error in Figure 9.0.1, but now the graphic object has been modified to include the overridden image definition, so the program no longer requests an unimplemented method.

The modification to Graceless with fresh inheritance to implement merged identity is provided in Figure 9.2.4. The Rule E-Inh is the only part of the original Graceless inheritance model which is changed. Where the original implementation of the rule in Figure 8.0.3 resulted in a new object expression with some of the methods from the super-object, this new rule skips straight to returning the resulting reference. This is necessary because the resulting reference is not fresh: it is the reference of the inherited object. The inherited object is updated with the new methods in the store, and overridden methods are removed and replaced.

One problem with implementing merged identity is that there is no longer an object which corresponds to super to substitute into the body of the object, since the parent identity is used as the identity of the child as well. In order to implement a super object, Rule E-Inh clones the parent object before mutating it with the definitions of the child. The super-object clone has all of the methods of the inherited object before it was modified, while the merged object has its methods replaced with up-calls to this new super-object. As under the concatenation model, the clone is invisible outside of the inheritance relation because it is impossible to get a direct reference to a super-object. All of the models that emulate classes with super-references create these 'part-objects', which exist purely to store the inherited methods for super calls, and are only ever referenced under an alias for the 'real' object that is bottom-most in the hierarchy.

The structure of part-objects in our model is visualised in Figure 9.2.5. The super reference is a clone of the graphic object before it was modified by the inheritance. The only reference to the clone is as a super reference in the inherited object. Since super is not a valid term by itself, it is impossible for it to appear in any other context as a bare reference.

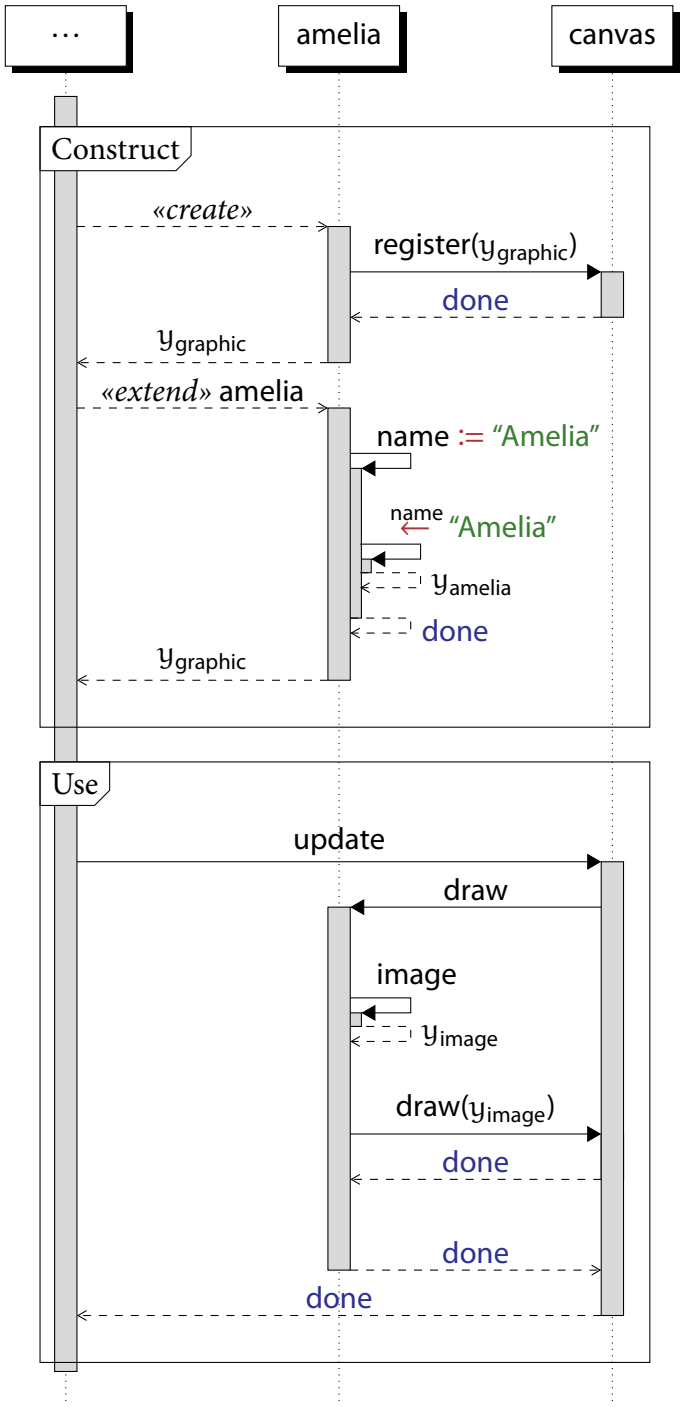

Figure 9.2.3: Sequence diagram of registration under merged identity

(E-Inh)

$$\frac{\sigma(y) = \overline{d_\uparrow} \qquad \overline{d'_\uparrow} = \mathit{extend}(\overline{d}, \overline{d_\uparrow}) \qquad y_\uparrow \text{ fresh} \qquad \overline{a = \mathit{identify}(d'_\uparrow; d)} \qquad \overline{a} \text{ unique} \qquad \overline{[s']} = \overline{[s][\mathsf{self}./a]}[(y_\uparrow \text{ as self})/\mathsf{super}]}{\sigma \mid \mathsf{object}\,\{\,\mathsf{inherit}\; y\; \overline{s}\; \overline{d}\; t\,\} \longrightarrow \sigma(y_\uparrow \mapsto \{\,\overline{d}\,\})(y \mapsto \{\,\overline{d'_\uparrow}\; \overline{[s']d}\,\}) \mid [y/\mathsf{self}]\overline{[s']}t; y}$$

Figure 9.2.4: Merged identity reduction

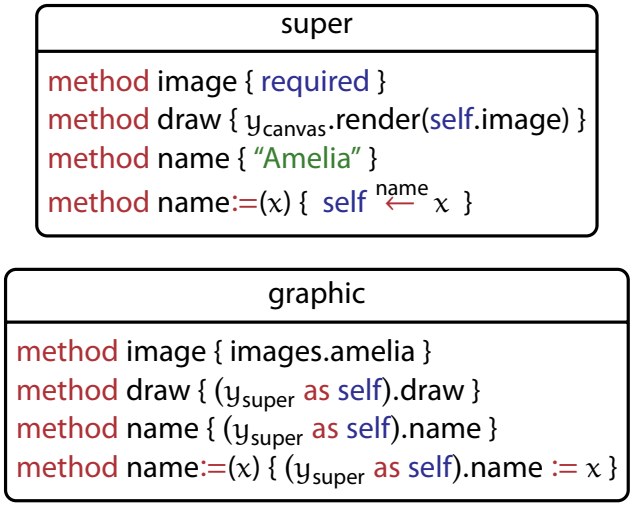

Figure 9.2.5: Visualisation of merged identity part-objects

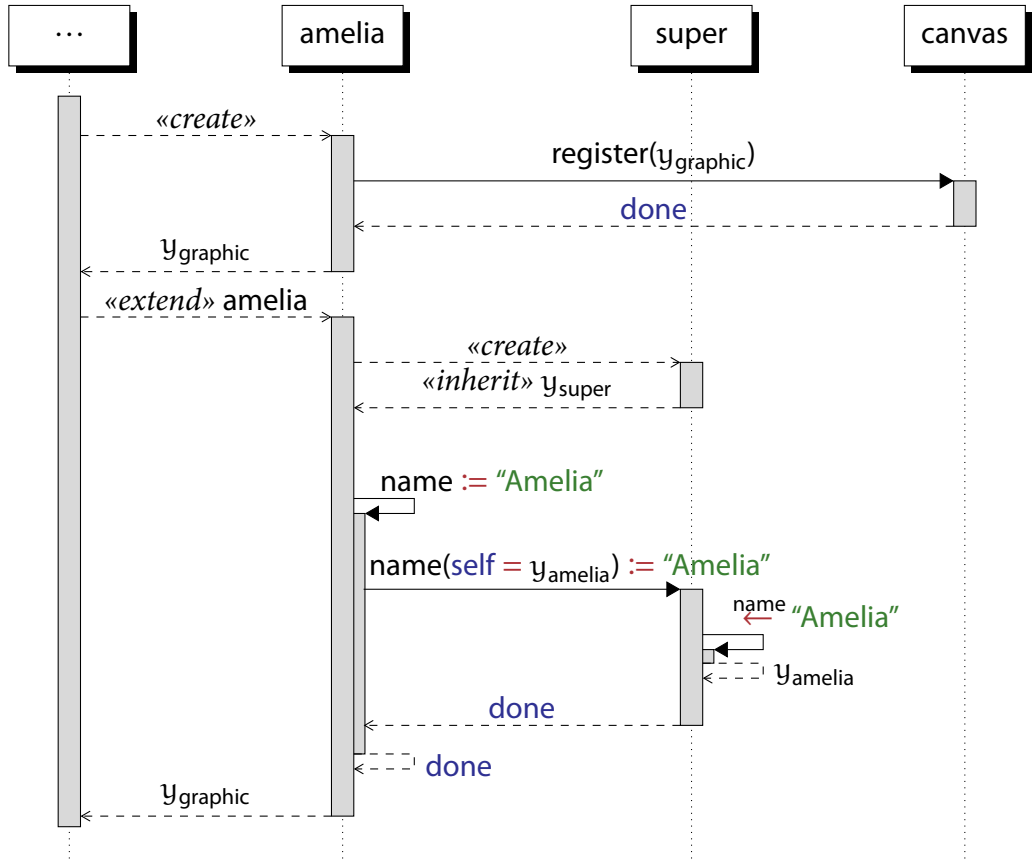

Figure 9.2.6: Sequence diagram of initialisation under merged identity model

A sequence diagram of our model undergoing the same process as in Figure 9.2.3 is presented in Figures 9.2.6 and 9.2.7. The outcome is the same, the only difference being the creation of the super-object and some extra communication between the graphic object and the super-object instead of graphic handling its own behaviour entirely by itself.

Rule E-Inh is a combination of the behaviour of the old Rule E-Inh from Figure 8.0.3 and Rule E-Obj from Figure 4.3.1. We have highlighted the changes to their combined behaviour. We take some liberties here on what it means for the two to be combined: note, for instance, that the inherited methods that are not overridden $\overline{d'_{\uparrow}}$ appear with the ordinary methods $\overline{d}$ in the series of calls to the signature function. While no such combination appeared syntactically in the old rules, the combination was implied by an application of E-Inh and then E-Obj, as the in-

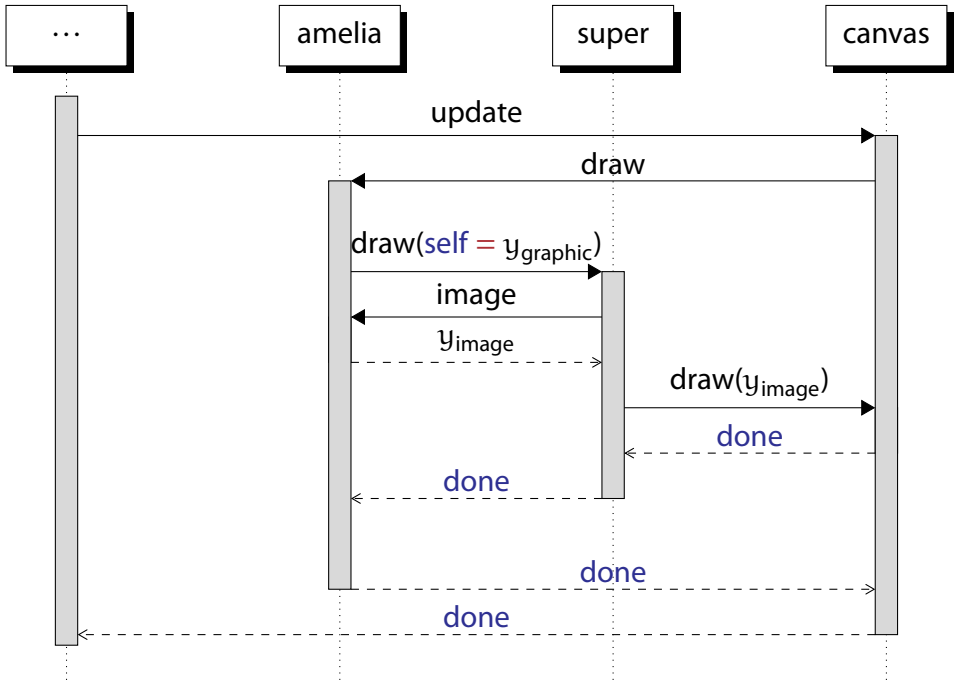

Figure 9.2.7: Sequence diagram of use under merged identity model

herited methods would be in the series $\overline{d}$ by the time the second rule was applied. Similarly, the use of $[\overline{s'}]$ is new, but it is just abbreviating an existing sequence of substitutions.

Like delegation and concatenation, merged identity enables down-calls after, but not during, initialisation. Unlike those models, a single identity is preserved throughout the construction process, so registration stores the final object identity. It does not provide stability during initialisation, but once objects are complete they cannot change again. The mutation of an object's definition when merged means that it unsafe to permit inheritance from arbitrary objects, so preëxisting objects cannot be inherited from. Merged identity is really the only model presented in this paper that does not lend itself well to multiple inheritance, because only a single parent can have its identity preserved.

