University of London Imperial College of Science, Technology and Medicine Department of Computing

# Existential Types for Variance — Java Wildcards and Ownership Types

Nicholas Cameron

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, April 2009

#### Statement of Originality

Chapter 2 is my own work. Sections 3.1 and 3.2 is collaborative work with Sophia Drossopoulou and Erik Ernst [24, 21]; the original formalism, descriptions and proofs are my own, but have been improved by much feedback over many iterations. In particular, section 3.2 was substantially re-written by Erik Ernst. Section 3.3 is my own work. Chapter 4 is substantially my own work but benefited from many discussions with Sophia Drossopoulou [19, 20] and was influenced and motivated by discussions on multiple ownership [23, 22] with Sophia Drossopoulou, James Noble, and Matthew Smith. Chapter 5 is my own work.

#### Abstract

Inclusion polymorphism (subclassing) and parametric polymorphism (generics) are often present in modern object oriented languages. However, their integration is usually limited; an important example is that variant subtyping of parametric types is usually forbidden, even though in some circumstances it is safe and desirable. Existential types have been used to bridge the gap between these two forms of polymorphism used in, for example, Java wildcards or Variant Parametric Types.

In this thesis we investigate how existential types can be used to implement variant subtyping. We contribute a soundness proof for Java with wildcards, and a new, minimal language that uses existential types to implement variance for ownership types.

Java wildcards provide subtype variance to Java generics in a powerful and programmer friendly way. Wildcards have been formalised using a variation on existential types; however, there has never been a type soundness proof for Java with wildcards. Our main contribution is a new formal model of Java with wildcards (Tame FJ) and a detailed proof of soundness.

Ownership types are a mechanism for structuring the topology of the heap in object oriented programs. In an ownership types system, types are parameterised by owners. Similarly to generics, owners are traditionally treated invariantly. There have been several, mostly ad hoc, attempts to add some form of subtype variance to owner-parametric types. Our second contribution is a minimal calculus (Jo $\exists$ ) where existential quantification of owners is used to uniformly and cleanly support variance. We include type parametricity in our language, and the interaction of this with existentially quantified owners allows us to more precisely specify the ownership properties of collections and similar code structures. We prove soundness and the owners-as-dominators property for this system.

#### Acknowledgements

It has been a privilege to work with, and learn from, my supervisor, Sophia Drossopoulou. Sophia has been consistently knowledgeable, insightful, and enthusiastic. I am particularly grateful to her for accepting me as a PhD student when I had little instruction in programming language theory and a slightly unorthodox academic record.

I have had many interesting and informative discussions and email exchanges with Erik Ernst. I have learnt a lot about Java wildcards, and language design and theory from Erik. I have gained much from working with James Noble; I have been inspired by his knowledge, creativity, and passion. I am also most grateful to him for taking me on as a research fellow. Working with Matthew Smith taught me lots about ownership types (and other topics); he also brightened my PhD experience with his humour. The anonymous reviewers of our papers gave us lots of useful feedback, which has improved the work in this thesis. I would like to thank my examiners for reading my thesis thoroughly, an interesting and constructive viva, and ideas for improvements and future work.

I am grateful to Alex Summers, Alex Buckley, Stephan Wehr, Christian Urban, Dave Clarke, Mariangiola Dezani-Ciancaglini, and Atsushi Igarashi for interesting and helpful discussions about this work; this thesis would be much poorer without their input.

Nobuko Yoshida, Susan Eisenbach and the rest of the SLURP group have given much useful feedback on my work as well as involving me in many interesting discussions about many aspects of programming languages. My time at Imperial College London has been greatly enhanced by being surrounded by such an intelligent and impassioned group of people.

My wife, Rujuta, has been a constant source of support and comfort throughout my PhD studies. It would have been a much less enjoyable and productive time without her. My parents have been extremely supportive (financially and otherwise) throughout my educational career; obviously, I owe them a great deal. I've met and become friends with many good people at Imperial College London, it has been fantastic to be part of such a supportive and friendly environment. In particular, I would like to thank Will Heaven and Karen Osmond for giving me a home for my final, frantic month of writing.

Finally, I would like to thank Cecilia Mascolo for encouraging me to pursue a PhD and helping me through the application process.

# Contents

A	bstract 3				
A	cknov	wledgements	5		
1	Intr	roduction 1			
	1.1	Subtyping and Variance	21		
	1.2	Existential Types and Variance	22		
	1.3	Tame FJ	23		
	1.4	Jo∃	24		
	1.5	Contributions	25		
	1.6	Thesis Organisation	26		
	1.7	Publications	26		
<b>2</b>	Bac	kground	28		
	2.1	Objects and Subtyping	28		
		2.1.1 Subtyping in Object-Oriented Languages	29		
		2.1.2 Featherweight Java	31		

	2.1.3	Type Safety	32
2.2	Param	netric Types	33
	2.2.1	Java Generics	34
	2.2.2	Variant Subtyping	40
2.3	Existe	ential Types	49
	2.3.1	A Language with Existential Types — Fun	49
	2.3.2	Static and Dynamic Semantics of Existential Types	52
	2.3.3	Open existential types	54
	2.3.4	Decidability	54
	2.3.5	Modelling Abstract Types	55
	2.3.6	Existential Types for Imperative Languages	57
	2.3.7	Existential Types for Subtype Variance	58
2.4	Wildc	ards	64
	2.4.1	Existential Types for Wildcards	66
	2.4.2	Wildcard Capture	67
	2.4.3	F-Bounds	69
	2.4.4	Using Wildcard Types	69
	2.4.5	Comparison with Variant Parametric Types	71
	2.4.6	Previous Models of Java with Wildcards	73
	2.4.7	Decidability	75
2.5	Owner	rship Types	76

	2.5.1	Encapsulation Properties	81
	2.5.2	Related Systems	83
	2.5.3	Variant Ownership	85
	2.5.4	Existential Types and Ownership	87
	2.5.5	Generics and Ownership Types	89
For	mal M	odels for Wildcards	93
3.1	Tame	FJ	93
	3.1.1	Notation and Syntax	94
	3.1.2	Subtyping	96
	3.1.3	Well-formedness	99
	3.1.4	Typing	101
	3.1.5	Operational Semantics	108
	3.1.6	Type Soundness	109
3.2	Transl	ating Java to Tame FJ	113
3.3	Discus	ssion — Modelling Wildcards	116
	3.3.1	Some Previous attempts	119
	3.3.2	Wildcards and Existential Types	120
	3.3.3	Decidability	124
3.4	Chapt	er Summary	125

4	Exi	stential types for Context Variance	126
	4.1	Motivation	127
		4.1.1 Existential Types	128
		4.1.2 Type Parameterisation	128
		4.1.3 Example	129
	4.2	Jo∃	132
		4.2.1 Syntax	133
		4.2.2 Types in Jo $\exists$	135
		4.2.3 Subtyping and the Inside Relation	136
		4.2.4 Well-formedness	137
		4.2.5 Typing	139
		4.2.6 Dynamic Semantics	142
	4.3	Deep Ownership	147
	4.4	Discussion	153
		4.4.1 An Application — Effects	157
	4.5	Wildcards-Style Existential Types	158
		4.5.1 $\operatorname{Jo}\exists_{wild}$	159
		4.5.2 Programs in $Jo\exists_{wild}$ and $Jo\exists$	162
	4.6	Chapter Summary	164

5	Con	nparisons and Discussion 10	<b>35</b>
	5.1	Comparison of Tame FJ and Jo $\exists$	66
		5.1.1 Comparison of Proofs	68
	5.2	Related Work — Wildcards	69
		5.2.1 Wild FJ	69
		5.2.2 Variant Parametric Types	72
		5.2.3 Pizza and $\mathcal{EX}_{upto}$	72
	5.3	Related Work — Ownership	75
		5.3.1 Variance	75
		5.3.2 Existential Types	80
	5.4	Chapter Summary	81
6	Con	clusions 18	33
	6.1	Further Work	85
		6.1.1 Tame FJ	85
		6.1.2 Improving Wildcards in Java	86
		6.1.3 Jo $\exists$	87
A	Pro	ofs of properties of Tame FJ 18	<b>3</b> 9
	A.1	Outline of proofs	89
	A.2	Proofs	90

В	Proofs of properties of $Jo\exists$			
	B.1 Outline of proofs	. 215		
	B.2 Proofs	216		
Bi	bliography	245		

# List of Figures

2.1	Observer pattern in Java.	37
2.2	The Box example using virtual types	44
2.3	The Observer pattern implemented using virtual types	45
2.4	Subtype relationships between variant parametric types	48
2.5	Example 1	68
2.6	Example 2	69
2.7	Testing the behaviour of F-bounded wildcard types	70
3.1	Syntax of Tame FJ	94
3.2	Tame FJ subclasses, extended subclasses, and subtypes	96
3.3	Tame FJ well-formed types and type environments	99
3.4	Tame FJ class and method typing rules.	100
3.5	Tame FJ expression typing rules.	101
3.6	Auxiliary functions for Tame FJ.	103
3.7	Method and field lookup functions for Tame FJ	104
3.8	Example of a derivation for field access.	105

3.9	Example of a derivation for method invocation
3.10	Example 2 in Tame FJ
3.11	Tame FJ reduction rules
3.12	Syntax of Java types
3.13	Translation from Java types to Tame FJ types
3.14	Syntax of ∃J
3.15	Selected $\exists J$ typing and subtyping rules
4.1	Example: existential types for context variance
4.2	Syntax of Jo∃
4.3	Jo $\exists$ subtyping, and the inside relation for owners and environments 136
4.4	Jo $\exists$ well-formed contexts, types, and environments
4.5	Jo $\exists$ expression typing rules
4.6	Field and method lookup functions for Jo $\exists$
4.7	Jo $\exists$ typing rules for classes and methods
4.8	$Jo\exists$ reduction rules
4.9	Jo $\exists$ reduction rules for <b>null</b> and error propagation
4.10	Jo $\exists$ well-formed heaps and configurations
4.11	Using the heap as an environment in Jo $\exists$
4.12	Owner lookup functions for $Jo\exists_{deep}$
4.13	Example: owners-as-dominators
4.14	An example of a Jo $\exists$ program with effects

4.15	Syntax of $Jo\exists_{wild}$	160
4.16	$Jo\exists_{wild}$ rules for method invocation	161
5.1	Pizza and Tame FJ type rules for existential types.	173
5.2	Usage and definition of a list in the universes system	178
5.3	The universes list in Jo $\exists$	179

# Chapter 1

# Introduction

Parametric polymorphism allows code to be written and type checked generically, and instantiated with specific types. This increases the flexibility of the type system and improves reuse. Various kinds of parametric polymorphism are found in functional languages and calculi, from *let-polymorphism* in ML to *universal types* in System F [46, 80]. In the procedural world, parametric polymorphism, known as *generic programming* or *generics*, first appeared in Ada [4]. This gave rise to parameterisation of classes in object-oriented languages such as C++ [2, 86], C# [1], and Java [14, 16, 47].

Parametric types in Java are called generic types, for example, List<String> represents a list of strings; as opposed to the non-generic type List which does not specify the contents of the list. Generic types are used throughout the Java libraries and are an important part of the Java language.

We say parameterised types are *variant* if subtyping relationships between parameters cause some subtyping relationship between parameterised types. Subtyping of generic types in Java is *invariant* (type parameters cannot change across subtypes: ArrayList<String> is a subtype of List<String>, but List<String> is not a subtype of List<Object>) with respect to their parameters, this is necessary to guarantee soundness. *Variance* consists of *covariance*, *contravariance*, and *bivariance*. Types are covariant if a subtype relation between parameters gives a subtype relation between parameterised types, contravariant if a supertype relation between parameters gives a subtype relation between parameterised types, and bivariant if both cases hold.

*Existential types* [26, 27, 49, 64, 74, 75] are another form of polymorphism. Existential types correspond to existential quantification in logic in the same way that universal types correspond to universal quantification. Existential types model abstraction; they express that some component of a type is unknown or partially known (of course, at some point in the program the hidden type must be known, it is then deliberately concealed and treated as unknown). This expression of partial knowledge is useful for dealing with variant generic types; if the compiler only relies on partial knowledge of a type parameter, then it is safe for generic types to support limited subtype variance (as opposed to in Java where variant subtyping of parametric types would be unsafe). That is, if only the bounds of a type parameter are known, then operations on variables of that type must be restricted; but, since fewer operations must be accommodated, subtyping of types parameterised by the partially unknown type parameter can be less restrictive.

In Java, *wildcards* are used to support subtype variance. Wildcards safely allow variant subtyping by restricting how objects with wildcard type can be used. Wildcard types correspond closely to existential types, and these are used in most formal models of Java with wildcards [54, 60, 90, 95].

Wildcards have been part of the Java language since 2004, but type soundness for Java with wildcards has been an open question until now. There are several informal, semi-formal, and formal descriptions of Java wildcards [14, 47, 60, 90] and soundness proofs for partial systems [24, 54]. However, a soundness proof for a type system exhibiting all of the interesting features of Java wildcards has been elusive.

In an *ownership types* system [30, 32, 34, 33, 97], objects are structured to enforce encapsulation policies or describe the heap. Rather than parameterising classes by types, classes are parameterised by *contexts* (entities to which objects may belong, usually objects; sometimes "context" is used to refer to the are of the heap owned by an object or other owner). By taking this parameterisation into account and applying implicit rules of ownership, objects in the heap can be structured into a tree or graph. This structure can be used to enhance reasoning about programs, restrict aliasing, and enforce encapsulation. Similarly to generic types, subtyping for ownership types is usually invariant with respect to ownership parameters. Where some form of variance has been supported [23, 57, 58, 66, 67], it has been severely restricted or ad hoc in its formalisation, and often both.

### 1.1 Subtyping and Variance

A parametric class in Java, e.g., class List<X> ..., can be instantiated to a *parameterised type*, for example, List<Shape>. Different actual type parameters create unrelated parameterised types (i.e., generic types are invariant), List<Circle> is not a subtype of List<Shape>, even if Circle is a subtype of Shape.

Ownership types are parameterised by a *context* (usually an object) rather than a type, for example, Shape<this> denotes a Shape object owned by this. Contexts are related by the *inside* relation, rather than subtyping. Context parameterisation and the rules of the inside relation define a hierarchical structure over the heap. Ownership types are invariant, Shape<this> is not a subtype of Shape<owner>, even if this is inside owner.

There have been many approaches to variance in parametric type systems [8, 42, 54, 51, 61, 71, 87, 89] and several in ownership systems [23, 57, 58, 66, 67]; the Java solution is wildcard types [14, 47, 90, 60]. These extend generics by allowing parameterised types to have actual type arguments which denote unknown or partially known types, such as List<?>. Wildcards allow for subtype relationships among parametric types; for example, List<? extends Circle> *is* a subtype of List<? extends Shape>.

The closest system to wildcards in the ownership world is variant ownership types [58]. In this system variance annotations (which also exist in the generics world [54]) are used to explicitly denote the variance properties of context parameters. Although very similar to wildcards, variance annotations are not as expressive. Furthermore, in ownership systems there are partial abstractions of ownership that cannot be expressed with variance annotations or wildcards, for

example, the difference between a list of objects where each object may be owned by a different context and a list of objects where each object is owned by the same context. Maintaining encapsulation properties is a further requirement on variance mechanisms in ownership types which is not found in generic systems.

### **1.2** Existential Types and Variance

Existential types are used to express abstraction in types. For example, the function type  $\exists X. (X \to X)$  denotes a function from *some* unknown type to that same type. Existentially quantified type variables may be given bounds to express partial abstraction.

Existential types are *introduced* by *packing* a concrete value; this creates an *abstract package*. Abstract packages are opaque, they cannot be used or accessed, only passed around. The only operation that may be performed on abstract packages is *unpacking*, which *eliminates* existential types. In traditional systems [27, 64, 74], only abstract packages have existential types; no concrete value (that is an object or address) has existential type.

When used to quantify a parametric type, existential types express uncertainty about a type parameter. For example, the type  $\exists X.List < X >$  describes a list of objects with unknown type. We use the notation  $\rightarrow [B_l \ B_u]$  to express lower and upper bounds ( $B_l$  and  $B_u$ , respectively) on quantified variables. Bounded existential types express partial knowledge about a type, for example,  $\exists X \rightarrow [Circle \ Object] .List < X >$  describes a list of objects with unknown type where that type is a subtype of Object and a supertype of Circle (Object and Circle are the upper and lower bounds, respectively). Following the standard rules for existential subtyping, we get variant subtyping of generic types, as expected in object-oriented languages. Packing is used to relate variant to invariant types. Unpacking must be performed to use variables with existential type. In Java, *wildcard capture* corresponds to unpacking. Packing and unpacking is implicit, i.e., it is not apparent to the programmer.

### 1.3 Tame FJ

Tame FJ is an extension of FGJ [53], a minimal functional calculus that models the interesting aspects of Java with generics. The syntax of types is extended with existential quantification to model wildcard types. Explicit packing and unpacking (as used in traditional existential types systems) are not expressive enough to model variance in Java (section 3.3.2), therefore, these operations are implicit in Tame FJ.

Subtyping in Tame FJ extends that in FGJ to wildcard types, encoded as existential types. This extension to subtyping includes existential packing and a limited form of unpacking. Thus, subtyping in Tame FJ is significantly more complex than in FGJ and similar systems.

In formalisations of Java without wildcards, the types of fields and methods are invariant with respect to subtyping. With wildcards, this no longer holds, in fact the relationship is considerably more complicated. To address this, we split subtyping into subclassing and subtyping.

Type variables in Tame FJ may have upper and lower bounds. We need to avoid this so that spurious subtype relationships cannot be derived; for example, if X is bounded by Shape and String, we could derive that Shape is a subtype of String, which is unsound. Although X could never be instantiated, this is difficult to reflect in the proofs. In Tame FJ, such relationships are prevented by further splitting subtyping into extended subclassing and subtyping, and using the former to check well-formedness of type environments. Only subtyping includes type variables and their bounds, so this ensures that all subtype relationships judged under well-formed environments reflect the subclass hierarchy.

Expressions must be unpacked and packed in the type system of Tame FJ. Unpacking is relatively straightforward and follows previous formalisms. Packing is more complex because it can create types that are not syntactically well-formed. Without correct packing, unpacked type variables could escape their scope and cause unsoundness. We track unpacked variables using *guarding environments*, an addition to the shape of the type checking judgement. The unpacked variables are then re-packed using subtyping via an enhanced subsumption rule. The Java compiler supports type parameter inference. If unpacking is implicit, as in Tame FJ, type parameter inference is required because existentially quantified type variables cannot be named. Formalising this inference was a crucial, but complicated step towards a soundness proof.

#### **1.4** Jo∃

The various systems that support (use-site) subtype variance (wildcards, variant parametric types, variant ownership types, etc.) can be thought of in terms of existential types; some features of existential types are used in all of their formalisations. The surface syntax of these systems is more restrictive than the underlying system using existential types. Therefore, to design an ownership language with expressive subtype variance, we describe the underlying existential types system, Jo∃.

Similarly to Tame FJ, Jo $\exists$  adds existential quantification to the syntax of types; however, in Jo $\exists$ , it is context variables that are quantified, not type variables. This is an important distinction because context variables cannot be used as types in their own right, they can only be used as type *parameters*. In contrast, quantified type variables can be used as both types and type parameters. Therefore, quantification of contexts is simpler to reason about than quantification of types.

Jo $\exists$  uses explicit packing and unpacking: it has open and close expressions. This makes Jo $\exists$  easier to reason about and closer to earlier work on existential types, at the expense of being less realistic. Due to this and other simplifications, Jo $\exists$  does not require some of the innovations of Tame FJ, such as different kinds of subtyping relations or guarding environments. Furthermore, the proof of soundness for Jo $\exists$  is much simpler than it would be in a system with implicit packing and unpacking.

By using existential types for variance,  $Jo\exists$  is more expressive, more uniform, and less ad hoc than previous ownership systems with variant contexts.  $Jo\exists$  also supports type parameterisation; this, in combination with quantification of contexts, allows  $Jo\exists$  to express even more types.

Jo $\exists$  emerged naturally from design decisions to use existential quantification of contexts, parameterisation of contexts and types, and explicit packing and unpacking. The standard rules for enforcing owners-as-dominators had to be re-thought to accommodate references with existential type. Indeed, because of these references, the proof of owners-as-dominators was much more challenging than the proof of soundness.

#### 1.5 Contributions

Soundness result for Java with wildcards Tame FJ is the first formalisation of wildcards with all of the salient features of Java's type system to be proved sound. A soundness result is important because it guarantees that type-correct programs will not cause type errors at runtime. If Java were unsound, then type-correct Java programs could crash, corrupt shared memory, or allow security violations.

**Jo** $\exists$  Jo $\exists$  is the first language with explicit existential types to support subtype variance with respect to ownership information. Jo $\exists$  is more expressive and uniform than previous languages. By using existential quantification to implement variance, rather than ad hoc mechanisms, variance properties are more easily explained and understood. Jo $\exists$  can be used to encode many languages and language features, and thus allows for easy comparison of different proposals. By making operations on existential types explicit, features of other languages can be clearly explained in terms of well-understood type theory. We prove that Jo $\exists$  is sound and that it can support the owners-as-dominators encapsulation property.

We expect that our research around Jo∃ will have a primary impact amongst language designers, rather than programmers. We do not offer a usable programming language, but give the theoretical underpinnings for a language to be designed that combines the encapsulation benefits of ownership types with the flexibility of subtype variance. We expect this work to be especially valuable to research on intermediate languages for compilers involving ownership and program

analyses using effects or similar techniques. These areas could benefit from the flexibility and encapsulation, respectively, offered by Jo∃.

### 1.6 Thesis Organisation

In chapter 2 we give the necessary background to this thesis. We describe object-oriented programming languages and parametric polymorphism, existential types, Java wildcards, and ownership types. In chapter 3 we describe Tame FJ and its soundness proof, and discuss some of the interesting aspects of Java wildcards in relation to existential types. In chapter 4 we describe Jo $\exists$  and discuss how existential types can be used to support subtype variance in ownership languages. In chapter 5 we compare subtype variance and the use of existential types in Tame FJ and Jo $\exists$ , and discuss related work. We conclude, and discuss related work, in chapter 6. We outline proofs of soundness for Tame FJ in appendix A and proofs of soundness and the owners-as-dominators property for Jo $\exists$  and Jo $\exists_{deep}$  in appendix section B.

### 1.7 Publications

In chronological order:

- [20] Existential Quantification for Variant Ownership. Nicholas Cameron and Sophia Drossopoulou.ESOP, 2009. This paper summarises the work in chapter 4.
- [21] A Model for Java Wildcards. Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. ECOOP, 2008. Describes Tame FJ and its soundness proof. The descriptions and discussion of Tame FJ are expanded on in chapter 3.
- [19] Variant Ownership with Existential Types. Nicholas Cameron and Sophia Drossopoulou.IWACO, 2008 (position paper). Proposes the work in chapter 4.

- [23] Multiple Ownership. Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. OOPSLA, 2007. This work is touched upon in chapters 2 and 5 and had some influence on the work in chapter 4.
- [24] Towards an Existential Types Model for Java Wildcards. Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. FTfJP, 2007. Describes a partial model for Java with wildcards, ∃J, that uses explicit packing and unpacking. This is briefly discussed in section 3.3.2 and contributed to the design of Jo∃.
- [37] A state abstraction for coordination in Java-like languages. Ferruccio Damiani, Elena Giachino, Paola Giannini, Nick Cameron, and Sophia Drossopoulou. FTfJP, 2006. Not directly relevant to this thesis.
- [22] More Expressive Ownership Types. Nicholas Cameron, Sophia Drossopoulou, and James Noble. Grant Proposal, awarded 2008. Motivates and outlines work in chapter 4 and further work.

http://www.doc.ic.ac.uk/~ncameron/papers/cameron\_proposal08.pdf

# Chapter 2

## Background

In this thesis we tackle wildcards in Java, and subtype variance for ownership types systems. We use existential types to investigate both subjects and both are set in the context of Javalike languages. In this chapter we give some background to these areas; we discuss object oriented programming languages, specifically Java, in section 2.1. We extend this discussion to parametric polymorphism, including Java generics and some alternatives, in section 2.2. In section 2.3, we discuss existential types, we then use these to help describe Java wildcards in section 2.4. Finally, in section 2.5 we give some background on ownership types and similar systems.

### 2.1 Objects and Subtyping

The work in this thesis is based in the context of *class-based* object-oriented languages. Java [47] is a widely used and studied class based object-oriented language. Because its syntax is typical of a large family of object-oriented languages, we adopt it in this thesis. In a class-based language, the programmer declares classes, objects are instantiations of these classes. For example,

```
class Shape {
    void draw() {...}
}
```

defines a class Shape with a method draw. Shape objects are instantiated using the expression new Shape().

Class names are often used as types; in Java-like languages, the object given by new Shape() has type Shape. Thus, classes play a dual role: they are templates for object instantiation and types. Java also includes interfaces; an interface defines a type, but cannot be instantiated. Interfaces allow for a limited form of multiple inheritance in Java and allow for type to be somewhat separated from class.

*Inheritance* supports code re-use. A *subclass* may declare that it inherits from a *superclass* using the **extends** keyword, for example:

```
class Circle extends Shape {
    Shape getBound() {
        return this;
    }
    int countEdges() {
        return 1;
    }
}
```

A subclass inherits all of the functionality of the superclass, thus Circle includes a draw method from Shape.

#### 2.1.1 Subtyping in Object-Oriented Languages

In many object-oriented languages (including Java, C#, C++, and Scala), subtyping follows from inheritance (subclassing). In our example, Circle is a subtype of Shape. Subtyping is transitive, so Circle is a subtype of the (implicit) superclass of Circle, Object.

Since a subclass inherits all the fields and methods of its superclass, it is safe to use an instance of a subclass anywhere an instance of the superclass is required. For example, if we have a method declared as void m(Shape x) {...}, it is safe to call m with a Circle object as a parameter, since any operation that m can perform on a Shape can also be performed on a Circle. This property of object oriented languages is known as *subsumption*. It can be expressed formally by the subsumption type rule:

$$\frac{\vdash T' <: T \qquad \vdash e : T'}{\vdash e : T}$$
(T-SUBS)

Where e is an expression and T and T' are types.

According to Cardelli and Wegner's classification of *polymorphism* [27], the above description of subtyping supports *inclusion polymorphism*. In a *monomorphically typed* language, a function or procedure and its operands (formal parameters) have a unique type. Thus, a monomorphic function can only be applied to objects with a specific and unique type. *Polymorphically* typed languages allow functions and their operands to have multiple types; functions are generic since they can be applied to objects with different types. Cardelli and Wegner split polymorphism into *universal* and *ad-hoc polymorphism*; universally polymorphic functions can be applied to a finite set. Universal polymorphism is further separated into *parametric polymorphism* (described in section 2.2) and inclusion polymorphism; ad-hoc polymorphism consists of *overloading* (for example, two methods in a single class may have the same name but take parameters with different types) and type *coercions* (for example, a *int* may be used as an *float*), both are found in many languages, including Java, but are beyond the scope of this thesis.

Inclusion polymorphism is so called because an object may belong to several *non-disjoint*<sup>1</sup> types, therefore, types may be *included* inside one another. For example, the type Circle is included in type Shape since any object with type Circle also has type Shape.

<sup>&</sup>lt;sup>1</sup>as opposed to parametric polymorphism

#### 2.1.2 Featherweight Java

Featherweight Java (FJ) [53] is a minimal, functional calculus that includes, or can model, all the important features of the Java programming language and type system; it is a strict subset of Java. The formal calculi presented in this thesis all extend FJ in some way.

FJ includes classes, methods, fields, and constructors (constructors are simplified in later versions of the calculus such as Wild FJ [60]). A method includes a single expression which may be a field access, method invocation, variable access, creation of an object (new), or a cast (omitted from some later systems). Types are class names, and subtyping is the reflexive, transitive closure of declared subclassing.

In this thesis, we use and adapt the notation of FJ: we use  $\emptyset$  to denote the empty sequence,  $\overline{\mathbf{x}}$  to denote the sequence  $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$ , commas are used to concatenate two sequences, and it is implicitly assumed that concatenation of two sequences of mappings only succeeds if their domains are disjoint. Our use of the overbar notation is slightly different to that of FJ (and hopefully more natural), in FJ,  $\overline{\mathbf{x}}$   $\overline{\mathbf{y}}$  and  $\vdash \overline{\mathbf{x}} <: \overline{\mathbf{y}}$  are used to denote a sequence of pairs and a sequence of relations, we prefer  $\overline{\mathbf{x}}$   $\overline{\mathbf{y}}$  and  $\vdash \overline{\mathbf{x}} <: \overline{\mathbf{y}}$ , since we believe the FJ notation (in the first case) could be interpreted as  $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n$ , rather than  $\mathbf{x}_0$   $\mathbf{y}_0, \mathbf{x}_1$   $\mathbf{y}_1, \ldots, \mathbf{x}_n$   $\mathbf{y}_n$ . FJ makes heavy use of substitution, denoted  $[\mathbf{e}/\mathbf{x}]\mathbf{e}'$ , which means replace all free occurrences of  $\mathbf{x}$  in  $\mathbf{e}'$  with  $\mathbf{e}$ .

To keep the type rules syntax directed, subsumption is handled implicitly — there is no subsumption rule. Instead, type checking assigns an expression its minimal type<sup>2</sup>; types are compared by subtyping rather than equality. For example<sup>3</sup>,

$$fields(C) = \overline{T f;}$$

$$\Gamma \vdash \overline{e: T'} \vdash \overline{T' <: T}$$

$$\Gamma \vdash new C(\overline{e}) : C$$
(FJ-T-New)

<sup>&</sup>lt;sup>2</sup>A minimal type can always be found, and is always unique, in FJ.

<sup>&</sup>lt;sup>3</sup>Here, and throughout this thesis, we adapt the others' notation for the sake of consistency.

At runtime, objects are represented by a **new** expression, where all parameters to the constructor are values (in FJ, values are only objects), **new**  $C(\overline{v})$ . Since FJ is a functional calculus, there is no need for a stack or heap.

#### 2.1.3 Type Safety

A programming language is type safe (aka type sound) if well-typed programs cannot go wrong. For formal languages this usually means that well-typed programs cannot *get stuck*. Being in a stuck state means that a program cannot be further reduced, but is not a final value of the language.

In Java and other object-oriented languages, type safety is taken to mean that only fields or methods that exist will be accessed; there will be no 'method not found' errors. In FJ, the standard definition of type safety implies the 'Java' meaning because the operational semantics demands that fields or methods exist; a program that involves accessing a field or method that doesn't exist would get stuck — none of the reduction rules could be applied.

Type safety is usually proved by proving *progress* and *preservation* properties [96]. Progress means that any well-typed term is not stuck, that is, can be further reduced or is a value; formally (we use  $\rightarrow$  to mean "reduces to"):

$$\mathbf{e}:\mathbf{T} \Rightarrow (\exists \mathbf{e}'st \ \mathbf{e} \rightsquigarrow \mathbf{e}') \lor (\exists \mathbf{v} \ st \ \mathbf{e} = \mathbf{v})$$

Preservation means that reducing a well-typed term results in a well-typed term; usually preservation states that the type of the resulting term is the same as, or a subtype of, the type of the original term; formally:

$$e: T \land e \rightsquigarrow e' \Rightarrow (e': T') \land (T' <: T)$$

Preservation is also known as *subject reduction* and is usually the more interesting property to prove. Preservation can be proved by structural induction on the derivation of the type checking judgement (e:T) or of the reduction ( $e \rightsquigarrow e'$ ); the latter is more common and is used

in this thesis. Inversion lemmas are used to relate the premises of type rules to the conclusion of the typing derivations of e, e', and sub-expressions, these must take into account subsumption if it is used in the system. Featherweight Java and similar languages (including those in this thesis) make heavy use of substitution, thus substitution lemmas are necessary and widely used in preservation proofs.

### 2.2 Parametric Types

Suppose a programmer wishes to write a simple container class; first let us consider an untyped version, using \* for the type of the object in the container:

```
class Box {
    * datum;
    * get() {
        return datum;
    }
    void set(* x) {
        datum = x;
    }
}
```

In early versions of Java, the programmer could write a different class for each type, for example a ShapeBox, where \* is replaced by Shape, StringBox, IntegerBox, etc. However, there are then many copies of nearly identical code with no reuse. This is bad software engineering practice. Alternatively, the programmer could use Object instead of \*, as in the Java libraries. This enables reuse, but forces the programmer to use a dynamic cast whenever an item is removed from the list. This can result in performance degradation and in runtime errors (ClassCastException in Java), compromising the static safety of the language.

In a parametrically polymorphic type system, type variables may be used in the definition of a type; such a polymorphic type is instantiated by providing actual type parameters. This enables reuse without requiring casts. In section 2.2.1 we describe generics, the implementation of parametric polymorphism in Java and C#. This is followed by some discussion of including subtype variance in generic systems, including *virtual types*, an alternative to parametric polymorphism, in section 2.2.2.

#### **2.2.1** Java Generics

In Java, parametric polymorphism is implemented by *generics* [14, 16, 47]. Classes, types, and methods may be parameterised by formal type parameters; actual type parameters are provided when a class is instantiated or a method is called. Writing the box example using Java generics gives us:

```
class Box<X> {
    X datum;
    X get() {
        return datum;
    }
    void set(X x) {
        datum = x;
    }
}
```

Box is a generic class and X is its formal type parameter. The box can be instantiated to be a box of shapes (Box<Shape>) or a box of strings (Box<String>), and so forth; Shape and String are actual type parameters.

By using generics, the programmer only needs to write a class once, it can be instantiated with many types. In each instantiation it is type safe. For example, in the case of Box<Shape>, the programmer can only call set with objects of type Shape (or a subtype), and is returned an object of type Shape by get. Thus, no casts are necessary.

Generics are useful for more than just collections such as boxes and lists; wherever a class involves managing objects of a given type, then type safety and re-usability can be improved by parameterising by the managed type. For example, class Class in the Java reflection API is (since Java 5.0) a generic class, parameterised by the class that it reflects.

Just as classes may be parameterised, so can methods. For example, a method, first, that extracts the first element from a list, can be parameterised by the type of the list:

```
<Y> Y first(List<Y> y) {
    ...
}
void m(List<Shape> x)) {
    Shape s1 = this.<Shape>first(x); //1
    Shape s2 = this.first<x>; //2
}
```

Here, Y is the formal type parameter of the method first. In call 1, the actual type parameter is Shape, first will expect a parameter of type List<Shape> and return an object with type Shape. Java may infer certain type parameters in a method call; in call 2 no actual type parameter is given, but Java will infer the parameter Shape from the types of x and y.

Classes and methods may have any number of type parameters, for example:

```
class Pair<X, Y> {
    X datum1;
    Y datum2;
    ...
}
```

This pair class has two formal type parameters. It may be instantiated as Pair<Shape, String> (a pair of a shape and a string) or Pair<Shape, Shape> (a pair of two shapes), and so forth.

Generics in C# [1] are very similar to those described here for Java. The major difference is in implementation; Java generics are implemented by *erasure*, all type parameters are erased during compilation. Type parameters in C# are preserved at runtime. The advantage of the C# approach is that operations that use types at runtime (such as casts or **instanceof** tests) have access to the whole type rather than just the class name, and where dynamic checks are required, these can check type parameters. Such a scheme was not adopted in Java since it does not preserve backward compatibility. Parametric polymorphism is supported in C++ by templates [2, 86]. These have a similar syntax to Java generics, but are very different. A C++ template, in effect, works by macro expansion; it is compiled into a new class for each actual type parameter. This can lead to an explosion in the number of classes, and therefore space requirements, and means that templates cannot be type checked until they are instantiated. C++ does not support bounds on type variables.