9.2.1 In Other Languages

As a reverse form of concatenation, merged identity is just as easy to implement for objects with mutable structure. A constructor can call into the super-constructor, then concatenate its own definitions into the resulting object. Copying the original object beforehand ensures that a super-reference is still available. Simulating merged identity in a JavaScript constructor simply requires referring to the parent object directly instead of this:

```
function Amelia() {
    const parent = graphic(canvas);
    parent.image = images.image;
    parent.name = "Amelia";
    return parent;
}
```

Such an implementation does not play well with the existing prototype system of JavaScript, though.

Emulating merged identity on top of JavaScript's existing prototype inheritance is also simple, as any use of inheritance where methods are added to an object in its constructor rather than through a prototype is effectively merged identity:

```
function Amelia() {
    // Inherit from the Graphic constructor
    Graphic.call(this, canvas);
    // Then add the image and name definitions to this object
    this.image = images.image;
    this.name = "Amelia";
};
```

Modifying methods in a JavaScript constructor isn't strictly the form of merged identity we have presented here because the identity of the resulting object is allocated at the bottom of the inheritance hierarchy and passed up to the top-most constructor, instead of actually being allocated at the top, but the behaviour is equivalent because the methods of the inheriting object are not added until after the initialisation of the inherited object, and there is still only a single object identity

at the end of the inheritance.

A form of merged identity can also be achieved in E, despite the fact that the implementation of an object is immutable, because you can pass the reference of an inheriting object into the closure of its inherited object (Miller 2006). Methods cannot be requested on this reference until after the super-constructor has completed, but it will behave correctly under registration, and down-calls are possible in the methods of the inherited object. In E:

```
# Take self as a parameter
def graphic(self, canvas) {
    def parent { # Use self instead of parent
        to draw() { canvas.render(self.image()) }
        ...
    }
    canvas.register(self)
    return parent # Except to return parent
}

# Refer to amelia before construction, get a promise instead
def amelia extends graphic(amelia, canvas) { ... }
```

The self reference in the call to the graphic method points to a *promise* of some inheriting object, to be fulfilled and replaced by amelia when the method completes, so it is safe to register the object with the canvas, and calling image on it in the draw method will correctly perform a down-call. The draw method must not be not called until after the graphic method is finished and amelia finishes extending from the parent object, since the promise has not yet been fulfilled and so no such method is present. There are still two unique object identities, but as long as we do not refer to parent in the graphic method other than to return it, we end up with the merged identity semantics.

The graphic method included a down-call to the draw method during initialisation that we have been omitting because it failed in every model we have presented so far. If we were to add this back in the E program above (in between the registration with canvas and the return of parent), we would have two options: either call

the method on self, or on parent. Neither one will work, though: either way we end up requesting a method on self before it has been constructed, which results in an error. This is a problem with merged identity, not with E: in fact, the language has a clever mechanism to get around this problem, which resembles our next model, uniform identity.

9.3 Uniform Identity

Uniform identity is a direct attempt to implement the behaviour of a typical class-based inheritance system, but based on objects rather than on classes. The *first* observable action is to create a new object identity in the bottom-most 'child' object constructor; this exactly mirrors the merged identity design, which creates a single identity of the *top-most* parent. All inherited declarations are assembled in a single object associated with the initial identity, but no fields are initialised and no initialisation code runs until the entire inheritance hierarchy is established. Inheritance occurs with the identity 'passed along': declarations are attached to the original object, without initialisation, until the topmost object with no parent is reached. Finally, the initialisation code runs from top to bottom.

All initialisation occurs in the context of the final object. No new methods or overrides are added visibly during construction. Initialisation code always sees objects as a consistent type, though it may observe uninitialised fields. The semantics of uniform identity essentially aligns with the semantics of Grace's inheritance, as well Java-like languages in general.

We illustrate the conceptual object structure of amelia inheriting from the graphic object mid-initialisation in Figure 9.3.1. This time the super-object is the one greyed-out, as amelia is the first object that is constructed, before inheriting from the graphic object. Rather than constructing a distinct identity for the graphic object, its definitions are added to amelia, and then its initialisation code is executed with self as amelia.

The addition of the graphic object's definitions to amelia is in reverse to all of the semantics we have described so far, as overriding methods are *already in the object* when it comes time to add the definitions that they are overriding. An overriding method prevents its corresponding super-definition from being added to the

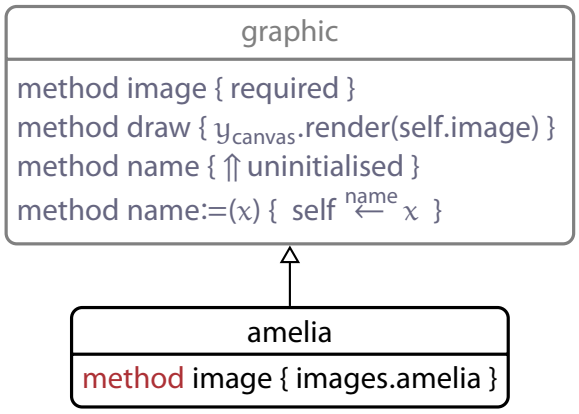

Figure 9.3.1: Conceptual visualisation of uniform identity initialisation

object instead of replacing it: the overridden method is added to a corresponding part-object instead. The resulting amelia object is the same as under merged identity, visualised in Figure 9.2.2, because only the initialisation behaviour is different.

A sequence diagram of amelia's initialisation behaviour under uniform identity is provided in Figure 9.3.2. The first thing that amelia does after being constructed is request the graphic method, extending amelia with the resulting object's definitions *before* the graphic object's initialisation is run (indicated by amelia's self application of the inherit special form), and then running the initialisation with self as amelia.

The modification to Graceless to implement uniform identity is provided in Figure 9.3.3. As with merged identity, the Rule E-Inh overwrites the existing Rule E-Inh. The Rule E-I/C has been further refined to also prevent the evaluation of object expressions directly inside an inherit clause. This is a direct consequence of the 'passed on' object identity behaviour of inheriting objects, where the identity of the inherited object constructor is rewritten into the inheriting object. It also illustrates how requiring fresh objects for inheritance is now a requirement of the semantics, rather than just a security concern.

Once again, the semantics of uniform inheritance in our model lines up with the conceptual behaviour discussed above, but the internal behaviour of the model differs. As under merged identity, we create part-objects to support super-calls and overriding field setters, and these part-objects are created from top to bottom (in contrast to our earlier assertion that the bottom identity is the first one that is

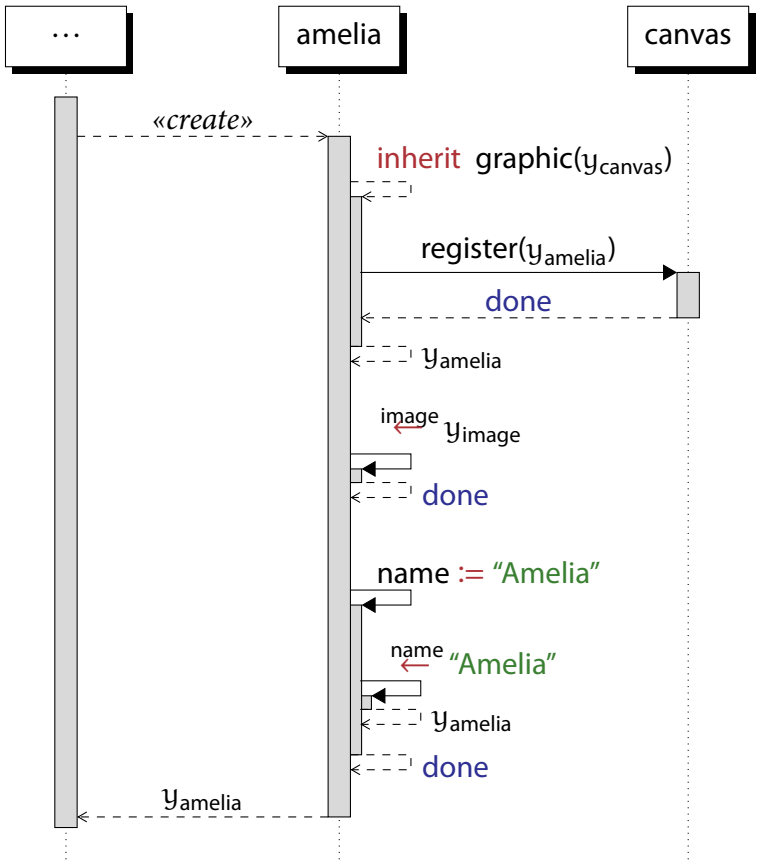

Figure 9.3.2: Sequence diagram of initialisation under uniform identity

$$\text{(E-I/C)}\quad \frac{t \neq \text{object}\,\{\,\overline{d_\uparrow}\;t_\uparrow\,\} \qquad \sigma \mid t \longrightarrow \sigma' \mid t' \qquad t = y.m(\overline{v}) \implies t' = \overline{t}; \text{object}\,\{\cdots\}}{\sigma \mid \text{object}\,\{\,\text{inherit}\;t\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,\} \longrightarrow \sigma' \mid \text{object}\,\{\,\text{inherit}\;t'\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,\}}$$

$$\text{(E-Inh)}\quad \frac{\begin{array}{l} y \text{ fresh} \qquad \overline{a = \mathit{identify}(d)} \qquad \overline{a} \text{ unique} \qquad \overline{d_\uparrow} = \overline{[\text{self}./a]d} \\ \overline{d'_\uparrow} = \mathit{extend}(\overline{d_\downarrow}, \overline{d_\uparrow}) \qquad \overline{[s']} = \overline{[s][\text{self}./a][(y\ \text{as self})/\text{super}]} \end{array}}{\begin{array}{c} \sigma \mid \text{object}\,\{\,\text{inherit object}\,\{\,\overline{d}\;t\,\}\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,\} \longrightarrow \\ \sigma(y \mapsto \{\,\overline{d_\uparrow}\,\}) \mid \text{object}\,\{\,\overline{d'_\uparrow}\;\overline{[s']}\overline{d_\downarrow}\;\overline{[\text{self}./a]}t;\;\overline{[s']}t_\downarrow\,\} \end{array}}$$

Figure 9.3.3: Uniform identity reduction

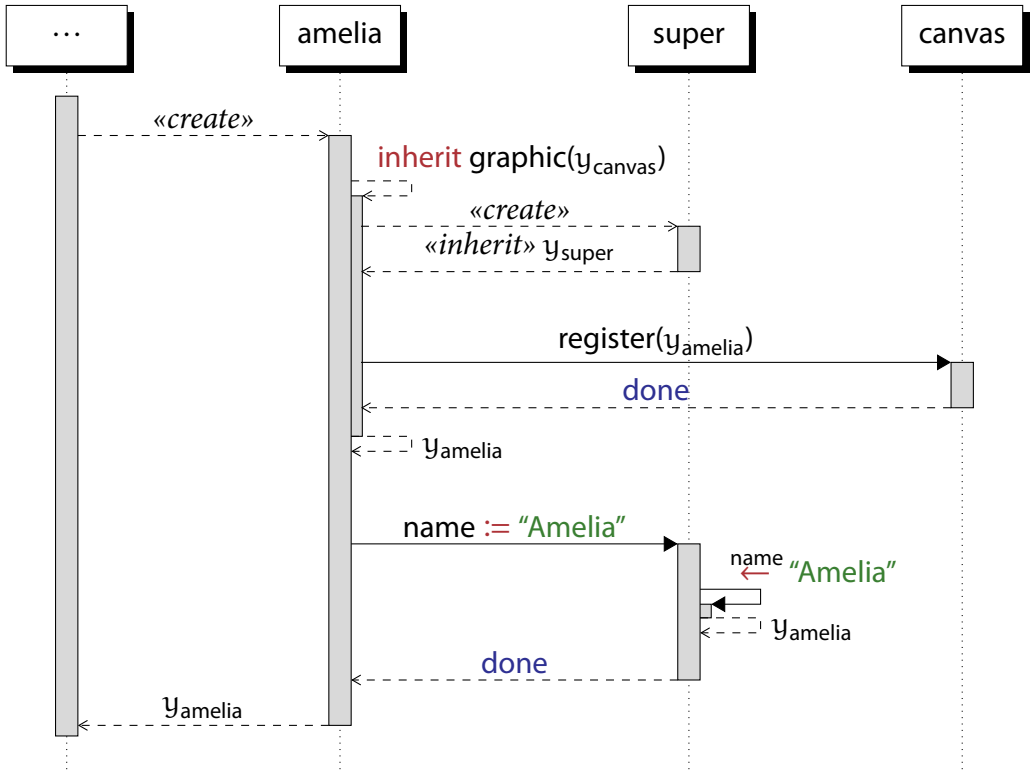

Figure 9.3.4: Sequence diagram of initialisation under uniform identity model

created). Super-references remain essentially invisible in these semantics, so the difference is not observable in the result of a computation.

As the semantics outside of initialisation is the same as merged identity, the resulting object structures are the same between the two formal semantics as visualised in Figure 9.2.5. The sequence of events that results in this structure is different, though. The relevant sequence diagram is presented in Figure 9.3.4. The creation of amelia immediately creates the super-object without running any initialisation code: the model does not actually create the identity of amelia y_{amelia} until after this point, at «*inherit*» y_{super}.

Where Rule E-Inh in Figure 9.2.4 was a modified application of the original Rules E-Inh and then E-Obj, the uniform identity modification applies this in reverse. As the body of the inherit clause is an object expression, we have to apply many of the same processes for reducing a regular object expression, but then move the sequence of expressions resulting from the body of the super-object down into

the body of the inheriting object. The evaluation *does* create a new object reference for the super-object, but only for up-calls to super, as was the case for merged identity.

In the new Rule E-Inh, the substitutions to local definitions $\overline{\alpha}$ need to be applied to the inherited methods (both those in the store, and the non-overridden methods in the inheriting object). The same substitution into the body of the inheriting object occurs, but now self is not bound in the body: self will be bound by the ultimate application of Rule E-Obj that constructs the final object. The substitutions applied simultaneously to $d_{\downarrow}$ and $t_{\downarrow}$ in the inheriting object by Rule E-Inh now have to be split between the definitions and the terms, as the super-body t appears in between, but it amounts to the same behaviour.

While the order of the object creation is different between the merged identity and uniform identity semantics, the observable behaviour we have investigated so far has been equivalent. We now consider the down-call to the draw method in the graphic method's object constructor, and can investigate how the uniform identity semantics differ from merged identity. The local request to draw in the initialisation of graphic now successfully down-calls into amelia's implementation, as the object *is* amelia during all initialisation in the hierarchy.

A sequence diagram of amelia's new initialisation path under uniform identity is provided in Figure 9.3.5. The call to draw down-calls to the definition of image in amelia's constructor, and successfully returns the object at images.amelia. Down-calls to field accessors during initialisation will always fail with ⇑ uninitialised, because every field value in inheriting definitions will not be initialised yet.

Uniform identity permits down-calls both during and after construction, as well as registration. The identity and structure of the object are both constant, guaranteeing stability. Uniform identity does not allow inheriting from preëxisting objects, instead requiring that parents be fresh. The exact definition of uniform identity presented here does not support multiple inheritance, but there is a logical extension to do so, discussed in Chapter 10.

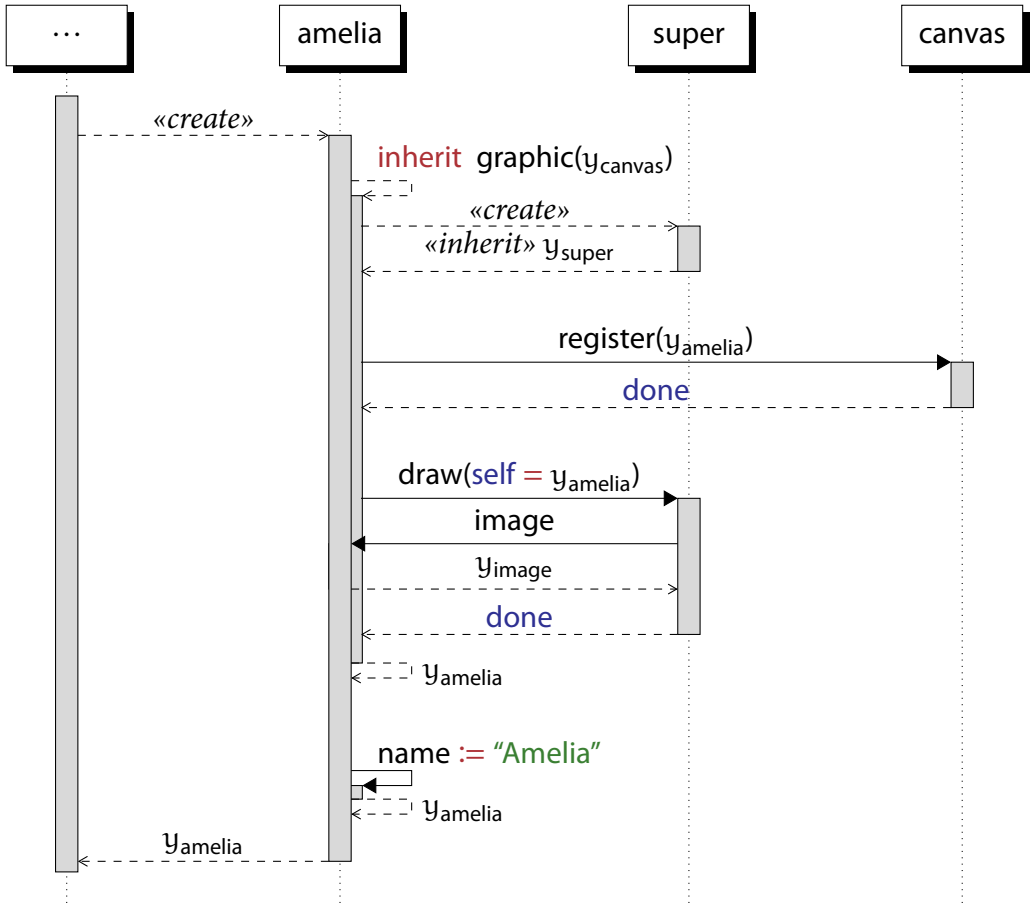

Figure 9.3.5: Sequence diagram of initialisation down-call under uniform identity

9.3.1 In Other Languages

Uniform identity is also supported in a number of objects-first languages. Constructors have a special role in JavaScript, and their behaviour can be manipulated to simulate classes by setting up their prototype property and then manually passing the value of this to any super-constructors. One of the main complications is that JavaScript constructors must have their prototype property set *before* they are used to construct a new object. As a result, inheriting from a preëxisting prototype object is simple (because it can just be directly assigned to prototype), but inheriting from another constructor, particularly one that requires arguments specific to a particular inheriting object, is more difficult, because no actual object exists to assign to prototype without calling the constructor. Fortunately, Object.create allows the creation of an object from a prototype without actually invoking the constructor:

```
function Child(arg) {
    // The arg value is only available when a Child is constructed
    Parent.call(this, arg)
}

Child.prototype = Object.create(Parent.prototype)
```

The ability to bind the value of this in a function call using the call method on a function allows JavaScript to implement uniform identity, ensuring that the initialisation code in a super-constructor can be executed in the context of the inheriting object instead of creating a fresh object and inheriting from that. JavaScript's class syntax, introduced by ECMAScript 2015 (ECMAScript Project 2016), is just sugar for this behaviour (with extra constraints such as sealing the structure of the objects involved).

The most natural forms of inheritance in JavaScript are the built-in single delegation and, by extension, single uniform inheritance, now codified directly in the language with the class syntax. Our primary concern is that without the particular dynamism of JavaScript, simulating class initialisation is impossible without reinterpreting factory methods as constructors with special semantics, at which point the language has arguably just implemented classes.

Similarly, the mechanism to permit down-calls during object initialisation in E resembles uniform identity (Miller 2006).

```
def graphic(self, canvas) {
    def parent { to draw() { canvas.render(self.image()) } }
    self.draw() # Results in an error
    return parent
}

def image := images.amelia
def amelia extends graphic(amelia, canvas) {
    to image() { return image }
}
```

Because the reference to the child object in the parent's initialisation is a promise, not a true reference, methods cannot be called on it directly. The call to self.draw() results in an error, because the self reference has not yet resolved to an object. Calling parent.draw() instead has the same problem, because parent's draw method attempts to call a method on self as well.

E allows interaction with a promise by calling a method *asynchronously*, to be called when the promise is resolved. This means we can schedule down-calls to happen during the initialisation of a super-constructor, to be called when has finished its entire initialisation process.

```
def graphic(self, canvas) {
    def parent { to draw() { canvas.render(self.image()) } }
    self ← draw() # Asynchronously calls draw()
    return parent
}

def image := images.amelia
def amelia extends graphic(amelia, canvas) {
    to image() { return image }
}

# Now amelia.draw() executes
```

The only difference between this behaviour and the uniform identity semantics is that down-calls in a super-constructor have to be executed at the end of the constructor. Interleaving these down-calls with other arbitrary code requires more complicated juggling of promises.

As in JavaScript, this more complicated behaviour is made possible only by the existence of other language features that combine to simulate classes. The E encoding of classes is clever, but the behaviour of promises in E (primarily that they actually transform into the object that satisfied the promise) are a domain-specific feature of the language (Miller 2006), and the uniform identity semantics generalises the behaviour to a language with a smaller number of core features.

10 Multiple Inheritance

Reusing behaviour from multiple parents is often desirable, but is less commonly supported in language designs in practice. In this section we show three logical extensions enabling multiple object inheritance. The first extends the uniform identity model with multiple inherit statements. The second is a separate model following the tradition of trait systems, where method names are unique in any object. The third provides the ability to include multiple inherit statements anywhere in an object's initialisation code, processed imperatively. All of these extensions process inherit clauses as under uniform identity, but they could all be simplified to perform inheritance from preëxisting objects as per forwarding, delegation, or concatenation. We have demonstrated this by constructing all possible combinations in our PLT Redex implementation.

All systems enable code reuse from arbitrarily many parents. All parents are treated symmetrically in the first two models, but further restrictions or privileges could be accorded to some parents without substantially affecting the models. In order to handle conflicts in symmetric inheritance, methods can be abstract in their body, which causes an error if that method is called. Alternatively, the implementations could ban the construction of an object with abstract methods, but this would only be valid for uniform identity, as the overriding of a concrete implementation would not take effect until the object was inherited under any of the other interpretations.

Extended Syntax

$\mathtt{i}$	::=	inherit $\mathtt{t}$ as z	*(Inherit clause)*
$\mathtt{i}^{\mathtt{o}}$	::=	inherit object $\{\cdots\}$ as z	*(Evaluated inherit clause)*
$\mathtt{t}$	::=	$\cdots \mid$ object $\{\, \overline{\mathtt{i}}\ \overline{\mathtt{s}}\ \overline{\mathtt{d}}\ \mathtt{t}\,\}$	*(Term)*
$\mathtt{r}$	::=	$\mathtt{t} \mid (y$ as $x)$	*(Receiver)*
$\mathtt{s}$	::=	$\cdots \mid (y$ as self$)/z$	*(Substitution)*

Figure 10.1.1: Multiple Parents grammar

10.1 Multiple Parents

The multiple parents model allows a sequence of inherit statements to appear at the start of an object body. Each statement includes an as clause, which binds a name in the body of the object as a super reference to the particular parent that is constructed by the expression in the inherit clause. The single-inheritance uniform identity model is immediately subsumed using inherit parent as super. When the same method name is inherited from multiple parents, none has priority and an abstract method of that name is inserted instead. The programmer must provide a local override calling the version from a particular named parent if desired. All methods are collected and installed before any initialisation code from the object bodies executes, so a consistent set of method implementations is seen throughout the initialisation.

The extended grammar of multiple uniform is presented in Figure 10.1.1. Multiple inherit clauses can now appear at the head of an object constructor. A method can be marked as unimplemented manually by using required as the method body, but unimplemented methods can also be generated by the inheritance mechanism. The super form is gone, as *which* super-object the reference refers to needs to be distinguished, so instead inherit clauses include a name binding with as x. Any abstract variable can be substituted as a reduction as a result (though the reduction rules will only ever apply a substitution for a super-variable).

Reduction rules for Multiple Parents are presented in Figure 10.1.2. The Rule E-

$$\text{(E-I/C)}\quad \frac{t \neq \text{object}\,\{\,\overline{d_\uparrow}\;t_\uparrow\,\} \qquad \sigma \mid t \longrightarrow \sigma' \mid t' \qquad t = y.m(\overline{v}) \implies t' = \overline{t}; \text{object}\,\{\cdots\}}{\sigma \mid \text{object}\,\{\,\overline{i^o}\;\text{inherit}\;t\;\text{as}\;z\;\overline{i}\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,\} \longrightarrow \sigma' \mid \text{object}\,\{\,\overline{i^o}\;\text{inherit}\;t'\;\text{as}\;z\;\overline{i}\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,\}}$$

$$\text{(E-Inh)}\quad \frac{\begin{array}{cc}\overline{y\ \text{fresh}} & \overline{\overline{a} = \mathit{identify}(d)} \qquad \overline{\overline{a}\ \text{unique}} \qquad \overline{\overline{d_\uparrow} = \overline{[\text{self}/a]}\overline{d}} \\ \overline{d'_\uparrow} = \mathit{join}(\mathit{extend}(\overline{d_\downarrow}, \overline{d_\uparrow})) & \overline{s'} = \overline{[s]}\overline{[\text{self}/a]}[(y\ \text{as self})/z]\end{array}}{\begin{array}{c}\sigma \mid \text{object}\,\{\,\overline{\text{inherit object}\,\{\,\overline{d}\;t\,\}\;\text{as}\;z\;\overline{s}\;\overline{d_\downarrow}\;t_\downarrow\,}\,\} \longrightarrow \\ \sigma(y \mapsto \{\,\overline{d_\uparrow}\,\}) \mid \text{object}\,\{\,\overline{d'_\uparrow}\;\overline{[s']d_\downarrow}\;\overline{[\text{self}/a]}t; \overline{[s']}t_\downarrow\,\}\end{array}}$$

$$\begin{array}{l}\mathit{join} : \textsc{Seq}(\textsc{Def}) \to \textsc{Seq}(\textsc{Def}) \\ \mathit{join}(\cdot) \quad = \cdot \\ \mathit{join}((d, \overline{d_i})) = \begin{cases} \mathit{identify}(d) \notin \overline{\mathit{identify}(d_i)} & d, \mathit{join}(\overline{d_i}) \\ \mathit{body}(d) = \text{required} & \mathit{join}(\overline{d_i}) \\ \mathit{abstract}(d) \in \overline{d_i} & \mathit{join}(\overline{d_i}, d) \\ \text{otherwise} & \mathit{join}(\mathit{extend}(\mathit{abstract}(d), \overline{d_i})) \end{cases}\end{array}$$

$$\begin{array}{l}\mathit{abstract} : \textsc{Def} \to \textsc{Def} \\ \mathit{abstract}(\text{method D \{ t \}}) = \text{method D \{ required \}}\end{array}$$

Figure 10.1.2: Multiple Parents reduction

I/C implements fresh inheritance in objects with potentially multiple inherit clauses, ensuring each clause is evaluated in order. Rule E-Inh is essentially the same as in the uniform identity implementation in Figure 9.3.3, but processing multiple inherit clauses at once (hence the extra multiplicities for many of the bindings). Once all of the super-methods are collected, conflicting methods are resolved with the *join* auxiliary function that, for each unique method name in the collection of methods, accepts exactly one concrete implementation of a method with that name and removes all of the abstract implementations, or provides a single abstract method with that name and removes all other implementations.