**Inheritance and subtyping** When we wish to inherit from a generic class we must provide actual type parameters; these may include the formal type parameters of the inheriting class. Thus, the following declarations are all legal.

class List<X> extends Object ...
class SpecialBox<X> extends Box<X> ...
class ShapeBox extends Box<Shape> ...
class DoublePair<X> extends Pair<X, X> ...

The above class declarations give rise to the following subtype relationships, by substituting the actual type parameters of the subclass into the superclass declaration.

List<Shape> <: Object</li>
SpecialBox<Shape> <: Box<Shape>
ShapeBox <: Box<Shape>
DoublePair<Shape> <: Pair<Shape, Shape>

Formal type parameters may be given upper bounds. Actual parameters must be subtypes of their bounds and formal parameters are assumed to be subtypes of their bounds. For example,

```
<X extends Shape> void render{List<X> 1} {
    for (X x : List<X>) {
        x.draw();
    }
}
```

X has the upper bound Shape; valid actual parameters include Square and Triangle, but not Object or Number (assuming the obvious class hierarchy). Within the method, X is a subtype
of Shape, we may call the draw method on elements of 1, since the type of the element must

be a subtype of Shape.

The bound of a type variable may be another type variable, as long as it is not a forward reference, class C<X extends Shape, Y extends X> is legal in Java, but class C<X extends Y, Y extends Shape> is not. A type variable may occur as a *parameter* of the bound of any type variable (subject to the usual scoping rules); thus,

class C<X extends Shape, Y extends List<X>> ...
class C<X extends List<Y>, Y extends Shape> ...
class C<X extends List<X>> ...

are all legal declarations.

**F-bounded polymorphism** *F-bounded quantification* differs from simply bounded quantification by binding the type variable in its bound as well as the scope of the quantification [25].

Java generics support F-bounded polymorphism [16]. Type parameters may appear recursively or mutually recursively in bounds. F-bounds are useful for modelling families of related classes. For example, to give parametric types to the Observer pattern [44], we must use F-bounded quantification of type variables as shown in figure 2.1

```
class Subject<S extends Subject<S, 0>, 0 extends Observer<S, 0>> {
    private final List<0> observers = new ArrayList<0>();
    void subscribe(0 ob) { observers.add(ob); }
    void update() {
        for (0 ob : observers)
            obs.notify(this);
    }
}
class Observer<S extends Subject<S, 0>, 0 extends Observer<S, 0>> {
    void notify(S sub);
}
class Button extends Subject<Button, EventManager> ...
class EventManager extends Observer<Button, EventManager> ...
```

Figure 2.1: Observer pattern in Java.

Type variables in the classes Subject and Observer are F-bounded, S and O appear in their own upper bounds; the classes Button and EventManager show how the classes are used and type

variables instantiated. F-bounds are necessary because the two classes are mutually recursive, we wish to refer to the Subject type from the Observer class and vice versa.

To show why this necessitates F-bounded quantification we will step through some simpler designs and show how they fail: we begin by not using parametric types at all — why not use notify(Subject sub) in Observer? This design does not work because there may be many subclasses of Subject, some of which are incompatible with certain observers.

We could try to parameterise each class with a single type variable, representing the 'opposite' class, i.e., class Observer<S>...; we could then use S as the parameter type in notify. But, this is not strong enough, we should give S a bound — class Observer<S extends Subject>.... However, we must do the same in the declaration of Subject, and now we must give a parameter to the bound of S — class Observer<S extends Subject<O>>.... Here, O is intuitively the type of this, but no such construct exists in Java and so we must use a type variable. Furthermore, we know that this type variable must be bounded by Observer and so we get class Observer<S extends Subject<O>, O extends Observer<...> Meanwhile, we must extend the signature of Subject in the same manner, and we end up with class Observer<S extends Subject<S, O>, O extends Observer<S, O>>..., both type parameters have f-bounds.

#### Invariant Subtyping

Subtyping between generic types is *invariant* with respect to their parameters. Generic types are only subtypes if their parameters are identical (up to the substitution of actual for formal type parameters due to subclassing). For example, List<Circle> is not a subtype of List<Shape>, even though Circle is a subtype of Shape.

Invariant subtyping is required to maintain type soundness. To see why covariant subtyping is, in general, unsound, consider the following example that uses the class declaration from section 2.1 (a similar argument can be made for contravariant subtyping):

Covariant subtyping *is* legal for Java arrays, Circle[] (an array of Circles) is a subtype of Shape[]. In order to prevent errors, a dynamic check is inserted whenever the programmer assigns into an array. Although this means that no non-existent field or method will be accessed, *static* type safety is lost. For example,

Circle[] arrayC = new Circle[4]; Shape[] arrayS = arrayC; //OK arrayS[0] = new Shape(); //statically OK, but caught with a dynamic check

#### Featherweight Generic Java

Featherweight Generic Java (FGJ) [53] extends Featherweight Java (see section 2.1.2) with generics. Classes and methods are parameterised by formal type parameters. Actual type parameters are given when a type is instantiated or a method is called. FGJ lacks type parameter inference (found in Java), all type parameters must be explicitly stated.

Typing in FGJ has the form  $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T}$  (we adjust the notation very slightly for uniformity with the other systems in this thesis);  $\mathbf{e}$  is an expression,  $\mathbf{T}$  a type,  $\Gamma$  is a variable environment which maps variables ( $\mathbf{x}$ ) to their types ( $\mathbf{T}$ ),  $\Delta$  is a type environment which maps type variables ( $\mathbf{X}$ ) to their upper bound (type variables cannot be bounded by other type variables in FGJ).

Subtyping has the form  $\Delta \vdash T <: T'$  and is reflexive and transitive. A type variable is a subtype of its upper bound, given by  $\Delta$ . Subclassing gives subtyping, given by the subclass hierarchy and substitution of type variables. The subclassing rule in FGJ is<sup>4</sup>

 $<sup>^4 \</sup>mathrm{we}$  use  $\lhd$  as a shorthand for <code>extends</code>

$$\frac{\text{class } \mathbb{C} \langle \overline{X} \triangleleft \ \mathbb{N} \rangle \ \triangleleft \ \mathbb{N} \ \{\dots\}}{\Delta \vdash \mathbb{C} \langle \overline{T} \rangle <: [T/X] \mathbb{N}}$$
(FGJ-S-CLASS)

Like FJ, FGJ is functional and so there is no need for a stack or heap in the dynamic semantics. Any variables or type variables will have been substituted away before an expression is reduced.

# 2.2.2 Variant Subtyping

We have seen that in Java with generics, subtyping is invariant (section 2.2.1). Although in general, invariant subtyping is required to maintain soundness, it is sometimes safe and desirable to have variant subtyping.

Given that A is a subtype of B, subtyping is covariant if C<A> is a subtype of C<B>, contravariant if C<B> is a subtype of C<A>, and bivariant if both results can be derived.

There are two categories of systems that support variance: those that support *use-site variance* and those that support *declaration-site variance*. Declaration-site variance is the older of the two variations; the variance of a type is specified when the class is declared; all instances of the class have the same variance properties. Use-site variance allows the variance properties of a type to be specified when the type is instantiated. Different instantiations of a single generic type may have different variance properties.

In the next few sections, we describe some systems that support different forms of subtype variance. The solution in Java is to use wildcards, a form of use-site variance that supports co-, contra-, and bivariance; wildcards are covered in section 2.4.

#### **Declaration-Site Variance Annotations**

Formal type parameters may be decorated with variance annotations, this gives declarationsite variance [8]. Usually, + denotes covariance in the annotated parameter and – denotes contravariance. For example, we could write a covariant box as:

```
class CoBox<+X> {
    readonly X datum;
    X get() {
        return datum;
    }
}
```

We can instantiate CoBox<Square> and, thanks to the variance annotations, it is a subtype of CoBox<Shape>. However, it would be unsafe for this class to include a set method, or allow field assignment to datum. In this way type safety is maintained. If a type parameter is covariant, it cannot appear in contravariant position (i.e., method parameter types or non-read-only fields); if it is contravariant, it cannot appear in covariant position (i.e., in method return types or fields).

Declaration-site variance annotations are found in several programming languages including POOL [8], Smalltalk [15], the CLR [43] (and C# from version 4.0), and Scala<sup>5</sup> [3, 71]. They have been proposed for Eiffel [35] and C# [42].

Reasoning about variance is often a complex and difficult task, and the declaration-site approach has the advantage that it is the library designer (usually a more expert programmer), rather than the client, that must consider the variance of types. On the other hand, Declaration-site variance is less flexible than use-site variance [42, 54]; once a class has been written it is not possible to change its variance properties without changing the class. Furthermore, it tends to lead to duplication of classes and interfaces, since a programmer often has to write co-, contra-, and invariant interfaces for each generic class. This gets even worse if there are multiple type parameters, the number of interfaces required to cover all the different combinations of variance rises exponentially with the number of parameters.

**Variance and Generalised Constraints** Generics in C# are invariant (as of version 2.0<sup>6</sup>); declaration-site variance with constraints on type variables [38] has been proposed [42] as an extension. The variance annotations part of the extension follows the description given above.

<sup>&</sup>lt;sup>5</sup>Scala also supports virtual types (see section 2.2.2) which support subtype variance; in fact, Scala's variance annotations are treated as syntactic sugar for virtual types [71].

<sup>&</sup>lt;sup>6</sup>Declaration-site variance is rumoured to be present in the forthcoming version 4.0

In addition, constraints involving type parameters may be placed on methods and classes. These constraints declare a subtype relation involving formal type variables. For example, the constraint Shape  $\leq X$  constraints X to be a supertype of Shape. If a method is constrained, then the constraint only has to hold in order to call that specific method. Constraints may contain other type variables and may be recursive and mutually recursive.

Generalised constraints make declaration-site variance more flexible. For example [42], a covariant list cannot usually allow elements to be set or appended, since the element type of the list must appear in contravariant position in these methods. Using generalised constraints, a functional<sup>7</sup> append method can be declared that is type safe and can be used on a covariant list. A list class declared as CoList<+X> may have an append method with signature CoList<Y> append<Y>(Y datum)  $\{\ldots\}$  where  $X \leq Y$ . The constraint in the signature means that Y must be a supertype of the element type of the list, X. The method returns a new list with type CoList<Y> with the element datum appended. For example, we could append a Shape to a CoList<Square> resulting in a CoList<Shape>. This is safe since each element in CoList<Square> is at least a Square, and, therefore, at least a Shape, as is the element we append.

#### Existential Types in Pizza

Pizza [70] is a precursor of GJ [16] and Java generics. Parametric types in Pizza are very similar to those in Java and GJ. The primary difference<sup>8</sup>, is that Pizza allows more type parameter inference than Java. To support inference of type parameters, Pizza supports existential types. To see why existential types are required, consider the following example [70]:

```
List<String> l = ...;
Object o = l;
int s = ((List)o).size{};
```

<sup>&</sup>lt;sup>7</sup>That is, the method returns a copy of its input with an appended element, rather than appending directly to the input.

<sup>&</sup>lt;sup>8</sup>Pizza also supports higher order functions and algebraic types.

The interesting part of the example is the cast. In Pizza, the programmer does not need to give the parameters of a type in a cast — only List is required, not List<String>, the type parameter is inferred. But, in order to infer List<String>, the compiler would have to check that every element of the list is a String. Instead, the compiler gives the result of the cast the existential type  $\exists X.List<X>$  which signifies that the object is a list of some type, but exactly which type is unknown.

Existential types are used only in type checking and cannot be written by the programmer. Pizza existential types are bivariant types, albeit types to which the programmer does not have access. We discuss how existential types are type checked in Pizza in section 2.3.7.

#### **Raw Types**

A raw type in Java 5.0 [47, 14, 16, 52] is a generic type without type parameters, for example Box. Raw types facilitate interaction between generic and non-generic (in particular, legacy) code. In the box example, the get and set methods can be called as if the parameter and return types are Object. However, unlike Box<Object>, raw types enjoy subtype variance, Box<Shape> is a subtype of Box. In fact, any parameterised box is a subtype of the raw box; in this way, raw types behave like unbounded wildcard types (see section 2.4). Unlike wildcard types, Box is a subtype of Box<Shape> or any other parameterisation. This is not type safe and so a dynamic check is inserted to prevent runtime errors and a compile-time warning is given (this would be an error in the case of a wildcard type). The language designers thought that this flexibility, for the sake of backward compatibility, was more important than strict static type safety [14].

#### Virtual Types

Virtual types [51, 61, 87, 89] are an alternative to parametric polymorphism. A virtual types system allows a class to have type members (as well as fields and methods), and these may be overridden (thus, they are virtual, in the C++ sense). This gives inherent covariance. We

demonstrate how to model our Box example using virtual types in figure 2.2.

```
class Box {
                        //a virtual type X
    type X;
    X datum;
    X get() { return datum; }
    void set(X x) { datum = x; }
}
class ShapeBox extends Box{
    type X < Shape;
                        //X is restricted,
                        //Shape is an upper bound on X
}
class SquareBox extends Box {
    type X = Square;
                        //X is bound to Square
}
```

Figure 2.2: The Box example using virtual types.

The syntax type X in class Box introduces the (unbounded) type member X to a class. The type member X is inherited in ShapeBox, it is *restricted* to be a subtype of Shape by type X < Shape. It is also inherited in SquareBox, but in this case it is *bound* to Square. Due to subclassing, SquareBox is a subtype of ShapeBox, this corresponds to the covariant relationship, Box<Square> is a subtype of Box<Shape>.

Virtual types have type safety issues; they are affected by the same problems that afflict arrays in Java (see section 2.2.1). There have been several proposed solutions, some systems insert dynamic checks [61, 87], others restrict instantiation of classes to those with bound type members [51, 89] (in our example, SquareBox could be instantiated, but ShapeBox and Box could not), and some distinguish between exact and variant types [17].

Virtual types are more verbose than parametric types when modelling collection classes. However, they have advantages for modelling families of classes. For example, the Observer pattern [44] can be concisely described without the complicated F-bounds necessary for the parametric version given in section 2.2.1, this is shown in figure 2.3. In this example, we assume that there is a List class defined using virtual types in the manner of the Box class above. Note that OList is an inner class that uses a virtual type, it is not a virtual class [61, 41].

Structural virtual types [88] are an extension to virtual types that combine the advantages of virtual types and parametric polymorphism. The type members of a class are extracted into a

```
class Subject {
    type O;
    private final OList observers = new ObList();
    void subscribe(O ob) { observers.add(ob); }
    void update() {
        for (O ob : observers)
            obs.notify(this);
    }
    class OList extends List{
        type X = 0;
    }
}
class Observer {
    type S;
    void notify(S sub);
}
class Button extends Subject {
    type 0 = EventManager;
    . . .
}
class EventManager extends Observer {
    type S = Button;
    . . .
}
```

Figure 2.3: The Observer pattern implemented using virtual types.

separate block (delimited with square brackets) and contribute to subtyping. For example, we can rewrite the above box example using structural virtual types as:

```
class Box [X] {
    X datum;
    X get() { return datum; }
    void set(X x) { datum = x; }
}
```

A box of squares is represented as ShapeBox[X = Square], ShapeBox[X < Shape] restricts X to a subtype of Shape. Subtyping follows from restriction of bounds (Box[X < Square] is a subtype of Box[X < Shape]) and binding (Box[X = Square] is a subtype of Box[X < Shape]).

Type members are bound or restricted where types are instantiated. Structural virtual types thus support use-site variance (in contrast to regular virtual types). Structural virtual types are statically safe, unsafe method calls and field assignments are forbidden (such as calling set on an object with type ShapeBox[T < Shape]).

The authors of [88] note that structural subtyping of type parameters could be used in a parametrically polymorphic language to support use-site variance (without the modelling advantages of virtual types). This inspired *variant parametric types* [54], the first rigorously investigated system to support use-site variance.

#### Variant Parametric Types

Variant parametric types [54] support covariance, contravariance, and bivariance in a systematic and safe way. In comparison to structural virtual types, variant parametric types offer support for contravariance and a syntax more natural to parametric types. Variant parametric types evolved into Java wildcards; a comparison between the two systems is made in section 2.4.5.

Variant parametric types extend Java generics by allowing actual type parameters to be annotated with symbols to denote variance: +T denotes a covariant type parameter, i.e., T or a subtype of T; -T denotes a contravariant type parameter, i.e., T or a supertype of T; \*T (or just \*, since in this case T is unimportant) denotes a bivariant type parameter, i.e., any type. The formal grammar of variant parametric types is ( $\circ$  is used to annotate an invariant type parameter):

v::=
$$\circ | + | - | *$$
variance annotationsN::= $C < vT >$ class typesT::=X | Ntypes

Using a variance annotation restricts the members of a class that are available. The user of a variant parametric type cannot access members where a covariant type parameter appears in contravariant position (i.e., in method parameter types); nor members where a contravariant type parameter appears in covariant position (i.e., in method return types and field types). If a type parameter is bivariant, then a member in which it appears cannot be accessed. For example, given the generic Box class from section 2.2.1, a variable of type Box<+Shape>, has

datum and get() available (both would have type Shape), but not set(). A variable of type Box<-Shape> only has set() available and nothing can be accessed from a variable of type Box<\*>. More usefully, in a List<\*> the size() method can be accessed because it takes no parameters and returns an int. Intuitively, member restriction means that for any generic class C<X>, instances of C<+T> are read-only with respect to X, and instances of C<-T> are write-only.

Variant parametric types support variant subtyping: type parameters marked + permit covariant subtypes, those marked – permit contravariant subtypes, and \* permits any parameterisation as a subtype. The following are all legal (these subtype relationships can be represented diagrammatically, see figure 2.4):

- ⊢ Box<Shape> <: Box<+Shape>
- $\vdash$  Box<Shape> <: Box<-Shape>
- ⊢ Box<Square> <: Box<+Shape>
- ⊢ Box<Object> <: Box<-Shape>
- $\vdash$  Box<+Square> <: Box<+Shape>
- ⊢ Box<-Object> <: Box<-Shape>
- $\vdash$  Box<Square> <: Box<\*>
- $\vdash$  Box<Shape> <: Box<\*>
- ⊢ Box<Object> <: Box<\*>

Variant parametric types are never subtypes of invariant parametric types (i.e., regular generic types). So, for example, Box<+Shape> is not a subtype of Box<Shape>. Variance annotations cannot be mixed across subtypes, so Box<-A> and Box<+B> will never be subtypes, in either direction, for any A and B. This principle extends to bivariant types, Box<+A> and Box<-A> are not subtypes of Box<\*> for any A.

To see why the member access restrictions guarantee soundness, we look in detail at some of the subtype relationships given above. By subsumption, an object with type Box<+Shape> may have type Box<Square>, in this case calling get and returning a Shape is safe because Square



Figure 2.4: Subtype relationships between variant parametric types.

is a subtype of Shape. However, calling set could be unsafe if called with a Shape parameter. Likewise, a variable with type Box<-Shape> may contain an object with type Box<Object>. Calling set(new Shape()) is safe because Shape is a subtype of Object, but it is unsafe to call get and expect a Shape, since our box may contain objects with any type.

By annotating actual type parameters, variant parametric types allow the variance of a container to be decided when it is instantiated. Different instantiations of a generic class may be invariant, covariant, etc. This means that reusability is increased and the user can decide on the trade-off between flexibility of typing and availability of members that is entailed by the different kinds of variance. For example, the single definition of the box class (section 2.2.1) is enough to generate in-, co-, contra-, and bivariant types (for example Box<Shape>, Box<+Shape>, Box<-Shape>, Box<\*>), this would require four class definitions using declaration-site variation.

Variant parametric types have been formalised as an extension to FGJ; this formalisation is influenced by work on existential types and is discussed in section 2.3.7. Variant parametric types have also been formalised using flow analysis [28]; this approach avoids existential types.

# 2.3 Existential Types

Existential types are a form of polymorphic type, they are based on the existential quantification of logic. Their principal use is in modelling abstraction. There have been many different formulations of existential types [26, 27, 49, 64, 74, 75]; although the concepts are similar, they often differ in the details of their syntax and formalisation.

Existential types were first investigated by Girard [46] in the context of proof theory in logic. Girard discovered a system called System F (also independently discovered by Reynolds as a type system [80]), a polymorphic extension of the typed lambda calculus. Existential types have been investigated as an extension to System F [27], and can be encoded within it [74, 45].

Abstract types (synonymously, abstract data types) support abstraction and modularisation in the type system of a programming language. An abstract type declaration binds an abstract interface to an implementation. Abstraction is provided by limiting access to the implementation; only the operations present in the interface may be used outside the declaration. Mitchell and Plotkin [64] discovered that abstract types in programming languages could be given existentially quantified types, taken from logic. Existential types have also been used to model modules [27] and objects [75]. More recently, they have been used for subtype variance in parametrically polymorphic object-oriented languages. [54, 60]

The first programming language with existential types was SOL [64], a notational variant of Girard's System F, and an extension of the polymorphic lambda calculus. To support existential types, SOL extends the lambda calculus with pack and abstype (unpack) expressions.

# 2.3.1 A Language with Existential Types — Fun

We will describe Fun [27], an adaption and extension of SOL and the foundation for nearly all existential type systems developed since. We prefer to describe Fun rather than SOL because it is more typical of existential types systems and is conceptually clearer.

In Fun, a type specification for an expression with existential type has the form  $\mathbf{e} : \exists \mathbf{X} . t(\mathbf{X})$ , where  $\mathbf{e}$  is an expression,  $\mathbf{X}$  is a type variable, and  $t(\mathbf{X})$  is a type expression. This may be read as "there exists some type  $\mathbf{X}$ , such that  $\mathbf{e}$  has type  $t(\mathbf{X})$ ". In general, we use upper case characters (or capitalised strings) for types and lower case for values,  $\exists$  for existential quantification, a full stop ('.') to separate the quantifying from the quantified parts of a type,  $\rightarrow$  for function types,  $\langle$  and  $\rangle$  to mark a pair or tuple, and  $\times$  to denote pair or tuple types. Relations and judgements always have the highest precedence, so, for example,  $\Delta \vdash \exists \Delta' . \mathbf{T} <: \mathbf{T}'$  could be written explicitly as  $\Delta \vdash (\exists \Delta' . \mathbf{T}) <: \mathbf{T}'$ .

The simplest example of an existential type is  $\exists X . X$ , where for any expression, there is a type, X, such that the expression can be given the type X. However, variables with such a type can only be passed around and cannot be used, because, in effect, we have no information about them. More complicated existential types allow the programmer to perform some operations on variables. For example, a value with type  $\exists X . X \times X \rightarrow int$  (with full bracketing:  $\exists X . (X \times (X \rightarrow int)))$  is a pair of a variable with some type, and a function that takes a parameter of that type and returns an integer; we can therefore apply the function to the variable and obtain an integer. The syntax for the existential types we use in this section is:

R	::=	$X \mid T {\rightarrow} T \mid \texttt{int} \mid T {\times} T$	non-existential types
Т	::=	$R \mid \exists X.R$	unbounded types
T'	::=	$R \mid \exists X \leq T.R$	bounded types

In an existential type  $\exists X.T$ , the type variable X is bound in T. Thus, X can be renamed in T (*alpha conversion*);  $\exists X.T$  is considered equivalent to  $\exists Y.[Y/X]T$ . As with lambda binding, Barendregt's variable convention [10] may be used with existential quantification; that is, we may assume that bound and free type variables are distinct.

**Packing and unpacking** In traditional treatments [27, 64, 74], existential types are introduced and eliminated explicitly in the expression syntax. An existential type is introduced using a *pack* expression (also known as a *close* expression); in Fun, this takes the form pack[X = T in t(X)](e), where e is the expression to be packed, T is the *witness type* (the type *hidden* by X), and t(X) is a type expression that is the *interface* of the *package* created as a result of the pack expression. For example, the pair  $\langle 2,3 \rangle$  with type  $int \times int$  can be abstracted to pack[X = int in X×X]( $\langle 2,3 \rangle$ ) with type  $\exists X.X \times X$ ; int is the witness type and is hidden by X<sup>9</sup>.

A value with existential type must be *unpack*ed (or *open*ed) before it can be used; the syntax **open p as x in e** is used in Fun, where **p** is a package with existential type, **x** is a label for the value portion of the contents of the package, and **e** is the scope in which **x** can be used. For example, **open p as x in (fst x)**<sup>10</sup>, where **p** has type  $\exists X.X \times X$ , takes the first value in the tuple contained in **p**<sup>11</sup>.

Continuing the  $\exists X.X \times X \rightarrow int$  example; given a float, x, and a function, round of type float $\rightarrow$ int which takes a float and returns the nearest integer, we can create a package, p, with type  $\exists X.X \times X \rightarrow int$ , by packing a tuple of these entities:

 $p = pack[X = float in X \times X \rightarrow int](\langle x, round \rangle)$ 

In order to use p we must unpack it, for example, open p as x in ((snd x) (fst x)). From the structure of the existential type, we can see that the second element of p is a function, with result type int, that can be applied to the first element; thus, the whole expression will have type int.

**Bounded quantification** Unbounded existential types (as described above) hide all information about the witness type. *Bounded* existential types [27, 73] give *partial* information

<sup>&</sup>lt;sup>9</sup>In previous work [24] and in the later chapters of this thesis, we adopt the syntax close e with X hiding T. The only significant difference in our approach is that we do not specify an interface type (t(X)). We can derive this type from the type of e, t(X) = [T/X]U, where U is the type of e. The Fun approach is a little more flexible, in that it allows for the 'partial hiding' of types. For example, the Fun expression pack[X = T in X×T] (e) can take an expression (e) with type T×T and pack it to a package with the type  $\exists X.X \times T$ ; only one of the occurrences of T is hidden. This is not possible with our notation, but such a facility is not required to model Java wildcards.

<sup>&</sup>lt;sup>10</sup>This expression does not actually type check according to the standard rules. It would have type X, except that this type variable escapes the scope of unpacking. A pack expression can be used to make the expression type check: open p as x in (pack [X=X in X](fst x))

<sup>&</sup>lt;sup>11</sup>In the later chapters of this thesis and in previous work [24], we also give a label to the witness type in the scope of the unpack expression (since we may have to refer to the type in the program syntax, for example new C<X> or this.<X>m(x); this cannot happen in Fun), thus we use the syntax open p as x,X in e. As we will see in section 2.3.2, both the witness type (T in pack[X = T in X×T](e)) and the type variable that hides it (X) must be known in the reduction rules. In Fun, both are given in the pack expression; in our notation, X is given in the open expression, and T in the close expression.

about the witness type. Early work on existential types allowed the quantified type variable to have an upper bound [27], work on wildcards also requires the use of lower bounds [60]. A bounded existential type in Fun has the form  $\exists X \leq T.t(X)$ , where T is an upper bound. Any type that can be hidden by X must be a subtype of T; therefore it is safe to make use of the assumption that X is a subtype of T.

## 2.3.2 Static and Dynamic Semantics of Existential Types

In this section we show how programs using existential types are type checked and evaluated. We discuss the type rules of Fun without bounds; adding bound checking to these rules is fairly straightforward; the bound checks are only interesting in subtyping. Since no dynamic semantics have been given for Fun [27], we adapt them from System  $F_{<:}$  [74].

#### Typing

$$\frac{\Delta; \Gamma \vdash e : [T/X]U}{\Delta; \Gamma \vdash pack[X = T in U](e) : \exists X.U} \qquad \begin{array}{c} \Delta; \Gamma \vdash e : \exists X.U & \Delta, X; \Gamma, x: U \vdash e' : T \\ & X \notin fv(T) \\ \hline \Delta; \Gamma \vdash open \ e \ as \ x \ in \ e' : T \\ & (Fun-T-Open) \end{array}$$

In these rules,  $\Gamma$  maps variables to types and  $\Delta$  is a list of type variables. The function fv(T) returns those type variables that appear free in type T.

A pack expression wraps a subexpression (e) to an abstract package. Its type is the type of e with the witness type (T) hidden by a type variable, X. The premise of the rule gives that e has the interface type U without T being hidden.

An open expression takes an abstract package (e) and unpacks it in the scope of the body of the expression (e'). This is done by using a fresh variable (x) to hold the unpacked package. The quantifying type variable from the type of e must be remembered when judging e', since it may appear free in the type of x. The open expression is given the same type as e', the third premise ensures that the unpacked type variable cannot escape its scope.

## Subtyping

There are two variations on subtyping between existential types, these are known as the *kernel* and *full* variants [74] (in these rules,  $\Delta$  maps type variables to their bounds):

$$\frac{\Delta, X \leq U \vdash T <: T'}{\Delta \vdash \exists X \leq U. T <: \exists X \leq U. T'} \qquad \qquad \frac{\Delta \vdash U <: U' \quad \Delta, X \leq U \vdash T <: T'}{\Delta \vdash \exists X \leq U. T <: \exists X \leq U'. T'}$$
(Fun-S-Kernel) (Fun-S-Full)

To compare the bodies of existential types, which may include type variables bound by quantification, these rules move the quantifying variables into the judging environment. This requires quantifying parts of the types to be related: in the kernel variant they must be identical, while in the more flexible full variant, the type variables must be identical, but their bounds may be subtypes. The more precise bound (U) is used to extend the type environment in the second premise. As we will see in section 3.3.2, the full variant of subtyping corresponds most closely to subtyping in Java with wildcards.

### **Operational Semantics**

Pack and unpack expressions may be reduced together to eliminate both expressions (we use  $e \rightarrow e'$  to mean e reduces in one step to e'):

open (pack[X = T in U](v)) as x in 
$$e \rightarrow [v/x, T/X]e$$
  
(FUN-R-OPEN-PACK)

The pack sub-expression packs v creating an abstract package where the witness type Y is hidden by X, the open expression unpacks this package, re-exposing v and T within the scope of the body of the open expression, e. Within e, the unpacked package is referred to by the variable  $\mathbf{x}$ . The reduction rule reflects these intuitions by replacing  $\mathbf{x}$  with  $\mathbf{v}$  and  $\mathbf{X}$  with  $\mathbf{T}$  in  $\mathbf{e}$ , allowing it to be reduced further since the free variables due to the open expression have been eliminated.

## 2.3.3 Open existential types

A modern alternative to formulating existential types is to separate packing and unpacking into more atomic operations. In  $F^{Y}$  [65], unpacking is split into *opening* and *scope restriction*. This is done so that defining a scope for an opened existential type variable is separated from the actual opening (existential types are opened into the environment and are known as open existential types). Packing is split into *existential introduction*, *open witness definition*, and *coercion*; this is done to reduce verbosity, by separating the creation of types from the abstraction of expression types, the witness type can be specified more succinctly. Existential introduction creates an existential type, open witness definition specifies which part of the hidden type are abstracted, and coercion gives a more abstract type to an expression.

The consequence of more atomic operations for the introduction and elimination of existential types is that module systems can be more closely modelled. By deconstructing unpacking, abstract types can be made available at any scope or freely throughout a program, this fits more closely to how modules are used. Open witness definition corresponds closely to type abstraction of modules. It would be interesting future work to examine how (and if) the finer grained operations of  $F^{\gamma}$  correspond to wildcard types in Java.

# 2.3.4 Decidability

Type checking in the full variant of System  $F_{<:}$  is undecidable [73] because subtyping is undecidable. This is shown by reducing the halting problem for two counter Turing Machines to subtype checking. However, subtyping is decidable in the kernel variant of System  $F_{<:}$  [73]. Several other restrictions of full System  $F_{<:}$  also make subtyping decidable. These results extend to System  $F_{<:}$  extended with existential types [74]: the kernel variant is decidable, the full variant is not. Furthermore, there is only a minimal typing<sup>12</sup> algorithm for the kernel variant of System  $F_{<:}$  with existential types. It has been shown that no algorithm can give a minimal type to **open** expressions in the full variant of System  $F_{<:}$  [45].

In the next few sections we discuss some of the applications of existential types.

# 2.3.5 Modelling Abstract Types

We use a stack of integers to show how existential types model abstract types. We wish to model a stack with functions empty (to provide an empty stack), push, and pop, and where the implementation is left abstract; the implementation should not concern clients of the stack. The abstract stack has the type  $\exists X.((empty:X) \times (push:(X \times int) \rightarrow X) \times (pop:X \rightarrow (int \times X)))$ . In this type, X hides the implementation type; the abstract interface consists of a record of functions: empty returns a stack, push takes a stack and an integer and returns a stack, pop takes a stack and returns a stack and an integer.

To create a stack we must provide a concrete implementation and then pack it. Assuming that we have functions lEmpty, lPush, and lPop that implement empty, push, and pop using lists, we create an abstract stack using

To use this stack we must first unpack it, we assume that x has type

 $\exists X.((empty:X) \times (push:(X \times int) \rightarrow X) \times (pop:X \rightarrow (int \times X))))$ 

(we simplify the example by using pattern matching notation in the open expression):

 $<sup>^{12}</sup>$ A minimal type is the most precise type that can be given to an expression. We define "most precise": if A is a subtype of B then A is more precise than B. In languages with forms of polymorphism other than subtyping, there are other definitions of "most precise".

```
open x as ((stEmpty:st)× (stPush:(st×int)\rightarrowst) × (stPop:st\rightarrow(int×st))) in (fst (stPop (stPush (stPush (stEmpty, 3)), 5)))
```

Executing this code will give the result 5 (we take an empty stack and push on 3, then 5, then pop off the top of the stack, which is 5). Note that we unpack the abstract stack as soon as possible, and then use its components several times.

Rather than using explicit unpacking, the *dot calculus* [26] models abstract types more directly. A dot is used to unpack and access elements of an abstract package. This system is fundamentally related to earlier work on existential types [27, 64].

The dot calculus [26] has abstract packages with existential type which are created by a 'pack' expression, similarly to Fun. The code p.f accesses a field, f, in an existentially typed package, p. This system is formalised by introducing two operators on packages, Fst and snd<sup>13</sup>. For a package p = pack[X = T in U] (e) (using the syntax of Fun [27], described above, which varies slightly from that used in [26]), p.Fst gives the witness type of p, X, and p.snd gives the value part, e.

The dot notation calculus can be translated to one that uses open notation and vice versa [26]. Both translations preserve typing and semantics; thus, the two calculi are equally expressive.

A more powerful version of the dot calculus also exists [26] where Fst and snd may be preceded by any existentially typed expression (e.Fst and e.snd), rather than simply variable names. This is motivated by programming language features such as nested modules. The authors show that this extended calculus is more expressive than calculi that use explicit unpacking or the original dot calculus.

*ML Modules* are an alternative abstraction mechanism to abstract types. A module is a selfcontained unit of code with an interface type. ML Modules have also been modelled using existential quantification [82]; this formalisation does not use the standard mechanisms of existential types such as packing and unpacking. A proposed extension [81] to ML uses open

 $<sup>^{13}</sup>$ The slightly odd capitalisation is taken from [26], Fst is capitalised because it returns a type, whereas snd returns a value.

and pack expressions similar to those described earlier. These constructs operate over multiple rather than single type variables, allowing quantification by many type variables at once (as in Tame FJ, section 3.1).

# 2.3.6 Existential Types for Imperative Languages

Existential types and low-level pointer operations can cause problems when mixed. Cyclone [49] is a type safe, C-like language that had such problems in its early development. In this section we describe Cyclone's problem and the solution.

In Cyclone, structs may be existentially quantified. Accessing data in existentially typed structs involves pattern matching, which performs unpacking. The address of a field in a struct can be matched using \*x, where x is a variable into which the address is copied; this is called a *reference pattern*. Thus, if we have a struct S with two fields, the pattern S(a, b) copies the first field to a and the second to b; the pattern S(a, \*b) copies the *address* of the second component to b. If S has the existential type  $\exists X. (void (*f)(int, X), X a)$ , i.e., consists of a function f that takes an integer and a value of some type, and a value of the same type, then in the scope of S(a, \*b), the variable b will contain the address of the second field; it will have type \*Y, where Y is the unpacked witness type.