Multiple Parents supports registration and downcalls exactly as in uniform identity, multiple inheritance from freshly-created parents, and is stable because methods are collected first. These are all properties of uniform inheritance: applying this modification to the simpler object inheritance models each retains their

Extended Syntax

i	::=	inherit t alias $\overline{\mathrm{a}\ \text{as}\ \mathrm{m}}$ exclude $\overline{\mathrm{m}}$	*(Inherit clause)*
i^{o}	::=	inherit object $\{\cdots\}$ alias $\overline{\mathrm{a}\ \text{as}\ \mathrm{m}}$ exclude $\overline{\mathrm{m}}$	*(Evaluated inherit clause)*
r	::=	t	*(Receiver)*
s	::=	$\mathrm{v}/\mathrm{z} \mid \text{self}/\mathrm{a} \mid \mathrm{y}/\text{self}\leftarrow$	*(Substitution)*

Figure 10.2.1: Method Transformation grammar

own particular properties as well.

10.2 Method Transformations

Multiple inheritance under Method Transformations resembles trait-like composition of objects, representing object values as a single mapping of method names to methods, with no equivalent to super in the previous designs. Instead, an inherit statement can have any number of alias or exclude clauses associated, which respectively create an alternative name for an inherited method and exclude its implementation. The syntax for these clauses — and the removal of super-references and up-calls in receivers and substitutions — is defined in Figure 10.2.1.

If a method is overridden locally and still needs to be accessed, the inherited method can be aliased to a different name and accessed through that name within the object instead of referring to a super-object. An object contains at most one method with any given name, and there are no part-objects, but referring to super-methods can be more verbose because they need to be explicitly aliased individually.

Multiple inherit statements can appear in an object, treated symmetrically. If the same name is inherited from multiple parents, all but one must be excluded, or a local overriding method declared, if a concrete implementation of that method is to appear in the object. All inheritance expressions are evaluated and the final method set of the object assembled before any initialisation code executes. Initialisation occurs from top to bottom (depth-first search) within each branch of the

inheritance hierarchy. Methods are decoupled from their names because aliases may provide multiple equivalent names all reaching the method, while exclusion means that local definitions may not be implemented in the final object.

If amelia wished to simultaneously be a graphic and a gunslinger, then two inherit clauses can be included under the Transformation model. If the gunslinger class also contained a draw method for drawing her gun, amelia would be required to choose a single implementation by excluding one of the two. The excluded method can still be accessed if it is aliased:

```
def amelia = object {
  inherit gunslinger
  inherit graphic alias draw as render exclude draw
  def image = images.amelia
  var name := "Amelia"
}
```

Even if the gunslinger class also has a name method, amelia has successfully combined the two by overriding both. Note that method resolution is consistent throughout the entire object: any request to draw on amelia will unholster instead of rendering, including from within the lexical context of the graphic method.

The implementation of Method Transformation inheritance is given in Figure 10.2.2 as a modification of the previous Multiple Parents extension. Inherit clauses no longer have super-names, using the method aliasing and excluding syntax instead. Rule E-Inh now pre-processes the methods of each inherited object before passing them to join, with the aliases and excludes auxiliary functions. Both of these functions proceed as expected: fold the transformations over the methods, applying the alias or exclude rules in order. Because these rules are ordered, it is possible to create an alias of an existing alias that occurs earlier in the list, and it is possible to exclude aliases. Exclusion is always processed after aliases, so a directly excluded method cannot be aliased.

Method Transformation permits both down-calls and registration. It provides stability through time, but not through local analysis of visible declarations. It imposes the same freshness requirements as the other models, and supports multiple inheritance. Applying Method Transformation to the simpler object inheritance

(E-I/C)

$$\frac{t \neq \mathsf{object}\,\{\,\overline{d_\uparrow}\ t_\uparrow\,\} \qquad \sigma \mid t \longrightarrow \sigma' \mid t' \qquad t = y.m(\overline{v}) \implies t' = \overline{t}; \mathsf{object}\,\{\cdots\}}{\begin{array}{c}\sigma \mid \mathsf{object}\,\{\,\overline{i^o}\ \mathsf{inherit}\ t\ \mathsf{alias}\ \overline{a_i\ \mathsf{as}\ m_i}\ \mathsf{exclude}\ \overline{a_j}\ \overline{i}\ \overline{s}\ \overline{d_\downarrow}\ t_\downarrow\,\} \longrightarrow \\ \sigma' \mid \mathsf{object}\,\{\,\overline{i^o}\ \mathsf{inherit}\ t'\ \mathsf{alias}\ \overline{a_i\ \mathsf{as}\ m_i}\ \mathsf{exclude}\ \overline{a_j}\ \overline{i}\ \overline{s}\ \overline{d_\downarrow}\ t_\downarrow\,\}\end{array}}$$

(E-Inh)

$$\frac{\begin{array}{c}\overline{y\ \text{fresh}} \qquad \overline{\overline{a = \mathit{identify}(d)}} \\ \overline{d_\uparrow = \overline{[\mathsf{self}/a]}d} \qquad \overline{d_a = \mathit{aliases}(\overline{\langle a_i, m_i\rangle}, \overline{d_\uparrow})} \qquad \overline{d_e = \mathit{excludes}(\overline{a_j}, \overline{d_a})} \\ \overline{d_e}\ \text{unique} \qquad \overline{d'_\uparrow} = \mathit{join}(\mathit{extend}(\overline{d_\downarrow}, \overline{d_e})) \qquad \overline{s'} = \overline{[s]\overline{[\mathsf{self}/a]}[(y\ \mathsf{as}\ \mathsf{self})/\mathsf{super}]}\end{array}}{\begin{array}{c}\sigma \mid \mathsf{object}\,\{\,\mathsf{inherit}\ \mathsf{object}\,\{\,\overline{d}\ t\,\}\ \mathsf{alias}\ \overline{a_i\ \mathsf{as}\ m_i}\ \mathsf{exclude}\ \overline{a_j}\ \overline{s}\ \overline{d_\downarrow}\ t_\downarrow\,\} \longrightarrow \\ \sigma \mid \mathsf{object}\,\{\,\overline{d'_\uparrow}\ \overline{[s']d_\downarrow}\ \overline{\overline{[\mathsf{self}/a]}t; [s']t_\downarrow}\,\}\end{array}}$$

$$\begin{aligned}
&\mathit{aliases} : \textsc{Seq}(\textsc{Def}) \times \textsc{Seq}(\textsc{Ident} \times \textsc{Name}) \to \textsc{Seq}(\textsc{Def}) \\
&\mathit{aliases}(\overline{d}, \cdot) = \overline{d} \\
&\mathit{aliases}(\overline{d}, (\langle a, m\rangle, \overline{\langle a_i, m_i\rangle})) = \mathit{alias}(a, m, \overline{d}), \mathit{aliases}(\overline{d}, \overline{\langle a_i, m_i\rangle})
\end{aligned}$$

$$\begin{aligned}
&\mathit{alias} : \textsc{Ident} \times \textsc{Name} \times \textsc{Seq}(\textsc{Def}) \to \textsc{Seq}(\textsc{Def}) \\
&\mathit{alias}(a, m, \cdot) = \cdot \\
&\mathit{alias}(a, m, (d, \overline{d_i})) = \begin{cases} \mathit{identify}(d) = a & \mathit{rename}(d, m), \cdot \\ \text{otherwise} & \mathit{alias}(a, m, \overline{d_i}) \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathit{rename} : \textsc{Def} \times \textsc{Name} \to \textsc{Def} \\
&\mathit{rename}(\mathsf{method}\ m_1(\overline{x : T_i}) \to T\ \{\ t\ \}, m_2) = \mathsf{method}\ m_2(\overline{x : T_i}) \to T\ \{\ t\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathit{excludes} : \textsc{Seq}(\textsc{Def}) \times (\textsc{Name} \times \mathbb{N}) \to \textsc{Seq}(\textsc{Def}) \\
&\mathit{excludes}(\overline{d}, \cdot) = \overline{d} \\
&\mathit{excludes}(\overline{d}, (a, \overline{a_i})) = \mathit{exclude}(a, \overline{d}), \mathit{excludes}(\overline{d}, \overline{a_i}) \\
&\mathit{exclude} : (\textsc{Name} \times \mathbb{N}) \times \textsc{Seq}(\textsc{Def}) \to \textsc{Seq}(\textsc{Def}) \\
&\mathit{exclude}(a, \cdot) = \cdot \\
&\mathit{exclude}(a, (d, \overline{d_i})) = \begin{cases} \mathit{identify}(d) = a & \mathit{abstract}(d), \cdot \\ \text{otherwise} & \mathit{exclude}(a, \overline{d_i}) \end{cases}
\end{aligned}$$

Figure 10.2.2: Method Transformation reduction

models continues to maintain their own particular properties, as they were never stable to begin with.

10.3 Positional

The previous multiple inheritance models treat the inherit clauses as symmetric, such that no definition in any one is preferred in the case of conflicts, requiring resolution either by overriding or Method Transformations. Moreover, they do not permit any initialisation until all of the inheritance relations are established, which can be a problem if fields need to be initialised for down-calls in super-initialisation code. Consider if amelia's overriding definition of the image method was provided as an accessor to a field: the image field would not be initialised during the graphic method's evaluation, causing an uninitialised error when the initialisation attempts to down-call to the image accessor. Positional inheritance addresses these concerns, at the cost of visibly mutating the object during construction, similarly to what can be achieved with mutable object structure.

Under Positional inheritance, multiple inherit clauses can appear in an object, amongst the typical statements; this syntax is presented in Figure 10.3.1. These clauses are processed imperatively where they appear, in order, using the same semantics as the selected base model of inheritance. As under Multiple Parents, each inherit statement can have a name associated. Positional inheritance could also be used for single inheritance, allowing some initialisation before the superconstructor is called, but the visible mutation is still present, and the distinct ordering of the inherit clauses in an object body implies a natural method conflict resolution: replace the methods that are already defined in the object.

In the most general version, an inherit clause can appear anywhere in the object body, and have other code before, after, and in between, with the semantics of the inheritance taking effect at the point of appearance and later parents having precedence over earlier. The ordering of the inherit clauses under Positional inheritance allows some interesting programmer choices with some of the base models. Altering the order of parents affects which versions of same-named methods are accessed, and interleaving other code in between exposes both at different times. The availability and safety of up-calls and down-calls are affected by the placement

Extended Syntax

i	::=	super inherit t as z $\overline{s}$ | $\overline{c}$ inherit t as x $\overline{s}$	*(Inherit clause)*
c	::=	$\langle y, \overline{d}, \overline{s} \rangle$	*(Inherit context)*
t	::=	$\cdots$ | i	*(Term)*
r	::=	t | (y as y)	*(Receiver)*
s	::=	$\cdots$ | (y as y)/z | $\overline{c}$/super	*(Substitution)*

Figure 10.3.1: Positional inheritance grammar

of the inheritance, field initialisation, and other code. A more restrained approach could limit inheritance to appearing all at the top (or bottom) of the object body.

Positional delegation with named super-objects is essentially the behaviour of Self (Chambers et al. 1991), where multiple parent pointers may exist in a single object; Self does not allow initialisation code to execute in the context of the object under construction or have any priority between parents, but does allow parent pointers to be mutable. The nature of concatenation fundamentally supports Positional multiple inheritance, simply copying in the contents of the inherited object in place of the inherit statement, and the limitation in the single inheritance concatenation model that the inherit clause must appear at the top of the object constructor was purely syntactic. Named super-objects (or a 'next-method' functionality) are also necessary to access overridden methods.

Positional inheritance with forwarding is quite straightforward, and strictly named super-objects are not required: because forwarding only accesses the public interface, an ordinary reference to each parent suffices. One of the primary use-cases of Positional inheritance — initialising fields before invoking a super-constructor — is irrelevant without uniform identity, as the super-constructor cannot make a down-call into the inheriting object anyway.

Merged identity does not lend itself to the Positional extension because it relies on taking over the identity of the parent object, and the identity of the child object is established before any inheritance occurs. We have already established that merged identity is not conducive to multiple inheritance, and multiple parents in

the Positional model would result in repeated identity changes, some of which may even lose methods. A different extension could merge multiple identities together, or resolve the resulting issues in some other way, but we do not address this combination here and simply exclude it from consideration as confusing at best.

Positional inheritance is the only one of our multiple-inheritance models that permits inheriting from something obtained from a parent. A parent could define a number of specialised 'inner classes', with the intention that its child would in turn inherit from one of those specialisations as well. It is not obvious to us that such an ability is useful, but nor is it obvious that it is not. We note this unique ability, but do not pursue it further.

Positional inheritance reintroduces mutation during construction to uniform identity, because each inherit adds new methods to the object. When multiple methods are inherited by the same name, the last-inherited method wins out. An unusual aspect is that while down-calls are always available, during construction 'side-calls' to co-parents of a common child can be made only to parents whose inherit preceded this one. An object can even define a fresh constructor directly inside of itself, and then inherit from it:

```
object {
    method parent { object { ··· } }
    inherit parent
}
```

Each line of initialisation occurs after preceding inheritance statements and before subsequent inheritance statements. If inherit precedes a field initialisation or other expression, upcalls to that parent are available from that expression; if inherit follows a field initialisation, down-calls from that parent accessing that field are safe. This means that amelia can implement the image method as a field accessor, and safely initialise it before inheriting from a request to the graphic method:

```
def amelia = object {
    def image = images.amelia
    inherit graphic(canvas)
    name := "Amelia"
}
```

Note that amelia must wait until after the inherit clause has completed before setting the name field, as it is not defined in amelia until after it is defined in the graphic method.

The implementation of Positional inheritance is provided in Figure 10.3.2. The primary difficulty with implementing Positional inheritance is that the binding of local definitions *can change imperatively*: a local request might refer to some definition in the surrounding scope, but after processing an inherit clause during the initialisation phase that request might now refer to an inherited definition instead. This presents a particular difficulty for substitution, which irreversibly binds an unqualified name to a particular definition. Substitutions are now delayed by all object expressions, as the inherit clauses are now nested in the object statements. Rule E-Obj modifies the Rule E-Obj originally defined in Figure 4.1.1, handling the new statement form with the updated body translation and applying the delayed substitution.

In order to implement the necessary dynamic scoping, we introduce an *inherit context* c, which records the reference y of the object that an inherit clause appeared inside, the source of the methods $\overline{\mathrm{d}}$ that appeared directly in that object, and the delayed substitutions $\overline{\mathrm{s}}$ that were on that object. Any processed inherit clause has an ordered stack of these contexts on it, from the actual object the inherit clause appeared in, down to the bottom-most inheriting object. The super prefix is used as a placeholder on newly created inherit expressions, so that Rule E-Obj can substitute it out for the initial inherit context.

By retaining the source of the methods and the substitution scope they appeared in, the methods can be re-substituted in any new scope that appears. The update auxiliary function applies this for any newly inherited methods $\overline{\mathrm{d}_{\uparrow}}$ to every object in the current stack of contexts. Rule E-Inh handles any inherit expression, constructing a new part-object y, and using update to include the new methods in the original object and every intervening part-object, after applying overrides from the existing methods. The value of self is bound in the inherited object body by the last object reference in the inherit context, as that is the location of the original object. The inherited body is processed in the same way as in Rule E-Obj. One of the key distinctions here is that the value of self to substitute in the inherited body already exists, whereas under single uniform identity the body is concatenated into

(E-I/C)

$$\frac{t \neq \mathsf{object}\,\{\,\overline{s_\uparrow}\ \overline{d_\uparrow}\ t_\uparrow\,\} \qquad \sigma \mid t \longrightarrow \sigma' \mid t' \qquad t = y.m(\overline{v}) \implies t' = \overline{t}; \mathsf{object}\,\{\cdots\}}{\sigma \mid \overline{c}\ \mathsf{inherit}\ t\ \mathsf{as}\ z\ \overline{s} \longrightarrow \sigma' \mid \overline{c}\ \mathsf{inherit}\ t'\ \mathsf{as}\ z\ \overline{s}}$$

(E-Obj)

$$\frac{y \text{ fresh} \qquad \overline{a = \mathit{identify}(d)} \qquad \overline{a} \text{ unique}}{\sigma \mid \mathsf{object}\,\{\,\overline{s}\ \overline{d}\ t\,\} \longrightarrow \sigma(y \mapsto \{\,\overline{[s][\mathsf{self}/a]d}\,\}) \mid \overline{[s]}[\langle y, \overline{d}, \overline{s}\rangle/\mathsf{super}][y/\mathsf{self}]\overline{[\mathsf{self}/a]}t; y}$$

(E-Inh)

$$\frac{\begin{array}{rr} y_\uparrow \text{ fresh} & \overline{a = \mathit{identify}(d_\uparrow)} \\ \overline{a} \text{ unique} \quad \overline{d'_\uparrow} = \overline{[s_\uparrow]}\overline{[\mathsf{self}/a]}\overline{d} & \overline{c'} = \langle y, \overline{d}, (\overline{s}, (y_\uparrow\ \mathsf{as}\ y_\downarrow)/z)\rangle, \overline{c}, \langle y_\downarrow, \overline{d_\downarrow}, \overline{s_\downarrow}\rangle \end{array}}{\begin{array}{c} \sigma \mid (\langle y, \overline{d}, \overline{s}\rangle, \overline{c}, \langle y_\downarrow, \overline{d_\downarrow}, \overline{s_\downarrow}\rangle)\ \mathsf{inherit\ object}\,\{\,\overline{s_\uparrow}\ \overline{d_\uparrow}\ t_\uparrow\,\}\ \mathsf{as}\ z\ \overline{s}; t \longrightarrow \\ \mathit{update}(\sigma(y_\uparrow \mapsto \{\,\overline{d'_\uparrow}\,\}), \overline{d'_\uparrow}, \overline{c'}) \mid \\ \overline{[\mathsf{self}/a]}((\overline{[s_\uparrow]}[(\langle y, \overline{d_\uparrow}, \overline{s_\uparrow}\rangle, \overline{c'})/\mathsf{super}][y_\downarrow/\mathsf{self}]t_\uparrow); \overline{[s]}[c'/\mathsf{super}][(y\ \mathsf{as}\ y_\downarrow)/x]t) \end{array}}$$

$$\begin{array}{l} \mathit{update} : (\textsc{Var} \rightharpoonup \textsc{Seq}(\textsc{Def})) \times \textsc{Seq}(\textsc{Def}) \times \textsc{Seq}(\textsc{Ctxt}) \to \textsc{Var} \rightharpoonup \textsc{Seq}(\textsc{Def}) \\ \mathit{update}(\sigma, \overline{d_\uparrow}, \cdot) = \sigma \\ \mathit{update}(\sigma, \overline{d_\uparrow}, (\langle y, \overline{d}, \overline{s}\rangle, \overline{c})) = \mathit{update}(\sigma(y \mapsto \{\,\overline{d'_\uparrow}, \overline{d'}, \overline{d'_\downarrow}\,\}), (\overline{d'_\uparrow}, \overline{d'}), \overline{c}) \\ \qquad \textbf{where}\ \overline{a_\uparrow} = \overline{\mathit{identify}(d_\uparrow)} \quad \textbf{and}\ \overline{a_\downarrow} = \overline{\mathit{identify}(\overline{d_\downarrow})} \\ \qquad\qquad \overline{d'_\uparrow} = \mathit{extend}(\overline{d}, \overline{d_\uparrow}) \qquad \overline{d'} = \overline{[s][\mathsf{self}/a_\uparrow][\mathsf{self}/a_\downarrow]}\overline{d} \\ \qquad\qquad \{\,\overline{d_\downarrow}\,\} = \sigma(y) \qquad \overline{d'_\downarrow} = \mathit{extend}(\overline{d_\downarrow}, (\overline{d}, \overline{d_\uparrow})) \end{array}$$

Figure 10.3.2: Positional reduction

an object expression and self is substituted once that is evaluated.

Note that inherit expressions still delay substitutions, preventing them from applying to expressions later in any sequence. After each inherit expression is evaluated, the substitutions are then applied to the following expressions, after being shadowed by inherited definitions and the super-name defined by the as clause. Positional inheritance preserves the other traits of uniform identity from §9.3, with the exception of stability: during construction, an object's apparent structure and behaviour can change. Applied to the simpler object inheritance models, it preserves each of their unique properties.

11 Classless Inheritance

Table 11.1 compares the models according to the criteria established in Section 7.1. Each model provides a different mix of the criteria, which may be appropriate for different circumstances or languages. The uniform identity design provides the closest match to Java semantics (given at the bottom of the table). No model provides every property; indeed, stability, down-calls, and inheriting from existing objects are fundamentally in conflict, particularly during initialisation. The complexity of each design and its implementation roughly increases down the table, which is a further trade-off for language designers to consider.

The key insight of this investigation is that there are gaps in the design space presented by Table 11.1: no model permits inheriting from preëxisting objects while also permitting down-calls during initialisation, for instance. Combinations not found in the table *do* exist in other languages, but they tend to require a significant amount of dynamism to achieve, such as JavaScript's dynamic binding of the this parameter and the special role of the prototype field on constructors.

While delegation, forwarding, and concatenation can fundamentally support inheriting from arbitrary objects, the other models lean towards supporting planned reuse rather than ad-hoc reuse — that is, inheriting from objects that have been designed to be inherited from, rather than from any arbitrary object. Both planned and unplanned reuse have solid software-engineering motivations; indeed, language features exist both specifically to prevent inheritance (final or sealed classes) and to enable ad-hoc reuse (structural types).

We do not wish to present one or another choice as better, but to draw attention to a potentially-unintended side effect of various points in the solution space. Nonetheless, it is possible for any of the fresh-object-based systems to support delegation or forwarding semantics simply by exposing a method, accepting any object

	Reg.	Down.	Dist.	Stable	Exist.	Mult.	Overl.	Par.
Forwarding	no	no	yes	yes	yes	no	yes	no
Delegation	no	no*	yes	no	yes	no	yes	no
Concatenation	no	no*	no	no	yes	no	yes	no
Merged	yes	no*	no	no*	fresh	no	yes	no
Uniform	yes	yes	no	yes	fresh	no	yes	no
Mult. Uniform	yes	yes	no	yes	fresh	yes	yes	no
Transform U.	yes	yes	no	no	fresh	yes	no	no
Positional U.	yes	yes	no	no	fresh	yes	yes	yes
Java	yes	yes	no	yes	class	no	yes	no

Table 11.1: Comparison of models of object-first inheritance. A * indicates answer holds during construction, but is reversed after. The Overl. column indicates multiple definitions of the same method name in an object, accessible through super-references. The Par. column indicates ability to inherit from something obtained through another parent. All other columns relate to criteria from Section 7.1.

as an argument, that returns a fresh object whose methods provide the behaviour in question. Concatenation semantics can similarly be supported by inheriting from a standard clone.

Diamond inheritance (repeatedly inheriting from the same class or trait two or more times) has long been recognised as a problem in object-oriented language design. Eiffel and C++ both offer the same essential solution to the problem: arranging that some classes can be replicated each time they are inherited, while other classes will be inherited only once. Malayeri and Aldrich present a good discussion of the problems diamonds cause for inheritance, and then argue that diamond inheritance can be prevented in languages, partly by supporting a *requires* clause inspired by Scala which indicates that a trait depends upon the eventual final self object providing a set of methods, but not actually implementing those methods itself (Malayeri and Aldrich 2009).

The two multiple-inheritance systems we describe are open to the sort of collaboration used to solve issues with diamond inheritance in other languages, but do not require it. Because they are object-based, rather than class-based, some issues of diamond inheritance do not arise, as each instance of a parent is (unavoidably) separately obtained and constructed: conceptual problems such as whether fields

should be duplicated if a class is inherited from multiple times are no longer relevant, since the objects are logically separated and fields are naturally duplicated. In an object-based system, coalescing similar ancestors is a dubious activity, as side effects may occur on the path to construction and be semantically meaningful, which cannot happen in a static, declarative class system.

11.1 Typing

In our formal model we have made a conscious effort to handle as many errors as possible in the operational semantics (i.e. at run-time) rather than by defining erroneous programs as ill-formed (i.e. at compile-time). There are several reasons for this, but most important is that we see further layers on top — such as type systems or checks for diamond inheritance — as an important, but separable part of the design process. Omitting such definitions highlights the inheritance designs, and enables the core language of the model to be smaller and more general.

Extending the static semantics of Graceless to type the inheritance systems that we have defined produces some interesting challenges. The primary difficulty is that we must know the *exact* type of an object in order to safely inherit from it, both in terms of width and depth: to add new methods, the type system must know that there is not a corresponding method in the super-object that is being overridden; to override existing methods, the type system must know exactly the type annotations on the parameters and return of the overridden method to ensure that the override does not generalise the signature type. Under the structural subtyping of Graceless, types need not describe the exact interface of an object, and there is no mechanism to express an exact type.

Annotating a method with the exact type of the object it returns instead of its public interface is potentially burdensome and may expose more information about its implementation than is necessary when the object is not being inherited from. In order to type a program without explicit and exact type annotations, exact return types need to be inferred for inheritable methods. Because method declarations in an object can mutually refer to one another, inference requires an unfortunate dependency between typing a definition and determining the exact type of a term.

Consider the following program:

```
method first {
    object {
        inherit second
        method inner { ··· }
    }
}

method second {
    object {
        inherit first.inner
    }
}
```

The first method constructs an object that inherits from the second method, but second constructs an object that inherits from a definition in the result of requesting first. An exact type for first is required to type second, but the reverse is also true. Typing a program such as the one above requires more advanced constraint solving.

Inherit clauses also introduce bindings into scope, so a further challenge is to integrate this feature into the type environment Γ. Exact typing is also required for this, since otherwise there may be definitions in the inherited object but not specified in the type that shadow definitions surrounding the object, causing unqualified references to be typed incorrectly otherwise. All of the methods defined in the super-object must be included in the environment Γ when typing the body of any inheriting object.