Since Cyclone is imperative, it allows assignment to variables, including those with existential type. Unfortunately, these features combine to produce an unsoundness. Continuing the example [49],

```
void ignore(int x, int y) {}
void assign(int x, int *y) { *y = x; }
void f(int* ptr) {
    struct S s1 = S(ignore, 0xabcd);
    struct S s2 = S(assign, ptr);
    let S(a,*b)<Y> = s2 in {
        s2 = s1;
        a(37,*b);
    }
}
```

Both s1 and s2 have the same type, but different witness types, namely int in s1 and \*int in s2. As in C, the assignment s2 = s1 copies the values of s2 into s1, this changes the witness type from int to \*int. Since a contains a *copy* of the pointer to assign it is unaffected by the assignment. However, b contains the address of the second field of s2, so when s2 is updated, b will point to the new value, in this case, the integer Oxabcd. When the function pointed to by a is called, it is assign that is executed, but it is passed an integer where it expects an address, thus an arbitrary location in memory can be overwritten.

To fix the problem, either assignment to unpacked existential types or using reference patterns to match existential types must be forbidden [49]. We discuss why this problem does not affect Java in section 3.3.2.

# 2.3.7 Existential Types for Subtype Variance

Recently, existential types have been used to support subtype variance in parametrically polymorphic object oriented languages. Parametric types (such as generic types in Java) must support invariant subtyping (section 2.2.1). Several systems address this restriction (described in sections 2.2.2 and 2.4); in this section we describe how existential types have been used to model these solutions.

#### Pizza

Pizza [70] uses bounded existential types during type checking to support inference of type parameters (see section 2.2.2).

Pizza is formalised as Mini-Pizza [70], a functional calculus that captures the extensions to Java proposed in Pizza. Existential types may not be present in the surface syntax of Mini-Pizza programs, but they may appear during type checking. In Mini-Pizza, many formal variables may be quantified at once. For example, in Fun, we must use one quantifier per variable,  $\exists X. \exists Y. \langle X, Y \rangle$ ; in Mini-Pizza, we can quantify several variables,  $\exists X, Y. Pair < X, Y >$ . The same approach is taken in Wild FJ [60] and section 3 to formalise wildcards. Although, to the best of

our knowledge, this issue has not been discussed in the context of Pizza [70], it had a significant impact on our formalisation of wildcards, and is discussed in section 3.3.

There are no pack or unpack expressions in Mini-Pizza, instead packing and unpacking occurs in the type and subtype rules. Packing is handled by the  $\exists \geq$  subtyping rule, and unpacking by the  $\exists \leq$  subtyping rule and  $\exists$  Elim type rule. We start by discussing the subtyping rules:

$$\frac{\Delta, \Delta' \vdash \mathsf{T} <: \mathsf{T}' \quad dom(\Delta') \cap fv(\mathsf{T}') = \emptyset}{\Delta \vdash \exists \Delta'. \mathsf{T} <: \mathsf{T}'} \qquad \underbrace{\Delta \vdash \mathsf{T} <: [\overline{\mathsf{A}/\mathsf{X}}] \mathsf{T}' \quad \Delta \vdash \overline{\mathsf{A}} <: [\overline{\mathsf{A}/\mathsf{X}}] \mathsf{B}}_{\Delta \vdash \mathsf{T} <: \exists \overline{\mathsf{X} \leq \mathsf{B}}. \mathsf{T}'}$$
(PIZZA-S- $\exists$ - $\leq$ )
(PIZZA-S- $\exists$ - $\geq$ )

T and T' are types, X is a type variable, A and B are non-existential types and  $\Delta$  and  $\Delta'$  are type environments, i.e., mappings from type variables to their upper bounds.

The rule PIZZA-S- $\exists$ - $\leq$  takes an existential type,  $\exists \Delta'.T$ , and finds a supertype, T', of the unquantified part T. To do this,  $\exists \Delta'.T$  is unpacked into  $\Delta'$  and T',  $\Delta'$  is used to judge the subtype relation between T and T'. To avoid free variable escape, T' must not contain variables bound in  $\Delta'$ .

For example, consider  $\exists X \leq Shape.Box < X > judged under \Delta = \emptyset$ ; the first premise allows us to find a supertype of Box < X > under {X \leq Shape}; Shape is such a supertype and it does not contain X; therefore, we can conclude that  $\exists X \leq Shape.Box < X >$  is a subtype of Shape.

PIZZA-S-∃-≥ packs a type T to an existentially quantified supertype  $\exists \overline{X \leq B} . T'$ . In the supertype,  $\overline{X}$  hide the witness types  $\overline{A}$ ;  $\overline{T}$  must, therefore, satisfy the bounds on  $\overline{X}$ .

For example, Box<Circle> is a subtype of  $\exists X \leq Shape.Box<X>$ . The first premise is satisfied by reflexivity because Box<Circle> = [Circle/X]Box<X>. We satisfy the second premise because Circle is a subtype of Shape.

These rules perform the same function, and operate in the same way, as FUN-T-OPEN and FUN-T-PACK in Fun (given in section 2.3.2). The difference is that the Fun rules are type rules and the Pizza rules are subtype rules. Subtyping in Fun does not permit packing or unpacking

of existential types. Subtyping between existential types (given by FUN-S-FULL in Fun) is also given in Pizza by these subtype rules.

PIZZA-S- $\exists$ - $\leq$  compares with FUN-T-OPEN. Both rules unpack a type and allow the unpacked type to be used within a limited scope, and ensure that unpacked variables cannot escape that scope. The first premise of FUN-T-OPEN finds an existential type for an expression  $(\Delta; \Gamma \vdash e : \exists X.U)$ , in Pizza, we start with an existential type. This existential type is separated in the same way in both systems; in Fun, using X with  $\Delta$  to judge the type of a second expression  $(\Delta, X; \Gamma, x: U \vdash e': T)$ ; and in Pizza applying the same concept to types rather than expressions, that is, using subtyping rather than typing. The final premise of each rule is identical, it ensures that no free variables escape via the result type or supertype, T'.

PIZZA-S- $\exists$ - $\geq$  compares with FUN-T-PACK. In Fun, we wrap an expression to pack it, in Pizza we pack a type by finding a supertype; otherwise, the two rules are almost identical. The first premise of PIZZA- $\exists$ - $\leq$  corresponds with the first premise of FUN-T-PACK ( $\Delta$ ;  $\Gamma \vdash e : [T/X]U$ ), except that we have a subtyping, rather than a typing, judgement. We gave FUN-T-PACK in the context of unbounded existential types, the version with bounds includes exactly the second premise of PIZZA-S- $\exists$ - $\leq$ .

The same pattern of packing and unpacking using subtype rules is found in JavaGI [94, 95] (section 2.4.6). The subtyping rules in Pizza are, in effect, combined into the 'env' subtyping rule of Wild FJ [60] (section 2.4.6) and Tame FJ (section 3.1); these two approaches to subtyping are compared in section 5.2.3.

In Pizza, unpacking is also found in the type rule  $\exists$  Elim, again, we adjust the notation for uniformity,

$$\frac{\Delta; \Gamma \vdash \mathbf{e} : \exists \Delta'. \mathsf{T}}{\Delta, \Delta'; \Gamma \vdash \mathbf{e} : \mathsf{T}}$$
(Pizza-T-∃-Elim)

Here,  $\mathbf{e}$  is an expression and  $\Gamma$  is a variable environment, i.e., a mapping from variables to their types. This rule allows an expression with existential type to be used without quantification.

We must add the quantifying environment,  $\Delta'$ , to the judging environment,  $\Delta$ . For example, if  $\Delta; \Gamma \vdash \mathbf{x} : \exists X.Box < X >$ , then we can derive  $\Delta, X; \Gamma \vdash \mathbf{x} : Box < X >$ .

PIZZA-T- $\exists$ -ELIM complements PIZZA-S- $\exists$ - $\leq$ ; PIZZA-S- $\exists$ - $\leq$  allows existential types to be unpacked during subtyping, PIZZA-S- $\exists$ -ELIM allows expressions to be unpacked during typing.

There does not appear to be a corresponding rule to pack expressions in Mini-Pizza and so it is unclear how  $\Delta'$  could ever be removed from a judging environment. For example, consider the derivation of a call to a function with type  $\forall Y.Box < Y > \rightarrow Box < Y >$ , a sketch of a possible derivation (with bounds omitted) is:

$$\frac{\Delta; \Gamma \vdash \mathbf{x} : \exists \mathbf{X} . \mathsf{Box} < \mathbf{X} >}{\Delta, \mathbf{X}; \Gamma \vdash \mathbf{x} : \mathsf{Box} < \mathbf{X} >} \qquad (\mathsf{Pizza-T-\exists-Elim})$$
$$\frac{\Delta, \mathbf{X}; \Gamma \vdash \mathbf{f}(\mathbf{x}) : \mathsf{Box} < \mathbf{X} >}{\Delta, \mathbf{X}; \Gamma \vdash \mathbf{f}(\mathbf{x}) : \mathsf{Box} < \mathbf{X} >} \qquad (\mathsf{Pizza-T-Apply})$$

We would like to be able to pack the resulting type, i.e., be able to derive  $\Delta; \Gamma \vdash f(x) : \exists X.Box < X >$ . Even though  $\Delta, X \vdash Box < X > <: \exists X.Box < X >$  holds, it does not seem to be possible to use this in the typing derivation to remove X from the left of the turnstile.

#### **Raw Types**

Raw types (section 2.2.2) behave like wildcard types, except that some type errors are demoted to warnings. Raw types behave like existentially quantified types [54, 52]. The only formalisation of Raw Types [52] does not use existential types, although they were used to gain an intuition for Raw Types' behaviour. The correspondence between wildcard types and existential types, described in section 2.4.1, also applies to raw types.

#### Variant Parametric Types

Variant parametric types [54] (section 2.2.2) are "existential types in disguise" [54]. Packing and unpacking is used in the formalisation of variant parametric types. The correspondence is very similar to the one between wildcards and existential types<sup>14</sup>, discussed in detail in section 2.4.1. Bivariant parametric types can be thought of as unbounded existential types and covariant parametric types as bounded existential types. For example, Box<\*> corresponds to  $\exists X . Box<X>$ and Box<+Shape> corresponds to  $\exists X \leq Shape.Box<X>$ .

To find a corresponding existential type for contravariant parametric types, we must introduce lower bounds on existentially quantified type variables. To the best of our knowledge, this idea has not been studied outside of the context of variant parametric types and wildcards. The correspondence follows the same pattern as covariant types, for example, Box<-Shape> corresponds to  $\exists X \geq Shape.Box<X>$ .

Subtyping between variant parametric types corresponds to the full variant of existential subtyping (FUN-S-FULL given in section 2.3.2).  $\vdash$  Box<+Square> <: Box<+Shape> corresponds to  $\vdash \exists X \leq$  Square.Box<X> <:  $\exists X \leq$  Shape.Box<X>, derived from  $\vdash$  Square <: Shape. Contravariant parametric types behave in the same way, but subtyping is contravariant with respect to the bounds. For example,  $\vdash$  Box<-Shape> <: Box<-Square> could be interpreted as  $\vdash \exists X \geq$  Shape.Box<X> <:  $\exists X \geq$  Square.Box<X>, from  $\vdash$  Square <: Shape.

Subtyping between variant and invariant parametric types is due to an implicit pack operation. Box<Square> is a subtype of Box<+Shape>, this corresponds to packing Box<Square> to  $\exists X \leq Shape.Box<X>$  by hiding Square with X.

As with existential types, a variant parametric type must be unpacked before it can be used, this is done during type checking, not by a dedicated expression. For example, a method call, x.get(), on a variable with type x:Box<+Shape> corresponds to unpacking x and then calling get on the opened variable, for example:

open x as y in
 y.get()

Here, y will have type Box<Z>, where Z is a fresh type variable. As with existential types, the introduced type variable (Z in the example) must not escape the scope of the open expression.

<sup>&</sup>lt;sup>14</sup>It is noted [54] that a system with existential types syntax and explicit open expressions could be used to provide a very expressive system of variance in a parametric, object oriented language.

This is done by finding a Z-free supertype of Z. In the example, a suitable supertype is Shape. This is the type that we would expect from the description of variant parametric types in section 2.2.2. Sometimes, it is not possible to find a supertype with no free type variables. This occurs when access to a member is restricted. For example, trying to call get on an object with type Box<-Shape> can be thought of in the same way as above. However, this time there are no supertypes of Z and so type checking fails; this is expected, access to get in Box<-Shape> is forbidden.

Variant parametric types are formalised as an extension to FGJ [53]). In this calculus, open and close operations (denoted  $\uparrow$  and  $\downarrow$ , respectively) are used in the type and subtype rules to unpack and pack variant parametric types.

The open operation has the form  $\Delta \vdash \mathbb{N} \uparrow^{\Delta'} \mathbb{N}'$ , where N is a variant parametric type and N' is an invariant parametric type. N is unpacked to N', this requires introducing fresh type variables and these are recorded in  $\Delta'$ . For example,  $\Delta \vdash Box<*>\uparrow^X Box<X>$ . This follows unpacking of existential types, where  $\exists X.Box<X>$  can be unpacked to Box<X>.

The close operation has the form  $\mathbb{N} \Downarrow_{\Delta} \mathbb{N}'$ , and is the 'reverse' of the open operation,  $\mathbb{N}$  is an invariant parametric type that is packed to  $\mathbb{N}'$ , a variant parametric type.  $\Delta$  records the type variables in  $\mathbb{N}$  that are abstracted by the close operation;  $\mathbb{N}'$  will not contain any of these variables. Thus, the close operation can be used to find a supertype of  $\mathbb{N}$  in which no free variables escape their scope. For example,  $Box<X> \Downarrow_X Box<*>$ . Again, this follows from the existential types interpretation of variant parametric types.

As an example of how these operations work together, we sketch the derivation of type checking for a field access expression. We assume that there is a class declaration class C<X> { C<X> f; } and that the variable x has type C<+A>; fType looks up the type of a field, the type it is passed must be invariant.

$$\begin{array}{c} \Delta; \Gamma \vdash \mathbf{x} : \mathsf{C} <+\mathsf{A} > & \Delta \vdash \mathsf{C} <+\mathsf{A} > \Uparrow^{\mathsf{X} \leq \mathsf{A}} \mathsf{C} <\mathsf{X} > \\ fType(\mathtt{f}, \mathsf{C} <\mathsf{X} >) = \mathsf{C} <\mathsf{X} > & \mathsf{C} <\mathsf{X} > \Downarrow_{\mathsf{X} \leq \mathsf{A}} \mathsf{C} <+\mathsf{A} > \\ \hline \Delta; \Gamma \vdash \mathtt{x} . \mathtt{f} : \mathsf{C} <+\mathsf{A} > \end{array}$$
 (VPT-T-FIELD)

The open and close rules are also used in the subtype rule for subclassing. Subtyping between variant parametric types (corresponding to subtyping between existential types) is given syntactically, without the use of open or close operations.

We compare variant parametric types with Tame FJ in section 3.1.

**Wildcards** Wildcards in Java may also be interpreted as existential types [90]. This correspondence is used to formalise wildcards in Wild FJ [60] and in chapter 3 of this thesis. More details of the correspondence are given in section 2.4.1.

# 2.4 Wildcards

Subtype variance in Java is given by wildcards [14, 47, 90, 60]. A *wildcard type* is a parameterised type where ? is used as an actual type parameter, for example Box<?>. Such a type can be thought of as a box of *some* type, where the wildcard is *hiding* that type. Where multiple wildcards are used, for example Pair<?, ?>, each wildcard hides a potentially different type.

A wildcard may be given upper or lower bounds using the extends and super keywords respectively. List<? extends Shape> is a list of some type where that type is a subtype of Shape; Box<? super Circle> is a box of some type where that type is a supertype of Circle. The bounding types may also have type parameters, including wildcards, so both Box<? extends List<Shape> and Box<? extends List<?>> are valid types.

If a class declaration involves bounds on the class's formal type parameters, then these bounds are 'inherited' by a wildcard used as an actual parameter. For example, consider the following class declaration:

```
class InheritBound<X extends Shape> {
    ...
}
```

If this class is instantiated with a wildcard, InheritBound<?>, then the Java compiler takes Shape as the upper bound of the wildcard. If the wildcard is given a bound, then the stricter of the two bounds is used; for example, the upper bound of InheritBound<? extends Object> is Shape, but the upper bound of InheritBound<? extends Circle> is Circle. It is therefore possible for a wildcard to have both upper and lower bounds, e.g., InheritBound<? super Circle> has bounds Circle and Shape; that is, the hidden type must be a subtype of Shape and a supertype of Circle.

Wildcard types enjoy variant subtyping; upper bounds give *covariance* and lower bounds give *contravariance*. So, where Circle <: Shape,

Box<? extends Circle> <: Box<? extends Shape>
Box<? super Shape> <: Box<? super Circle>

Non-wildcard types are subtypes of wildcard types according to similar variance rules (Wildcard types are never subtypes of non-wildcard types),

Box<Circle> <: Box<? extends Shape>
Box<Shape> <: Box<? super Circle>

Unbounded wildcard types are *bivariant*, that is, both co- and contravariant,

Box<? extends Shape> <: Box<?>
Box<? super Shape> <: Box<?>
Box<?> <: Box<?>

A wildcard type can be used as a type parameter to give nested wildcard types; for example, Box<Box<?>>, the actual parameter is the wildcard type Box<?>. In this case, the top level type (Box<...>) is *not* a wildcard type and so has invariant typing properties. For example, Box<Box<?>> is not a supertype of Box<Box<Shape>>. Such nesting can be extended to any depth and may include bounds; List<Box<Box<?>>>>, Box<Box<? extends Shape>>, and Box<Box<? extends Box<Pox<? extends Box<?>>>> are all valid types.

# 2.4.1 Existential Types for Wildcards

Existential types (section 2.3) can be used to understand and formalise Java wildcards [90, 60]. A wildcard type can be thought of as an existentially quantified type; for example, Box<?> can be represented as  $\exists X.Box<X>$ . Many of the above observations about wildcard types can be expressed using existential types: the scope of existential quantification ( $\exists X.Box<X>$  rather than  $Box<\exists X.X>$ ) expresses that a wildcard hides a single type and not potentially many types, the uniqueness of quantified variables (for example,  $\exists X,Y.Pair<X$ , Y> rather than  $\exists X.Pair<X$ , X> for Pair<?, ?>) reflects the uniqueness of wildcards, nested quantification ( $Box<\exists X.Box<X>$  for Box<Rox<?>>) reflects the invariance of nested wildcards — only top level quantification gives rise to variant subtyping.

Bounds on wildcards can be thought of as bounds on the existentially quantified type variable, thus Box<? extends Shape> corresponds to  $\exists X \rightarrow [\bot \text{ Shape}] .Box<X>$ . We use  $\bot$  as a lower bound where the wildcard is unbounded below and Object where a wildcard is unbounded above<sup>15</sup>. We will omit bounds in examples where they do not play an important part. We take into account bounds inherited from the class declaration and represent the precise bounds known to the type checker in our existential types. In the example from the previous section, we would write InheritBound<?> as  $\exists X \rightarrow [\bot \text{ Shape}] .InheritBound<X>.$ 

The major difference between type checking existential types and wildcards is that, in Java, packing and unpacking are implicit. Similarly to traditional existential types, wildcard types must be unpacked before they can be used (used in the Java sense means acting as the receiver of a field access or assignment, or method call). However in Java, this unpacking is done automatically without an explicit **open** expression. In Java this implicit unpacking is known as *capture conversion*; for example, the type C<?> is capture converted to C<Z>, where Z is a fresh type variable.

Subtyping between wildcard and non-wildcard types reflects packing of existential types. For example,

<sup>&</sup>lt;sup>15</sup>This is possible because unbounded wildcard types behave exactly like bounded ones. This is not the case in variant parametric types [54].

Box<Circle> <: Box<? extends Shape>

corresponds to

```
Box<Circle> <: \exists X \rightarrow [\bot \text{ Shape}] . Box<X>
```

This can be thought of as packing Box<Circle> to  $\exists X \rightarrow [\bot \text{ Shape}]$ .Box<X>, where Circle is the witness type, hidden by X.

Subtyping between wildcard types reflects subtyping between existential types. For example,

Box<? extends Circle> <: Box<? extends Shape>

corresponds to

 $\exists X \rightarrow [\bot \text{ Circle}].Box < X > <: \exists X \rightarrow [\bot \text{ Shape}].Box < X >$ 

Which follows from the full variant of existential subtyping (see section 2.3.2).

We formalise the translation from wildcard types to existential types in section 3.2.

# 2.4.2 Wildcard Capture

Wildcard capture is the mechanism by which a wildcard is promoted to a fresh type variable. This occurs most visibly at method calls: Tree<?> is not a subtype of Tree<X>, and yet the Java code in figure 2.5 is legal. At the method invocation, the wildcard in the type of y is capture converted to a fresh type variable, say Z, and the method invocation can then be thought of as this.<Z>walk(y). In figure 3.8 we show how this example is type checked in Tame FJ.

As mentioned in section 2.4.1, capture conversion corresponds to existential unpacking [60, 90]. Using an explicit **open** expression, the above method invocation becomes:

open y as z,Z in
 this.<Z>walk(z);

```
<X> List<X> walk(Tree<X> x) {...} {
List<?> walkAny(Tree<?> y)
this.walk(y);
}
```

Figure 2.5: Example 1.

Here it is clear where the fresh type variable Z comes from and how it is used<sup>16</sup>.

In the above examples we have written the type parameter to the method calls (Z). This can always be done in Java generics without wildcards. However, if wildcards are used, then there exists programs for which there is no source to source translation that produces a program with all type parameters named. This is because the hidden type parameters of wildcards cannot be named.

Wildcard capture may give rise to types during type checking that cannot be denoted using the Java syntax, these types are *expressible but not denotable*. This is a serious obstacle for a direct formalisation of Java wildcards using the Java syntax, because type soundness requires typability of every step of the computation. This is the primary motivation for using existential types to model Java wildcards. For example in figure 2.6, the method invocation at 1 is type incorrect because the method compare requires a Pair parameterised by a single type variable twice. Pair<?, ?> cannot be capture converted to this type because the two wildcards may hide different types. Its existential type,  $\exists X, Y.Pair<X, Y>$ , makes this clear. The result of the call to make at 2 has a type which is expressible but not denotable. The type checker knows that the wildcards hide the same type (even though this cannot be denoted in the surface syntax) and so capture conversion, and thus type checking, succeeds. This type can be denoted using existential types as  $\exists X.Pair<X, X>$ . We show how this example is type checked in figure 3.9.

close (this.<Z>walk(z)) with X hiding Z;

The result will have type  $\exists X.List < X >$ , equivalent to List <?>.

<sup>&</sup>lt;sup>16</sup>In this case, the result of the method invocation has type List<Z>. This must be repacked to List<?> to prevent the escape of Z. In Java this is done implicitly by subtyping; the fully explicit version of the method invocation is:

open y as z,Z in

```
<X>Pair<X, X> make(List<X> x) {}
<X>Boolean compare(Pair<X, X> x) {}
void m()
{
    Pair<?, ?> p;
    List<?> b;
    this.compare(p); //1, type incorrect
    this.compare(this.make(b)); //2, OK
}
```

Figure 2.6: Example 2.

# 2.4.3 F-Bounds

Where a wildcard type instantiates a class with an F-bounded parameter, the wildcard is also F-bounded. For example, given class F<X extends F<X>>>, the wildcard type F<?> is given the bound F<?>, where the wildcards hide the *same* type parameter. Using existential type notation, we write  $\exists Z \rightarrow [\bot F < Z >]$ . F<Z>. The bound F<X> is inherited from the class declaration, substituting the actual parameters gives F<Z>.

Surprisingly, F<? extends F<?>> also corresponds to the existential type  $\exists Z \rightarrow [\bot F < Z >]$ .F<Z>. The existential type found directly is  $\exists Z \rightarrow [\bot \exists Y.F < Y >]$ .F<Z>, whereas the type found by inheriting the bound is  $\exists Z \rightarrow [\bot F < Z >]$ .F<Z>. Since F<Z> is a subtype of  $\exists Y.F < Y >$ , the inherited bound is used instead of the declared one.

We can test these translations using capture conversion; for example in figure 2.7 the call to testF succeeds because f has type  $\exists Z \rightarrow [\bot F < Z >]$ .F<Z>, which can be capture converted to F<X>. The call to testNotF fails because notF has type  $\exists Z \rightarrow [\bot \exists Y.NotF < Y >]$ .NotF<Z>. This type cannot be capture converted to NotF<X> because the bounds on Z do not match the bounds declared on X in the testNotF method.

# 2.4.4 Using Wildcard Types

A wildcard type gives only partial information about an object, therefore, there must be some restrictions on using such objects. These restrictions ensure soundness in the presence of variant

```
class F<X extends F<X>> { }
class NotF<X extends NotF<?>> { }
class Test{
    <X extends F<X>> void testF(F<X> x) { }
    <X extends NotF<X>> void testNotF(NotF<X> x) { }
    void m(F<? extends F<?>> f, NotF<? extends NotF<?>> notF) {
        testF(f); //OK
        testNotF(notF); //error
    }
}
```

Figure 2.7: Testing the behaviour of F-bounded wildcard types.

subtyping.

We can use a wildcard's upper bound where the corresponding formal variable appears in covariant position, and the lower bound where the variable appears in contravariant position. If a bound is missing, then we can still use Object in covariant position or the bottom type  $(\perp)$  in contravariant position. In the box example, the return type of get is covariant and the parameter of set is contravariant. We thus get the following types:

	b:Box extends Shape	b:Box super Shape	b:Box
b.get():	Shape	Object	Object
b.set(x)	x:⊥	x:Shape	x:⊥

We can think of these rules in terms of existential types. If **b** has type  $\exists X \rightarrow [A \ B] . Box<X>$ , then calling **b.get()** requires us to unpack **b**, then type check the invocation on type Box<X>. Thus, the result of **b.get()** has type X, but if this is given as the result type, then X would escape its scope. We can use subsumption to give an X-free supertype of X, which in this case can be the upper bound (B) of X. This corresponds to the type of **b.get()** in the second column of the table.

To call set, we must again unpack b to find the parameter type, X. X is fresh in the scope of the method invocation so to derive that any type is a subtype of X, we must derive that that type is a subtype of X's lower bound — A. This corresponds with the parameter type of b.set in the third column of the table.

We can see that there is a trade-off between flexible subtyping and usefulness. We can get

precise types for all operations on an invariant generic type. But to use co- or contravariance, we must sacrifice some information about the types of methods and fields, thus restricting usefulness. An unbounded wildcard type has the most flexible subtyping, however, we can only call methods with null parameters and return Objects.

Crucial to understanding the rules for using wildcard types, is that a wildcard hides a specific actual type argument. A wildcard's bound is a bound on this hidden type, not a bound on the type of objects with the hidden type. The difference is due to the implicit subsumption due to inheritance found in object-oriented languages (section 2.1.1). A box with type Box<? extends Shape> may have witness type Circle or Shape (amongst others). If the hidden type is Circle, then the box may contain objects of type Circle and its subclasses. If the hidden type is Shape then it may contain objects with type Shape or its subclasses. A box with type Box<? super Shape>, may have hidden type Shape or Object, the objects in the box must be a subtype of the hidden type, so in the latter case, the box may contain objects of any type. These intuitions lead straightforwardly to the rather arbitrary feeling rules about types in co- and contravariant positions.

# 2.4.5 Comparison with Variant Parametric Types

Java wildcards evolved from variant parametric types [54] (section 2.2.2). Other than the difference in notation, the two systems appear very similar, and both have a strong correspondence with existential types (see section 2.3.7). However, there are some subtle but important differences; we describe them in this section.

There is no equivalent of wildcard capture in variant parametric types. In Java, a method with signature <X>void m(List<X> x) can be called on an actual parameter with wildcard type, such as List<?>; this is not permitted with List<\*> or any other variant parametric type. This makes wildcards far more flexible; instances requiring wildcard capture occur frequently in practice [54].

Wildcard capture is a form of existential unpacking with type parameter inference. Existen-

tial unpacking without inference occurs in both systems, where a wildcard type or variant parametric type is the receiver of a method call or field access.

Types can be expressed in Java programs, by using wildcard capture, that cannot be denoted in the wildcards syntax (see section 2.4.2). For this reason, explicit existential types are required to formalise wildcards. Since variant parametric types lack wildcard capture, such types cannot be expressed. Thus, the natural syntax of variant parametric types is sufficient for their formalisation. Although the syntax of types of variant parametric types and wildcards are equally expressive, the Java language with wildcards can express more types than the variant parametric types system. For example both syntaxes can express a type equivalent to  $\exists X, Y.Pair < X, Y >: Pair <*, *>$  in variant parametric types and Pair <?, ?> in Java. However, an expression with type equivalent to  $\exists X.Pair < X, X >$  can only be expressed in Java (see section 2.4.2). Explicit existential types are more expressive than either system's notation.

In Java, there is an equivalence between unbounded and bounded wildcards. A wildcard without an upper bound is equivalent to a wildcard with Object as its upper bound. A wildcard without a lower bound is equivalent to a wildcard with the bottom type as its lower bound. Such an equivalence does not exist with variant parametric types. The practical consequence of this, is that variant parametric types restrict access to some members completely, whereas wildcards allow all members to be accessed. For example, no methods may be called on a variable with type Box<\*>, but, on a variable with type Box<?>, we may call set(null), and call get(), returning an Object.

Partially as a consequence of the previous difference, subtyping between variant parametric types is more restrictive than between wildcards types. Java with wildcards allows subtyping between bounded and unbounded wildcards, whereas, variant parametric types forbid subtyping between types with different variance annotations. For example,

 $\vdash$  Box<? extends T> <: Box<?> and  $\vdash$  Box<? super T> <: Box<? extends Object> are true in Java with wildcards, but the corresponding variant parametric types relations,  $\vdash$  Box<+T> <: Box<\*> and  $\vdash$  Box<-T> <: Box<+Object> are not.
# 2.4.6 Previous Models of Java with Wildcards

There are several existing models that are, or could be regarded as, models for wildcards. None of these models offer a proof of type soundness for Java with wildcards. Some are informal, some are only partial models, and some do not address soundness.

#### Informal Descriptions

There have been several informal descriptions of Java with wildcards. The Java specification [47] is the definitive reference for the behaviour of wildcards. However, it is rather vague in some of the subtler areas of the specification, such as the definition of legal bounds on wildcards that involve inherited bounds and F-bounds.

In the first description of wildcards [90], the authors describe the typing and subtyping properties of wildcards, capture conversion, and inference of type parameters including wildcards. Wildcards are described in terms of existential types, but there is no formalisation.

Wildcards have been described in terms of access restriction [92]. Access restriction is used to describe variant parametric types [54] in terms of the members that can or cannot be accessed in a parameterised class. This is extended to cover wildcards, where true access restriction is replaced by restriction of types (see section 2.4.5). Access restriction is formalised in terms of a close relation, similar to that used for variant parametric types. Capture conversion is ignored and no formalism is given for a full language.

#### Wild FJ

Wild FJ [60] was the first formalism to include all of the interesting features of Java wildcards. The syntax of Wild FJ is a subset of Java with wildcards, but requires explicit type arguments to polymorphic method calls. Java types are converted to existential types 'on the fly', and this conversion of types complicates the typing, subtyping, well-formedness, and auxiliary rules. Type soundness has never been proved for Wild FJ. Wildcard types are used in the surface syntax of Wild FJ, but existential types are used in type checking, and throughout the formalism. Wildcard types are translated to existential types by the *snap* function. *snap* operates on types using *fix to translate type parameters*, fix uses *merge to merge the declared and inherited bounds on wildcards (see the start of section 2.4). There is a slight problem with* merge: if there are both declared and inherited bounds, it always uses the declared bound. The JLS [47] states that the most strict bound is used, as is done in our translation (section 3.2).

In the type rules (WT-INVK and WT-FIELD), packing is done by combining a type that may contain free variables (say, T) and the environments that may contain those free variables ( $\Delta$ ) to make an existential type,  $\exists \Delta$ .T. Packing also takes place in WS-ENV, in the same way as in XS-ENV of Tame FJ (described in section 3.1.2).

Unpacking in Wild FJ is done by separating an existential type,  $\exists \Delta . T$ , into  $\Delta$  and T. Unpacked types can be used to infer type parameters using the *capture* function. If an actual type parameter of a method is explicitly declared, then the *capture* function returns it. If it is omitted (actually marked with  $\star$ ), then a limited form of inference is performed (this may infer unpacked variables and performs a similar role to *match* in Tame FJ (section 3.1.4)).

See section 5.2.1 for a comparison of Wild FJ with Tame FJ.

The subtyping rules of the JLS [47] have been formalised and compared (informally) with those of Wild FJ [60]. The main difference between JLS and Wild FJ subtyping, is that in the JLS, conversion to existential types is only done where necessary, whereas in Wild FJ, Java types must be converted to existential types before other rules can be applied. Under the formalisation of the JLS rules, conversion to existential type (using *snap*) is combined with unpacking in a rule similar to PIZZA-S- $\exists$ - $\leq$  [70] (section 2.3.7). Subtyping between wildcard and non-wildcard types (which involves packing of the underlying existential representation) is handled without conversion to existential types. These two rules could be combined into an approximation of the WS-ENV rule.

#### Variant Parametric Types

Variant parametric types [54] (see sections 2.2.2 and 2.3.7) have been formalised and proved sound, and this formalism can be regarded as a partial model for wildcards in Java. Capture conversion and some other flexibility is missing from a full model for wildcards, the differences are described in section 2.4.5.

#### JavaGI and $\mathcal{EX}_{upto}$

JavaGI [94] is a proposed extension to Java that generalises Java interfaces. In particular, it allows interfaces to be dealt with in terms of existential types. Packing and unpacking takes place in separate subtype rules, these rules are similar to those of Pizza [70], described in section 2.3.7.

 $\mathcal{EX}_{upto}$  [95] is a formulation of subtyping of wildcard types in Java. It follows subtyping in JavaGI, in particular, packing and unpacking occur in separate rules, as in Pizza [70].  $\mathcal{EX}_{upto}$  uses explicit existential types; wildcard types must first be translated to existential types. We compare  $\mathcal{EX}_{upto}$  subtyping with Tame FJ in section 5.2.3.

Subtyping in JavaGI and  $\mathcal{EX}_{upto}$  is undecidable [95]. There are no type or reduction rules for  $\mathcal{EX}_{upto}$  and soundness is not investigated. Furthermore, it is a simplification of Java subtyping (for example, subclassing and inheritance are not considered).

# 2.4.7 Decidability

Decidability of typing in Java with wildcards is an interesting and open problem. Wildcards are closely related to existential types, and the standard formulation of existential types is undecidable [73] (see section 2.3.4). The main questions concerning decidability of wildcards involve subtyping and the decidability of type parameter inference. The latter has not yet been addressed [62].

Java subtyping does not satisfy any of the conditions for decidability suggested by Pierce [73]: quantification of types can occur at any depth, there is no stratification of types, and subtyping follows the full variant. A particular formalisation of subtyping (using separate rules for packing and unpacking as in Pizza, see section 2.3.7) has been shown to be undecidable [95]. On the other hand, a decidable type unification algorithm for Java 5.0 has been described [76], which is a step toward a decidable subtyping algorithm, and hints that such an algorithm may exist.

Subtyping in the general case for declaration-site variance is undecidable [55]. The fact that there exists a simple encoding from declaration-site to use-site variance suggests that subtyping in Java may be undecidable (several example programs are given in [55] that cause the Java compiler to crash). However, as Kennedy and Pierce point out [55], the argument does not apply if subtyping is restricted in certain ways; for example, subtyping on the .NET CLR (the virtual machine of C# and its intermediate language) is decidable because each type has a finite number of supertypes. Java satisfies one of these restrictions, multiple inheritance from different instantiations of the same type is forbidden<sup>17</sup>. However, this does not mean that Java subtyping is decidable, only that this specific argument cannot be applied.

# 2.5 Ownership Types

In Java and similar languages, the heap is unstructured in the sense that any object can refer to or access and modify any other object (subject to class-level access restrictions such as private). This means that reasoning about programs (either by the programmer or some tool) must take the entire heap into account. In particular, *Aliasing* (allowing multiple references to an object) allows an aggregate's representation to change without the aggregate being aware of the change.

There have been many attempts to overlay some *structure* on the heap, in the sense of restricting the parts of the heap that a given object can access, reference, or modify. This limits the scope or effect of aliasing and allows for easier and more powerful reasoning about programs.

<sup>&</sup>lt;sup>17</sup>For example, a class cannot implement Box<Circle> and Box<Square> (if Box were an interface).

Islands [50] and Balloons [7] made the first attempts to structure the heap. They enforced *full encapsulation*, that is, they restrict both the ingoing and outgoing references of an aggregate. The heap is divided into distinct partitions and references cannot breach the partition boundaries. Flexible alias protection [69] was proposed as an alternative where references (and thus aliases) to an object's internal state (its *representation* in the ownership terminology) are limited, while references to its interface may be shared more freely.

Ownership types [30, 32, 34, 33, 97] implement the core features of flexible alias restriction. In an ownership types system each object is owned by a *context* (contexts are usually objects, although the definition of context changes from system to system). This structures the heap as a tree and the type system ensures that this structure cannot be changed at runtime. Ownership types can be used to control aliasing [34, 18, 12, 85, 84] by restricting references according to the ownership hierarchy.

Various flavours of ownership types have been suggested, and have been successfully applied in many areas: annotating large Java library classes and multi-threaded server programs to prevent data races in all their studies [13]; supporting memory management in real time systems, with applications such as flying unmanned aircraft [9]; enforcing software architectures in large, real-world software [5].

We describe the ownership hierarchy with an example. In this graphical representation of a small heap, objects are represented as shaded boxes with rounded corners, and encapsulation boundaries with square cornered rectangles. Not all possible or illegal references are shown; since the root object is not a real object, it is also not shown.



In this example, object 1 owns object 3, which owns objects 4 and 5. The representation of object 1 is objects 3, 4, and 5. Object 2 owns no object and is only owned by the theoretical root object (as is object 1).

The relation between objects in the ownership hierarchy is expressed by the inside relation. We write  $\vdash a \leq b$  — a is inside b — if a can be derived to be transitively owned by b. In the example, we have:

- dash 3  $\preceq$  1
- $dash 4 \preceq 3$
- $\vdash$  5  $\leq$  3
- $dash 4 \preceq 1$
- $\vdash 5 \preceq 1$

We denote the root of the ownership hierarchy with  $\bigcirc$ . All objects are considered inside  $\bigcirc$ . All flavours of ownership type share a notion of objects being owned by contexts and have the effect of overlaying a graph (usually tree) structure on the heap. There are also several ways to describe the ownership structure in the program syntax. One common solution, and the one adopted in this thesis, is to parameterise classes by contexts, similarly to parametric types (see section 2.2). Each instance of a class can have different actual context parameters. The first context parameter of a type is the owner of objects of that type and is mandatory. For example, a type C<o> describes objects owned by o. In the heap diagram, and assuming that all objects are instances of the class C which has a single context parameter, the objects have the following types:

1:C<)> 2:C<)> 3:C<1> 4:C<3> 5:C<3>

Context parameters are fixed for the lifetime of an object and are immutable. Declared subclasses must have the same owner as their superclasses [32] and subtyping is usually invariant. Thus, the owner of an object cannot change due to subtyping.

Although ownership is a property of objects and we speak about a structure on the heap, ownership and the inside relation are purely static properties. No runtime checks are necessary (except when casting, as in other systems). Furthermore, all type information can safely be erased at compile time. All contexts are named statically and the ownership hierarchy is known statically from declared relations between contexts; no new ownership information is available at runtime.

Classes may be parameterised by more than one formal context, these extra contexts can be used within the class declaration as actual contexts in the types of fields and methods. For example, we can define a linked list as:

```
class List<o, do> {
    Object<do> datum;
    List<o, do> next;
    ...
}
```

A list with type List<a, b> is owned by a, the data in the list is owned by b. A heap containing a list with three nodes is shown in the diagram below:



*Owner polymorphic methods* [30, 97] are methods parameterised by contexts. They allow for code reuse with different owners. They have a similar syntax and semantics to type polymorphic methods. Since the scope of context parameters is limited to the method, it is not possible for the receiver to store a reference to any object owned by a method-level context parameter. Owner polymorphic methods thus increase flexibility of ownership systems without compromising integrity.

In many systems (for example [30, 32, 72]), formal context parameters may have explicit bounds; a bound states that the parameter is inside or outside a context (we use outside to mean **a** is outside **b** if and only if **b** is inside **a**). These bounds can then be reflected in the inside relation. Using bounds makes ownership systems more descriptive; they are necessary (implicitly or explicitly) for the owners-as-dominators property, described below.

# 2.5.1 Encapsulation Properties

Many ownership types systems, including the original work on ownership types, enforce the owners-as-dominators property [34, 30, 79]. A system satisfies owners-as-dominators if every path of references to an object,  $\circ$ , from the root object, passes through the owner of  $\circ$ . That is, objects are *dominated* by their owners. Clarke [30, 34] showed that owners-as-dominators is satisfied if every reference to an object  $\iota'$  comes from an object  $\iota$  that is inside the owner of  $\iota'$ . More formally,

$$\forall \iota, \iota' \in dom(\mathcal{H}) : \iota \text{ refers to } \iota' \Rightarrow \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\iota')$$

We use  $own_{\mathcal{H}}(\iota')$  to denote the owner of  $\iota'$  in the heap  $(\mathcal{H})$  and dom to give the addresses of all objects in  $\mathcal{H}$ .

We extend the heap example above with references. To satisfy owners-as-dominators, legal references are represented with solid arrows, and illegal references with dashed arrows.



Encapsulation in most context-polymorphic ownership systems is given by the ability to name contexts. In order to write a type, its actual context parameters must be named. By ensuring that all contexts that can be named in a class are outside the owner of that class, we get the owners-as-dominators property. Systems that satisfy owners-as-dominators enforce *deep* ownership, encapsulated objects are protected from direct and indirect access.

Owners-as-dominators is sometimes *too* strong; some common programming idioms cannot be written if it is strictly enforced. The most common example is an iterator. An iterator can be owned either by its collection or its user. In the first case, it cannot be accessed from outside the collection so is useless. In the second case, it cannot access the contents of the collection, again, it's useless. There have been two proposals to relax ownership systems to support iterators and similar idioms: allowing dynamic references to encapsulated objects [32] and allowing inner classes access to an object's representation [30, 11]. These two approaches are described in the next paragraphs.

By allowing final variables to act as context parameters, this can be safely named as a context outside of its class declaration. This allows encapsulation to be temporarily violated and an object's representation accessed. Access to encapsulated objects is limited to the lifetime of a method by variable scoping. Storing references to an object's representation (*dynamic aliases*) is limited to the stack; the owners-as-dominators property still applies to the heap. An iterator is implemented by being owned by its collection, it thus has access to the objects in the collection. It can be accessed from outside the collection using a dynamic alias, for example:

final List<this, d> list = ...;
Iterator<list, d> it = list.makeIterator();

Inner classes are classes declared within a class definition and considered to exist on a per-object basis. Inner class objects can be considered to have special privileges to the representation of the object in which they are declared [30, 11]. In an inner class, the programmer can name the outer class object in which the inner class resides using the syntax C.this, where C is the name of the outer class. An iterator can be implemented as an inner class of its collection. Since the collection does not own the iterator, it can be used freely.

An alternative encapsulation property is *owners-as-modifiers*, supported by the *universes* type system [66, 67] and others. In these systems, there are fewer restrictions on references, but objects can only be modified by their (transitive) owners. Owners-as-modifiers encapsulation is useful for reasoning with invariants [68].

# 2.5.2 Related Systems

In this section we describe some of the different flavours of ownership types systems. We focus on systems that are related to our work with existential types and variant ownership (section 4).

#### Multiple Ownership

MOJO [23] extends standard ownership systems by allowing objects to have any number of owners. This makes for a much more flexible system; however, strong encapsulation properties cannot be enforced. The heap is structured as a directed acyclic graph rather than a tree. This graph is used to describe effects and show disjointness of expressions.

In MOJO, classes are declared with a single formal owner; but many objects may be bound to a single context parameter, thus, an object can have multiple owners. Using ? expresses an unknown context. Since the unknown context may only be expressible as the intersection of many named contexts, ? expresses uncertainty in the number of objects that own an object.

#### **Ownership Domains**

In the standard ownership types systems, contexts are defined by objects. *Ownership domains* [6, 5, 56, 83] relaxes this restriction by using domains as contexts; a domain simply represents some part of an object's representation. An object may have multiple domains, allowing for more flexible heap topologies than ownership types, whilst still maintaining a tree structure. By allowing the access policies of domains to be specified by the programmer, the representation

of an object can safely be partially exposed, allowing for the implementation of iterators and similar constructs.

#### Universes

Universes [40, 66, 67, 68, 36] use a simpler notation than ownership types. There is no parameterisation of classes or types. Instead, types may be annotated with peer (owned by the same context as this), rep (owned by this, i.e., in the representation of this), or any (owned by some unknown owner). The universes type system enforces owners-as-modifiers by disallowing field assignment and (non-pure) method call to objects annotated with any. Flexibility is given by allowing any variables to be passed around and accessed without restriction.

#### Effects

An *effect* [48] describes the state that is accessed during the execution of a piece of code. By analysing the effects of expressions and methods, a tool can reason about the interference or disjointness of pieces of code. This knowledge is useful for many applications, such as program transformations, concurrency analysis, and debugging. Ownership types allow effects to be specified in terms of the ownership hierarchy [32]. This allows for the precise denotation of effects, including of parts of the heap that are unknown (because of the hierarchical nature of object ownership). Effects have been combined with multiple ownership [23], ownership domains [84], and invariants [59].

In *effective ownership* [57], effects are used to enforce encapsulation. The same object hierarchy is used as in ownership types, however, encapsulation is enforced by restricting write effects rather than references. Effective ownership enforces an owners-as-modifiers discipline. The type system is more flexible than the Universes system, but requires more syntactic overhead.

#### Uniqueness

A pointer is *unique* if it is the only pointer to an object. Uniqueness systems [50, 63] are an alternative to ownership types for controlling aliasing. Since a unique reference can have no aliases, the reference is the only possible way to access, and thus change, the referenced object. This allows programmers and compilers to reason locally about unique references and avoid many of the problems usually associated with reasoning about object-oriented programs.

Uniqueness is a very strict protocol and causes abstraction problems: minor changes to the implementation of a class can require changes to the interface of a class. *External uniqueness* [29, 31, 72] addresses this issue by requiring that there is only one *external* reference to a unique object; there may be many references to a unique object from within its representation. Owner-ship types are used to define and enforce representation encapsulation. External uniqueness is a stronger property than owners-as-dominators. In addition to forbidding references to within an object's representation, it only allows one reference to the object itself.

#### Confinement

Confinement [93] is an alternative to ownership types. The scope of encapsulation is drawn from packages rather than objects and is defined on a per-class rather than per-instance basis. Confinement has the advantage that the number of scopes is statically known (equal to the number of packages), as opposed to the unbounded number of contexts in parametric ownership types. There is also less annotation overhead and the system is conceptually simpler. However, encapsulation is weaker and more coarse grained than in ownership types<sup>18</sup>.

### 2.5.3 Variant Ownership

Ownership types are usually invariant (section 2.5). To increase flexibility, there have been several attempts to introduce some kind of variant subtyping to ownership systems. Variant

<sup>&</sup>lt;sup>18</sup>This is not a weakness of confinement types, because their motivation is encapsulation within modules, not objects.

contexts increase flexibility by abstracting contexts: a type can be written without having to be able to name that type's context parameters. This leads to increased genericity and less duplication of code: methods and types can be written that operate on objects with different owners. Furthermore, context variance improves abstraction, because interfaces can be written without specifying precise contexts in types, which may be an implementation detail. See section 4.1.3 for an example with many uses of variant contexts.

In this section we describe previous approaches for adding context variance to ownership languages. We compare these approaches with  $Jo\exists$  in section 5.3.1.

#### The 'any' Context

Several systems include the notion of an unknown context: any (sometimes readonly) in universes [66, 67, 36] and effective ownership [57], and ? in MOJO [23]. For all classes C and contexts o, C<o> is a subtype of C<any><sup>19</sup> (peer C and rep C are subtypes of any C in the universes type system). There must be some restriction on the use of objects owned by the unknown context in order to preserve soundness and encapsulation properties. In universes and effective ownership such objects cannot be modified.

An 'any' context is a useful addition to an ownership system, it allows for the abstraction of types and thus gives the benefits of variant contexts such as collections of objects with different owners. However, it only allows complete abstraction of contexts, it is more useful to have partial abstraction so that some information about contexts is retained. Partial abstraction would allow the programmer to make a trade-off between increased flexibility and better ownership information.

#### Variant Ownership Types

Variant ownership types [58] support variance annotations [54] (see section 2.2.2) on context

<sup>&</sup>lt;sup>19</sup>Note the difference in behaviour to the  $\bigcirc$  context. C<o> is not a subtype of C< $\bigcirc$ >, even though  $\bigcirc$  is outside all contexts.

parameters. Actual context parameters<sup>20</sup> may be marked as co-, contra-, or bivariant. Subtyping follows variant parametric types [54] except that variance is with respect to the inside relation, not subtyping. For example, C<+o2> denotes an object of class C owned by some object that is inside o2; C<o1> is a subtype of C<+o2> if o1 is inside o2. A bivariant context (marked \*) is equivalent to the unknown contexts discussed above. By allowing variant contexts, the naming restrictions of ownership types that give rise to encapsulation are lifted. Encapsulation is maintained because a separate mechanism for accessibility is used. By separating ownership into accessibility and reference capability and by supporting variant contexts, the type system is flexible enough to support iterators and similar idioms.

The type system handles variant ownership types as if they were existential types; packing and unpacking are part of the type system, and are not made explicit in the syntax.

Variant ownership does not support owner polymorphic methods or both upper and lower bounds on variant contexts. The scope of variance in a type is fixed for a given class; for example, we could write a list where each element has the same owner, and, by using variance annotations, we can abstract this owner. Or, we can write a list class where the owner of each element is potentially different and unknown. However, it is not possible to write a generic list where the user of the list can choose if the elements have the same or different owners.

# 2.5.4 Existential Types and Ownership

Existential types have appeared in the ownership literature in several guises and for several purposes. Here we describe explicit uses of existential types in ownership languages; we compare these to  $Jo\exists$  in section 5.3.2.

#### Infinitary Ownership Types

In his PhD thesis, Clarke [30] uses existential quantification of contexts to avoid dependent typing in his *infinitary ownership types* system. This system is a formalisation of ownership

 $<sup>^{20}</sup>$ Therefore, variant ownership types are a form of use-site variance, see section 2.2.2.

as an object calculus and uses dynamically created contexts to model an infinite number of contexts (present in class-based systems because of class instantiation). Using dynamically created contexts as parameters in types would lead to dependent typing, Clarke avoids this by hiding the dynamic context with existential quantification.

Existential quantification of contexts in infinitary ownership types follows quantification of types in existential types systems (see section 2.3.2). Quantification is explicit and existential types are introduced and eliminated using pack and unpack expressions, respectively. There are no lower bounds on contexts.

Quantification of an object's owner is forbidden to prevent information about an object's owner being forgotten by subsumption. This ensures that ownership remains invariant with subtyping and that encapsulation properties are not violated.

#### System $\mathbf{F}_{own}$

System  $F_{own}$  [56] is a formalisation of an ownership domains [6] language as an extension of System F. System  $F_{own}$  is imperative, higher order, and includes existential types using pack and unpack expressions. It does not include subtyping, so does not have a notion of subtype variance. There is no explicit treatment of objects or classes<sup>21</sup>. Ownership is enforced using permissions. A permission allows functions in one domain to *access* or create entities in another domain. "Access" includes reading, dereferencing, storing a reference, unpacking an abstract package, and function application.

In System  $F_{own}$ , types may be existentially quantified by type variables or domains. Quantification of a domain means that the domain does not need to be named. This allows abstract packages to be exported out of the scope of a domain declaration (which is similar to a let expression). Safety is ensured because a domain must still have access to an abstract package to unpack it, even if the abstract package's domain cannot be named<sup>22</sup>.

<sup>&</sup>lt;sup>21</sup>These can be encoded using existential types.

<sup>&</sup>lt;sup>22</sup>This is possible because permissions can be to or from the domain they are attached to.

System  $F_{own}$  satisfies a strong encapsulation property called *access correctness*. This states that any access to an entity occurs from a domain that has permission to access that entity's domain. That is, execution abides by the permissions defined by the programmer.

#### **Existential Downcasting**

To ensure type safety, downcasts<sup>23</sup> must be dynamically checked. In an ownership system this dynamic check must include checks on ownership information. This means keeping ownership information around at runtime, which is a potentially huge overhead. To address this problem, an object can be cast to an existential type, known as *existential downcasting* [98].

Wrigstad and Clarke [98] use a programmer friendly syntax without explicit existential quantification. Fresh names for contexts are used in casts and these may be used outside of the casts. These fresh contexts resemble unpacked existential types. Their formalism supports this view (which is not explicitly expressed in the paper): existential owners are introduced by a let expression, the syntax and semantics of which closely resemble existential unpacking.

The following example demonstrates existential downcasting:

```
void m(Object<this> x) {
   List<this, d> l = (List<this, d>) x;
}
```

The cast introduces the context d. The owner of l (this) is not affected by the cast since it is named in the type of x. Since d cannot escape the scope of the method, elements of l (and l itself) cannot be permanently stored or returned from the method.

## 2.5.5 Generics and Ownership Types

Ownership types and generics fulfil different roles; in terms of a list, generics allow us to say "this is a list of X" and ownership types to say "this list belongs to x". It is likely that a

 $<sup>^{23}</sup>$ A downcast is a cast that gives a type T to an expression e with type T' where T is a subtype of T. In Java casting has the syntax (T)e.

programmer will wish to use both generics *and* ownership, to say "this is a list of X belonging to x". There have been two approaches to combining ownership and generics: support the two systems orthogonally or use generics to implement ownership.

#### **Combining Generics and Ownership Orthogonally**

Type genericity can be combined with ownership types by allowing classes to be parameterised by contexts *and* types. For example (we write context parameters first and use lower case, we write type parameters using upper case),

```
class GenericList<o, X> {
  X datum;
  GenericList<o, X> next;
}
```

In comparison to the list example in section 2.5, only the owner of the list is mentioned explicitly, the owners of the elements in the list are part of the actual type parameter that will be bound to X. For example, List<this, Shape<>>> represents a list of Shapes owned by this where each shape in the list is owned by  $\bigcirc$ . In fact, using unbounded type variables is a convenient shorthand [30]. To specify the list fully, we must use a formal context for the owner of the elements:

```
class GenericList<o, d, X extends Object<d>> {
    X datum;
    GenericList<o, X> next;
}
```

The invariance of owners ensures that any subtype of Object < d > (and thus any valid actual parameter for X) will have owner d.

As with most topics in ownership types, Clarke discusses adding type genericity to ownership types in his thesis [30]. His object calculus for infinitary ownership supports type variables and allows methods to be universally quantified by types. He goes on to discuss adding generics to a class-based language with ownership types. He also hints at the possibility of using type parameters to implement ownership, discussed in the next section.

Universes also come in a generic variety [39]. Since types in the universes system do not require ownership parameterisation, a class is parameterised by types only. Type generic classes allow more precise specification of ownership than in the non-generic universes system. For example, without generics, a list must have elements with an **any** modifier:

```
class UList {
    peer UList next;
    any Object datum;
    ...
}
...
UList l = ...
```

With generic universes, the ownership modifier of elements can be specified in the type of list objects, rather than in the list's class declaration:

```
class GUList<X> {
    peer GUList<X> next;
    X datum;
    ...
}
...
GUList<rep Shape> 1 = ...
```

Generic universes enforce the owners-as-modifiers discipline, as in non-generic universes.

Confined types have been extended with generics [99]. Confinement can be specified by type parameters, as well as in the class declaration, so classes can be generic in their confined-ness. Generic confinement is formalised as an extension of FGJ [53] and proved sound.

#### Implementing Ownership with Generics

Rather than treating generics and ownership separately, it is possible to extend a generic type system to implement ownership types. OGJ [77, 78] does exactly that, implementing deep

ownership with only small additions to Java with generics. OGJ uses a *type* parameter to signal the owner of objects of a class. This parameter is given last in the sequence of type parameters and is usually called **Owner**. Contexts are indicated by type parameters. Types that indicate contexts extend the **World** class. This is used to indicate that objects are owned by **this**. This requires special treatment in the type system to enforce per-object, rather than per-class, ownership; all accesses to objects owned by **This** must occur using the **this** variable as receiver.

Context parameters may be implicitly bound, Object<O> may be used as a bound even if O is not declared as a parameter. Potanin et al. note that implicitly bound contexts are treated much like wildcards in Java [78].

```
A generic list is written in OGJ as
```

```
class OGJList<X extends Object<XOwner>, Owner> {
    X datum;
    OGJList<X, Owner> next;
    ...
}
...
OGJList<Shape<World>, This> 1 = ...
```

XOwner is treated as a fresh type variable bounded by World.

OGJ satisfies the deep ownership property. This is enforced in a similar way to standard ownership systems, by restricting context parameters to being outside the owner of an object and invariant ownership — the owner of a class may not change due to subclassing or subtyping.

# Chapter 3

# Formal Models for Wildcards

Wildcards (described in section 2.4) are the mechanism adopted in Java to implement subtype variance. Although there have been several descriptions and formalisations of Java wildcards [14, 47, 60, 90], none have been proved type sound except for ours.

In the first section of this chapter, we show type soundness for Java with wildcards using a new formal model, Tame FJ. We use explicit existential types (such as  $\exists X.List<X>$ ) to model Java wildcard types, but implicit packing and unpacking of existential types. In section 3.2, we define and discuss a translation to Tame FJ from a subset of the Java language that includes wildcards. In section 3.3, we discuss how Tame FJ relates to more traditional existential types systems; in particular,  $\exists J$ , a model for wildcards using explicit packing and unpacking. We also discuss some of the other interesting aspects of Tame FJ.

# 3.1 Tame FJ

Tame FJ is an extension of FGJ [53]. The major extension to FGJ is the addition of existential types, used to model wildcard types. Typing, subtyping and reduction rules must be extended to accommodate these new types, and to handle wildcard capture. As is common [53, 60], we regard Java's inference of type parameters for method calls (except where this involves wildcards) as a separate pre-processing step and do not model this in Tame FJ.

We use existential types in the surface syntax and, in contrast to Wild FJ, do not create them during type checking; this simplifies the formal system and our proofs significantly. In particular, capture conversion is dealt with more easily in our system because fresh type variables do not have to be supplied. We also pack existential types more declaratively, by using subtyping, rather than explicitly constructing existential types.

е	::=	$x \mid e.f \mid e.<\overline{P}>m(\overline{e}) \mid new C<\overline{T}>(\overline{e})$	expressions
Q M	::= ::=	class C< $\overline{X \triangleleft T}$ > $\triangleleft$ N { $\overline{Tf}$ ; $\overline{M}$ } < $\overline{X \triangleleft T}$ > Tm( $\overline{Tx}$ ) {return e;}	class declarations method declarations
v	::=	new $C < \overline{T} > (\overline{v})$	values
N R T,U P	::= ::= ::= ::=	C <t̄>   Object&lt;&gt; N   X ∃∆.N   ∃Ø.X T   *</t̄>	class types non-existential types types type parameters
$\Delta \\ \Gamma \\ { m B}$	::= ::= ::=	$ \begin{array}{c} \overline{\mathbf{X}} \longrightarrow \begin{bmatrix} \mathbf{B}_l & \mathbf{B}_u \end{bmatrix} \\ \overline{\mathbf{x}} : \overline{\mathbf{T}} \\ \overline{\mathbf{T}} \mid \perp \end{array} $	type environments variable environments bounds
x C X,Y			variables classes type variables

Figure 3.1: Syntax of Tame FJ.

## 3.1.1 Notation and Syntax

Tame FJ is a calculus in the FJ [53] style, see section 2.1.2. We use vector notation for sequences; for example,  $\bar{\mathbf{x}}$  stands for a sequence of 'x's. In our rule definitions, we use the layout and ordering of premises only to aid comprehension and organisation of rules, there is no formal significance. Each rule consists of one or more conclusions and zero or more premises, there is no nesting of derivation rules or tree structure. We use  $\triangleleft$  as a shorthand for **extends** and  $\triangleright$  for **super**. The function fv() returns the free variables of a type or expression, and dom() returns the domain of a mapping. We assume that all type variables, variables, and fields are uniquely named.

The syntax for Tame FJ is given in figure 3.1. The syntax for expressions and class and method declarations is very similar to Java, except that we allow  $\star$  as a type parameter in method invocations. In Tame FJ (and as opposed to Java) all actual type parameters to a method invocation must be given. However, where a type parameter is existentially quantified (corresponding to a wildcard in Java), we may use  $\star$  to mark that the parameter should be inferred. Such types cannot be named explicitly because they cannot be named outside of the scope of their type. The marker  $\star$  is not a replacement for ? in Java;  $\star$  cannot be used as a parameter in Tame FJ types, whereas ? cannot be used as a type parameter to method calls in Java. Note that we treat this as a regular variable.

The syntax of types is that of FGJ [53] extended with existential quantification. Non-existential types consist of class types (e.g., C<D<>>) and type variables, X. Types (T) are existential types, that is non-existential types (R) quantified by environments ( $\Delta$ , i.e., sequences of formal type variables and their bounds), for example,  $\exists X \rightarrow [\exists \emptyset. D <> \exists \emptyset. Object<>].C<X>$ . Type variables may only be quantified by the empty environment, e.g.,  $\exists \emptyset. X$ . In the text and examples, we use the shorthands: C for C<>,  $\exists X.C<X>$  for  $\exists X \rightarrow [\bot Object<>].C<X>$ , and R for  $\exists \emptyset. R$ . We use  $\exists \emptyset. R$  (which subsumes  $\exists \emptyset. X$ ) as an element of T, rather than R, so that types are always existential type. This simplifies packing and unpacking.

Existential types in Tame FJ correspond to types parameterised by wildcards in Java. Using T as an upper or lower bound on a formal type variable corresponds to using extends T or super T, respectively, to bound a wildcard. This correspondence is discussed further in section 3.2. The bottom type,  $\perp$ , is used only as a lower bound and is used to model the situation in Java where a lower bound is omitted. This is a faithful modelling because bounded and unbounded wildcard types behave in similar ways (as opposed to, for example, variant parametric types [54]), see section 2.4.5.

Substitution in Tame FJ is defined in the usual way but with a slight modification. For the sake of consistency (see section 3.3) formal type variables are quantified by the empty set when used as a type in a program  $(\exists \emptyset. X)$ . Therefore, we define substitution on such types to replace

the whole type, that is  $[T/X] \exists \emptyset . X = T$ .

A variable environment,  $\Gamma$ , maps variables to types. A type environment,  $\Delta$ , maps type variables to their bounds. Where the distinction is clear from the context, we use "environment" to refer to either kind of environment.

Subclasses: $\vdash R \boxplus : R$		
$\texttt{class C<\overline{X} \triangleleft T_u} \texttt{>} \triangleleft \texttt{N} \ \{\ldots\}$		$\vdash R \boxplus: R'' \qquad \vdash R'' \boxplus: R'$
$\vdash \mathbb{C} < \overline{\mathbb{T}} > \boxplus : [\overline{\mathbb{T}/\mathbb{X}}] \mathbb{N}$		$\vdash \texttt{R} \sqsubseteq \texttt{R}'$
(SC-SUB-CLASS)	(SC-Reflex)	(SC-TRANS)
Extended subclasses: $\Delta \vdash B$	□: B	
class $C < \overline{X \lhd T_u} > \lhd \mathbb{N} \{ \}$	.}	
$\Delta \vdash \exists \Delta' . C \lt T > \sqsubset : \exists \Delta' . [T].$	XJN $\Delta \vdash \perp \Box$ : B	$\Delta \vdash B \sqsubset: B$
(XS-SUB-CLASS)	(XS-Bottom	) (XS-Reflex)
$\Delta \vdash B \sqsubset : B''$	$dom(\Delta') \cap fv(\exists \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}$	$(\mathbf{N}) = \emptyset \qquad fv(\overline{\mathbf{T}}) \subseteq dom(\Delta, \Delta')$
$\Delta \vdash B'' \sqsubset : B'$	$\Delta, \Delta' \vdash \overline{[T/X]} B_l <: T$	$\Delta, \Delta' \vdash \overline{\mathtt{T} <: [\overline{\mathtt{T/X}}] \mathtt{B}_u}$
$\Delta \vdash B \sqsubset: B'$	$\Delta \vdash \exists \Delta' . [\overline{T/X}]$	$\mathbb{N} \sqsubseteq : \exists \overline{\mathbb{X} \longrightarrow [\mathbb{B}_l \ \mathbb{B}_u]} . \mathbb{N}$
(XS-TRANS)	(XS	-Env)
Subtypes: $\Delta \vdash B <: B$		
$\frac{\Delta \vdash B \sqsubset : B'}{\Delta \vdash B \triangleleft : B'}$	$\Delta \vdash B <: B'' \qquad \Delta \vdash B'' <:$	$\underline{\mathbf{B}'} \qquad \underline{\mathbf{\Delta}(\mathbf{X}) = [\mathbf{B}_l \ \mathbf{B}_u]}$
$\Delta \mid D \leq D$	$\Delta \mid D \leq D$	$\Delta \vdash \exists \psi \cdot \mathbf{A} \leq \cdot \exists u$ $\Delta \vdash B_l <: \exists \emptyset \cdot X$
(S-SC)	(S-TRANS)	(S-BOUND)
		(u-u00u-u)

Figure 3.2: Tame FJ subclasses, extended subclasses, and subtypes.

# 3.1.2 Subtyping

The subclassing relation ( $\Box$ :, which relates non-existential types, R), reflects the class hierarchy. Subclassing for type variables is restricted to reflexivity as type variables have no place in the subclass hierarchy. Subtyping (<:) extends subclassing by adding subtyping between existential types and between type variables and their bounds. Extended subclassing ( $\Box$ :) is an intermediate relation that expresses the class hierarchy (with the addition of a bottom type) and the behaviour of wildcards and type variables *as type parameters*; extended subclassing is used mainly to simplify the proofs of soundness. The three relations are defined in figure 3.2.

The rule XS-ENV, adapted from Wild FJ [60], gives all the interesting variance properties for wildcard types. It gives a subtype relationship between two existentially quantified class types, where the bounds of the type parameters of the subtype are 'more precise' than those of the supertype. The following relationships are given by this rule, given the class hierarchy described in section 2.1 and using the shorthands described in section 3.1.1:

 $\emptyset \vdash$  Shape  $\Box$ : Shape

- $\emptyset \vdash \texttt{List} < \texttt{Shape} : \exists \texttt{X}.\texttt{List} < \texttt{X} >$
- $\emptyset \vdash \texttt{List} \leq \texttt{Shape} \sqsubset : \exists X \rightarrow [\texttt{Circle Object}] . \texttt{List} \leq X > \texttt{Circle Object}]$
- $\emptyset \vdash \exists X \rightarrow [Circle Shape].List < X > \Box : \exists X \rightarrow [Circle Object].List < X >$
- $\emptyset \vdash \exists X.Pair < X, X > \Box : \exists Y, Z.Pair < Y, Z >$

That the bounds of the type parameters are 'more precise' is expressed through the substitution  $[\overline{T/X}]$ , where  $\overline{X}$  are some of the parameters of the supertype and  $\overline{T}$  are the corresponding parameters in the subtype. The subtype checks in the premises of XS-ENV ensure that  $\overline{T}$  are 'more precise' than  $\overline{X}$ ; that is, that  $\overline{T}$  are within the bounds of  $\overline{X}$ . The first premise ensures that free variables in the supertype cannot be captured in the subtype, thus forbidding erroneous subtypes such as  $\Delta \vdash \exists X.C < X > \Box$ : C < X >. The second premise ensures that variables are not introduced to the subtype which are not bound either in  $\Delta$  or  $\Delta'$ . This is a limited form of well-formedness constraint on the subtype, and is only used in the proof of soundness.

The rule XS-ENV performs existential packing; the subtype is packed into the supertype. XS-ENV operates on types (as in Pizza [70], see section 2.3.7), as opposed to expressions in traditional systems (see FUN-T-UNPACK in section 2.3.2). The quantification of the subtype in XS-ENV (by  $\Delta'$ ) allows for a limited kind of *un*packing in finding a supertype. It is limited since there is no scope for the unpacked type variables to be used except in the bounds checking premises. We use a nested overbar notation in the premises of XS-ENV, for clarity we give the expansion of the third premise,  $\Delta, \Delta' \vdash \overline{[\overline{T/X}]}B_l <: T$ :

$$\begin{split} \Delta, \Delta' \vdash [\mathsf{T}_0/\mathsf{X}_0, \ \mathsf{T}_1/\mathsf{X}_1, \ \ldots, \ \mathsf{T}_n/\mathsf{X}_n] \mathsf{B}_{l0} <: \mathsf{T}_0 \\ \Delta, \Delta' \vdash [\mathsf{T}_0/\mathsf{X}_0, \ \mathsf{T}_1/\mathsf{X}_1, \ \ldots, \ \mathsf{T}_n/\mathsf{X}_n] \mathsf{B}_{l1} <: \mathsf{T}_1 \\ \cdots \\ \Delta, \Delta' \vdash [\mathsf{T}_0/\mathsf{X}_0, \ \mathsf{T}_1/\mathsf{X}_1, \ \ldots, \ \mathsf{T}_n/\mathsf{X}_n] \mathsf{B}_{ln} <: \mathsf{T}_n \end{split}$$

35 links extended subclassing to subclassing.

Lemma 17 (*uBound* refines subtyping) If  $\Delta \vdash T \ll T'$  and  $\vdash \Delta OK^1$  then  $\Delta \vdash uBound_{\Delta}(T) \sqsubset uBound_{\Delta}(T').$ 

This lemma states that if two types are subtypes then their upper bounds are extended subclasses. The *uBound* function (defined in figure 3.7) returns a non-variable type by recursively finding the upper bound of a type until a non-variable type is reached. The interesting cases in the proof are from the S-BOUND rule, where  $T = \exists \emptyset . X$  and  $T' = B_u$ ; then, by the definition of *uBound*, we have that  $uBound(\exists \emptyset . X) = uBound(B_u)$ , and are done by reflexivity. The other S-BOUND sub-case is where  $T = B_l$  and  $T' = \exists \emptyset . X$ , here we use  $\Delta \vdash uBound(B_l) \sqsubset : uBound(B_u)$ from F-ENV and  $uBound(\exists \emptyset . X) = uBound(B_u)$ , again from the definition of uBound. A corollary to this lemma is that any two non-variable types which are subtypes, are also subclasses.