The different dynamic semantics of our inheritance models also require different modifications to the type system. Typically an object-oriented type system would aim to prevent the construction of objects with unimplemented methods, just as an abstract class cannot be instantiated in Java. In the object inheritance models it *must* be possible to create objects with required methods, because the inherited objects have a distinct identity and are created individually before being inherited from.

In a type system for the object inheritance models, direct requests to methods

that create unimplemented objects should not be possible. Unimplemented objects still need to exist for the purposes of inheritance, so we must build a parallel type system for terms that are safe, yet remain 'abstract'. Terms typed by this parallel system are not safe to interact with directly, since any of there methods may raise required, but are safe when inherited by an object that completes their implementation.

11.2 Conclusion

Object-based inheritance is unexpectedly complicated, especially when commonplace desires for functionality available in classical models are involved, and programmers have resorted to increasingly complex workarounds in existing object-based languages. We have demonstrated that object inheritance without classes is both viable and desirable, avoiding the conceptual complexity of an additional conceptual entity (the class) in an object-oriented language without losing functionality through careful feature selection, and set out a range of options with their various trade-offs made explicit.

We have presented several models of object inheritance, including the well-known approaches of delegation, forwarding, and concatenation. We have presented a novel extended operational semantics for a base language incorporating advanced but standard features affected by inheritance, and formalised the models as extensions to that single base language, formally demonstrating the subtle behavioural differences of each model. In particular, we have addressed the complex questions of down-calls, object registration, stability, inheriting from preëxisting objects, action at a distance, and multiple inheritance, as well as their interactions.

We have illustrated that object-based inheritance has the full range of possibilities of classical inheritance, and showed that many of these models can be used as effectively as purely declarative classes, but particular combinations — especially class initialisation semantics combined with inheritance from preëxisting objects — require a specific set of features usually reserved for very dynamic and reflective languages. Crucially, some of the concerns we have considered are directly at odds with each other, and cannot be implemented at the same time without the intervention of other, invariably more complicated language features.

Part IV

Conclusions

12 Classless Object Semantics

This dissertation has investigated the role of classes as a fundamental component of the object-oriented paradigm. The overall conclusion of our investigation is that classes need not be a foundational component of object-orientation, and that objects alone can express much of the necessary functionality made available by classes. This conclusion is couched in the premise that a classless language is inherently different to one that is class-based, and programs built in such a language must take this into account. We have validated this conclusion by analysing the design and formal semantics of classless languages, comparing their properties and considering case studies.

12.1 Graceless

We have developed Graceless, a classless object-oriented language that has served as a useful platform for exploring the implementation of class features in terms of objects. The Graceless language encodes much of the Grace programming language, and with it much of the functionality of an object-oriented languages without the need for classes as a fundamental construct. We have designed Graceless to focus on the practical aspects of classless languages, particularly object initialisation and self references as a point of differentiation from existing object theory.

We have demonstrated how Graceless is capable of checking the shallow type of an object with coercions, and extended the language with casts to allow the expression of assumptions about the type of an object. We have considered these casts in the context of gradual typing, demonstrating how Graceless casts are more appropriate than traditional casts when typing with subsumption.

12.2 Brand Typing

Brand objects implement the dynamic semantics of nominal types within the existing semantics of Grace, and are interpreted as nominal types in Grace's pluggable type checker. We have demonstrated how brand objects can be used to describe the nominal type of a class, hiding the permission component of the brand within the scope surrounding a constructor so that only the objects created by the constructor can inhabit the associated type.

We have extended Graceless to encode much of this design in the formal model of Branded Graceless, and proven that the resulting type system soundly describes the safe use of brand guards as nominal types. We have also considered a number of recent developments to place our work in context, considering the role of type members to pair a class with its type and express nominal subtyping relationships outside of the scope of the brand, as well as other attempts to encode nominal types on top of existing structurally-typed languages.

12.3 Object Inheritance

We have extended Graceless to model a number of different semantic models of object inheritance, in order to better understand the differences between the models. A driving part of this investigation is the expectations of 'class-like' behaviour, and we chose each semantics in order to model the behaviour of different programming languages, including the different phases of design that inheritance underwent in Grace itself.

Our comparison of these models illustrate our conclusion that the interaction of inheritance with object initialisation and self references means that standard object inheritance mechanisms such as delegation are not sufficient to simulate the behaviour of classes from popular languages such as Java. Several classless object-oriented languages — most prominently JavaScript — only manage to emulate classes through the confluence of other, unrelated language features. We have augmented our investigation by considering a number of different designs for supporting multiple inheritance as extensions to all of the models that are capable of it.

12.4 Implementation

Implementation has been an important part of these contributions, both for the formal models with PLT Redex, and in the Grace language proper with our custom Hopper interpreter and contributions to other Grace implementations. Using Grace's dialect system, we have implemented both structural and nominal type systems in a platform independent manner, and have taken care in Hopper to ensure that modules written under any type checker, including none at all, can safely interact with one another.

The implementation of the inheritance semantics in Redex has allowed us to run automatic tests of particular properties and immediately visualise the reduction of any program, as well as demonstrate the many combinations of our semantics with multiple inheritance without having to put every different combination to paper. All of the implementation work that we have developed in support of these contributions is publicly available as free software.

13 Future Work

Each of our contributions has the potential for further research; we have collected and summarised our earlier discussions here. Where relevant, we refer back to the section that contained a more in-depth discussion of the possible future work.

13.1 Graceless

Extending the type system of Graceless to embrace the discipline of gradual typing is an obvious first step to more fully encoding the realities of the full Grace language. Casts and coercions allow the expression of interoperation between typed and untyped languages, but they differ significantly from the standard gradual typing literature by only describing down-casts and not including the unknown type as part of the cast calculus. A cast insertion judgement that erases the unknown component of every type as part of its translation from a gradually-typed program to one in the cast calculus may be sufficient to encode the expected behaviour of gradually-typed languages, including Grace, but that remains to be seen.

Graceless fails to uphold the gradual guarantee, as we have discussed in §5.5.3, despite the fact that the language's match construct cannot inspect the type annotations on an object's methods: the original impetus for the gradual guarantee was earlier work of ours that included matching on type annotations (Jones and Noble 2014; Boyland 2014). When an assumption in a Graceless cast is invalidated, the result is a raise of the underlying object.

Since a raise can be rescued, Graceless cannot meet the refined criteria for a gradually-typed language (Siek, Vitousek, Cimini, et al. 2015): two programs that are equivalent save for the precision of their type annotations can have different

behaviour. Since it does not seem reasonable to require that a run-time type error *must* fatally crash a program, formal languages that allow rescuing type errors will require a more nuanced expression of the gradual guarantee.

13.2 Brand Typing

A proper theory of bounded type members are the missing ingredient for fully supporting the conceptual design of brand objects as nominal types, discussed in §6.4.1. Branded Graceless cannot reason about subtyping between nominal types without the brand permission object — which should be private to the class — exposed to the client code. While there are other potential solutions to this problem, the recent advent of the Dependent Object Types calculus as a well developed and sound theory of type members for small-step semantics is the logical solution, since this feature would also be useful for the goal of modelling the semantics of Grace programs with public type exports in their objects (Rompf and Amin 2016).

Type members also allow the easy pairing of the nominal type and the object constructor of a class. Branded Graceless must pass these two components as separate arguments to a client instead of grouping them together in a single object. The Tagged Objects language uses dependent sums in order to permit the return type of the constructor to depend on the value of the accompanying tag (Lee et al. 2015), but such sums are subsumed by type members.

The outstanding question for a language like Branded Graceless extended with type members is how this affects the run-time match construct — which cannot match directly on a type, only perform a shallow structural test — so it is likely the case that a class would need to include a third component to act as a pattern for the purposes of determining if an object was constructed by a particular class at run-time. The type of this pattern would also need to indicate that a matching object satisfied the associated type of the class. This is not a concern for the practical implementation, which can match on types directly.

13.3 Object Inheritance

As discussed in §11.1, we have not presented accompanying type systems for any of the object inheritance systems that we modelled. The primary challenge here is collecting *exact* information about the type of the inherited object, since both width and depth subtyping of the structural types of Graceless are at odds with the needs of typing inheritance. The exact information is needed to ensure both that method overrides respect the type annotations on the overridden methods, and that new methods are not accidentally overriding methods that are in the object but not included in the type.

There has been some work in typing first-class classes with row typing and the ability to express in a type what methods do *not* appear in any inhabiting object (Takikawa et al. 2012), but Grace opts for the simpler mechanism that the relevant object constructor must be statically resolved. Even collecting this information can be difficult, since each constructor may also inherit from another definition, and there is no inherent ordering in the class declarations. Unlike the declarative classes of languages like Java, the relevant constructors can be buried arbitrarily deep inside of a term.

While we have explored a variety of semantic models, there is the potential for both a wider and deeper analysis of object inheritance. Many classless object-oriented programming languages feature mutable object structure (as objects members introduced imperatively): as discussed in §8.2.1, modelling this feature makes a language significantly more difficult to soundly type-check, but there are more opportunities to find a semantics that can more accurately simulate the behaviour of classes. Other features — such as the promises of E that resolve in-place (Miller 2006) — could also be worth modelling, though they are also likely to present a significant challenge for developing a coherent and sound type system.

Bibliography

Abadi, M. "Baby Modula-3 and a Theory of Objects". In: *Journal of Functional Programming* 4.2 (1994), pp. 249–283.

Abadi, M. and Cardelli, L. *A Theory of Objects*. New York: Springer-Verlag, 1996.

Abadi, M., Cardelli, L., Pierce, B. C., and Plotkin, G. D. "Dynamic Typing in a Statically Typed Language". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 13.2 (1991), pp. 237–268. DOI: 10.1145/103135.103138.

Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. "Typestate-oriented programming". In: *Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2009. Orlando, FL, USA, Oct. 2009, pp. 1015–1022. DOI: 10.1145/1639950.1640073.

Amin, N. "Dependent Object Types". PhD thesis. Lausanne, Switzerland: École polytechnique fédérale de Lausanne, Aug. 2016.

Amin, N. and Tate, R. "Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers". In: *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'16. Amsterdam, Netherlands, 2016, pp. 838–848. DOI: 10.1145/2983990.2984004.

Andreae, C., Noble, J., Markstrum, S., and Millstein, T. D. "A framework for implementing pluggable type systems". In: *Proceedings of the 21st International Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA'06. Portland, OR, USA, Oct. 2006, pp. 57–74. DOI: 10.1145/1167473.1167479.

Apel, S., Kästner, C., and Lengauer, C. "Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement". In: *Proceedings of the 7th International Conference on Generative Programming and Component Engi-*

neering. GPCE'08. Nashville, TN, USA, Oct. 2008, pp. 101–112. DOI: 10.1145/1449913.1449931.

Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000. ISBN: 0-201-70433-1.

Baars, A. I. and Swierstra, S. D. "Typing dynamic typing". In: *Proceedings of the 7th International Conference on Functional Programming*. ICFP'02. Pittsburgh, PA, USA, Oct. 2002, pp. 157–166. DOI: 10.1145/581478.581494.

Barendregt, H. P. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1981. ISBN: 978-0444875082.

Baumgartner, G. and Russo, V. F. "Implementing Signatures for C++". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 19.1 (1997), pp. 153–187. DOI: 10.1145/239912.239922.

Bettini, L., Capecchi, S., and Venneri, B. "Featherweight Java with multi-methods". In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. PPPJ'07. Lisbon, Portugal, Sept. 2007, pp. 83–92. DOI: 10.1145/1294325.1294337.

Bierman, G. M., Abadi, M., and Torgersen, M. "Understanding TypeScript". In: *Proceedings of the 28th European Conference on Object-Oriented Programming*. ECOOP'14. July 2014, pp. 257–281. DOI: 10.1007/978-3-662-44202-9_11.

Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *Simula Begin*. Chartwell-Bratt Ltd., 1979. ISBN: 978-0862380090.

Black, A. P., Bruce, K. B., Homer, M., and Noble, J. "Grace: the absence of (inessential) difficulty". In: *Proceedings of the 11th Symposium on New Ideas in Programming and Reflections on Software*. Onward!'12. Tucson, AZ, USA: ACM, Oct. 2012, pp. 85–98. DOI: 10.1145/2384592.2384601.

Black, A. P., Bruce, K. B., Homer, M., Noble, J., Ruskin, A., and Yannow, R. "Seeking Grace: a new object-oriented language for novices". In: *Proceedings of the 44th Technical Symposium on Computer Science Education*. SIGCSE'13. Mar. 2013, pp. 129–134. DOI: 10.1145/2445196.2445240.

Black, A. P., Bruce, K. B., and Noble, J. *The Grace Programming LanguageDraft Specification Version 0.7.0*. 2016. URL: http://gracelang.org/documents/grace-spec-0.7.0.pdf.

Black, A. P., Hutchinson, N. C., Jul, E., and Levy, H. M. "The Development of the Emerald Programming Language". In: *Proceedings of the 3rd Conference on History of Programming Languages*. HOPL-III. San Diego, California, 2007, pp. 11-1–11-51. DOI: 10.1145/1238844.1238855.

Black, A. P. and Palsberg, J. "Foundations of Object-Oriented Languages — Workshop Report". In: *SIGPLAN Notices* 29.3 (1994), pp. 3–11.

Borning, A. H. "Classes Versus Prototypes in Object-Oriented Languages". In: *Proceedings of 1986 ACM Fall Joint Computer Conference*. ACM'86. Dallas, Texas, USA: IEEE Computer Society Press, 1986, pp. 36–40. ISBN: 0-8186-4743-4.