Lemma 35 (Extended subclassing gives subclassing) If  $\Delta \vdash \exists \Delta' . \mathsf{R}' \sqsubset$ :  $\exists \overline{\mathsf{X} \to [\mathsf{B}_l \ \mathsf{B}_u]} . \mathsf{R} \text{ and } \Delta \vdash \text{ ok then there exists } \overline{\mathsf{T}} \text{ where } \vdash \mathsf{R}' \sqsubset : [\overline{\mathsf{T}/\mathsf{X}}] \mathsf{R} \text{ and } \Delta, \Delta' \vdash \overline{\mathsf{T} <: [\overline{\mathsf{T}/\mathsf{X}}]} \mathsf{B}_u} \text{ and } \Delta, \Delta' \vdash [\overline{\mathsf{T}/\mathsf{X}}] \mathsf{B}_l <: \overline{\mathsf{T}} \text{ and } fv(\overline{\mathsf{T}}) \subseteq dom(\Delta, \Delta').$ 

<sup>&</sup>lt;sup>1</sup>Well-formed environments are defined in section 3.1.3.

This lemma states that for any types in an extended subclass relationship, a substitution can be found so that there is a subclass relationship between the subtype and the substituted supertype. The difference between subclassing and extended subclassing is, essentially, the XS-ENV rule. This rule finds an extended subclass of an existential type by substituting away its existentially quantified type variables. This substitution corresponds to the one in the conclusion of the lemma.

Well-formed types:  $\Delta \vdash B \text{ OK}, \Delta \vdash P \text{ OK}, \Delta \vdash R \text{ OK}$ 

$\mathtt{X}\in\Delta$		
$\Delta \vdash \mathtt{X} \text{ ok}$	$\Delta \vdash \perp \ \mathrm{ok}$	$\Delta \vdash \texttt{Object}{<\!\!\!\!\!>} OK$
(F-VAR)	(F-Воттом)	(F-OBJECT)

	$\texttt{class C<\overline{X} \triangleleft T_u} > \ \lhd \ \texttt{N} \ \{\dots\}$	$\Delta\vdash\Delta' \text{ ok}$
	$\Delta \vdash \overline{T} \text{ ok} \qquad \Delta \vdash \overline{T <: [\overline{T/X}] T_u}$	$\Delta, \Delta' \vdash \mathbf{R} \text{ ok}$
$\Delta \vdash \star \text{ ok}$	$\Delta \vdash C < \overline{T} > OK$	$\Delta \vdash \exists \Delta'. \mathbf{R} \text{ ok}$
(F-STAR)	(F-CLASS)	(F-EXIST)

Well-formed type environments:  $|\Delta \vdash \Delta \text{ ok}|$ 

	$\Delta, \mathbf{X} \rightarrow [\mathbf{B}_l  \mathbf{B}_u], \Delta' \vdash \mathbf{B}_l \text{ ok} \qquad \Delta, \mathbf{X} \rightarrow [\mathbf{B}_l  \mathbf{B}_u], \Delta' \vdash \mathbf{B}_u \text{ ok}$
	$\Delta \vdash uBound_{\Delta}(B_l) \sqsubset : uBound_{\Delta}(B_u)$
	$\Delta \vdash B_l <: B_u \qquad \Delta, X \rightarrow [B_l  B_u] \vdash \Delta' \text{ ok}$
$\Delta \vdash \emptyset \text{ ok}$	$\Delta \vdash \mathtt{X} {\rightarrow} [\mathtt{B}_l  \mathtt{B}_u], \Delta' \text{ ok}$
(F-ENV-EMPTY)	$(F-E_{NV})$

Figure 3.3: Tame FJ well-formed types and type environments.

# 3.1.3 Well-formedness

Rules for judging well-formed types and type environments are given in figure 3.3. The rules for well-formed type environments are the most interesting. There are two motivating issues: we must not allow type variables which have upper and lower bounds that are unrelated in the class hierarchy; and we must restrict forward references whilst allowing F-bounds.

The first issue can cause a problem where an environment could judge a subtype relation which

does not reflect the class hierarchy. For example, an environment containing  $Z \rightarrow [Fish Plant]$  could judge (by using rule S-Bound and transitivity) that Fish is a subtype of Plant, which is presumably incorrect. We therefore check that the bounds of a type variable are related by subtyping under an environment without that type variable. We also require the stronger subclass relationship to hold for the upper bounds of the type variable's immediate bounds. This ensures that subtype relationships judged by a well-formed environment respect the class hierarchy. We need this property to prove lemma 17, described in section 3.1.2.

Method typing:  $\Delta \vdash M \text{ OK}$  in C

$$\begin{array}{c} \Delta' = \overline{\mathbb{Y} \rightarrow [\bot \ \mathbb{T}_u]} & \Delta \vdash \Delta' \ \text{OK} & \Delta, \Delta' \vdash \mathbb{T}, \overline{\mathbb{T}} \ \text{OK} \\ \text{class } \mathbb{C} \triangleleft \overline{\mathbb{X} \dots} \geqslant & \triangleleft \mathbb{N} \ \{ \dots \} \\ \Delta, \Delta'; \overline{\mathbb{x}:\mathbb{T}}, \ \text{this}: \exists \emptyset.\mathbb{C} \triangleleft \overline{\mathbb{X}} \succ \vdash \mathbb{e}: \mathbb{T} \mid \emptyset \quad override(\mathbb{m}, \mathbb{N}, \triangleleft \overline{\mathbb{Y}} \rightarrow \overline{\mathbb{T}} \rightarrow \mathbb{T}) \\ \hline \Delta \vdash \triangleleft \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{T}_u} \urcorner \mathbb{T} \ \mathbb{m}(\overline{\mathbb{T}} \mathbb{x}) \ \{ \text{return } \mathbb{e} \} \ \text{OK in } \mathbb{C} \end{array}$$

(T-Method)

 $\begin{array}{l} \underline{mType(\mathtt{m}, \mathtt{N}) = \langle \overline{\mathtt{X} \lhd \hspace{0.5mm} \mathtt{U} > \overline{\mathtt{T}} \rightarrow \mathtt{T}}}_{override(\mathtt{m}, \mathtt{N}, \langle \overline{\mathtt{X} \lhd \hspace{0.5mm} \mathtt{U} > \overline{\mathtt{T}} \rightarrow \mathtt{T})} & \underline{mType(\mathtt{m}, \mathtt{N}) \quad undefined} \\ \hline override(\mathtt{m}, \mathtt{N}, \langle \overline{\mathtt{X} \lhd \hspace{0.5mm} \mathtt{U} > \overline{\mathtt{T}} \rightarrow \mathtt{T})} \\ (\mathtt{T}\text{-} \texttt{OVERRIDE}) & (\mathtt{T}\text{-} \texttt{OVERRIDEUNDEF}) \end{array}$   $\begin{array}{l} \texttt{Class typing:} \quad \left| \vdash \mathtt{Q} \ \texttt{OK} \right| \\ \Delta = \overline{\mathtt{X} \rightarrow [\bot \quad \mathtt{T}_u]} & \emptyset \vdash \Delta \ \texttt{OK} & \Delta \vdash \mathtt{N}, \ \mathtt{T} \ \texttt{OK} & \Delta \vdash \overline{\mathtt{M}} \ \texttt{OK} \ \texttt{in } \mathtt{C} \end{array}$ 

$$\vdash \text{class } \mathbb{C} < \overline{\mathbb{X} \lhd \mathbb{T}_u} > \lhd \mathbb{N} \{ \overline{\texttt{Tf}}; \overline{\mathbb{M}} \} \text{ OK}$$
$$(\text{T-CLASS})$$

Figure 3.4: Tame FJ class and method typing rules.

Forward references are only allowed to occur as *parameters* of the bounding type. In the well-formedness rule, this is addressed by allowing forward references when checking that the bounds are well-formed types, but not when checking the subtype and subclass relationships of the bounds. This reflects Java where (in a class or method declaration) <X < Y, Y < Object> is illegal, due to the forward reference in the bound of X; however, <X < List<Y>, Y < Object> is legal.

# 3.1.4 Typing

Method and class type checking judgements are given in figure 3.4 and are mostly straightforward. The *override* relation allows method overriding, but does not allow overloading.

<b>Expression typing:</b> $\Delta; \Gamma \vdash e : T \mid \Delta$	
	$\Delta \vdash C < \overline{T} > OK$
	$fields(C) = \overline{f}$ $\overline{fType(f, C<\overline{T}>)} = U$
	$\Delta; \Gamma \vdash e: \cup   \emptyset$
$\Delta; \Gamma \vdash \mathbf{x} : \Gamma(\mathbf{x}) \mid \emptyset$	$\Delta; \mathbf{I} \vdash new \ C(e) : \exists \emptyset. C   \emptyset$
(T-VAR)	(T-NEW)
$\Delta; \Gamma \vdash \mathbf{e} : \exists \Delta' . \mathbb{N}     \emptyset$	$\Delta; \Gamma \vdash \mathbf{e} : \mathbf{U}   \Delta' \qquad \Delta, \Delta' \vdash \mathbf{U} <: \mathbf{T}$
$fType(\mathtt{f},\mathtt{N})=\mathtt{T}$	$\Delta \vdash \Delta'$ ok $\Delta \vdash T$ ok
$\overline{\Delta;\Gamma\vdash\texttt{e.f}:\texttt{T} \Delta'}$	$\Delta;\Gamma\vdash e:T \emptyset$
(T-FIELD)	(T-SUBS)
$\Delta; \Gamma \vdash e : \exists \Delta' . \mathbb{N} \mid \emptyset \qquad mT$	$Type(\underline{m},\underline{N}) = \langle \overline{Y} \triangleleft \underline{B} \rangle \overline{U} \to U$
$\Delta \vdash P \text{ ok}  \Delta;$	$\Gamma \vdash \mathbf{e} : \exists \Delta . \mathbf{R} \mid \emptyset$
<i>match(sift</i> (R	, U, Y), P, Y, T)
$\Delta, \Delta', \overline{\Delta} \vdash \mathtt{T} <: [\overline{\mathtt{T/Y}}]$ b	$\Delta, \Delta', \overline{\Delta} \vdash \exists \emptyset. \mathtt{R} <: [\overline{\mathtt{T/Y}}] \mathtt{U}$
$\Delta; \Gamma \vdash \texttt{e.<}\overline{P}\texttt{>}\mathtt{m}(\overline{e}$	$\mathbf{D}$ : $[\overline{\mathbf{T}/\mathbf{Y}}]\mathbf{U} \mid \Delta', \overline{\Delta}$
(T-I:	NVK)

Figure 3.5: Tame FJ expression typing rules.

Type rules for expressions are given in figure 3.5. Auxiliary functions used in typing are given in figures 3.6 and 3.7.

The type checking judgement has the form  $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} \mid \Delta'$ , and should be read as

expression **e** has type **T** under the environments  $\Delta$  and  $\Gamma$ , guarded by environment  $\Delta'$ .

 $\Delta'$  contains variables that have been unpacked from an existential type during type checking and that could escape their scope. These variables are used with  $\Delta$  to judge some premises of a rule. Any free variables in T are bound in either  $\Delta$  or  $\Delta'$ . T-SUBS is an extended subsumption rule: when  $\Delta'$  is empty, it allows an expression to be typed with a supertype of the expression's type in the usual way; when  $\Delta'$  is non-empty, it can be used to remove the guarding environment from the judgement. Type checking of a Tame FJ expression is finished when a type is found using an empty guarding environment (non-empty guarding environments may only occur at intermediate stages in the derivation tree). This ensures that no bound type variables escape the scope in which they are unpacked. The scope covers the conclusions, some premises, and the derivations of these premises in the type rule in which the variables are unbound. For example when  $\Delta'$  is empty, if x has type Square, we can derive that x has type Shape:

 $\begin{array}{c|c} \Delta; \Gamma \vdash \mathtt{x} : \texttt{Square} \mid \emptyset & \Delta \vdash \texttt{Square} <: \texttt{Shape} \\ \hline \Delta \vdash \emptyset \text{ ok} & \Delta \vdash \texttt{Shape ok} \\ \hline \Delta; \Gamma \vdash \mathtt{x} : \texttt{Shape} \mid \emptyset \\ & (\text{T-SUBS}) \end{array}$ 

In the next example we have a non-empty  $\Delta'$ , we elide bounds for clarity; note that  $\Delta, X \vdash$ List<X> <:  $\exists Z.List<Z>$  by XS-ENV and S-SC:

 $\begin{array}{c|c} \Delta; \Gamma \vdash \mathtt{x} : \texttt{List}<\mathtt{X} > | \, \mathtt{X} & \Delta, \mathtt{X} \vdash \texttt{List}<\mathtt{X} > <: \exists \mathtt{Z}.\texttt{List}<\mathtt{Z} > \\ & \Delta \vdash \mathtt{X} \text{ OK} & \Delta \vdash \exists \mathtt{Z}.\texttt{List}<\mathtt{Z} > \text{ OK} \\ \hline & \Delta; \Gamma \vdash \mathtt{x} : \exists \mathtt{Z}.\texttt{List}<\mathtt{Z} > | \, \emptyset \\ & (T-SUBS) \end{array}$ 

Typing of variables and 'new' expressions is done in the usual way. The lookup function *fields* returns a sequence of the field names in a class, and fType takes a field and a class type and returns the field's type.

Type checking field access and method invocation expressions follows similar patterns: subexpressions are type checked and their types are unpacked, then some work is done using these unpacked types, and a result type is found. The rule T-SUBS may then be used to find a final result type that does not require a guarding environment. Auxiliary Functions:  $uBound_{\Delta}(B)$  and  $match(\overline{R}, \overline{U}, \overline{P}, \overline{Y}, \overline{T})$  and  $sift(\overline{R}, \overline{U}, \overline{Y})$ 

$$uBound_{\Delta}(\mathbf{B}) = \begin{cases} uBound_{\Delta}(\mathbf{B}_u), & \text{if } \mathbf{B} = \exists \emptyset . \mathsf{X} \text{ and } \Delta(\mathsf{X}) = [\mathbf{B}_l \ \mathbf{B}_u] \\ \mathbf{B}, & \text{if } \mathbf{B} = \exists \emptyset . \mathsf{X} \end{cases}$$

$$\forall j \ where \ \mathsf{P}_{j} = \star : \mathsf{Y}_{j} \in fv(\overline{\mathsf{R}'}) \qquad \forall i \ where \ \mathsf{P}_{i} \neq \star : \mathsf{T}_{i} = \mathsf{P}_{i} \\ \vdash \overline{\mathsf{R}} \boxplus : [\overline{\mathsf{T}/\mathsf{Y}}, \overline{\mathsf{T}'/\mathsf{X}}] \mathsf{R'} \\ dom(\overline{\Delta}) = \overline{\mathsf{X}} \qquad fv(\overline{\mathsf{T}}, \overline{\mathsf{T}'}) \cap \overline{\mathsf{Y}}, \overline{\mathsf{X}} = \emptyset \\ match(\langle \overline{\mathsf{R}}, \overline{\exists \Delta . \mathsf{R}'} \rangle, \overline{\mathsf{P}}, \overline{\mathsf{Y}}, \overline{\mathsf{T}})$$

$$\begin{array}{c} X \in \overline{Y} \\ \hline \textit{sift}((R,\overline{R}), \ (\exists \emptyset. X, \overline{U}), \ \overline{Y}) = \operatorname{sift}(\overline{R}, \ \overline{U}, \ \overline{Y}) \end{array}$$

$$\begin{array}{ccc} \mathbf{X} \not\in \overline{\mathbf{Y}} & sift(\overline{\mathbf{R}}, \ \overline{\mathbf{U}}, \ \overline{\mathbf{Y}}) = (\overline{\mathbf{R}'}, \ \overline{\mathbf{U}'}) \\ \hline sift((\mathbf{R}, \overline{\mathbf{R}}), \ (\exists \emptyset . \mathbf{X}, \overline{\mathbf{U}}), \ \overline{\mathbf{Y}}) = \langle (\mathbf{R}, \overline{\mathbf{R}'}), \ (\exists \emptyset . \mathbf{X}, \overline{\mathbf{U}'}) \rangle \end{array}$$

	$sift(\overline{\mathtt{R}}, \ \overline{\mathtt{U}}, \ \overline{\mathtt{Y}}) = (\overline{\mathtt{R}'}, \ \overline{\mathtt{U}'})$
$sift(\emptyset, \ \emptyset, \ \overline{\mathbf{Y}}) = \langle \emptyset, \ \emptyset \rangle$	$sift((\mathbf{R},\overline{\mathbf{R}}), \ (\exists \Delta . \mathbf{N}, \overline{\mathbf{U}}), \ \overline{\mathbf{Y}}) = \langle (\mathbf{R}, \overline{\mathbf{R}'}), \ (\exists \Delta . \mathbf{N}, \overline{\mathbf{U}'}) \rangle$

Figure 3.6: Auxiliary functions for Tame FJ.

In the following paragraphs we describe unpacking and packing, descriptions of type checking using T-FIELD and T-INVK, and give examples.

Unpacking an existential type  $(\exists \Delta . R)$  entails separating the environment  $(\Delta)$  from the quantified type (R).  $\Delta$  can be used to judge premises of a rule and must be added to the guarding environment in the rule's conclusion. R can be used without quantification in the rule; bound type variables in R will now be free, we must take care that these do not escape the scope of the type rule.

If the result of type checking an expression contains escaping type variables (indicated by a non-empty guarding environment), then we must find a supertype (using T-SUBS) in which there are no free variables, and use this as the expression's type. An escaping type variable may be the whole type or a type parameter. If the escaping type variable is a type parameter (e.g., X in C<X>), the type may be packed to an existential type (e.g.,  $\exists X.C<X>$ ) using the subtyping

**Lookup Functions** 

	class C< $\overline{X \triangleleft T_u}$ > $\triangleleft$ D<> { $\overline{Uf}$ ; $\overline{M}$ } $fields(D) = \overline{g}$
$fields(\texttt{Object}) = \emptyset$	$fields(C) = \overline{g}, \overline{f}$
$\texttt{class C} < \overline{X \lhd T_u} > \ \lhd \ \texttt{N} \ \{ \overline{\texttt{Uf; M}} \} \qquad \texttt{f} \not\in \overline{\texttt{f}}$	class C< $\overline{X} \lhd T_u$ > $\lhd$ N {Uf; $\overline{M}$ }
$fType(\texttt{f},\texttt{C}{<}\overline{\texttt{T}}{>}) = fType(\texttt{f},[\overline{\texttt{T/X}}]\texttt{N})$	$\overline{fType(\mathbf{f}_i, \mathbf{C} < \overline{\mathbf{T}} >) = [\overline{\mathbf{T} / \mathbf{X}}] \mathbf{U}_i}$
$\begin{array}{c c} \texttt{class } \mathbb{C} < \overline{\mathtt{X} \lhd \mathtt{T}_u} > \lhd \mathtt{N} \ \{ \overline{\mathtt{Uf}}; \ \overline{\mathtt{M}} \} & \mathtt{m} \not\in \overline{\mathtt{M}} \\ \hline mBody(\mathtt{m}, \mathtt{C} < \overline{\mathtt{T}} >) = mBody(\mathtt{m}, [\overline{\mathtt{T}/\mathtt{X}}] \mathtt{N}) \end{array}$	$\begin{array}{c} \begin{array}{c} \texttt{class } \mathbb{C} < \overline{\mathbb{X} \lhd  \mathbb{T}_u } > \ \lhd \ \mathbb{N} \ \left\{ \overline{\mathbb{U}'  \texttt{f}  ;} \ \overline{\mathbb{M}} \right\} \\ \hline < \overline{\mathbb{Y} \lhd  \mathbb{T}'_u } > \ \mathbb{U}  \texttt{m}  (\overline{\mathbb{U}  \texttt{x}}) \ \left\{ \texttt{return } \mathbf{e}_0  ; \right\} \in \overline{\mathbb{M}} \\ \hline \\ \hline mBody(\texttt{m}, \mathbb{C} < \overline{\mathbb{T}} >) = \left( \overline{\texttt{x}} ; \ [\overline{\mathbb{T} / \mathbb{X}}]  \mathbf{e}_0 \right) \end{array}$
$\begin{array}{c c} \texttt{class } C < \overline{X \lhd T_u} \texttt{>} \ \lhd \ N \ \{ \overline{Uf} \texttt{;} \ \overline{M} \} & \texttt{m} \not\in \overline{M} \\ \hline mType(\texttt{m}, C < \overline{T} \texttt{>}) = mType(\texttt{m}, [\overline{T}/X] N) \end{array}$	$\begin{array}{c c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{$



rule XS-ENV. If the escaping type variable is the whole type, i.e.,  $\exists \emptyset . X$ , then the upper bound of X can be used as the result type by using S-BOUND.

#### Field access

In T-FIELD, the fType function applied to the unpacked type (N) of the receiver gives the type of the field (T). Because T may contain type variables bound in the environment  $\Delta'$ , the judgement must be guarded by  $\Delta'$ .

#### Example — Field access

The following example of the derivation of a type for a field access expression demonstrates the sequence of unpacking, finding the field type, and finding a supertype that does not contain free variables. In the example (figure 3.8), the type labelled 1 is unpacked to 2. The type labelled 3 would escape its scope, but its supertype (4) has no free variable and so may be used as the

$ \begin{split} \emptyset; \Gamma \vdash \mathbf{x} : \exists \mathbf{X} {\rightarrow} [ \bot ~ \texttt{Shape}] ~\texttt{.TreeNode} {<} \mathbf{X} {>}^1     \emptyset \\ fType(\texttt{datum}, \texttt{TreeNode} {<} \mathbf{X} {>}^2) = \mathbf{X}^3 \end{split} $	$\emptyset, \mathtt{X} { ightarrow} [ot ] dash \mathtt{X}^3 <: \mathtt{Shape}^4$
$\emptyset; \Gamma \vdash \texttt{x.datum} : \texttt{X}^3 \mid \texttt{X} \rightarrow [\bot \text{ Shape}]^2$	$\emptyset \vdash X \longrightarrow [\bot$ Shape] OK
(T-FIELD)	$\emptyset \vdash \mathtt{Shape}^4$ OK
$\emptyset; \Gamma \vdash \texttt{x.datum}: \texttt{S}$	$\mathtt{hape}^4   \emptyset$
(T-Subs)	

result type. We assume that the TreeNode<Y> class declaration has a field datum with type Y and that  $\Gamma = x: \exists X \rightarrow [\bot \text{ Shape}]$ . TreeNode<X>.

Figure 3.8: Example of a derivation for field access.

#### Method Invocation

In T-INVK, function mType applied to the unpacked type (N) of the receiver gives the method's signature,  $\langle \overline{Y} \triangleleft | \overline{B} \rangle \overline{U} \rightarrow U$ . We use the unpacked types ( $\overline{R}$ ) of the actual parameters and the *match* function to infer any 'missing' (actual) type parameters (denoted by  $\star$  in our syntax, following Wild FJ). The (possibly inferred) actual type parameters are substituted for formal parameters ( $[\overline{T/Y}]$ ) in the method's type signature. After substitution, the actual type parameters ( $\overline{T}$ ) must be within the formal bounds ( $\overline{B}$ ), and the types of the actual parameters must be subtypes of the types of the formal parameters ( $\overline{U}$ ). These checks are performed under the type environment  $\Delta, \Delta', \overline{\Delta}$ . Similarly to T-FIELD, we must guard the conclusion of the type rule with the environments extracted by unpacking ( $\Delta', \overline{\Delta}$ ).

The substitution [T/Y] is determined using the types of actual  $(\overline{R})$  and formal parameters  $(\overline{U})$ . These types are filtered using the *sift* function before being passed to *match*. This ensures that where the type of a formal parameter is one of the formal type parameters  $(U_i \in \overline{Y})$ , the formals and actuals at this position are not used for inference. Hence, we only infer the value of a type variable based on its usage as a type parameter in the formal type of a value argument. The *match* relation takes five arguments, but the first two are passed as a pair to facilitate the use of *sift*. For example,

 $sift(\{\texttt{M},\texttt{N},\texttt{O},\texttt{P},\texttt{Q}\},\{\texttt{C}<\texttt{X}>,\texttt{C}<\texttt{Y}>,\exists\texttt{X}.\texttt{C}<\texttt{X}>,\texttt{X},\texttt{Z}\},\{\texttt{Y},\texttt{Z}\})=\langle\{\texttt{M},\texttt{N},\texttt{O},\texttt{P}\},\{\texttt{C}<\texttt{X}>,\texttt{C}<\texttt{Y}>,\exists\texttt{X}.\texttt{C}<\texttt{X}>,\texttt{X}\}\rangle$ 

Type parameter inference is done using the *match* relation (figure 3.6). All formal type parameters ( $\overline{\mathbf{Y}}$ ) are substituted by types  $\overline{\mathbf{T}}$ . These types are either given explicitly, or are inferred if left unspecified (i.e., marked with  $\star$ ). The first premise of *match* ensures that any unspecified type parameter can be inferred, i.e., it appears as a type parameter in a type of at least one of the method's formal value parameters. The second premise ensures that each specified type parameter is used in the returned sequence. The remaining premises find a substitution that allows subclassing between the formal and actual parameter types. Part of this substitution will be the substitution of actual type parameters for formals, and these actual type parameters are  $\overline{\mathbf{T}}$ . The remainder ( $\overline{\mathbf{T}}$ ) account for existentially quantified type variables in the formal parameter types. These are forgotten, since in T-INVK we use full subtyping which allows us to use the XS-ENV rule to fulfil the same role. For example (assuming that B<X> is declared as a subclass of C<X>):

$$\begin{split} \textbf{X}, \textbf{Z} \in fv(\{\textbf{C}, \textbf{C}, \exists \textbf{U}. \textbf{D}\}) & \textbf{A} = \textbf{A} \\ & \vdash \textbf{B} \boxplus: [\textbf{V/X}, \textbf{A/Y}, \textbf{B/Z}, \textbf{A/U}]\textbf{C} \\ & \vdash \textbf{C} \boxplus: \[\textbf{V/X}, \textbf{A/Y}, \textbf{B/Z}, \textbf{A/U}\]\textbf{C} \\ & \vdash \textbf{D} \boxplus: \[\textbf{V/X}, \textbf{A/Y}, \textbf{B/Z}, \textbf{A/U}\]\textbf{D} \\ & dom\(\{\textbf{U}\}\) = \{\textbf{U}\} \quad fv\(\{V, A, B\}, \{A\}\) \cap \{\textbf{X}, \textbf{Y}, \textbf{Z}\}, \{\textbf{U}\} = \emptyset \\ \hline match\(\langle\{\textbf{B}, \textbf{C}, \textbf{D}\}, \{\textbf{C}, \textbf{C}, \exists \textbf{U}. \textbf{D} \}\rangle, \{\textbf{X}, \textbf{Y}, \textbf{Z}\}, \{\textbf{X}, \textbf{Y}, \textbf{Z}\}, \{\textbf{V}, \textbf{A}, \textbf{B}\}\\) \end{split}$$

#### Examples — Method invocation

*Example 1* from figure 2.5 demonstrates method invocation with a simple case of wildcard capture. The existential type  $\exists Z.Tree < Z >$  is unpacked to Tree < Z >, and Z is inferred and substituted for X. The return type (List < Z >) is then packed to the existential type  $\exists Z.List < Z >$ . We show how the example can be type checked using the T-INVK and T-SUBS rules (the bounds of type variables are omitted for clarity) in figure 3.9; the type labelled 1 is unpacked to 2 and the type labelled 3 is packed to 4. We omit from the derivation tree the call to *sift* for clarity, note that *sift*(Tree < Z ><sup>2</sup>, Tree < X >, X) = (Tree < Z ><sup>2</sup>, Tree < X >). In this example  $\Gamma = \{\text{this:C}, y: \exists Z.Tree < Z >\}.$ 

Example 2 from figure 2.6 expresses types that cannot be denoted using Java syntax. Using the

$\begin{array}{l} \emptyset; \Gamma \vdash \texttt{this} : \texttt{C}     \emptyset \\ mType(\texttt{walk},\texttt{C}) = \texttt{} \ \texttt{Tree}\texttt{} \rightarrow \texttt{List}\texttt{} \\ \emptyset; \Gamma \vdash \texttt{y} : \exists \texttt{Z}. \texttt{Tree}\texttt{}^1     \emptyset \end{array}$		
$match(\langle Tree < Z >^2, Tree < X > \rangle, \star, X, Z^2)$ $Z^2 \vdash Tree < Z >^2 <: Tree < Z >$	$Z^2 \vdash List < Z >^3 <: \exists Z.List < Z >^4$	
$\emptyset; \Gamma \vdash \texttt{this.<} \texttt{*}\texttt{valk}(\texttt{y}) : \texttt{List} \texttt{Z}^3   \texttt{Z}^2$	$\emptyset \vdash Z^2$ ok	
(T-Invk)	$\emptyset \vdash \exists \mathtt{Z.List} < \mathtt{Z} >^4 \mathrm{OK}$	
$\emptyset; \Gamma \vdash \texttt{this.<} \texttt{*} \texttt{valk}(\texttt{y}) : \exists \texttt{Z.List} \texttt{<} \texttt{Z} \texttt{>}^4     \emptyset$		
(T-SUBS)		

Figure 3.9: Example of a derivation for method invocation.

syntax of existential types, it becomes clear why type checking fails at 1 (figure 3.10). Namely, for the expression at 1 to be type correct, a T would need to be found so that

 $match(\langle Pair < U, V >, Pair < X, X > \rangle, \star, X, T)$ 

From the definition of *match* we see that T would have to satisfy  $\vdash$  Pair<U, V>  $\boxplus$ : [T/X]Pair<X, X>; no such T exists, and hence *match*ing, and thus type checking, fails.

Figure 3.10: Example 2 in Tame FJ.

### **Type Inference**

As is usual with formal type systems, we consider type inference to be performed in a separate phase before type checking. Due to the presence of existential types, some inferred type parameters cannot be named and are marked with  $\star$ . These parameters must be inferred during type checking. In T-INVK we only allow the inference of types where they are used as parameters to an actual parameter type (e.g., X in <X>void m(Tree<X> x)...). This is enforced by the *sift* function (defined in figure 3.6), which excludes pairs of actual and formal parameter types where the formal parameter type is a formal type variable of the method.

## 3.1.5 Operational Semantics



Figure 3.11: Tame FJ reduction rules.

The operational semantics of Tame FJ is defined in figure 3.11. Most rules are simple and similar to those in FGJ. The interesting rule is R-INVK, which requires actual type parameters which do not include  $\star$ , these are found using the *match* relation. Avoiding the substitution of  $\star$  for a formal type variable in the method body prevents the creation of invalid expressions, such as **new C**<\*>(). Since we are dealing only with values when using this rule, there will be no existential types and so all type parameters *could* be specified. However, there is no safe way to substitute the appropriate types for  $\star$ s during execution because each  $\star$  may mark a different type. In this rule, *mBody* (defined in figure 3.7) is used to lookup the body (an expression) and the formal parameters of the method.
### 3.1.6 Type Soundness

We show type soundness for Tame FJ by proving progress and subject reduction theorems (see section 2.1.3), stated below. We prove these with empty environments since, at runtime, variables and type variables should not appear in expressions. A non-empty guarding environment is required in the statement of the progress theorem, because we use structural induction over the type rules; if this environment were empty, the inductive hypothesis could not be applied in the case of T-SUBS.

In the remainder of this section, we summarise some selected lemmas. All lemmas used in the proof of these theorems are stated, along with the proofs of some interesting lemmas, in appendix A. Full proofs of all lemmas can be downloaded from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron\_ecoop08\_full.pdf

**Theorem 1 (Progress)** For any  $\Delta$ , e, T, if  $\emptyset$ ;  $\emptyset \vdash e : T \mid \Delta$  then either  $e \rightsquigarrow e'$  or there exists a v such that e = v.

**Theorem 2 (Subject Reduction)** For any e, e', T, if  $\emptyset; \emptyset \vdash e : T \mid \emptyset$  and  $e \rightsquigarrow e'$ then  $\emptyset; \emptyset \vdash e' : T \mid \emptyset$ .

To prove these two theorems, 40 supporting lemmas are required. These establish 'foundational' properties of the system, properties of substitution, properties of subtyping and subclassing (discussed in section 3.1.2), which functions and relations always give well-formed types, and properties specific to each case of subject reduction and progress. Two of the most interesting lemmas concern the *match* relation:

Lemma 36 (Subclassing preserves *matching* (receiver))

 $If \quad \Delta \vdash \exists \Delta_1 . \mathbb{N}_1 \sqsubset : \exists \Delta_2 . \mathbb{N}_2$ and  $mType(m, \mathbb{N}_2) = \langle \overline{\mathbb{Y}_2 \to [\mathbb{B}_{2l} \ \mathbb{B}_{2u}]} \rangle \overline{\mathbb{U}_2} \to \mathbb{U}_2$ and  $mType(m, \mathbb{N}_1) = \langle \overline{\mathbb{Y}_1 \to [\mathbb{B}_{1l} \ \mathbb{B}_{1u}]} \rangle \overline{\mathbb{U}_1} \to \mathbb{U}_1$ and  $match(sift(\overline{\mathbb{R}}, \overline{\mathbb{U}_2}, \overline{\mathbb{Y}_2}), \overline{\mathbb{P}}, \overline{\mathbb{Y}_2}, \overline{\mathbb{T}})$ and  $\emptyset \vdash \Delta \text{ OK and } \Delta, \Delta' \vdash \overline{\mathbb{T}} \text{ OK}$ then  $match(sift(\overline{\mathbb{R}}, \overline{\mathbb{U}_1}, \overline{\mathbb{Y}_1}), \overline{\mathbb{P}}, \overline{\mathbb{Y}_1}, \overline{\mathbb{T}})$  Lemma 37 (Subclassing preserves *matching* (arguments))

$$\begin{array}{rcl} If & \Delta \vdash \overline{\exists \Delta_1 . \mathtt{R}_1 \sqsubset : \exists \Delta_2 . \mathtt{R}_2} \\ and & match(sift(\overline{\mathtt{R}_2}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}), \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}}) \\ and & fv(\overline{\mathtt{U}}) \cap \overline{\mathtt{Z}} = \emptyset \text{ and } \overline{\Delta_2} = \overline{\mathtt{Z}} \rightarrow [\mathtt{B}_l \ \mathtt{B}_u] \\ and & \emptyset \vdash \Delta \ \mathtt{OK} \\ and & \Delta \vdash \overline{\exists \Delta_1 . \mathtt{R}_1} \ \mathtt{OK} \\ and & \Delta \vdash \overline{\mathtt{P}} \ \mathtt{OK} \end{array}$$

$$then there exists \quad \overline{\mathtt{U}'}$$

$$\begin{array}{lll} such \ that & match(sift(\overline{\mathbf{R}_{1}},\overline{\mathbf{U}},\overline{\mathbf{Y}}),\overline{\mathbf{P}},\overline{\mathbf{Y}},\overline{[\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{T}}) \\ and & \Delta,\overline{\Delta_{1}}\vdash\overline{\mathbf{U}'<:[\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{B}_{u}} \\ and & \Delta,\overline{\Delta_{1}}\vdash\overline{[\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{B}_{l}}<:\mathbf{U}' \\ and & \vdash\overline{\mathbf{R}_{1}}\boxplus:[\overline{[\mathbf{U}'/\mathbf{Z}]}\mathbf{R}_{2} \\ and & fv(\overline{\mathbf{U}'})\subseteq\Delta,\overline{\Delta_{1}} \end{array}$$

Lemma 36 states that if *match* succeeds with the formal parameter types of a superclass, then *match* will succeed where the formal parameter types are taken from the (extended) subclass (and the other arguments remain unchanged). Since overriding methods must have the same parameter types and formal type variables as the methods they override, the proof should be straightforward. However, it is complicated by extended subclassing of existential types; for example, if a method **m** is declared to have a parameter with type Z in the class declaration of class C<Z< Object>, then the type of **m**'s formal parameter will have type X when looked up in  $\exists X.C<X>$  and A in C<A>. Type X may not be a subtype of A, even if C<A> is an extended subclass of  $\exists X.C<X>$ . We show in the proof that such issues do not affect  $\overline{T}$ , because these types are found only from the actual parameter types of the method call.

Lemma 37 performs a similar duty, but for the types of the actual parameters. The conclusion defines a 'valid' substitution which is given by lemma 35 (see section 3.1.2). The types  $\overline{T}$  in *match* are found from the actual parameter types and so, in contrast to lemma 36, these types are affected by the substitution in the conclusion of the lemma.

### Lemma 31 (Inversion Lemma (object creation))

If  $\Delta; \Gamma \vdash \text{new } C < \overline{T} > (\overline{e}) : T \mid \Delta'$ 

 $\begin{array}{ll} then & \Delta' = \emptyset \\ and & \Delta \vdash \mathsf{C} < \overline{\mathsf{T}} > \text{ OK and } fields(\mathsf{C}) = \overline{\mathsf{f}} \\ and & \overline{fType(\mathsf{f},\mathsf{C} < \overline{\mathsf{T}} >)} = \mathsf{U} \\ and & \Delta; \Gamma \vdash \overline{\mathsf{e}}: \mathsf{U} \mid \emptyset \\ and & \Delta \vdash \exists \emptyset. \mathsf{C} < \overline{\mathsf{T}} > <: \mathsf{T} \end{array}$ 

### Lemma 33 (Inversion Lemma (method invocation))

If  $\Delta; \Gamma \vdash e. \langle \overline{P} \rangle m(\overline{e}) : T \mid \Delta'$ and  $\emptyset \vdash \Delta \text{ OK}$ and  $\Delta \vdash \Delta' \text{ OK}$ and  $\forall \mathbf{x} \in dom(\Gamma) : \Delta \vdash \Gamma(\mathbf{x}) \text{ OK}$ 

```
then there exists
                                                         \Delta_n
                      such that
                                                        \Delta', \Delta_n = \Delta'', \Delta
                                                         \Delta \vdash \Delta', \Delta_n ok
                                       and
                                                         \Delta; \Gamma \vdash \mathbf{e} : \exists \Delta'' . \mathbb{N} \mid \emptyset
                                       and
                                                        mType(\mathbf{m}, \mathbf{N}) = \langle \overline{\mathbf{Y} \triangleleft} \ \overline{\mathbf{B}} \rangle \overline{\mathbf{U}} \to \mathbf{U}
                                       and
                                                        \Delta; \Gamma \vdash e : \exists \Delta . R \mid \emptyset
                                       and
                                                        match(sift(\overline{R}, \overline{U}, \overline{Y}), \overline{P}, \overline{Y}, \overline{T})
                                       and
                                                         \Delta \vdash \overline{P} \text{ ok}
                                       and
                                                         \Delta, \Delta'', \overline{\Delta} \vdash \mathsf{T} <: [\overline{\mathsf{T/Y}}]\mathsf{B}
                                       and
                                                         \Delta, \Delta'', \overline{\Delta} \vdash \exists \emptyset. \mathbb{R} <: [\overline{\mathbb{T}/\mathbb{Y}}] \mathbb{U}
                                       and
                                                         \Delta, \Delta'', \Delta_n \vdash [\overline{\mathsf{T/Y}}] \mathsf{U} <: \mathsf{T}
                                       and
```

The formulation of the inversion lemmas is made more interesting by the presence of the guarding environment ( $\Delta'$ ) in the typing judgement ( $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} | \Delta'$ ). In the case of object creation (lemma 31) we show that the guarding environment must be empty. Intuitively, this is because no existential types may be unpacked in the application of T-NEW, and T-SUBS can only shrink the guarding environment, but not add to it. This property of object creation is used heavily in the proof of subject reduction since values in Tame FJ are object creation expressions.

Method invocation is more complex; the guarding environment of T-INVK is formed from the environments unpacked from the types of the receiver and arguments, but these may be repacked by applying T-SUBS. The conclusion of lemma 33 is that there exists some environment,

 $\Delta_n$ , which, when concatenated with  $\Delta'$  will be equal to the unpacked environments from the receiver and arguments.

#### Alpha conversion and Barendregt's variable convention

As well as the standard use of alpha conversion to rename bound variables in existential types, we also need to be able to rename type variables in the guarding environment, as in the following lemma:

### Lemma 7 (Alpha renaming of guarding environments)

 $If \Delta; \Gamma \vdash \mathbf{e} : \mathsf{T} \mid \overline{\mathsf{X} \to [\mathsf{B}_l \ \mathsf{B}_u]} \text{ and } \overline{\mathsf{Y}} \text{ are fresh, then } \Delta; \Gamma \vdash \mathbf{e} : [\overline{\mathsf{Y}/\mathsf{X}}] \mathsf{T} \mid \mathsf{Y} \to [[\overline{\mathsf{Y}/\mathsf{X}}] \mathsf{B}_l \ [\overline{\mathsf{Y}/\mathsf{X}}] \mathsf{B}_u].$ 

Lemma 7 guarantees that we can rename variables in  $\Delta'$  and T and preserve typing. Thus, the guarding environment can be thought of as binding its type variables; the scope of the binding is T, the result of type checking. Note that we do not need to rename types in e. This is because any type variables in the domain of the guarding environment ( $\overline{X}$ ) come from unpacked existential types, and so cannot be explicitly named in the expression syntax; instead they would be marked with  $\star$ .

In order to reduce the number of places where we need to apply alpha conversion in our proofs, we make use of Barendregt's variable convention [10]; i.e., we assume that bound and free variables are distinct. For example, consider the proof of lemma 2 (appendix A):

Lemma 2 (Substitution preserves matching) If  $match(\overline{\mathbb{R}}, \overline{\exists \Delta . \mathbb{R}'}, \overline{\mathbb{P}}, \overline{\mathbb{Y}}, \overline{\mathbb{U}})$  and  $(\overline{\mathbb{X}} \cup fv(\overline{\mathbb{T}})) \cap \overline{\mathbb{Y}}) = \emptyset$  then  $match(\overline{[\overline{\mathbb{T}/\mathbb{X}}]}\mathbb{R}, \overline{[\overline{\mathbb{T}/\mathbb{X}}]}\exists \Delta . \mathbb{R}', \overline{[\overline{\mathbb{T}/\mathbb{X}}]}\mathbb{P}, \overline{\mathbb{Y}}, \overline{[\overline{\mathbb{T}/\mathbb{X}}]}\mathbb{U}).$ 

We reach a point in the proofs where we have shown that

 $\vdash [\overline{\mathsf{T/X}}] \, \mathsf{R} \boxplus : [\overline{\mathsf{T/X}}] \, [\overline{\mathsf{U/Y}}, \overline{\mathsf{U'/Z}}] \, \mathsf{R'} dom(\overline{\Delta}) = \overline{\mathsf{Z}}(\overline{\mathsf{X}} \cup fv(\overline{\mathsf{T}})) \cap \overline{\mathsf{Y}}) = \emptyset$ 

we wish to show

and for this we require that  $\overline{Z}$  are not free in  $\overline{T}$ . We could have used alpha conversion on  $\exists \Delta . \mathbb{R}'$  to accomplish this; however, this would have required extensive renaming throughout the proof. Instead, we use the variable convention and assume that  $\overline{Z}$  are fresh at the point of becoming free and we can proceed with an elegant proof.

The use of Barendregt's variable convention is not always safe [91]. Sufficient conditions are that all rules are *equivariant* and that any binders in a rule do not appear free in that rule's conclusion [91]. A rule is equivariant if any variables in the rule can be substituted (respecting scope) by other variables and the rule remains valid. This requirement is easily satisfied by most sensible inference rules, and is true of all rules in Tame FJ. Binders in Tame FJ are the variables in quantified environments. The interesting rule is XS-ENV, we must ensure that none of  $\overline{X}$  or the type variables in the domain of  $\Delta'$  appear free in  $\exists \Delta'$ .  $[\overline{T/X}]N$  or  $\exists \overline{X} \rightarrow [B_l \ B_u] .N$ . This is ensured by the first two premises of XS-ENV. Therefore, Tame FJ satisfies the stated conditions and using Barendregt's convention *is* safe.

### 3.2 Translating Java to Tame FJ

In this section we describe a possible translation from the Java subset which accommodates wildcards into Tame FJ. Such a translation would consist of two phases. In the first phase, local type inference [16], as performed by Java compilers, would decorate calls of any polymorphic method with explicit type arguments, using  $\star$  where the arguments are wildcards. Wherever a type parameter cannot be inferred (because it is existentially quantified), a  $\star$  can be used to give a correct Tame FJ program. After this phase, the only remaining task, is to map Java types to Tame FJ types. We give a few examples of translating (using  $\mathcal{E}$ ) method calls to show how  $\star$  s are introduced, assuming that m in the type of x has type  $\langle U \rangle . U \rightarrow void$ ,  $n:\langle U \rangle . C \langle U \rangle U \rightarrow void$ , and  $z: C \langle ? \rangle$ ,

$$\mathcal{E}(x.m("hello")) = x.m("hello")$$
$$\mathcal{E}(x.m("hello")) = x.m("hello")$$
$$\mathcal{E}(x.n(new C())) = x.<*>n(new C())$$
$$\mathcal{E}(x.n(z)) = x.<*>n(z)$$

We work in a setting where we expect the first phase to have happened. Here we describe the second phase, and define it in figure 3.13. In figure 3.12 we give the syntax of the relevant subset of Java types, which are also those of Wild FJ.

$\mathbb{N}_s$	::=	$C < \overline{T_s} >$	Java class types
$T_s$	::=	$C < \overline{P_s} >   X$	Java types
$P_s$	::=	$T_s \mid ? \mid ? \lhd T_s \mid ? \vartriangleright T_s$	Java type parameters

Figure 3.12: Syntax of Java types.

The second phase is defined in terms of the functions  $\mathcal{T}$ ,  $\mathcal{P}$ , and  $\mathcal{M}$ , where  $\mathcal{T}$  translates Java types to Tame FJ types;  $\mathcal{P}$  translates a type parameter to an environment and a Tame FJ type; and  $\mathcal{M}$  gives the minimal types out of two. The function  $\mathcal{T}$  maps each occurrence of a wildcard, ?, in a Java type onto an existentially quantified type variable. To do this, it uses the function  $\mathcal{P}$ , which maps any Java type onto an environment and a Tame FJ type.  $\mathcal{T}$  uses the collected environments to create an existential type, using the  $\mathcal{M}$  function to find the appropriate upper bounds, and replaces each type argument by its image through  $\mathcal{P}$ . Note that, in order to reduce the notational complexity, the translation of non-wildcard type parameters introduces a type variable which is never used; this is harmless.

We now highlight some of the finer points of the translation in terms of examples.

A wildcard that occurs as a type parameter is replaced by a quantified type variable. Bounds on the wildcard become bounds on the quantifying type variable. Where bounds are not given we use  $\exists \emptyset. \texttt{Object}$  as the default upper bound and  $\bot$  as the default lower bound. For instance,  $C <? \lhd \texttt{Shape}$  is translated to  $\exists X \rightarrow [\bot \exists \emptyset. \texttt{Shape}] . C < X >$ , and the translation of  $C <? \triangleright \texttt{Shape} >$ 

$$\begin{array}{c|c} \texttt{class } \mathsf{C} < \overline{\mathsf{X}} \lhd \ \overline{\mathsf{T}_s} > \dots & \overline{\mathcal{P}_\Delta}(\mathsf{P}_s) = (\mathsf{Y} \to [\mathsf{U}_s \ \mathsf{U}'_s], \mathsf{T}) \\ \hline \mathcal{T}_\Delta(\mathsf{C} < \overline{\mathsf{P}_s} >) = \exists \overline{\mathsf{Y}} \to & [\mathcal{T}_\Delta(\mathsf{U}_s) \ \mathcal{M}_\Delta(\mathcal{T}_\Delta(\mathsf{U}'_s), [\overline{\mathsf{Y}}/\overline{\mathsf{X}}] \ \mathcal{T}_\Delta(\mathsf{T}_s))] & . \ \mathsf{C} < \overline{\mathsf{T}} > \end{array}$$

$$\frac{\Delta \vdash T <: T'}{\mathcal{T}_{\Delta}(X) = \exists \emptyset. X} \qquad \qquad \frac{\Delta \vdash T <: T'}{\mathcal{M}_{\Delta}(T, T') = T = \mathcal{M}_{\Delta}(T', T)} \qquad \qquad \frac{\Delta \vdash T \not\leq: T'}{\mathcal{M}_{\Delta}(T', T) = T}$$

$$\begin{array}{c} \textbf{X is fresh} \\ \hline \mathcal{P}_{\Delta}(?) = (\textbf{X} \rightarrow [\bot \ \exists \emptyset.\texttt{Object}], \textbf{X}) \\ \mathcal{P}_{\Delta}(? \lhd \textbf{T}_s) = (\textbf{X} \rightarrow [\bot \ \mathcal{T}_{\Delta}(\textbf{T}_s)], \textbf{X}) \\ \mathcal{P}_{\Delta}(? \rhd \textbf{T}_s) = (\textbf{X} \rightarrow [\mathcal{T}_{\Delta}(\textbf{T}_s) \ \exists \emptyset.\texttt{Object}], \textbf{X}) \\ \mathcal{P}_{\Delta}(\textbf{T}_s) = (\textbf{X} \rightarrow [\bot \ \exists \emptyset.\texttt{Object}], \mathcal{T}_{\Delta}(\textbf{T}_s)) \end{array}$$

Figure 3.13: Translation from Java types to Tame FJ types.

amounts to  $\exists X \rightarrow [\exists \emptyset.Shape \exists \emptyset.Object].C<X>$ . We must distinguish different occurrences of the wildcard symbol by translating them to distinct type variables. Hence, Pair<?, ?> translates to  $\exists X, Y.Pair<X, Y>$ . Finally, nested wildcards are quantified at the immediately enclosing level, so C<C<C<?>>> translates to  $\exists \emptyset.C<\exists \emptyset.C<\exists X.C<X>>>$ .

A subtle aspect of the translation is that wildcards can inherit their upper bound from the upper bound of the corresponding formal type variable in the class declaration. Since we want to avoid doing this in the calculus, we must take care of this in the translation. For example, for a class C declared as class C < Z < Circle > ..., the type C < ?> is translated to  $\exists X \rightarrow [\bot \exists \emptyset.Circle].C < X>$ .

When an upper bound is declared both for a wildcard and in the corresponding class declaration, then the 'smallest' type is taken as the upper bound, if the types are subtypes of each other  $(\mathcal{M})$ . Hence,  $C<? \lhd$  Shape> is translated to the same type as in the previous example, and is *not* a type error. Finally, if the bounds are unrelated, then the bound from the declaration is taken as the upper bound of the wildcard, which means that even the type  $C<? \lhd$  Serializable> is translated into the same type as the previous two examples.

This last behaviour implies that the Java type analysis uses a more general type for some expressions than it would have to in order to maintain soundness (in the example it could have used the intersection of Circle and Serializable, but it just uses Circle), and this means that some reasonable and actually type safe programs are rejected by the Java compiler. However, it poses no problems for the soundness of Java, nor for our translation.

The most interesting aspect of the translation is where wildcards meet F-bounds. An F-bounded type is a type where the formal type variable is bounded by an expression in which the variable itself occurs (see sections 2.2.1 and 2.4.3). In the following example both instantiations of F using wildcards are legal.

class F<X  $\lhd$  F<X>> {...} void m(F<?> x1, F<?  $\lhd$  F<?>> x2) {...}

The translation of the types F<?> and  $F<?\lhd F<?>>$  is not immediately obvious, because in Java there is no finite type expression for the least supertype of all legal type arguments to F, i.e., the least upper bound of the type argument X is not denotable in Java. However, in Tame FJ this upper bound *is*, in fact, denotable: it is just  $\exists Y \rightarrow [\bot F<Y>]$ . F<Y>. Indeed, our translation of F<?> gives this type. In the case of  $F<? \lhd F<?>>$  where the wildcard is translated to the fresh variable Y, the upper bound will be the least subtype of  $\exists Z.F<Z>$  (the translation of the given bound; where Z is fresh) and F<Y> (the bound derived from the class declaration). Since the latter is more strict, it is used, even though this appears to contradict the rule of using fresh type variables for each wildcard; in fact it does no such thing, the second wildcard *is* translated to a fresh type variable, but is then forgotten. This is under-defined in the Java Language Specification [47], but we have tested this in several compilers (section 2.4.3). Note that because the inherited bound is used in both types, x1 and x2 have the same type in Java (which is denoted in Tame FJ as  $\exists Y \rightarrow [\bot F<Y>]$ .

## 3.3 Discussion — Modelling Wildcards

It took many iterations and much work to arrive at Tame FJ. In this section we describe some of the difficulties in formalising Java with wildcards. In section 3.3.2 we discuss the relationship between Tame FJ and traditional existential types systems and show why explicit packing and unpacking cannot model Java wildcards.

Accommodating lower bounds in Tame FJ turned out to be delicate because we can always derive that a declared lower bound is a subtype of the corresponding upper bound (by rules S-BOUND and S-TRANS). We must ensure that, irrespective of the bounds declaration, the bounds are indeed subtypes, i.e., that all subtyping in Tame FJ reflects the class hierarchy. We do this in the rules for well-formed environments. We cannot simply check that the bounds of a variable are subtypes without that variable, because we need to accommodate F-bounded quantification (see section 2.2.1) and, therefore, the bounds of a variable might involve that variable. The solution is to use extended subclassing, rather than subtyping, in F-ENV; this is described in section 3.1.3.

We separated subclassing from extended subclassing in order to reason about properties that are invariant over the subclass hierarchy, but not over extended subclassing, such as field and method types. In Tame FJ, the types of fields and methods can change between subtypes; this cannot occur in FGJ [53] and most other object-oriented systems. In order to prove subjectreduction we must regain some form of invariance of member types, this is done using the subclassing relation. Subclassing has the same properties as subtyping in FGJ. Of course we must now find circumstances for reducing subtyping to subclassing. This takes several lemmas (described in section 3.1.2) that were delicate to formulate precisely and to prove.

Java does not have problems with well-formed bounds because only wildcards may have lower bounds and the scope of these variables is limited to the expression where a wildcard type is unpacked. However, in order to formalise type checking we must unpack the wildcard types and use the quantified environments in the same way as environments from class or method declarations. Separating these two kinds of environment complicates the rest of the formalism unreasonably, and so we adopt the stratified subtyping approach described above.

Independently of lower bounds, F-bounded quantification means that we must quantify types by a whole environment rather than a single formal variable (as is normally done with existential types [27, 64, 74]). Quantification by environments means that we must deal with equivalent types. There are two ways that types may be equivalent: by concatenation, for example  $\exists \Delta_1 . \exists \Delta_2 . T$  is equivalent to  $\exists \Delta_1, \Delta_2 . T$ , and involving unused variables, for example,  $\exists X, Y. C < X >$  is equivalent to  $\exists X. C < X >$ . An important special case of the second kind involves empty environments,  $\exists \emptyset . T$  would be equivalent to T. Introducing explicit equivalence rules is problematic, because then the syntax of a type would not always correspond to its behaviour. Instead, we force Tame FJ types into a normal form; all types are existential types quantified by a single environment ( $\exists \Delta . R$ ) that may be empty (which corresponds to non-wildcard types). Unfortunately this results in the somewhat ugly substitution rules concerning  $\exists \emptyset . X$ .

Capture conversion of wildcards is difficult to model because there is no explicit unpacking and, therefore, no easy way to name unpacked type variables. To type check a method call where an actual type variable is quantified in the type in which it appears, we must be able to either name the argument from outside its scope, or to infer it. Despite some effort, we were unable to name the arguments, mainly because of the requirement to alpha convert quantified type variables. For example,

```
<X>void m1(C<X> x) {...}
<X1,X2> void m2(C<X1> x1, C<X2> x2) {...}
<X>void m3(P<X,X> x) {...}
void ma(∃Y.C<Y> y, ∃Y,Z.P<Y,Z> p) {
    this.<Y>m1(y); //1
    this.<Y,Y>m2(y, y); //2
    this.<Y>m3(p); //3
}
```

This code assumes that quantified type variables can be named outside of their scope. Method call 1 is a call where this facility is used to avoid type parameter inference. However, there would be a problem with call 2: y must be unpacked twice, which adds Y into the type environment twice; however, this violates the invariant that type variables only occur once in the domain of type environments. In Tame FJ we address this by renaming Y, but we cannot do this if Y can be named once it is unpacked. Call 3 shows why it is necessary to maintain our invariant about the domain of type environments. It should not type check, but would if we relax this invariant (because Z could be renamed to Y).

Therefore, we chose to infer type parameters. Formalising inference was difficult, in particular we needed to be careful to avoid problems with subject reduction [60]. The essential design decision is that we only infer substitutions of type variables which appear as parameters, not those that appear as types in their own right. This alleviates soundness worries<sup>2</sup> due to the invariance of (non-quantified) type parameters. For example,

```
<X>void m4(C<X> x) {...}
<X>void m5(X x) {...}
void mb(∃Z.C<Z> z) {
this.<*>m4(z); //4
this.<*>m5(z); //5
}
```

This program has been incorrectly translated from Java to Tame FJ. Both method calls are supported in Java<sup>3</sup>. However in Tame FJ, inferences is only supported in call 4 (call 5 can be modelled, as long as the inference is performed as a preliminary step<sup>4</sup>; it is supported as this.< $\exists Z.C < Z >> m5(z)$ , which can be expressed in Java syntax as this.<C <? >> m5(z)). In m4 the formal type parameter that is inferred (X) appears as a parameter in the type C<X>, whereas in m5, X appears as a type in its own right.

### **3.3.1** Some Previous attempts

We went through many failed attempts at modelling Java with wildcards. The most successful was using fully explicit existential types which is described in the next section. Here we outline some of the other ideas to illustrate the difficulties we encountered.

Straightforward approach using Java syntax. We attempted to model wildcards using the Java syntax, this failed because there are types that can be expressed but not denoted in Java and some therefore a more sophisticated syntax is required.

 $<sup>^{2}</sup>$ We do not mean that Java is unsound or that there are problems with our formalisation, only that to include the preliminary step of type inference in Tame FJ rather than keep it separate would prevent us proving soundness.

<sup>&</sup>lt;sup>3</sup>Although the syntax is this.m4(z).

 $<sup>^4\</sup>mathrm{In}$  contrast, call 4 cannot be expressed in Java without inference.

**'On the fly' existential types.** We tried to prove Wild FJ [60] sound and, when that failed due to the complexity of the system, tried to formulate our own system which generated existential types on the fly. However, this makes for a complex system where it is difficult to compare types between expressions.

**Explicit unpacking, implicit packing.** We used open expressions to unpack existential types but subtyping to pack them. This caused problems with reduction, it was unclear how to reduce open expressions without corresponding close expressions.

Allowing quantified variables to be used out of scope. In this system we allowed out of scope quantified variables to be used as type parameters to method calls. This made for a very simple system, but could not handle alpha conversion correctly.

**Non-normalised existential types.** We allowed quantification by multiple environments and defined equivalence rules between existential types. Unfortunately the equivalence rules broke the soundness proof.

**Quantification of a single type variable.** This is simpler than quantifying by an entire environment, but does not allow us to model some F-bounded wildcard types.

There were many other attempts, which were, frankly, too embarrassingly wrong-headed to describe here.

### 3.3.2 Wildcards and Existential Types

We developed  $\exists J \ [24]^5$  with the goal of formalising wildcards using explicit packing and unpacking, keeping the formalism as close to that of classical existential types [26, 27, 64, 74, 75] as possible. In this section we describe some of the elements of  $\exists J$ , compare it to Tame FJ, and explain why  $\exists J$  cannot be extended to be a full model of wildcards.

Q class C< $\Delta$ >  $\triangleleft$  N {Tf; M} class declarations ::=М  $<\Delta> \text{Tm}(\overline{\text{Tx}}) \{\text{return e};\}$ method declarations ::= $x \mid e.f \mid e.\overline{T} > m(\overline{e}) \mid new C < \overline{P} > (\overline{e})$ expressions е ::=| open e as x, $\overline{X}$  in e | close e with  $\Delta$  hiding  $\overline{T}$ new C $\overline{T}$  ( $\overline{v}$ ) | close v with  $\Delta$  hiding  $\overline{T}$ values v ::=Ν ::=C<T> | Object<> class types N | X R non-existential types ::=Κ  $\exists \Delta. \mathsf{K} \mid \mathsf{N}$ non-variable types ::=T,U K | X types ::= $\Delta$  $X \rightarrow [B_l \ B_u]$ type environments ::=Γ x:T ::=environments x,y variables C, D, E, F classes type variables Χ,Υ,Ζ

```
Figure 3.14: Syntax of \exists J.
```

$\begin{array}{c c} \underline{\Delta \vdash \mathtt{U} <: \mathtt{U}' & \Delta, \mathtt{X} \lhd \mathtt{U} \vdash \mathtt{K} <: \mathtt{K}' \\ \hline \Delta \vdash \exists \mathtt{X} \lhd \mathtt{U}  .  \mathtt{K} <: \exists \mathtt{X} \lhd \mathtt{U}'  .  \mathtt{K}' \\ (\text{S-Full}) \end{array}$	$\begin{array}{ccc} \Delta \vdash \overline{\mathbf{T}'} & \boldsymbol{\Delta}; \Gamma \vdash \mathbf{e} : \mathbf{N} \\ mType(\mathbf{m} < \overline{\mathbf{T}'} >, \mathbf{N}) = < \overline{\mathbf{X} < \mathbf{T}} > \overline{\mathbf{U}} \rightarrow \mathbf{U} \\ \underline{\Delta}; \Gamma \vdash \mathbf{e} : [\overline{\mathbf{T}'/\mathbf{X}}] \mathbf{U} & \Delta \vdash \overline{\mathbf{T}'} <: [\overline{\mathbf{T}'/\mathbf{X}}] \mathbf{T} \\ \hline \Delta; \Gamma \vdash \mathbf{e} . < \overline{\mathbf{T}'} > \mathbf{m}(\overline{\mathbf{e}}) : [\overline{\mathbf{T}'/\mathbf{X}}] \mathbf{U} \\ (\mathrm{T-Invk}) \end{array}$
$\begin{array}{c} \Delta; \Gamma \vdash \mathbf{e}_1 : \exists \overline{\mathbf{X} \lhd \mathbf{T}}.\mathbf{K} \\ \Delta \vdash \exists \overline{\mathbf{X} \lhd \mathbf{T}}.\mathbf{K} \text{ OK} \\ \hline \Delta, \overline{\mathbf{X} \lhd \mathbf{T}}; \Gamma, \mathbf{x}:\mathbf{K} \vdash \mathbf{e}_2 : \mathbf{T}  \Delta \vdash \mathbf{T} \text{ OK} \\ \hline \Delta; \Gamma \vdash \text{ open } \mathbf{e}_1 \text{ as } \mathbf{x}, \overline{\mathbf{X}} \text{ in } \mathbf{e}_2 : \mathbf{T} \\ \hline (\mathbf{T} \text{-} \text{OPEN}) \end{array}$	$\Delta' = \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}$ $\Delta; \Gamma \vdash \mathbf{e} : [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{K} \qquad \Delta \vdash \mathbf{T} <: [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{T}'$ $\Delta \vdash \exists \Delta'. \mathbf{K} \ \mathbf{OK}$ $\overline{\Delta; \Gamma \vdash \text{close e with } \Delta' \text{ hiding } \overline{\mathbf{T}} : \exists \Delta'. \mathbf{K}}$ $(\mathbf{T} \ \mathbf{Chose})$

Figure 3.15: Selected  $\exists J$  typing and subtyping rules.

We give the syntax of  $\exists J$  in figure 3.14 and some of its interesting rules in figure 3.15. Subtyping in  $\exists J$  extends FGJ subtyping with the rule S-FULL. S-FULL allows subtyping between existential types and is almost identical to the subtyping rule from traditional systems we described in section 2.3.2. Subtyping between existential and non-existential types does not exist in  $\exists J$ . Explicit packing using **close** expressions is used instead. In comparison, Tame FJ subtyping subsumes  $\exists J$  subtyping and also allows for packing types to existential types (in the XS-ENV

<sup>&</sup>lt;sup>5</sup>The version of  $\exists J$  we describe here is a slightly updated and extended version of our earlier work [24].

rule). This models subtyping between wildcard and non-wildcard types in Java.

Method invocation (T-INVK) is much simpler than in Tame FJ since we do not have to perform capture conversion of the receiver or parameters, nor infer type parameters. The requirements that the receiver has class type (N) and that the types of formal and actual parameters match, ensure that subexpressions will have been unpacked if necessary. Unlike Tame FJ, all actual type parameters can be named, there is no need to infer type parameters. Unpacked type variables will be nameable because they are declared in an open expression.

Packing and unpacking (in open and close expressions) proceeds as in traditional systems (section 2.3.2). The only difference is that in  $\exists J$ , packing and unpacking operates on quantifying environments as opposed to individual variables.

Java types are translated to  $\exists J$  types in the same way as in Tame FJ (section 3.2). To get a valid  $\exists J$  program we must also translate expressions. Wherever capture conversion occurs in a Java program, we must insert **open** expressions. Wherever subsumption would pack a type to an existential type, we must insert **close** expressions. For example,

```
<X>void m (Box<X> x, Box<?> y) {
    this.m(y, x);
}
```

Is translated to the following Tame FJ code:

```
<X>void m (Box<X> x, ∃Y.Box<Y> y) {
this.<*>m(y, x);
}
```

The same Java code is translated to the following  $\exists J$  code:

```
<X>void m (Box<X> x, ∃Y.Box<Y> y) {
open y as z,Z in
this.<Z>m(z, close x with Y hiding X);
}
```

In Java Box<X> (the type of x) is a subtype of Box<?> (the type of y), to represent this relationship in the  $\exists J$  program we must wrap x in a close expression. The type of close x... is  $\exists Y.Box<Y>$ , which corresponds to Box<?>. So that we can name the fresh type variable Z in the  $\exists J$  program, y must be unpacked. This is done implicitly in Java and Tame FJ, but requires the open expression that surrounds the method call in  $\exists J$ .

Although  $\exists J$  is an elegant model for Java with wildcards, it is incomplete. There are two major restrictions:  $\exists J$  does not support lower bounds on existential types and only expressions can be packed.

The first problem may actually be orthogonal to the use of explicit packing and unpacking. We believe that by adopting the Tame FJ treatment of subtyping,  $\exists J$  can be made to support lower bounds. The only possible problem is that in Tame FJ (as in Java), the scope of an unpacked variable (and thus any lower bound) is restricted to a single sub-expression, whereas in  $\exists J$  it can be freely defined. Thus, there is more room for problems with lower bounds than in Tame FJ; however, it is likely that there will not be a problem.

Because unpacking in  $\exists J$  is done in an expression rather than in subtyping, there are places where a type could be packed in Java but not in  $\exists J$ . For example, when a type is instantiated, subtyping is used to check that actual type parameters are within the declared bounds. Existential packing in this context cannot be done in  $\exists J$ . So, if a type variable is bounded by an existential type, then it can only ever be instantiated with an existential type. This is not the case in Java; for example, if X is declared to have bound Box<?> we can instantiate X with Box<Shape>, or any other Box. We have not found any encoding of such examples into  $\exists J$ .

#### Imperative Systems

Existential types can cause problems in low level imperative languages [49] (see section 2.3.6). These problems do not affect Java because all objects are referred to by reference, objects cannot be overwritten in memory. We show this by adapting the example from Cyclone given in section 2.3.6 [49]. We use an imperative version of  $\exists J$  and the Box definition from section 2.2.1;

A and B are unrelated classes:

```
∃X.Box<X> ba = new Box<A>;
∃X.Box<X> bb = new Box<B>;
open ba as x,Z in {
    Z z = x.get();
    ba = bb;
    x.set(z);
}
```

The assignment ba = bb only affects the variable ba it does not affect the object that was stored there. Therefore, x still holds a reference to the object of type Box<A>, not Box<B>. The call to set assigns z (at runtime, containing an object with type A) to a field of type A, so there is no unsoundness.

It would be straightforward, but time consuming, to add imperative features to Tame FJ. We would add a assignment and a heap in a similar manner to  $Jo\exists$  (section 4.2); this should not present any technical difficulty, nor make the proofs any different, only longer.

### 3.3.3 Decidability

Tame FJ is not designed with decidability in mind. Subtyping in Tame FJ is not syntax directed, which is an important first step toward an algorithm for subtype checking. It would be easy to refactor transitivity into the other subtype rules, however, the system would still not be syntax directed because of overlap between XS-ENV and XS-SUB-CLASS. A further challenge to giving an algorithmic version of subtyping for Tame FJ is that in XS-ENV, there are many valid choices for  $\overline{T}$ , any algorithm using a similar rule would involve backtracking.

We show in section 5.2.3 that Tame FJ subtyping and subtyping using separate rules for packing and unpacking (as used in Pizza [70], see section 2.3.7) are almost equivalent. Since subtyping using these rules is undecidable [95], Tame FJ subtyping may also be undecidable. However, the argument from [95], and another for declaration-site variance [55], rely on multiple inheritance (of interfaces for Java), and as such, it does not necessarily apply for Tame FJ. A decidability result for Tame FJ would not show that Java with wildcards is decidable since Tame FJ is a significant abstraction of Java. An undecidability result would carry over to Java if an expression type checks in Tame FJ if and only if it type checks in Java; we expect this equivalence to hold.

## 3.4 Chapter Summary

In this chapter we have presented Tame FJ, an existential types model for Java with wildcards, and shown that it is type safe. This is the first model for Java with wildcards that has been proved sound. Tame FJ is innovative in that it uses guarding environments to ensure correct repacking of existential type variables, several layers of subtyping, and a novel implementation of type parameter inference. Our proofs have several novel features involving the relationship between the subtype-like relations and auxiliary functions. We have discussed Tame FJ and the difficulties in its design, namely, type parameter inference, lower bounds, expressible but not denotable types, and capture conversion. We have presented a formal translation from a subset of Java to Tame FJ, focusing on the translation of types. We have also presented  $\exists J$ , an alternative and partial model for Java with wildcards that uses explicit packing and unpacking and is thus closer to traditional models of existential types.  $\exists J$  cannot be made into a full model for Java with wildcards because it cannot support unpacking of existential types outside of the expression syntax, such as in well-formedness checks;  $\exists J$  does not support lower bounds, but we do not believe this restriction is intrinsic to the model.

# Chapter 4

# **Existential types for Context Variance**

In this chapter, we suggest a novel use of existential types to support subtype variance in ownership types systems. We begin by motivating subtype variance in ownership systems and existential types as a possible solution (section 4.1). In section 4.2, we present and discuss  $Jo\exists$ , a calculus with existential types for subtype variance, combined with type genericity for added expressivity. In section 4.3, we show how  $Jo\exists$  can be modified to support deep ownership in  $Jo\exists_{deep}$ . In section 4.5, we discuss  $Jo\exists_{wild}^1$ , a more realistic variation of  $Jo\exists$  with implicit packing and unpacking. We give proofs of type soundness and deep ownership in appendix B.

**Contributions** Although systems exist with variant ownership or existential quantification, to the best of our knowledge,  $Jo\exists$  is the first language to use explicit existential types for variance; it is also the first language to combine type parameterisation with variant contexts and is thus more expressive than existing languages. We give a soundness proof for  $Jo\exists$  and a proof of owners-as-dominators for  $Jo\exists_{deep}$ — to the best of our knowledge the first such proof for an ownership language with variance.

<sup>&</sup>lt;sup>1</sup>The 'wild' subscript refers to wildcards, since  $Jo\exists_{wild}$  uses implicit packing and unpacking of existential types, similarly to Java wildcards.