Boyland, J. T. "The problem of structural type tests in a gradual-typed language". In: *Proceedings of the 21st International Workshop on Foundations of Object-Oriented Languages*. FOOL'14. 2014.

Bracha, G. *Newspeak Programming Language Draft Specification Version 0.096*. Tech. rep. Apr. 2016.

Bracha, G. "Pluggable Type Systems". In: *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages*. Oct. 2004.

Bracha, G. "The Strongtalk type system for Smalltalk". In: *Proceedings of the OOPSLA Workshop on Extending the Smalltalk Language*. 1996.

Bracha, G. and Griswold, D. "Strongtalk: Typechecking Smalltalk in a Production Environment". In: *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA'93. Washington, D.C., USA: ACM, 1993, pp. 215–230. DOI: 10.1145/165854.165893.

Bruce, K. B. "A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics". In: *Journal of Functional Programming* 4.2 (1994), pp. 127–206. DOI: 10.1017/S0956796800001039.

Bruce, K. B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

Büchi, M. and Weck, W. "Compound Types for Java". In: *Proceedings of the 13th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'98. Vancouver, BC, Canada, Oct. 1998, pp. 362–373. DOI: 10.1145/286936.286975.

C# Project. *C# Reference*. July 2015. URL: https://msdn.microsoft.com/en-us/library/618ayhy6.aspx.

Cameron, N., Drossopoulou, S., and Ernst, E. "A Model for Java with Wildcards". In: *Proceedings of the 22nd European Conference on Object-Oriented Programming*. ECOOP'08. Paphos, Cyprus, July 2008, pp. 2–26. DOI: 10.1007/978-3-540-70592-5_2.

Cardelli, L. "Structural Subtyping and the Notion of Power Type". In: *Proceedings of the 15th Symposium on Principles of Programming Languages*. POPL'88. San Diego, CA, USA, Jan. 1988, pp. 70–79. DOI: 10.1145/73560.73566.

Cardelli, L., Donahue, J. E., Jordan, M. J., Kalsow, B., and Nelson, G. "The Modula-3 Type System". In: *Proceedings of the 16th Symposium on Principles of Programming Languages*. POPL'89. Austin, TX, USA, Jan. 1989, pp. 202–212. DOI: 10.1145/75277.75295.

Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. "An Extension of System F with Subtyping". In: *Information and Computation* 109.1 (1994), pp. 4–56. DOI: 10.1006/inco.1994.1013.

Cardelli, L. and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism". In: *Computing Surveys* 17.4 (1985), pp. 471–522. DOI: 10.1145/6041.6042.

Castanos, J. G., Edelsohn, D., Ishizaki, K., Nagpurkar, P., Nakatani, T., Ogasawara, T., and Wu, P. "On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages". In: *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'12. Tucson, AZ, USA, Oct. 2012, pp. 195–212. DOI: 10.1145/2384616.2384631.

Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. FMCO'05. Amsterdam, The Netherlands, Nov. 2005, pp. 342–363. DOI: 10.1007/11804192_16.

Chambers, C., Ungar, D., Chang, B., and Hölzle, U. "Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF". In: *Lisp and Symbolic Computation* 4.3 (1991), pp. 207–222.

Cimini, M. and Siek, J. G. "Automatically Generating the Dynamic Semantics of Gradually Typed Languages". In: *Proceedings of the 44th Symposium on Princi-*

ples of Programming Languages. POPL'17. Paris, France, 2017, pp. 789–803. DOI: 10.1145/3009837.3009863.

Cimini, M. and Siek, J. G. "The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems". In: *Proceedings of the 43rd Symposium on Principles of Programming Languages*. POPL'16. St. Petersburg, FL, USA, 2016, pp. 443–455. DOI: 10.1145/2837614.2837632.

Clarke, D. G., Potter, J., and Noble, J. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 13th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'98. Vancouver, BC, Canada, Oct. 1998, pp. 48–64. DOI: 10.1145/286936.286947.

Cook, W. R. "A Proposal for Making Eiffel Type-Safe". In: *Proceedings of the 3rd European Conference on Object-Oriented Programming*. ECOOP'89. Nottingham, UK, July 1989, pp. 57–70.

Cook, W. R. "On Understanding Data Abstraction, Revisited". In: *Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2009. Orlando, FL, USA: ACM, Oct. 2009, pp. 557–572. DOI: 10.1145/1640089.1640133.

Cook, W. R., Hill, W. L., and Canning, P. S. "Inheritance Is Not Subtyping". In: *Proceedings of the 17th Annual Symposium on Principles of Programming Languages*. POPL'90. San Francisco, CA, USA, Jan. 1990, pp. 125–135. DOI: 10.1145/96709.96721.

Cook, W. R. and Palsberg, J. "A Denotational Semantics of Inheritance and Its Correctness". In: *Proceedings of the 4th International Conference on Object-oriented Programming Systems, Languages and Applications*. OOPSLA'89. New Orleans, Louisiana, USA, Oct. 1989, pp. 433–443. DOI: 10.1145/74877.74922.

Dart Project. *Dart Programming Language Specification*. Standard ECMA-408. Ecma International, June 2015.

Dubochet, G. and Odersky, M. "Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS'09. Genova, Italy, July 2009, pp. 34–41. DOI: 10.1145/1565824.1565829.

ECMAScript Project. *ECMAScript 2016 Language Specification*. Ed. by A. Wirfs-Brock. 6th Edition. ECMA-262. Ecma International, June 2016.

Fähndrich, M. and Xia, S. "Establishing object invariants with delayed types". In: *Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'07. Montreal, Quebec, Canada, Oct. 2007, pp. 337–350. DOI: 10.1145/1297027.1297052.

Felleisen, M., Findler, R. B., and Flatt, M. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J. A., and Tobin-Hochstadt, S. "The Racket Manifesto". In: *1st Summit on Advances in Programming Languages*. SNAPL'15. May 2015, pp. 113–128. DOI: 10.4230/LIPIcs.SNAPL.2015.113.

Findler, R. B. and Felleisen, M. "Contracts for Higher-Order Functions". In: *Proceedings of the 7th International Conference on Functional Programming*. ICFP'02. Pittsburgh, PA, USA, Oct. 2002, pp. 48–59. DOI: 10.1145/581478.581484.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. "Extended Static Checking for Java". In: *Proceedings of the 23rd Conference on Programming Language Design and Implementation*. PLDI'02. June 2002, pp. 234–245. DOI: 10.1145/512529.512558.

Furr, M., An, J., Foster, J. S., and Hicks, M. "Static Type Inference for Ruby". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC'09. Honolulu, Hawaii: ACM, 2009, pp. 1859–1866. DOI: 10.1145/1529282.1529700.

Garcia, R. "Calculating Threesomes, with Blame". In: *Proceedings of the 18th International Conference on Functional Programming*. ICFP'13. Boston, MA, USA, Oct. 2013, pp. 417–428. DOI: 10.1145/2500365.2500603.

Garcia, R. and Cimini, M. "Principal Type Schemes for Gradual Programs". In: *Proceedings of the 42nd Symposium on Principles of Programming Languages*. POPL'15. Mumbai, India, Jan. 2015, pp. 303–315. DOI: 10.1145/2676726.2676992.

Garcia, R., Clark, A. M., and Tanter, É. "Abstracting Gradual Typing". In: *Proceedings of the 43rd Symposium on Principles of Programming Languages*. POPL'16. St. Petersburg, FL, USA, 2016, pp. 429–442. DOI: 10.1145/2837614.2837670.

Gil, J. and Maman, I. "Whiteoak: introducing structural typing into Java". In: *Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'08. Nashville, TN, USA, Oct. 2008, pp. 73–90. DOI: 10.1145/1449764.1449771.

Gil, J. and Shragai, T. "Are We Ready for a Safer Construction Environment?" In: *Proceedings of the 23rd European Conference on Object-Oriented Programming*. ECOOP'09. Genoa, Italy, July 2009, pp. 495–519. DOI: 10.1007/978-3-642-03013-0_23.

Glew, N. "Type Dispatch for Named Hierarchical Types". In: *Proceedings of the 4th International Conference on Functional Programming*. ICFP'99. Paris, France, Sept. 1999, pp. 172–182. DOI: 10.1145/317636.317797.

Go Project. *The Go Programming Language Specification*. Version of May 31. 2016. URL: https://golang.org/ref/spec.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. ISBN: 978-0201113716.

Graver, J. O. and Johnson, R. E. "A Type System for Smalltalk". In: *Proceedings of the 17th Annual Symposium on Principles of Programming Languages*. POPL'90. San Francisco, CA, USA, Jan. 1990, pp. 136–150. DOI: 10.1145/96709.96722.

Hall, C. V., Hammond, K., Peyton-Jones, S. L., and Wadler, P. "Type Classes in Haskell". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 18.2 (1996), pp. 109–138. DOI: 10.1145/227699.227700.

Harper, R. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. ISBN: 9781107150300.

Henglein, F. "Dynamic typing: syntax and proof theory". In: *Science of Computer Programming* 22.3 (1994), pp. 197–230. DOI: http://dx.doi.org/10.1016/0167-6423(94)00004-2.

Herman, D., Tomb, A., and Flanagan, C. "Space-efficient gradual typing". In: *Higher-Order and Symbolic Computation* 2 (2010), pp. 167–189. DOI: 10.1007/s10990-011-9066-z.

Homer, M., Bruce, K. B., Noble, J., and Black, A. P. "Modules As Gradually-typed Objects". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA'13. Montpellier, France, July 2013, 1:1–1:8. DOI: 10.1145/2489798.2489799.

Homer, M., Jones, T., Noble, J., Bruce, K. B., and Black, A. P. "Graceful Dialects". In: *Proceedings of the 28th European Conference on Object-Oriented Programming*. ECOOP'14. July 2014, pp. 131–156. DOI: 10.1007/978-3-662-44202-9_6.

Homer, M., Noble, J., Bruce, K. B., Black, A. P., and Pearce, D. J. "Patterns as objects in Grace". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS'12. Tucson, AZ, USA, Oct. 2012, pp. 17–28. DOI: 10.1145/2384577.2384581.

Horwat, W. and Miller, M. S. *ES6 Strawman: Trademarks*. 2011. URL: http://wiki.ecmascript.org/doku.php?id%20=strawman:trademarks.

Ierusalimschy, R., Figueiredo, L. H. de, and Filho, W. C. "The Evolution of Lua". In: *Proceedings of the Third History of Programming Languages Conference*. HOPL-III. San Diego, CA, USA, June 2007, pp. 1–26. DOI: 10.1145/1238844.1238846.

Igarashi, A., Pierce, B. C., and Wadler, P. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 23.3 (May 2001), pp. 396–450. DOI: 10.1145/503502.503505.

Ina, L. and Igarashi, A. "Gradual typing for generics". In: *Proceedings of the 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'11. Portland, OR, USA, Oct. 2011, pp. 609–624. DOI: 10.1145/2048066.2048114.

Ingalls, D. "The Smalltalk-76 Programming System". In: *Proceedings of the 5th Symposium on Principles of Programming Languages*. POPL'78. Tucson, AZ, USA, Jan. 1978, pp. 9–16. DOI: 10.1145/512760.512762.

James, P. R. and Chalin, P. "Extended static checking in JML4: benefits of multiple-prover support". In: *Proceedings of the 24th Annual Symposium on Applied Computing*. SAC'09. Honolulu, HI, USA, Mar. 2009, pp. 609–614. DOI: 10.1145/1529282.1529410.

Jantz, M. R. and Kulkarni, P. A. "Performance potential of optimization phase selection during dynamic JIT compilation". In: *Proceedings of the 9th Conference on Virtual Execution Environments*. VEE'13. Houston, TX, USA, Aug. 2013, pp. 131–142. DOI: 10.1145/2451512.2451539.

Jones, T. *Hopper*. 2016. URL: https://github.com/zmthy/hopper.

Jones, T., Homer, M., and Noble, J. "Brand Objects for Nominal Typing". In: *Proceedings of the 29th European Conference on Object-Oriented Programming*. ECOOP'15. July 2015, pp. 198–221. DOI: 10.4230/LIPIcs.ECOOP.2015.198.

Jones, T., Homer, M., Noble, J., and Bruce, K. "Object Inheritance Without Classes". In: *Proceedings of the 30th European Conference on Object-Oriented Programming*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. ECOOP'16. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 13:1–13:26. DOI: 10.4230/LIPIcs.ECOOP.2016.13.

Jones, T. and Noble, J. "Tinygrace: A simple, safe, and structurally typed language". In: *Proceedings of the 16th International Workshop on Formal Techniques for Java-like Programs*. FTfJP'14. Uppsala, Sweden, July 2014, 3:1–3:6. DOI: 10.1145/2635631.2635848.

Kiczales, G., des Rivières, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN: 978-0262610742.

King, G. *The Ceylon Lanugage*. Version 1.3. 2016. URL: https://ceylon-lang.org/documentation/1.3/spec/.

Läufer, K., Baumgartner, G., and Russo, V. F. "Safe Structural Conformance for Java". In: *The Computer Journal* 43.6 (2000), pp. 469–481. DOI: 10.1093/comjnl/43.6.469.

Lee, J., Aldrich, J., Shaw, T., and Potanin, A. "A Theory of Tagged Objects". In: *Proceedings of the 29th European Conference on Object-Oriented Programming*. ECOOP 2015. Prague, Czech Republic, July 2015, pp. 174–197. DOI: 10.4230/LIPIcs.ECOOP.2015.174.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. *The OCaml System Release*. 4.03. Institut National de Recherche en Informatique et en Automatique, 2016. URL: http://caml.inria.fr/pub/docs/manual-ocaml.

Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". In: *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA'86. Portland, Oregon, USA: ACM, 1986, pp. 214–223. DOI: 10.1145/28697.28718.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. *Java Virtual Machine Specification*. Java SE 7 Edition. 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

Liquori, L. and Spiwack, A. "FeatherTrait: A modest extension of Featherweight Java". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 30.2 (Mar. 2008), 11:1–11:32. DOI: 10.1145/1330017.1330022.

Liskov, B. "Keynote Address — Data Abstraction and Hierarchy". In: *Addendum to the Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA'87. Orlando, FL, USA, Oct. 1987, pp. 17–34. DOI: 10.1145/62138.62141.

Lua-Users. *Object Oriented Programming*. [Online; accessed 30-November-2015]. 2014. URL: http://lua-users.org/wiki/ObjectOrientedProgramming.

Mackay, J., Hannes, Potanin, M. A., Groves, L., and Cameron, N. "Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant". In: *Proceedings of the 14th International Workshop on Formal Techniques for Java-like Programs*. FTfJP'12. Beijing, China, July 2012, pp. 11–19. DOI: 10.1145/2318202.2318206.

Malayeri, D. and Aldrich, J. "Combining structural subtyping and external dispatch". In: *Companion to the 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'07. Montreal, Quebec, Canada, Oct. 2007, pp. 789–790. DOI: 10.1145/1297846.1297889.

Malayeri, D. and Aldrich, J. "CZ: Multiple Inheritance Without Diamonds". In: *Proceedings of the 24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'09. Orlando, FL, USA, Oct. 2009, pp. 21–40. DOI: 10.1145/1640089.1640092.

Malayeri, D. and Aldrich, J. "Integrating Nominal and Structural Subtyping". In: *Proceedings of the 22nd European Conference on Object-Oriented Programming*. ECOOP'08. Paphos, Cyprus, July 2008, pp. 260–284. DOI: 10.1007/978-3-540-70592-5_12.

Markstrum, S., Marino, D., Esquivel, M., Millstein, T. D., Andreae, C., and Noble, J. "JavaCOP: Declarative pluggable types for Java". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 32.2 (2010). DOI: 10.1145/1667048.1667049.

Martelli, A. *Type checking in Python?* 2000. URL: https://groups.google.com/forum/#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ.

Martin, R. C. "The Interface Segregation Principle". In: *C++ Report* (June 1996).

Meyer, B. *Design by Contract*. Tech. rep. TR-EI-12/CO. Interactive Software Engineering Inc., 1986.

Miller, M. S. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control". PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University, May 2006.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997. ISBN: 978-0262631815.

Morris Jr., J. H. "Protection in Programming Languages". In: *Commun. ACM* 16.1 (Jan. 1973), pp. 15–21. ISSN: 0001-0782. DOI: 10.1145/361932.361937. URL: http://doi.acm.org/10.1145/361932.361937.

Nelson, G., ed. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

Nistor, L., Kurilova, D., Balzer, S., Chung, B., Potanin, A., and Aldrich, J. "Wyvern: A Simple, Typed, and Pure Object-Oriented Language". In: *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance*. MASPEGHI'13. Montpellier, France, July 2013, pp. 9–16. DOI: 10.1145/2489828.2489830.

Noble, J., Homer, M., Jones, T., Black, A., and Bruce, K. "Grace's Inheritance". In: *Journal of Object Technology* (2017).

Noort, T. van, Achten, P., and Plasmeijer, R. "Ad-hoc polymorphism and dynamic typing in a statically typed functional language". In: *Proceedings of the 6th Workshop on Generic Programming*. WGP'10. Baltimore, MD, USA, Sept. 2010, pp. 73–84. DOI: 10.1145/1863495.1863505.

Norell, U. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers Institute of Technology, 2007.

Odersky, M. *The Scala Language Specification: Version 2.9*. Programming Methods Laboratory, EPFL, Switzerland, June 2014.

Owens, S. "A Sound Semantics for OCamlLight". In: *Proceedings of the 17th European Symposium on Programming*. ESOP'08. London, UK, 2008, pp. 1–15. DOI: 10.1007/978-3-540-78739-6_1.

Palsberg, J. and Schwartzbach, M. I. *Object-Oriented Type Systems*. Chichester: John Wiley & Sons, 1994.

Papi, M. M., Ali, M., Jr., T. L. C., Perkins, J. H., and Ernst, M. D. "Practical pluggable types for Java". In: *Proceedings of the 7th International Symposium on Software*

Testing and Analysis. ISSTA'08. Seattle, WA, USA, July 2008, pp. 201–212. DOI: 10.1145/1390630.1390656.

Pearce, D. J. "JPure: A Modular Purity System for Java". In: *Proceedings of the 20th International Conference on Compiler Construction*. CC'11. Saarbrücken, Germany, Mar. 2011, pp. 104–123. DOI: 10.1007/978-3-642-19861-8_7.

Pearce, D. J. "Sound and Complete Flow Typing with Unions, Intersections and Negations". In: *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'13. Rome, Italy, Jan. 2013, pp. 335–354. DOI: 10.1007/978-3-642-35873-9_21.

Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

Pierce, B. C. and Turner, D. N. "Simple Type-Theoretic Foundations for Object-Oriented Programming". In: *Journal of Functional Programming* 4 (1994), pp. 207–247. DOI: 10.1017/S0956796800001040.

Python Project. *The Python Language Reference*. Version 2.7.13. 2016. URL: https://docs.python.org/2/reference/.

Qi, X. and Myers, A. C. "Masked types for sound object initialization". In: *Proceedings of the 36th Symposium on Principles of Programming Languages*. POPL'09. Savannah, GA, USA, Jan. 2009, pp. 53–65. DOI: 10.1145/1480881.1480890.

Rastogi, A., Chaudhuri, A., and Hosmer, B. "The ins and outs of gradual type inference". In: *Proceedings of the 39th Symposium on Principles of Programming Languages*. POPL'12. Philadelphia, PA, USA, Jan. 2012, pp. 481–494. DOI: 10.1145/2103656.2103714.

Rompf, T. and Amin, N. "Type soundness for dependent object types (DOT)". In: *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'16. Oct. 2016, pp. 624–641. DOI: 10.1145/2983990.2984008.

Ruby Project. *Ruby*. Standard ISO/IEC 30170:2012. International Organization for Standardization, Apr. 2012.

Rust Project. *The Rust Programming Language*. 2016. URL: https://doc.rust-lang.org/book/.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. "An Introduction to Trellis/Owl". In: *Proceedings of the 1st Conference on Object-Oriented Pro-*

gramming Systems, Languages and Applications. OOPSLA'86. Portland, Oregon, USA: ACM, 1986, pp. 9–16. DOI: 10.1145/28697.28699.

Servetto, M., Mackay, J., Potanin, A., and Noble, J. "The Billion-Dollar Fix — Safe Modular Circular Initialisation with Placeholders and Placeholder Types". In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Montpellier, France, July 2013, pp. 205–229. DOI: 10.1007/978-3-642-39038-8_9.

Shaughnessy, P. *Ruby Under A Microscope*. No Starch Press, 2013. ISBN: 978-1593275273.

Siek, J. G. and Taha, W. "Gradual Typing for Functional Languages". In: *Proceedings of the Scheme and Functional Programming Workshop*. SFP'06. Portland, OR, USA, Sept. 2006, pp. 81–92.

Siek, J. G. and Taha, W. "Gradual Typing for Objects". In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. July 2007, pp. 2–27. DOI: 10.1007/978-3-540-73589-2_2.

Siek, J. G. and Vachharajani, M. "Gradual typing with unification-based inference". In: *Proceedings of the 4th Symposium on Dynamic Languages*. DLS'08. Paphos, Cyprus, July 2008, 7:1–7:12. DOI: 10.1145/1408681.1408688.

Siek, J. G., Vitousek, M. M., and Bharadwaj, S. *Gradual Typing for Mutable Objects*. Unpublished manuscript. 2012. URL: https://ecee.colorado.edu/~siek/gtmo.pdf.

Siek, J. G., Vitousek, M. M., Cimini, M., and Boyland, J. T. "Refined Criteria for Gradual Typing". In: *Proceedings of the 1st Summit on Advances in Programming Languages*. SNAPL'15. May 2015, pp. 274–293. DOI: 10.4230/LIPIcs.SNAPL.2015.274.

Siek, J. G. and Wadler, P. "Threesomes, with and Without Blame". In: *Proceedings of the 37th Symposium on Principles of Programming Languages*. POPL'10. Madrid, Spain, Jan. 2010, pp. 365–376. DOI: 10.1145/1706299.1706342.

Standish, T. A. "Extensibility in programming language design". In: *Proceedings of the American Federation of Information Processing Societies National Computer Conference*. AFIPS'75. Anaheim, CA, USA, May 1975, pp. 287–290. DOI: 10.1145/1499949.1500003.

Stein, L. A., Lieberman, H., and Ungar, D. "A Shared View of Sharing: The Treaty of Orlando". In: *Object-Oriented Concepts, Databases, and Applications*. 1989, pp. 31–48.

Strickland, T. S. and Felleisen, M. "Contracts for first-class classes". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS'10. Reno, NV, USA, Oct. 2010, pp. 97–112. DOI: 10.1145/1869631.1869642.

Strickland, T. S., Tobin-Hochstadt, S., Findler, R. B., and Flatt, M. "Chaperones and impersonators: run-time support for reasonable interposition". In: *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'12. Tucson, AZ, USA, Oct. 2012, pp. 943–962. DOI: 10.1145/2384616.2384685.

Stroustrup, B. "Evolving a language in and for the real world: C++ 1991-2006". In: *Proceedings of the 3rd Conference on History of Programming Languages*. HOPL-III. San Diego, California, 2007, pp. 1–59. DOI: 10.1145/1238844.1238848.

Strub, P.-Y., Swamy, N., Fournet, C., and Chen, J. "Self-certification: bootstrapping certified typecheckers in F* with Coq". In: *Proceedings of the 39th Symposium on Principles of Programming Languages*. POPL'12. Philadelphia, PA, USA, 2012, pp. 571–584. DOI: 10.1145/2103656.2103723.

Summers, A. J. "Modelling Java Requires State". In: *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*. FTfJP'09. Genova, Italy, July 2009, 10:1–10:3. DOI: 10.1145/1557898.1557908.

Summers, A. J. and Müller, P. "Freedom before commitment: a lightweight type system for object initialisation". In: *Proceedings of the 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'11. Portland, OR, USA, Oct. 2011, pp. 1013–1032. DOI: 10.1145/2048066.2048142.

Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., and Yang, J. "Secure distributed programming with value-dependent types". In: *Proceedings of the 16th International Conference on Functional Programming*. ICFP'11. Tokyo, Japan, Sept. 2011, pp. 266–278. DOI: 10.1145/2034773.2034811.

Taivalsaari, A. "Classes Versus Prototypes: Some Philosophical and Historical Observations". In: *Journal of Object-Oriented Programming* 10.7 (1997), pp. 44–50.

Taivalsaari, A. "Delegation versus Concatenation or Cloning is Inheritance too". In: *OOPS Messenger* 6.3 (1995), pp. 20–49. DOI: 10.1145/219260.219264.

Takikawa, A., Strickland, T. S., Dimoulas, C., Tobin-Hochstadt, S., and Felleisen, M. "Gradual typing for first-class classes". In: *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'12. Tucson, AZ, USA, Oct. 2012, pp. 793–810. DOI: 10.1145/2384616.2384674.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.5pl3. 2016. URL: http://coq.inria.fr/distrib/V8.4/refman.

Torgersen, M., Ernst, E., and Hansen, C. P. "Wild FJ". In: *Proceedings of the 12th Workshop on Foundations of Object-Oriented Languages*. FOOL'05. Long Beach, CA, USA, Jan. 2005.

Ungar, D. and Smith, R. B. "SELF: The Power of Simplicity". In: *Lisp and Symbolic Computation* 4.3 (1991). DOI: 10.1007/BF01806105.

Urban, C., Berghofer, S., and Norrish, M. "Barendregt's Variable Convention in Rule Inductions". In: *Proceedings of the 21st International Conference on Automated Deduction*. CADE'07. Bremen, Germany, July 2007, pp. 35–50. DOI: 10.1007/978-3-540-73595-3_4.

Wadler, P. and Findler, R. B. "Well-Typed Programs Can't Be Blamed". In: *Proceedings of the 18th European Symposium on Programming*. ESOP'09. Mar. 2009, pp. 1–16. DOI: 10.1007/978-3-642-00590-9_1.

Wehr, S., Lämmel, R., and Thiemann, P. "JavaGI : Generalized Interfaces for Java". In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. Berlin, Germany, July 2007, pp. 347–372. DOI: 10.1007/978-3-540-73589-2_17.

Xu, D. N. "Extended static checking for Haskell". In: *Proceedings of the 10th Haskell Workshop*. Haskell'06. Portland, OR, USA, Sept. 2006, pp. 48–59. DOI: 10.1145/1159842.1159849.

Zibin, Y., Cunningham, D., Peshansky, I., and Saraswat, V. A. "Object Initialization in X10". In: *Proceedings of the 26rd European Conference on Object-Oriented Programming*. Beijing, China, June 2012, pp. 207–231. DOI: 10.1007/978-3-642-31057-7_10.