# 4.1 Motivation

Ownership types, as described in section 2.5, usually display invariant subtyping. As with generics (section 2.2.1), allowing co- or contravariance without restrictions is unsound. Furthermore, the encapsulation properties of ownership systems often rely on invariant subtyping. For example, allowing restricted covariance could allow the programmer to violate the owners-as-dominators property, even if the system was otherwise sound:

```
class C<oc> {
    C<+oc> f;
}
class D<od> {
    void m(C<od> c) {
        c.f = new C<this>();
    }
}
```

The class C appears safe, since (assuming standard variance annotation typing properties) we cannot perform actions on f that require more information than the upper bound of the type parameter (oc). The assignment in method m is legal using variance annotation typing (since this is inside od). But c now holds a reference to an object that is neither owned by c, nor outside c, thus violating owners-as-dominators.

Subtype variance for ownership types is an interesting and desirable feature that has been tackled in many previous language designs [57, 23, 66, 67, 58] (section 2.5.3). However, none of these approaches are entirely satisfactory: most permit only bivariance, none can express constraints over more than one context parameter or between elements of an aggregate.

We address these issues by using existential quantification of contexts and type parametricity to provide variance. This approach is more expressive, uniform, and transparent than previous approaches. We compare our approach with related work in section 5.3 after comparison with Tame FJ.

### 4.1.1 Existential Types

The formalisation of variance using existential types is expressive and uniform. Compared to existing approaches to ownership variance, existential types allow the expression of more types; for example, a variant list with a single owner can be denoted  $\exists o.List < o, o>$ . Existential types allow variant contexts to have both upper and lower bounds, previous approaches do not support bounds or allow only one kind of bound on each context variable. Existential quantification facilitates even more expressivity by integrating with type parameterisation (see below); combining type parameterisation with variance annotations or 'any' contexts does not improve expressivity.

Supporting owner polymorphic methods (see section 2.5) requires existential types. Variance annotations are not expressive enough because capture conversion could be used to express types that cannot be denoted in the syntax. Expressible but not denotable types also motivate existential types for modelling wildcards (section 2.4.2).

Using existential types for variance allows us to connect this new problem with well-understood type theoretic foundations for subtype variance and abstraction [26, 27, 49, 64, 74, 75] (section 2.3).

### 4.1.2 Type Parameterisation

Type parameterisation lets us differentiate between, for example, GenericList<o1, Shape<o2>>, a list of Shapes, and GenericList<o1, Animal<o2>>, a list of Animals; where GenericList is defined as:

```
class GenericList<owner, X> {
    X datum;
    GenericList<owner, X> next;
}
```

By combining existential types with type parameterisation we can differentiate between a list where each item has the same (unknown) owner, and a list where each item has a different owner. Using either existential types or variance annotations alone would require different classes for each situation. By allowing classes to be parameterised by types as well as contexts, we can express these different constraints using a single class, see section 4.2.2.

### 4.1.3 Example

In the following example (figure 4.1), we use  $o \rightarrow [a \ b]$  to denote that the formal context parameter o has the lower bound a and upper bound b, that is, any instantiation of o must be inside b and outside a in the ownership hierarchy.

Our example figure 4.1 is part of a human resources system for a large company. Each worker in the company is owned<sup>2</sup> by its manager; the employees form a hierarchy with the director at its root. In the Worker class, each worker keeps a list of his colleagues. This list uses the GenericList class defined in section 4.1.2 to specify that it is a list of Workers and that each colleague has the same manager as this (and works for the same company). In the Company class, we store references to the director and the head of marketing, whose immediate manager is the director<sup>3</sup>. We also keep a list of all workers managed (owned) by the headOfMarketing, representing the marketing team.

So far, we have only used features present in classical ownership types systems. We use existential types where the precise owner of objects is unknown. In the Worker class, mentor is some worker whose owner either works with or indirectly manages that worker, but who's exact position in the management hierarchy is not specified. A worker may work with some other team of workers in the company (a team is assumed to have a single manager). For example, an engineer may have contact with the management team. This group (workGroup) may have any manager in the company, and this is represented by the existential type. Since we assume all members of the group have the same owner, the existential quantification is outside the

### GenericList.

<sup>&</sup>lt;sup>2</sup>In terms of our ownership types, hopefully not in the real world.

<sup>&</sup>lt;sup>3</sup>In the example we use fields as context parameters. This is not implemented in  $Jo\exists$ , but is a relatively easy extension. It is present in, for example, MOJO [23].

```
class Worker<manager, company outside manager> {
     GenericList<this, Worker<manager, company>> colleagues;
     \exists o \rightarrow [\perp \text{ company}].GenericList<this, Worker<o, company>> workGroup;
     \exists o \rightarrow [manager company] . Worker < o, company > mentor;
    void mixGroups() {
         workGroup = close colleagues with o \rightarrow [\perp \text{ company}] hiding manager;
         open workGroup as w,m in {
              //colleagues = w;
                                                               ERROR.
              //colleagues.add(w.get(0));
                                                               ERROR
              //w.add(colleagues.get(0));
                                                               ERROR
         }
     }
}
class Company extends Object<O> {
     Worker <this, this > director;
     Worker<director, this> headOfMarketing;
     GenericList<this, Worker<headOfMarketing, this>> marketing;
     \exists o \rightarrow [\bot \text{ director}]. Worker < o, this > employeeOfTheMonth;
     GenericList<this, \exists o \rightarrow [\bot \text{ this}].Worker<o, this>> payroll;
     <m> void processColleagues(Worker<m, this> w) {
         for (Worker<m, this> c : w.colleagues) {
               . . .
          }
     }
     void processPayroll() {
         for (\exists o \rightarrow [\bot \text{ this}].Worker < o, this > w : payroll) {
              open w as x,m in {
                   this.<m>processColleagues(x);
              }
          }
     }
     void mentorEmpMonth() {
         open employeeOfTheMonth as x,m in {
              x.mentor =
                   close director with o \rightarrow [m \text{ this}] hiding this;
              //x.mentor =
                                                               ERROR
              11
                     close new Worker<headOfMarketing, this>()
                     with o \rightarrow [m \text{ this}] hiding headOfMarketing;
              11
         }
     }
}
```

Figure 4.1: Example: existential types for context variance.

In the Company class, the employeeOfTheMonth may be any Worker in the company, her manager is not important. The payroll keeps track of every worker in the company. Each worker on the payroll may have a different manager. The method **mixGroups** could be expressed in a more user-friendly syntax using implicit packing and unpacking:

<pre>void mixGroups() {</pre>	
<pre>workGroup = colleagues;</pre>	
<pre>//colleagues = workGroup;</pre>	ERROR
<pre>//colleagues.add(workGroup.get(0));</pre>	ERROR
<pre>//workGroup.add(colleagues.get(0));</pre>	ERROR
}	

We can set workGroup to colleagues because manager (the owner workers in colleagues) is within the bounds specified in the type of workGroup. We cannot set colleagues to workGroup, nor add an element of colleagues to workGroup, because the workers in workGroup may have any owner, not necessarily manager. Even though we can set workGroup to colleagues, we cannot add an element of colleagues to workGroup because although the owner of the workGroup may be any owner, it is a specific owner and not necessarily manager (here, the manager is manager due to the earlier assignment, but in general it will be unknown). The close syntax introduces a new context variable (o in the assignment in mixGroups) this is bound by an existential quantifier in the type of the close expression; it cannot capture any similarly named context variables.

The method processColleagues takes a worker (w) as a parameter and performs some action on each of his colleagues. Since the method is polymorphic in the manager of w, we can name the manager (m) as the owner of w's colleagues, c. The method processPayroll performs some action on each worker w in the payroll and their colleagues. Since the manager of each worker in the payroll is abstracted, we have to open w in order to be able to name their manager (m), in the example, to call processColleagues. This corresponds to wildcard capture in Java (sections 2.4.2 and 3.1.4).

**Owners-as-dominators** Even in a deep ownership system it can be safe and desirable to support subtype variance. A Worker instance and his mentor (though not his workGroup) satisfy owners-as-dominators in  $Jo\exists_{deep}$ . mentorEmpMonth sets the mentor of the employeeOfTheMonth to the director. This preserves owners-as-dominators since the director must transitively

manage (own) the employeeOfTheMonth, no matter who that is. The employeeOfTheMonth must be unpacked and the director packed, as in previous methods.

Setting the employeeOfTheMonth's mentor to a new worker owned by the headOfMarketing would violate owners-as-dominators and is not allowed. This is because the employeeOfTheMonth may not be transitively owned by the headOfMarketing. Type checking fails in the close expression because we cannot derive that m is inside headOfMarketing.

### **4.2** Jo∃

In this section we present Jo $\exists$ , a minimal, imperative, object-oriented language with parameterisation of methods and classes by context and type parameters, and existential quantification of contexts. Jo $\exists$  extends FGJ [53] (Jo $\exists$  also restricts FGJ by omitting casting and type bounds on type variables) and is similar in spirit to  $\exists$ J in its treatment of existential types [24] (section 3.3.2). Some of the features of Jo $\exists$  can be found in other formalisations of ownership types, including Joe [32], ownership types for flexible alias protection [34], Joe<sub>3</sub> [72], and OGJ [77, 78]. In order to demonstrate ownership properties, we include field assignment and a mutable heap.

We present Jo $\exists$  in three flavours; firstly, we present Jo $\exists$ , a purely descriptive language, the type system describes an ownership hierarchy but does not enforce encapsulation properties. Type safety (section 4.2.6) guarantees that types accurately reflect the ownership hierarchy. Secondly, we present Jo $\exists_{deep}$ , an extension of Jo $\exists$  that supports owners-as-dominators.

Subtype variance in Jo $\exists$  is implemented by existential quantification. Existential types are explicit and are introduced and eliminated by close and open expressions. In this way, we follow the more traditional model of existential types [24], rather than the Java 5.0 approach of using implicit packing and unpacking (see section 3.1). Thirdly, we discuss a system with implicit packing and unpacking, Jo $\exists_{wild}$ , in section 4.5.

Neither the ownership or existential quantification features of  $Jo\exists$  interact with subclassing. Furthermore, the benefits of existential quantification in  $Jo\exists$  do not depend on subclassing, nor the absence of subclassing. For these reasons, and because the standard solution to subclassing in ownership types systems is long known [32], we elide subclassing and inheritance. This simplifies the presentation of Jo $\exists$  and its proofs. Jo $\exists$  could be extended to include subclassing by extending the subtyping and method and field lookup rules following FGJ [53]. Subclassing must preserve the formal owner of an object [32]. There are no changes to any of the rules involving quantification.

We are primarily interested in type parameterisation to increase expressiveness of ownership types, rather than to investigate features of generic types. We therefore treat type parameterisation simply and do not support bounds on formal type parameters, nor allow existential quantification of type variables.

### 4.2.1 Syntax

The syntax of Jo $\exists$  is given in figure 4.2. Entities only used at runtime are in grey. Jo $\exists$  includes expressions for accessing variables (**x**, which may include **this**) and addresses ( $\iota$ ), object creation, field access and assignment, method invocation, and packing and unpacking of existential types. Jo $\exists$  also includes **null** to handle uninitialised fields. These expressions are discussed in detail in section 4.2.5.

Class and method declarations (Q and W) are parameterised by context ( $\circ$ ) and type (X) parameters. The former have upper and lower bounds (bounds are actual context parameters — not subtype bounds — and limit the bounded formal context to some part of the ownership hierarchy), and so methods and classes are considered to be parameterised by *context environments* ( $\Delta$ ). These are mappings from formal context parameters to their bounds ( $\circ \rightarrow [b_l \ b_u]$ ).

Contexts (a) consist of context variables (o), variables (x) and the *world context*,  $\bigcirc$ , the root of the object hierarchy and the indirect owner of all objects. At runtime we may also use addresses. Runtime contexts (r) are restricted to addresses and  $\bigcirc$ . Bounds on context variables (b) consist of contexts and the bottom owner  $\bot$ . This is a theoretical context that is owned by all objects<sup>4</sup>.

<sup>&</sup>lt;sup>4</sup>Due to the hierarchical nature of ownership, no object in any non-trivial program can have this property.

е	::=	$\begin{array}{l c c c c c c c c c c c c c c c c c c c$	expressions
Q W	::= ::=	class C< $\Delta$ , $\overline{X}$ > { $\overline{Tf}$ ; $\overline{W}$ } < $\Delta$ , $\overline{X}$ > Tm( $\overline{Tx}$ ) {return e;}	class declarations method declarations
v	::=	close v with $\overline{o \rightarrow [b \ b]}$ hiding $\overline{r} \mid \iota \mid null \mid e$	rr values
N	::=	C<ā, T>	class types
R	::=	$C < \overline{r}, \overline{T} >$	runtime types
М	::=	N   X	non-existential types
Т	::=	$M \mid \exists \Delta . N$	types
a	::=	o   x   ()   <i>l</i>	actual owners
r	::=	$\bigcirc \mid \iota$	runtime owners
b	::=	a   ⊥	bounds
$\Psi$	::=	$\overline{X \rightarrow [b_l \ b_u]}$	type environments
$\Delta$	::=	$\overline{o \rightarrow [b_l \ b_u]}$	$context\ environments$
$\gamma$	::=	$\mathbf{x} \mid \iota \mid null$	variables and addresses
Γ	::=	$\gamma$ : T	variable environments
$\mathcal{H}$	::=	$\overline{\iota \to \{ R; \ \overline{f \to v} \}}$	heaps
x,y v v			variables
л, і			formal owners
C			classes
l			addresses
_			

Figure 4.2: Syntax of  $Jo\exists$ .

Contexts may not be directly quantified, i.e., there is no entity of the form  $\exists o.o.$ 

Variable environments,  $\Gamma$ , map variables to their types. Type environments,  $\Psi$ , map type variables to bounds on a context. In contrast to Tame FJ, type variables do not have bounds on the types they may take. The bounds contained in  $\Psi$  define upper and lower bounds on the owner of the actual types. If the lower and upper bounds on the owner of X are  $b_l$  and  $b_u$ , then for C<o> to instantiate X,  $\circ$  must be outside  $b_l$  and inside  $b_u$ . These bounds are manufactured by the type system (in T-CLASS and T-METHOD in figure 4.7) and cannot be defined by the programmer. In Jo∃ and Jo∃<sub>deep</sub>, upper bounds in  $\Psi$  are always  $\bigcirc$  and, in effect, never used;

however, we keep upper bounds for symmetry and to allow for easy extension. We only use the lower bound to support deep ownership in  $Jo\exists_{deep}$  (in section 4.3). We could therefore elide  $\Psi$  and just use a list of type variables, however, this does not make for a clean transition to  $Jo\exists_{deep}$ .

To model execution we use a heap,  $\mathcal{H}$ , which maps addresses ( $\iota$ ) to records representing objects. Each record contains the type of the object and a mapping from field names to values. Values (v) are addresses or close expressions that pack addresses. The type of an object is a runtime type, R.

### 4.2.2 Types in $Jo\exists$

The syntax of types in  $Jo\exists$  is also given in figure 4.2. Class types (N) are class names parameterised by actual type and context parameters. The first context parameter is the owner of objects with that type. Class types may be existentially quantified by a context environment to give existential types. For example,  $\exists o.GenericList < o, Animal > denotes a list owned by$ *some*owner. For conciseness in examples, we omit bounds and empty parameter lists whereconvenient. Only context variables (*not*type variables) may be quantified. For conciseness inexamples, we will omit bounds and empty parameter lists where convenient.

By combining existential quantification with type parameterisation we can express many interesting and useful types:

denotes a list owned by some unknown owner where each element is an Animal owned by this, while

denotes a list owned by some owner where all elements are owned by the same owner which may be different from the owner of the list, and

∃o1.GenericList<o1, ∃o2.Animal<o2>>

denotes a list where each element is owned by some owner and the owner of each element may be different, finally,

```
∃o.GenericList<o, Animal<o>>
```

denotes a list where each element in the list and the list itself are owned by the same, unknown, owner.

### 4.2.3 Subtyping and the Inside Relation

Subtyping  $\Delta; \Gamma \vdash T <: T$   $\overline{\Delta; \Gamma \vdash M <: M}$  (S-REFLEX) (S-REFLEX) (S-FULL) Inside relations  $\Delta; \Gamma \vdash b \leq b$   $\Delta; \Gamma \vdash \Delta \leq \Delta$  $\overline{\Delta; \Gamma \vdash b \leq b}$   $\overline{\Delta; \Gamma \vdash b \leq b'}$   $\Delta; \Gamma \vdash b' \leq b'$   $\Delta; \Gamma \vdash b \otimes C$  (I-REFLEX) (I-TRANS) (I-WORLD) (I-BOTTOM) (I-BOTTOM)

Figure 4.3: Jo∃ subtyping, and the inside relation for owners and environments.

The inside relation relates contexts and is defined by the rules given in figure 4.3. We say that  $o_1$  is inside  $o_2$  ( $\Delta; \Gamma \vdash o_1 \leq o_2$ ), if  $o_1$  is transitively owned by  $o_2$ . The inside relation is judged by  $\Delta$  and  $\Gamma$ , however,  $\Gamma$  is only ever used to find the owner of variables, never there entire type. The inside rule could be simplified by using a mapping from variables to their owners rather than  $\Gamma$ , but this would complicate typing and well-formedness rules. The inside relation

is reflexive, transitive, and has top and bottom elements — the world and bottom contexts, respectively. I-OWNER asserts that every variable and address is inside the declared owner of its type (if its type is a class type). For example, if this has type C<o>, then this is inside o. I-BOUND gives that a formal context is within its bounds.

I-ENV extends the inside relation to owner environments. We say that  $\Delta_1$  is inside  $\Delta_2$  ( $\Delta; \Gamma \vdash \Delta_1 \preceq \Delta_2$ ), if the domains of the two environments are identical and if the upper bounds of  $\Delta_1$  are inside those of  $\Delta_2$  and vice versa for the lower bounds. The intuition is that  $\Delta_1$  is more precise than  $\Delta_2$ .

Subtyping is also given in figure 4.3. Since there is no subclassing in Jo $\exists$ , subtyping of nonexistential types is given only by reflexivity. Subtyping between existential types follows the full variant of existential subtyping (see FUN-S-FULL in section 2.3.2). Existential types are subtypes where the bounds of quantified contexts in the subtype are more strict than in the supertype (given by I-ENV). For example,  $\exists o \rightarrow [\bot \text{this}].C<o>$  is a subtype of  $\exists o \rightarrow$  $[\bot \bigcirc].C<o>$ , since this is inside  $\bigcirc$ . Non-existential types remain invariant.

### 4.2.4 Well-formedness

Well-formed contexts, types, and environments are given in figure 4.4. Owner variables ( $\circ$ ) and expression variables ( $\mathbf{x}$ ) are well-formed if they are in the domains of their respective environments ( $\Delta$  and  $\Gamma$ ). In addition, the type of a variable must be a class type, this guarantees precise information about all unquantified contexts, and that the set of contexts is closed under substitution<sup>5</sup>. This restriction abides by the philosophy of existential types, whereby entities with existential type must be unpacked to be used.

Well-formed class types (F-CLASS) require the class name to have been declared, the actual context parameters to be within the bounds of the formal context parameters, the number of

<sup>&</sup>lt;sup>5</sup>Forcing variables  $(\mathbf{x})$  used as contexts to have class type prevents them from having existential type. Since the subtyping rules of Jo∃ preclude subtyping between existential and non-existential types, any value substituted for  $\mathbf{x}$  must also have non-existential type (since subtyping is required for substitution). Since close values have existential type, a close value cannot be substituted for  $\mathbf{x}$ . Therefore, a type parameterised by a close expression cannot be created by substitution.

Well-formed contexts  $|\Delta; \Gamma \vdash b \text{ OK}|$ 

$\mathbf{o} \in dom(\Delta)$	$\Gamma(\gamma) = \mathtt{N}$		
$\Delta;\Gamma\vdash o \ \mathrm{OK}$	$\Delta;\Gamma\vdash\gamma \text{ ok}$	$\Delta;\Gamma\vdash\bigcirc \text{ ok}$	$\Delta; \Gamma \vdash \perp \text{ ok}$
(F-OWNER)	(F-VAR)	(F-WORLD)	(F-BOTTOM)

Well-formed types  $\Psi; \Delta; \Gamma \vdash T$  ok

class  $C < \overline{o \rightarrow [b_l \ b_u]}$ ,  $\overline{X} > \dots$   $\Delta; \Gamma \vdash \overline{a} \ OK$  $\Delta; \Gamma, \text{this:} C < \overline{a}, \ \overline{X} > \vdash \ \overline{[a/o]} b_l \preceq a$   $\Delta; \Gamma, \text{this:} C < \overline{a}, \ \overline{X} > \vdash \ \overline{a} \preceq \overline{[a/o]} b_u$  $\Psi; \Delta; \Gamma \vdash \overline{T} \ OK$   $|\overline{T}| = |\overline{X}|$  $\Psi; \Delta; \Gamma \vdash C < \overline{a}, \ \overline{T} > OK$ 

(F-CLASS)

	$\Delta; \Gamma \vdash o \rightarrow [b_l \ b_u] \text{ OK}$
$\mathbf{X} \in dom(\Psi)$	$\Psi; \Delta, \overline{o \to [b_l \ b_u]}; \Gamma \vdash \mathbb{N} \text{ ok}$
$\Psi; \Delta; \Gamma \vdash X \text{ ok}$	$\Psi; \Delta; \Gamma \vdash \exists \overline{o \to [b_l \ b_u]} . \mathbb{N} \text{ OK}$
(F-TYPE-VAR)	(F-EXIST)

Well-formed environments  $|\Delta; \Gamma \vdash \Delta \text{ ok}|$ 

	$\Delta; \Gamma \vdash b_l, b_u \text{ OK} \qquad \Delta; \Gamma \vdash b_l \preceq b_u$
	$\Delta, o \rightarrow [b_l \ b_u]; \Gamma \vdash \Delta' \text{ ok}$
$\Delta;\Gamma \vdash \emptyset \text{ ok}$	$\Delta; \Gamma \vdash o {\rightarrow} [b_l \ b_u], \Delta' \text{ OK}$
(F-EMPTY)	(F-ENV)

Figure 4.4: Jo∃ well-formed contexts, types, and environments.

actual type parameters to match the number of formal type parameters, and the actual context and type parameters to be well-formed.

To check that actual context parameters are within their corresponding bounds, the judging environments are extended with this mapped to  $C < \overline{a}$ ,  $\overline{X} > ^6$ . That is, the class type with actual context parameters and formal type parameters. This is necessary because  $\overline{b}_l$  and  $\overline{b}_u$  may involve this. We cannot substitute for this, because there may not be a variable or address

<sup>&</sup>lt;sup>6</sup>An alternative to adding this to  $\Gamma$  is to add this  $\rightarrow [\perp a_0]$  to  $\Delta$ . This avoids the need to use  $\overline{X}$  out of scope, but if this is in  $dom(\Gamma)$ , then we must explicitly rename it in  $\Gamma$  and  $\overline{a}$ .

that contains the object to be substituted. For example, to type check **new** N, we must check that N is well-formed; we cannot substitute **new** N for **this** in a type. We use a mixture of actual context parameters ( $\overline{a}$ ) and formal type parameters ( $\overline{X}$ ) because of the order of application of substitution lemmas in the proofs. Using  $\overline{X}$  is safe, even though  $\overline{X}$  are not in scope, because the type parameters of types are never used in the rules defining the inside relation. If **this** already occurs in  $\Gamma$  then it must be renamed; we assume this happens implicitly by alpha conversion<sup>7</sup>.

A type variable is well-formed if it is in the domain of  $\Psi$ . An existential type,  $\exists \Delta . \mathbb{N}$ , is well-formed if  $\Delta$  is well-formed and  $\mathbb{N}$  is well-formed under the judging environments extended with  $\Delta$ .

A context environment,  $\Delta$ , is well-formed if all bounds in the environment are well-formed and, for each formal context, the lower bound is inside the upper bound. Well-formedness is defined inductively, so bounds may not include forward references.<sup>8</sup>

### 4.2.5 Typing

Type rules for expressions are given in figure 4.5. Field and variable access (T-FIELD and T-VAR) are close to those of FGJ [53]. Field assignment (T-ASSIGN) is a straightforward extension of field access. We adopt the standard subsumption rule (T-SUB), which is simpler than in Tame FJ, because we do not have guarding environments. In object creation (T-NEW), we create uninitialised objects and do not have constructors. T-NULL allows null to take any well-formed type. Method invocation is also close to FGJ, with the addition that actual context parameters must be well-formed and within their corresponding formal bounds.

In T-FIELD, T-ASSIGN, and T-INVK, the receiver is restricted to  $\gamma$ . If we didn't make this

$$\begin{array}{l} \Delta_1 = \{ \mathsf{o}_0 \to [\bot \ \bigcirc] \text{, } \mathsf{o}_1 \to [\mathsf{o}_0 \ \bigcirc] \} \\ \Delta_2 = \{ \mathsf{o}_0 \to [\mathsf{o}_1 \ \bigcirc] \text{, } \mathsf{o}_1 \to [\bot \ \bigcirc] \} \end{array}$$

Here,  $\Delta_1$  is well-formed, but  $\Delta_2$  is not, due to the forward reference in the lower bound of  $o_0$ .

<sup>&</sup>lt;sup>7</sup>For example, assume we wish to check that C<this> is well-formed under  $\{x:T_1,this:T_2\}$  and  $\Delta$  where C is defined as class C<o→[⊥ this]> {...}. The object this must be inside the upper bound of o. To check this, we must extend  $\Gamma$  with this, which means renaming this in  $\{x:T_1,this:T_2\}$  and C<this>. The judgement could become  $\Delta$ ;  $\{x:T_1,that:T_2,this:C<that>\} \vdash$  that  $\preceq$  this.

<sup>&</sup>lt;sup>8</sup>For example,

**Expression typing**  $\Psi; \Delta; \Gamma \vdash e : T$ 

-

$\Psi;\Delta;\Gammadash$ e: T'		$\Psi;\Delta;\Gamma\vdash\gamma:\mathtt{N}$
$\Delta;\Gamma\vdash {\rm T}'<:{\rm T}$	$\Psi;\Delta;\Gamma\vdash\gamma:\mathtt{N}$	$fType(\mathtt{f},\gamma,\mathtt{N})=\mathtt{T}$
$\Psi; \Delta; \Gamma \vdash T$ ок	$fType(\mathtt{f},\gamma,\mathtt{N})=\mathtt{T}$	$\Psi;\Delta;\Gamma\vdash \texttt{e}:\texttt{T}$
$\Psi;\Delta;\Gamma\vdash {\tt e}:{\tt T}$	$\Psi; \Delta; \Gamma \vdash \gamma.\texttt{f}:\texttt{T}$	$\Psi; \Delta; \Gamma \vdash \gamma  \texttt{.f} \; \texttt{=}\; \texttt{e}: \texttt{T}$
(T-SUBS)	(T-FIELD)	(T-Assign)

	$\Psi; \Delta; \Gamma \vdash C < \overline{a}, \ \overline{U} > OK$	$\Psi; \Delta; \Gamma \vdash T$ ок
$\Psi; \Delta; \Gamma \vdash \gamma : \Gamma(\gamma)$	$\Psi; \Delta; \Gamma \vdash \texttt{new C<\overline{a}, \overline{U}> : C<\overline{a}, \overline{U}>}$	$\Psi;\Delta;\Gamma\vdash\texttt{null}:\texttt{T}$
(T-VAR)	(T-NEW)	(T-NULL)



Figure 4.5:  $Jo\exists$  expression typing rules.

restriction, then an arbitrary expression could be substituted for **this** which would require dependent typing. There is no loss of expressivity because to use an expression  $\mathbf{e}$  as a receiver (for example to access  $\mathbf{f}$ ) we can use an open expression with empty  $\overline{\mathbf{o}}$  as a let expression, for example, open  $\mathbf{e}$  as  $\mathbf{x}, \emptyset$  in  $\mathbf{x}.\mathbf{f}$ .

The lookup functions used to type check field access and assignment, and method invocation are given in figure 4.6, these are fairly standard for this kind of calculus. fType, mBody, and mType all take an extra parameter, the receiver of the field or method. This is substituted for this, since this may occur in types in Jo $\exists$ . Similarly to F-CLASS, we must add this to  $\Gamma$ . The odd mixture of actual context parameters and formal type parameters in the type of this is a convenience for our proofs.

class C< $\overline{o \rightarrow [b_l \ b_u]}$ , $\overline{X}$ > { $\overline{Uf}$ ; $\overline{W}$ }	class $C < \overline{o \rightarrow [b_l \ b_u]}$ , $\overline{X} > \{ \overline{Uf}; \overline{W} \}$
$fields(C) = \overline{f}$	$\overline{fType(\mathtt{f}_i,\gamma,\mathtt{C<\overline{a},\ \overline{T}>})=[\overline{\mathtt{a/o},\ \overline{\mathtt{T/X}},\ \gamma/\mathtt{this}]\mathtt{U}_i}$
class C< $\overline{o}$	$\overline{[\mathbf{b}_l \ \mathbf{b}_u]}$ , $\overline{\mathbf{X}} > \{\overline{\mathbf{U}'' \mathbf{f}}; \overline{\mathbf{W}}\}$
$<\overline{{ m o}'}  ightarrow [{ m b}'_l \ { m b}'_u]$ , $\overline{{ m X}'}$	> $\operatorname{Tm}(\overline{\operatorname{Tx}})$ {return e;} $\in \overline{W}$
$\overline{mBody}(m<\overline{a'}, \overline{U'}>, \gamma, C<\overline{a}, \overline{U}>) =$	$=(\overline{\mathbf{x}}; [\overline{\mathbf{a/o}}, \overline{\mathbf{a'/o'}}, \overline{\mathbf{U/X}}, \overline{\mathbf{U'/X'}}, \gamma/\mathtt{this}]e)$
class C <o<math>\rightarrow[b<sub>l</sub> b<sub>u</sub>], X&gt; {U''I; W}</o<math>	$\langle o' \rightarrow [b'_l \ b'_u], X > Tm(Tx) $ {return e;} $\in W$
$\Delta; \Gamma, \texttt{this:C<\overline{a}, } \overline{X} \succ \texttt{a'} \preceq [\overline{\texttt{a/o}, } \overline{\texttt{a'/o'}}]$	$\mathbf{b}'_u = \Delta; \Gamma, \mathtt{this:C<\overline{a}}, \ \overline{\mathtt{X}} \vdash [\overline{\mathtt{a/o}}, \ \overline{\mathtt{a'/o'}}] \mathbf{b}'_l \preceq \mathtt{a'}$
$mType_{\Delta:\Gamma}(m<\overline{a'}, \overline{U'}>, \gamma, C<\overline{a}, \overline{U}>) =$	= $[\overline{a/o}, \overline{a'/o'}, \overline{U/X}, \overline{U'/X'}, \gamma/\text{this}](\overline{T} \to T)$

Figure 4.6: Field and method lookup functions for  $Jo\exists$ .

To type check open and close expressions we follow Fun [27] and other classical existential types systems (see section 2.3.2). T-OPEN is similar to FUN-T-OPEN, except that we are dealing with contexts rather than types. The type of expression  $\mathbf{e}$  is unpacked to an owner environment,  $\overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]}$ , and un-quantified type, N. We then judge the body of open ( $\mathbf{e}'$ ) by extending  $\Delta$  with  $\overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]}$  and adding a fresh variable,  $\mathbf{x}$ , with type N to  $\Gamma$ ; where  $\mathbf{x}$  stands for the unpacked value of  $\mathbf{e}$ . We ensure no variables escape the scope of the open expression by checking that the result type, T, is well-formed in an environment which does not contain  $\overline{\mathbf{o}}$  or  $\mathbf{x}$ .

A close expression packs an expression e by hiding some of the context parameters present in e's type. For example, if e has type C<this>, then the expression close e with o hiding this has the existential type  $\exists o.C < o >$ .

Type rules for methods and classes are given in figure 4.7. A class is well-typed if for all methods, the types of all fields, and the bounds of all formal context parameters are well-formed. Similarly, a method is well-typed if the return type, types of parameters, and bounds on formal context parameters are well-formed. The method body (e) must have the declared return type of the method. In these rules we use type environments,  $\Psi$  and  $\Psi'$ , always using  $\bot$  and  $\bigcirc$  as bounds. In section 4.3 we will use different lower bounds.

Class typing  $\vdash Q \text{ OK}$ 

$$\begin{split} \Psi &= \overline{X \rightarrow [\bot \bigcirc]} \\ \underline{\emptyset; \texttt{this}: \texttt{C} < \overline{\texttt{o}}, \ \overline{X} > \vdash \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]} \ \texttt{OK} \qquad \Psi; \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]}; \texttt{this}: \texttt{C} < \overline{\texttt{o}}, \ \overline{X} > \vdash \overline{\texttt{W}}, \ \overline{\texttt{T}} \ \texttt{OK} \\ &\vdash \texttt{class} \ \texttt{C} < \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]}, \ \overline{X} > \ \{\overline{\texttt{Tf}}; \ \overline{\texttt{W}}\} \ \texttt{OK} \\ & (\texttt{T-CLASS}) \end{split}$$

$$\begin{split} \mathbf{Method typing} \ \overline{\Psi; \Delta; \Gamma \vdash \mathbb{W} \ \texttt{OK}} \\ & \Psi' = \Psi, \overline{X \rightarrow [\bot \bigcirc]} \qquad \Delta' = \Delta, \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]} \qquad \Gamma = \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} >, \ \overline{\texttt{x}: \texttt{T}} \\ & \Delta; \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} > \vdash \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]} \ \texttt{OK} \qquad \Psi'; \Delta'; \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} >, \ \overline{\texttt{x}: \texttt{T}} \\ & \Phi'; \Delta; \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} > \vdash \overline{\texttt{o} \rightarrow [\texttt{b}_l \ \texttt{b}_u]} \ \texttt{OK} \qquad \Psi'; \Delta'; \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} > \vdash \texttt{T}, \ \overline{\texttt{T}} \ \texttt{OK} \\ & \Psi'; \Delta ; \texttt{this}: \texttt{C} < \overline{\texttt{o}'}, \ \overline{\texttt{Y}} > \vdash \texttt{c} \rightarrow [\texttt{b}_l \ \texttt{b}_u], \ \overline{\texttt{Y}} > \texttt{T} \ \texttt{m}(\overline{\texttt{Tx}}) \ \{\texttt{return e}; \} \ \texttt{OK} \\ & (\texttt{T-METHOD}) \end{split}$$

Figure 4.7: Jo $\exists$  typing rules for classes and methods.

### 4.2.6 Dynamic Semantics

The operational semantics of  $Jo\exists$  is defined by the reduction rules given in figure 4.8 and figure 4.9.

The notation  $m[i \mapsto x]$ , used in R-ASSIGN, gives the mapping m, modified to map i to x. Method invocation, R-INVK, is similar to Tame FJ; however, most substitutions are performed by the function mBody (figure 4.6). For example, if  $\mathcal{H} = \{1 \to 0bj_1, 2 \to 0bj_1, 3 \to 0bj_1\}$ where we use  $0bj_i$  to denote an object in the heap, then  $\mathcal{H}[2 \mapsto 0bj_4] = \{1 \to 0bj_1, 2 \to 0bj_1, 2 \to 0bj_4, 3 \to 0bj_1\}$ .

R-OPEN-CLOSE is taken from the classical formulations of existential types [64, 74] (see FUN-R-OPEN-PACK in section 2.3.2). It reduces open and close expressions together by eliminating the open and close sub-expressions, leaving the body of open (e) with formal variables replaced by the packed value and hidden contexts. For example, where 2 and 3 are addresses in the heap,

```
open
    close 3 with o hiding 2
as x,o in
    this.<o>m(x);
```

Computation  $| e; \mathcal{H} \rightsquigarrow e; \mathcal{H}$ 

 $\iota.\langle \overline{\mathbf{r}}, \overline{\mathbf{T}} \rangle \mathbf{m}(\overline{\mathbf{v}}); \mathcal{H} \rightsquigarrow [\mathbf{v}/\mathbf{x}] \mathbf{e}; \mathcal{H}$ (R-INVK)

open (close v with 
$$\overline{o \rightarrow [b_l \ b_u]}$$
 hiding  $\overline{r}$ ) as x, $\overline{o}$  in e; $\mathcal{H} \rightsquigarrow [\overline{r/o}, v/x]e;\mathcal{H}$   
(R-OPEN-CLOSE)

Congruence  $e; \mathcal{H} \rightsquigarrow e; \mathcal{H}$ 

 $\mathcal{H}'$ 

$$\begin{array}{c} \underline{\mathsf{e}'; \mathcal{H} \leadsto \mathsf{e}''; \mathcal{H}' \quad \mathsf{e}'' \neq \mathsf{err}} \\ \hline \iota \cdot \mathsf{f} = \mathsf{e}'; \mathcal{H} \leadsto \iota \cdot \mathsf{f} = \mathsf{e}''; \mathcal{H}' \\ (\text{RC-Assign}) \end{array} \qquad \begin{array}{c} \underline{\mathsf{e}_i; \mathcal{H} \leadsto \mathsf{e}_i'; \mathcal{H}' \quad \underline{\mathsf{e}_i' \neq \mathsf{err}} \\ \hline \iota \cdot \langle \overline{\mathsf{r}}, \ \overline{\mathsf{R}} \rangle \mathsf{m}(\overline{\mathsf{v}}, \ \mathbf{e}_i, \ \overline{\mathsf{e}}); \mathcal{H} \leadsto \iota \cdot \langle \overline{\mathsf{r}}, \ \overline{\mathsf{R}} \rangle \mathsf{m}(\overline{\mathsf{v}}, \ \mathbf{e}_i', \ \overline{\mathsf{e}}); \mathcal{H}' \\ \end{array}$$

$$(\text{RC-Invk})$$

$$\begin{array}{c} e; \mathcal{H} \rightsquigarrow e''; \mathcal{H}' \quad e'' \neq err\\ \hline \text{open e as x,} \overline{\circ} \text{ in } e'; \mathcal{H} \rightsquigarrow \text{ open } e'' \text{ as x,} \overline{\circ} \text{ in } e'; \mathcal{H}'\\ (\text{RC-OPEN}) \end{array}$$

$$\begin{array}{c} \mathsf{e}; \mathcal{H} \rightsquigarrow \mathsf{e}'; \mathcal{H}' \quad \mathsf{e}' \neq \mathtt{err} \\ \hline \mathtt{close \ e \ with \ } \overline{\mathsf{o} \rightarrow [\mathtt{b}_l \ \mathtt{b}_u]} \ \mathtt{hiding \ } \overline{\mathtt{r}}; \mathcal{H} \rightsquigarrow \mathtt{close \ e' \ with \ } \overline{\mathsf{o} \rightarrow [\mathtt{b}_l \ \mathtt{b}_u]} \ \mathtt{hiding \ } \overline{\mathtt{r}}; \mathcal{H}' \\ \hline \mathtt{(RC-CLOSE)} \end{array}$$

Figure 4.8:  $Jo\exists$  reduction rules.

reduces to: this.<2>m(3) (we replace x by 3 and o by 2).

Object creation is performed in R-NEW which appends a new object record to the heap at an unused address. The new object is created with all its fields set to null; i.e., we do not support constructors. This approach contrasts with Tame FJ, Wild FJ [60], and similar formalisms, where objects are created with all fields assigned and no null entity. Whilst that is a simple protocol, it does not extend to ownership systems, since we might have to name the newly Exceptional computation  $| e; \mathcal{H} \rightarrow e; \mathcal{H}$ 

$\texttt{null.f}; \mathcal{H} \leadsto \texttt{err}; \mathcal{H}$	$\texttt{null.f} \texttt{ = e}; \mathcal{H} \leadsto \texttt{err}; \mathcal{H}$	null.< $\overline{r}$ , $\overline{T}$ >m( $\overline{e}$ ); $\mathcal{H} \rightsquigarrow err; \mathcal{H}$
(R-FIELD-NULL)	(R-Assign-Null)	(R-INVK-NULL)

 $\frac{\mathbf{e}'; \mathcal{H} \rightsquigarrow \operatorname{err}; \mathcal{H}'}{\iota. \mathbf{f} = \mathbf{e}'; \mathcal{H} \rightsquigarrow \operatorname{err}; \mathcal{H}'} \qquad \qquad \frac{\mathbf{e}_i; \mathcal{H} \rightsquigarrow \operatorname{err}; \mathcal{H}'}{\iota. \langle \overline{\mathbf{r}}, \ \overline{\mathbf{T}} \rangle \mathbf{m}(\overline{\mathbf{v}}, \ \mathbf{e}_i, \ \overline{\mathbf{e}}); \mathcal{H} \rightsquigarrow \operatorname{err}; \mathcal{H}'}$ (RC-ASSIGN-ERR) (RC-INVK-ERR)

$$\begin{array}{c} \mathsf{e}; \mathcal{H} \rightsquigarrow \mathsf{err}; \mathcal{H}' \\ \hline \mathsf{close \ e \ with \ } \overline{\mathsf{o} \! \rightarrow \! [\mathsf{b}_l \ \mathsf{b}_u]} \ \mathsf{hiding \ } \overline{\mathsf{r}}; \mathcal{H} \rightsquigarrow \mathsf{err}; \mathcal{H}' \\ \hline (\mathsf{RC}\text{-}\mathsf{CLose\text{-}Err}) \end{array}$$

 $\begin{array}{c} \mathsf{e}; \mathcal{H} \rightsquigarrow \mathsf{err}; \mathcal{H}' \\ \hline \\ \hline \mathsf{open } \mathsf{e} \ \mathsf{as } \mathsf{x}, \overline{\mathsf{o}} \ \mathsf{in } \mathsf{e}'; \mathcal{H} \rightsquigarrow \mathsf{err}; \mathcal{H}' \\ \\ (\mathrm{RC-OPEN-ERR}) \end{array}$ 

Figure 4.9: Jo $\exists$  reduction rules for null and error propagation.

created object to assign into a field. For example, if we have a class C<o> with a single field f with type D<this>, there is no way to initialise f in a Tame FJ style constructor. This can be done using Jo∃ constructors:

C<o1> c = new C<o1>; c.f = new D<c>;

An alternative object construction protocol is used in MOJO [23]. In this system there are no constructors, but fields are initialised to default objects (each field is initialised to an object of the declared type) rather than null. The MOJO protocol is motivated by final fields, not present in Jo $\exists$ . It is more complex than the Jo $\exists$  approach, but avoids null expressions and the associated definitions.

In order to support null we give rules (R-...-NULL) to handle null pointer exceptions, these occur when null appears as the receiver in an expression. These rules result in an error configuration consisting of a heap and err. Rules RC-...-ERR propagate these errors to the top level.
Values in Jo∃ consist of addresses in the heap, addresses wrapped in a close expression, and null and err values (which are described above). A close value represents an address that is referred to in a variable with existential type. The close wrapper does not affect the address's behaviour or place in the heap, except to restrict access by obscuring the address's type.

Well-formed heaps and configurations  $|\Delta \vdash \mathcal{H} \text{ OK}| |\Delta; \mathcal{H} \vdash e \text{ OK}$ 

$\forall \iota \to \{ \mathbb{C} < \overline{\mathbb{r}}, \ \overline{\mathbb{T}} > ; \overline{\mathbb{f} \to \mathbb{v}} \} \in \mathcal{H} :$	
$\emptyset; \Delta; \mathcal{H} \vdash C < \overline{r}, T > OK$	
$fType(\mathtt{f},\iota,\mathtt{C}{<}\overline{\mathtt{r}}$ , $\overline{\mathtt{T}}{>})=\mathtt{T}'$ $\emptyset;\Delta;\mathcal{H}\vdash\mathtt{v}:\mathtt{T}'$	$\Delta \vdash \mathcal{H} \text{ OK}$
$\forall \mathbf{v} \in \overline{\mathbf{v}} : add(\mathbf{v}) \ defined \Rightarrow add(\mathbf{v}) \in dom(\mathcal{H})$	$\forall \iota \in fv(\mathbf{e}) : \iota \in dom(\mathcal{H})$
$\Delta \vdash \mathcal{H}$ ок	$\Delta; \mathcal{H} \vdash$ e OK
(F-HEAP)	(F-CONFIG)

Figure 4.10:  $Jo\exists$  well-formed heaps and configurations.

In figure 4.10 we give the definitions of well-formed heaps and configurations; these are used in the proof of properties of Jo $\exists$ . A heap is well-formed if each object in the heap has a well-formed runtime type and each value stored in the fields of each object has the type of that field. Most premises are standard. We insist that the addresses of all referenced values are in the domain of the heap. The address of a value is given by the partial function *add*, defined as:

$$add(\mathbf{v}) = \begin{cases} \iota, & \text{if } \mathbf{v} = \iota \\ add(\mathbf{v}'), & \text{if } \mathbf{v} = \texttt{close } \mathbf{v}' \dots \\ undefined, & \text{otherwise} \end{cases}$$

which recursively unwraps abstract packages, returning the address within. Thus, add(v) is defined if v is neither null nor null wrapped in a close expression.

A configuration consists of a heap and an expression. A configuration is well-formed if the heap is well-formed and any addresses in the expression are in the domain of the heap.

At runtime, we wish to use a heap as a variable environment ( $\Gamma$ ) in Jo $\exists$  judgements. We formalise this notion in the rules H-..., given in figure 4.11. In these rules we extract the addresses and their types from the heap and add these to the environment  $\Gamma$ .

$ \begin{array}{c} \mathcal{H} = \overline{\iota \to \{\mathtt{R}; \ \ldots\}} \\ \underline{\Delta; \overline{\iota: \mathtt{R}}, \Gamma \vdash \mathtt{b} \preceq \mathtt{b}'} \\ \hline \overline{\Delta; \mathcal{H}, \Gamma \vdash \mathtt{b} \preceq \mathtt{b}'} \end{array} $	$\mathcal{H} = \overline{\iota \to \{\mathtt{R}; \ldots\}}$ $\Delta; \overline{\iota: \mathtt{R}, \Gamma \vdash \mathtt{T} <: \mathtt{T}'}$ $\Delta; \mathcal{H}, \Gamma \vdash \mathtt{T} <: \mathtt{T}'}$	$ \begin{array}{c} \mathcal{H} = \overline{\iota \to \{\mathtt{R}; \ldots\}} \\ \underline{\Delta; \overline{\iota: \mathtt{R}}, \Gamma \vdash \mathtt{b} \text{ ok}} \\ \hline \Delta; \mathcal{H}, \Gamma \vdash \mathtt{b} \text{ ok} \end{array} $
(H-I)	(H-S)	(H-FO)
$\mathcal{H} = \overline{\iota \to \{R; \ldots\}}$ $\Delta; \overline{\iota:R}, \Gamma \vdash \Delta' \text{ ok}$	$\mathcal{H} = \overline{\iota \longrightarrow \{\mathtt{R}; \ldots\}}$ $\Psi; \Delta; \overline{\iota: \mathtt{R}, \Gamma \vdash \mathtt{T} \text{ ok}}$	$\mathcal{H} = \overline{\iota \longrightarrow \{\mathtt{R}; \ldots\}}$ $\Psi; \Delta; \overline{\iota: \mathtt{R}}, \Gamma \vdash \mathtt{e} : \mathtt{T}$
$\Delta; \mathcal{H}, \Gamma \vdash \Delta'$ ок	$\Psi; \Delta; \mathcal{H}, \Gamma \vdash T \text{ ok}$	$\Psi;\Delta;\mathcal{H},\Gamma\vdash e:\mathtt{T}$
(H-F-ENV)	(H-F)	(H-T)

Figure 4.11: Using the heap as an environment in  $Jo\exists$ .

#### **Type Soundness**

Type soundness in Jo $\exists$  guarantees that the types of variables accurately reflect their contents, including ownership information. Furthermore, the ownership hierarchy defined statically in a program describes the heap when that program is executed. Although these properties do not constitute an encapsulation property, they are necessary when using ownership information to reason about programs, for example using effects [32]. We show type soundness for Jo $\exists$  by proving progress and preservation (subject reduction) properties (see section 2.1.3):

**Theorem (progress)** For any  $\mathcal{H}, \mathbf{e}, \mathbf{T}$ , if  $\emptyset; \emptyset; \mathcal{H} \vdash \mathbf{e} : \mathbf{T}$  and  $\emptyset \vdash \mathcal{H}$  OK then either there exists  $\mathbf{e}', \mathcal{H}'$  such that  $\mathbf{e}; \mathcal{H} \rightsquigarrow \mathbf{e}'; \mathcal{H}'$  or there exists  $\mathbf{v}$  such that  $\mathbf{e} = \mathbf{v}$ .

**Theorem (subject reduction)** For any  $\Delta$ ,  $\mathcal{H}$ ,  $\mathcal{H}'$ , e, e', T, if  $\emptyset$ ;  $\Delta$ ;  $\mathcal{H} \vdash e$  : T and e;  $\mathcal{H} \rightarrow e'$ ;  $\mathcal{H}'$  and  $\Delta$ ;  $\mathcal{H} \vdash e$  OK and  $\emptyset$ ;  $\mathcal{H} \vdash \Delta$  OK and  $e' \neq err$  then  $\emptyset$ ;  $\Delta$ ;  $\mathcal{H}' \vdash e'$  : T and  $\Delta$ ;  $\mathcal{H}' \vdash e'$  OK.

In these proofs we make use of well-formed variable environments ( $\Gamma$ ). We simply require that all types in the range of  $\Gamma$  are well-formed:

 $\frac{\Psi; \Delta; \Gamma \vdash \mathsf{T} \text{ OK}}{\Psi; \Delta \vdash \overline{\gamma:\mathsf{T}} \text{ OK}}$ (F-Gamma)

The proof of soundness for Jo $\exists$  is fairly straightforward, though long and intricate (53 lemmas). We require many common sense properties (weakening, inversion, etc.); three sets of substitution lemmas: to deal with substitution of types for type variables (T/X), actual context parameters for formal (a/o), and values for variables (v/x); properties of well-formed types; properties that ensure well-formedness as the result of typing; and properties of the heap at runtime.

# 4.3 Deep Ownership

Owners-as-dominators is a strong encapsulation property found in deep ownership systems (section 2.5.1). A heap satisfies the owners-as-dominators property if, for any reference in the heap  $\mathcal{H} \vdash \iota \rightarrow \iota'$  (we use  $\rightarrow$  to mean that  $\iota$  holds a reference to  $\iota'$ ),  $\iota$  is inside the owner of  $\iota'$ [30], see section 2.5.1. It has not been clear that a system with variant subtyping can support owners-as-dominators; invariance of an object's owner has been a crucial element in proving owners-as-dominators in previous systems. We show that it is possible to enforce owners-asdominators in a system with subtype variance by relying on the lower bounds of existential types.

**Owner Lookup Functions**  $|own_{\Psi}(\mathtt{T})||glb(\mathtt{b})||own_{\mathcal{H}}(\mathtt{v})|$ 

 $\begin{array}{c} \hline \hline wn_{\Psi}(\mathbb{C}\langle\overline{\mathbf{a}},\ \overline{\mathbf{T}}\rangle) = \mathbf{a}_{0} & \hline \underline{\Psi}(\mathbb{X}) = [\mathbb{b}_{l}\ \mathbb{b}_{u}] \\ \hline wn_{\Psi}(\mathbb{X}) = \mathbb{b}_{l} & \hline wn_{\Psi}(\exists\Delta.\mathbb{C}\langle\overline{\mathbf{a}},\ \overline{\mathbf{T}}\rangle) = glb_{\Delta}(\mathbf{a}_{0}) \\ \hline \underline{b \notin dom(\Delta)} \\ glb_{\Delta}(\mathbb{b}) = \mathbb{b} & \hline \underline{\Delta}(\mathbf{o}) = [\mathbb{b}_{l}\ \mathbb{b}_{u}] \\ \hline glb_{\Delta}(\mathbf{o}) = glb_{\Delta}(\mathbb{b}_{l}) \\ \hline \underline{\mathcal{H}}(l) = \{\mathbb{C}\langle\overline{\mathbf{r}},\ \overline{\mathbf{T}}\rangle \dots\} \\ \hline wn_{\mathcal{H}}(l) = \mathbf{r}_{0} & \hline wn_{\mathcal{H}}(\operatorname{close}\ v\ \text{with}\ \overline{\mathbf{o}} \rightarrow [\mathbb{b}_{l}\ \mathbb{b}_{u}]\ \operatorname{hiding}\ \overline{\mathbf{r}}) = own_{\mathcal{H}}(\mathbf{v}) \end{array}$ 

Figure 4.12: Owner lookup functions for  $Jo \exists_{deep}$ .

 $Jo\exists_{deep}$  enforces the owners-as-dominators property. It differs from  $Jo\exists$  only in its definition of

well-formed types, heaps, and classes. We give auxiliary functions used to find the owner of an object in the heap  $(own_{\mathcal{H}}(\mathbf{v}))$  and the owner of objects with type T  $(own_{\Psi}(T))$  in figure 4.12. We use  $own_{\mathcal{H}}$  to define the owners-as-dominators property of a well-formed heap.

In effect,  $own_{\mathcal{H}}$  does not distinguish regular objects from objects that are abstracted by close expressions. Our first attempt was to use the first context parameter of the type of  $\mathbf{v}$  as  $own_{\mathcal{H}}(\mathbf{v})$ . Although simple, the result may be a context that is out of scope, if it is existentially quantified in the type of  $\mathbf{v}$ , and less precise than the owner of the hidden object. For example, if  $\mathbf{v}$  has type  $\exists \mathbf{o} \rightarrow [\mathbf{b} \ \mathbf{a}] . \mathbb{C} < \mathbf{o}$ ,  $\mathbf{b} >$ , then using  $\mathbf{o}$  as the result of  $own_{\mathcal{H}}$  then it would be out of scope. Including the quantifying environment in the result of  $own_{\mathcal{H}}$  addresses the scope issue, but the result is still not precise enough. In our example, the hidden owner of  $\mathbf{v}$  will be some precise context, but the bounds on  $\mathbf{o}$  will only give a partial description of that context. Our second attempt substituted the hidden contexts of close for the hiding contexts. For example,  $own_{\mathcal{H}}(\texttt{close } \mathbf{v} \text{ with } \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]$  hiding  $\mathbf{\bar{r}}$ ) would be  $[\mathbf{r}/\mathbf{o}] \mathbf{o}_0$  where  $\mathbf{o}_0$  is the first context parameter in the type of  $\texttt{close } \mathbf{v} \text{ with } \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]$  hiding  $\mathbf{\bar{r}}$ . This gives the same results as the current definition, but is more complicated.

The owner of objects of type X is the lower bound on the owner of X, recorded in  $\Psi$ . To find the owner of objects with existential type ( $\exists \Delta . C < \overline{a}, \overline{T} >$ ), we must find a context that is not quantified and that is inside the declared owner of the type ( $a_0$ ). This is accomplished by the *glb* function; *glb*<sub> $\Delta$ </sub>(b) finds the outermost object that is inside b and not in the domain of  $\Delta$ . This is done by repeatedly taking the lower bound of any formal owner in  $\Delta$ . To see how this works, consider the following three examples:

1. 
$$own_{\Psi}(\exists o \rightarrow [o2 \bigcirc] . C < o1, o>) = glb_{\{o \rightarrow [o2 \bigcirc]\}}(o1) = o1$$
  
2.  $own_{\Psi}(\exists o \rightarrow [o2 \bigcirc] . C < o>) = glb_{\{o \rightarrow [o2 \bigcirc]\}}(o) = o2$   
3.  $own_{\Psi}(\exists o1 \rightarrow [o3 \bigcirc] , o2 \rightarrow [o1 \bigcirc] . C < o2>) = glb_{\{o1 \rightarrow [o3 \bigcirc] , o2 \rightarrow [o1 \bigcirc]\}}(o2) = o3$ 

Example 1 demonstrates the simplest case where the owner (o1) in a type is unquantified, the result of  $own_{\Psi}$  is simply 01. The function *glb* has no affect on o1 since o1 is not in the environment passed to *glb* ( $o \rightarrow [o2 \bigcirc]$ ). Example 2 the owner in the type (o) is a quantified type, but o's lower bound is not in the domain of the quantifying environment and so can be used as the greatest lower bound. In example 3 both the owner in the type (o2) and its lower bound (o1) are in the quantifying environment so we must keep taking the lower bound to get o3 which is not in the environment.

The owners-as-dominators property manifests itself as an extra constraint on well-formed heaps, highlighted with a box:

$$\begin{array}{c} \forall \iota \to \{\mathbb{C} < \overline{\mathbb{r}}, \ \overline{\mathbb{T}} >; \overline{\mathbb{f}} \rightarrow \overline{\mathbb{v}}\} \in \mathcal{H} :\\ & \emptyset; \Delta; \mathcal{H} \vdash \mathbb{C} < \overline{\mathbb{r}}, \ \overline{\mathbb{T}} > \text{OK} \\ \hline fType(\mathbb{f}, \iota, \mathbb{C} < \overline{\mathbb{r}}, \ \overline{\mathbb{T}} >) = \mathbb{T}' \qquad \emptyset; \Delta; \mathcal{H} \vdash \overline{\mathbb{v} : \mathbb{T}'} \\ \forall \mathbb{v} \in \overline{\mathbb{v}} : add(\mathbb{v}) \ defined \Rightarrow add(\mathbb{v}) \in dom(\mathcal{H}) \qquad \overline{\Delta}; \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\mathbb{v}_i) \\ \hline \Delta \vdash \mathcal{H} \text{ OK} \\ & (\mathrm{F}\text{-HEAP}) \end{array}$$

 $Jo\exists_{deep}$  requires some modifications to the well-formedness rules for classes and class types of  $Jo\exists$ :

$$\begin{split} & \underbrace{\Psi = \overline{X \rightarrow [o_0 \bigcirc]}}_{\label{eq:this:C$$

$$\begin{array}{c} \text{class } \underline{\mathsf{C}} < \overline{\mathsf{o} \to [\overline{\mathsf{b}}_l \ \mathsf{b}_u]}, \ \overline{\mathsf{X}} > \dots & \Delta; \Gamma \vdash \overline{\mathsf{a}} \ \mathsf{OK} \\ \Delta; \Gamma, \texttt{this} : \underline{\mathsf{C}} < \overline{\mathsf{a}}, \ \overline{\mathsf{X}} > \vdash \overline{[\overline{\mathsf{a}}/\mathsf{o}]} \mathbf{b}_l \preceq \mathsf{a} & \Delta; \Gamma, \texttt{this} : \underline{\mathsf{C}} < \overline{\mathsf{a}}, \ \overline{\mathsf{X}} > \vdash \overline{\mathsf{a}} \preceq \overline{[\overline{\mathsf{a}}/\mathsf{o}]} \mathbf{b}_u \\ \hline \forall \mathsf{a}_i \in \overline{\mathsf{a}} : \Delta; \Gamma \vdash \mathsf{a}_0 \preceq \mathsf{a}_i \\ \Psi; \Delta; \Gamma \vdash \overline{\mathsf{T}} \ \mathsf{OK} & |\overline{\mathsf{T}}| = |\overline{\mathsf{X}}| \\ \Psi; \Delta; \Gamma \vdash \mathsf{C} < \overline{\mathsf{a}}, \ \overline{\mathsf{T}} > \mathsf{OK} \\ (\text{F-CLASS}) \end{array}$$

The extra premises in F-CLASS (together with the well-formedness rules for contexts) ensure that only contexts that are outside an object can be formed by substitution of actual for formal parameters in its class. The owner of an object  $(\mathbf{a}_0)$  is, by definition, outside that object.

The first extra premise ensures that the actual context parameters are outside  $a_0$ . The second premise ensures that the owners of any actual type parameters are outside  $a_0$ . Therefore, all types formed by substitution of either contexts or types will have an owner outside this.

In the following example (figure 4.13), all types are well-formed in Jo∃ and the types of f1, f3, and f5 are well-formed in Jo∃<sub>deep</sub>, but the types of f2, f4, and f6 are not. The types of f2, f4, and f6 are malformed because of the additional premises in F-CLASS. The data in the lists f2 and f4 are owned by this, which is inside o. The data in f6 is owned by some owner, but there is no guarantee that this owner is outside o. Therefore, these lists can hold references into an object's representation, as shown in m(), which violates owners-as-dominators.

```
class C<o> {
    List<this, o> f1;
                                                               //OK
    List<o, this> f2;
                                                               //type error
    GenericList<this, Object<o>> f3;
                                                               //OK
    GenericList<o, Object<this>> f4;
                                                               //type error
    \exists oo \rightarrow [o \bigcirc].GenericList<this, Object<oo>> f5;
                                                               //OK
    \exists oo \rightarrow [\bot o].GenericList < this, Object < oo >> f6;
                                                               //type error
    void m() {
         f1.datum = new Object<o>();
         f2.datum = new Object<this>();
                                                               //breaks o-as-d
         f3.datum = new Object<o>();
         f4.datum = new Object<this>();
                                                               //breaks o-as-d
         f5.datum = close new Object<o>() with oo \rightarrow [o \bigcirc] hiding o;
         f6.datum = close new Object<this>() with oo \rightarrow [\perp o] hiding this;
                                                               //breaks o-as-d
    }
}
```

Figure 4.13: Example: owners-as-dominators.

In T-CLASS we change the way  $\Psi$  is created; the lower bounds in  $\Psi$  are the formal owner of the class rather than  $\perp$ . This is required because of the changes we made to F-CLASS. F-CLASS requires that the owner of actual type parameters are outside that object's owner, the changes to T-CLASS allow formal type parameters to satisfy this requirement. For example, the following class declaration would not type check without this change:

```
class C<o, X> {
    C<o, X> f;
}
```

If  $\Psi$  had not been changed, C<o, X> would not be well-formed because  $own_{\Psi}(X)$  could not be derived to be outside o.

The second extra premise of T-CLASS ensures that  $\perp$  cannot appear as a bound in the formal context parameters of the class<sup>9</sup>, nor in any existential types given to fields in the class<sup>10</sup>. The intention is to ensure that the owner of all objects referenced by objects of the class (including the hidden owner of objects with existential type) is outside the referring object. This is done by restricting nameable contexts to the class's formal context parameters (which are outside this by definition in the case of the owner, and F-CLASS and T-CLASS for the other contexts), this, and  $\bigcirc$ . Thus, any type in the class declaration has an owner that is guaranteed (together with the constraints on substitution in F-CLASS) to be outside this. If a field **f** is declared to have an existentially quantified owner, the hidden owner of objects in **f** must be in the range defined by the declared bounds. Since we can only name contexts outside this.

We state that the owners-as-dominators property in  $Jo \exists_{deep}$  holds:

**Theorem (Owners-as-dominators)** For any  $\mathcal{H}$ , if  $\Delta \vdash \mathcal{H}$  OK then  $\forall \iota \rightarrow \{\mathbb{R}; \{\overline{\mathbf{f}} \rightarrow \mathbf{v}\}\} \in \mathcal{H}$ ,  $\forall \mathbf{v}_i \in \overline{\mathbf{v}} : \Delta; \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\mathbf{v}_i)$ 

This theorem states that For all well-formed heaps, all references to a value come from inside the owner (as defined by  $own_{\mathcal{H}}$ ) of that value. It is given by the added premise to F-HEAP; we prove that this premise is maintained under execution as part of the proof of subject-reduction. The proof is given in appendix B.

We require several extra lemmas (12) and adjustments to lemmas (such as weakening and substitution lemmas) to handle the extra functions and premises we have introduced. The real work is in proving that the ownership invariant in F-HEAP is preserved in the R-ASSIGN case of subject reduction. In this case, we must show that the owner of the new value (v) in the

<sup>&</sup>lt;sup>9</sup>In this premise, we use  $\notin$  to mean "does not appear anywhere in", this is a little informal, but corresponds to not being in the set of free variables of an expression or type, except that  $\perp$  is not a variable.

<sup>&</sup>lt;sup>10</sup>It would be easier to forbid  $\perp$  occurring at all by removing F-BOTTOM and I-BOTTOM. However, to allow maximum flexibility of polymorphic methods, we allow  $\perp$  in the bounds of their formal context and type parameters.

field of the assigned object (with address  $\iota$ ) is outside  $\iota$ . We will describe the lemmas that are used to do this.

We show that an object is inside (or equal to) the declared owner of each of its fields (lemma 61). The proof of this lemma relies on the premises added to F-CLASS and T-CLASS. We show that the only context parameters that can be named (and thus the declared owners of all fields) are outside (or equal to) the object in which the fields occur. If fields have existential type then we show that all possible hidden owners are outside the object by considering the lower bounds of the existential type.

We show in lemma 65 that the owner of a value with type T is outside the owner declared in T. Existentially typed values (close values) are interesting; we must show that the owner of the packed value is within the bounds declared in the close value and thus outside the declared owner, which is defined in terms of the lower bound on the quantified context parameter. We make use of the property that subtyping restricts ownership (lemma 63). This property relaxes the usual property of ownership systems that owners are invariant with respect to subtyping (which holds in Jo $\exists$  for non-existential types). Our property follows from the definition of subtyping, and, together with our definition of ownership for existentially typed values, is strong enough to prove lemma 65.

Lemma 44 states that the context parameters in an inside relation are either equal or wellformed. This property is interesting since there is no equivalent property of subtyping for Tame  $FJ^{11}$ . The lemma allows us to reason about the ownership hierarchy in ways which we cannot about subtype hierarchies. For example, in lemma 62, where we show that the *glb* function preserves the inside relation between contexts.

<sup>&</sup>lt;sup>11</sup>Indeed we spent some time wrestling with variations on such a lemma for subtyping in our early work with wildcards. Our lack of success led to the splitting of subtyping in Tame FJ into subclassing, extended subclassing, and subtyping (section 3.1.2).

# 4.4 Discussion

The expressivity of types in Jo $\exists$  comes from the combination of existential quantification of contexts and type parameterisation. The formalisation of Jo $\exists$  follows from these starting points and the decision to use explicit packing and unpacking, which simplifies the type rules and proofs for Jo $\exists$ . The natural and uniform emergence of the calculus is reassuring. In this section we discuss some of the decisions taken in the design of Jo $\exists$  and some of Jo $\exists$ 's interesting features.

**Explicit packing and unpacking**  $Jo\exists$  (in contrast to Tame FJ) uses close and open expressions to explicitly pack and unpack existential types. This makes for a simpler formalism and easier reasoning. On the other hand, it means that  $Jo\exists$  is further from a realistic, usable programming language. However, as we discuss in section 4.5, there is a simple translation from  $Jo\exists$  to a language with implicit operations on existential types.

Furthermore, (as opposed to models for wildcards, section 3.3.2), there is no loss of expressivity when dealing with quantification of owner variables. This is because in Jo $\exists$ , packing and unpacking is not required to check well-formedness; open and close expressions could not be used in these checks because they are subtype checks and there is no place for expressions. This is discussed in section 5.1.

Allowing abstract packages to be values (and thus stored in the heap) follows earlier work [27, 64, 45] on existential types and is a natural consequence of explicit packing. However, the owners-as-dominators property is usually phrased assuming that all values are objects (addresses in  $Jo\exists_{deep}$ ). We must therefore consider how to describe owners-as-dominators in the presence of abstract packages. We do this by not distinguishing between abstract packages and the objects that they abstract. This ensures that existential quantification cannot hide violations of owners-as-dominators.

**F-bounds** There is no need to support F-bounded polymorphism [25] (section 2.2.1) in Jo $\exists$ . A formal context parameter cannot appear as a parameter in a bound, since bounds are contexts,

not types (as in Tame FJ, for example,  $X \rightarrow [\perp C < X >]$ ). This simplifies the treatment of bounds in well-formed environments and the inside relation. Since we do not have to consider F-bounds, there can be no recursion in the definition of environments. As a specific example of why this is an advantage, we can always find a lower bound of a formal owner that does not involve any formal context parameters (as is done by the *glb* function in figure 4.12); this is not possible in Java with wildcards [60].

I-OWNER The rule I-OWNER is the basis of the relationship between objects and their owners. I-OWNER is interesting because only variables with class type can be derived to be inside their declared owner. To see why this must be the case, we consider the alternatives. If a variable  $(\gamma)$  has a type variable (X) as its type, we would like to derive that  $\gamma$  is inside the declared owner of X; however, no such owner is known. Extending I-OWNER to give the upper bound from  $\Psi$  is problematic; namely, if X was instantiated by an existential type then we could no longer apply I-OWNER and substitution is unsound.

If  $\gamma$  has existential type, we would also face problems. It seems possible to derive that  $\gamma$  is inside the upper bound of its declared owner. For example, if **x** has type  $\exists o \rightarrow [y \ z].C<o>$ , to derive that **x** is inside **z**. We take the position that abstract packages (which have existential type) are not contexts and therefore not part of the ownership hierarchy. Therefore, it does not make sense to state that close... is inside any context.

This restriction on I-OWNER and the similar restriction in F-VAR (see section 4.2.4) are used in the proofs of lemmas 33 to 43. They ensure that substituting  $\mathbf{v}$  for  $\mathbf{x}$  into a context or type results in a syntactically well-formed context or type. Specifically, that a close value is not substituted into a type, although it may be substituted into an expression. This is accomplished because if  $\mathbf{x}$  appears in a well-formed type (T), then  $\mathbf{x}$  must have class type; the type of  $\mathbf{v}$  must be a subtype of the type of  $\mathbf{x}$ , and thus also a class type (lemma 3); close values always have existential type, and therefore, if  $\mathbf{v}$  is a close value, then the substitution  $[\mathbf{v}/\mathbf{x}]$  does not affect T. Syntax of existential types Our syntax of existential types forbids nested quantification of types, such as  $\exists \Delta_1, \exists \Delta_2.N$ . However, any such type can be rewritten using single quantification, such as  $\exists \Delta_1, \Delta_2.N$ . By restricting the syntax in this way, we avoid the need for equivalence rules between types. Alternatively, we could use quantification by a single formal context and allow nested existential types. This scheme follows the classical formulations of existential types (see section 2.3) more closely. For example,  $\exists o1, o2, o3.C < o1, o2, o3 >$  would be written as  $\exists o1. \exists o2. \exists o3.C < o1, o2, o3 >$ . Since we do not allow forward references in bounds, the two versions are equally expressive. If we adopted the single quantification syntax then the open and close expressions would need to be adapted to deal with a single context variable at a time, rather than a context environment at a time. Although this would make their formalisation marginally simpler, it would also mean that we would need a let expression, since open could not be used to encode let.

**Quantification of contexts** Our syntax only allows quantification of types, not contexts; we cannot write  $C<\exists o.o>$ . This type would represent an invariant type parameterised by a variant context. We do not support quantification of contexts in Jo∃ because we are unsure of its precise behaviour, and because it would require significant adjustment to Jo∃.

Following the treatment of nested existential types in  $Jo\exists$  and Tame FJ, C< $\exists o.o>$  should be invariant. How  $\exists o.o$  would then behave as a context is not clear. Furthermore, it is hard to see if an invariant type with unknown owner is useful because variables with this type can only contain objects with an unknown owner; whether it is sensible to construct such objects is also unclear.

To support  $\exists o.o$  as a valid context in Jo $\exists$  would require extensions to well-formedness of contexts and the inside relation. Introduction and elimination of such existential quantification would require additional work. It is not clear how this would be done, as usually packing and unpacking only affects quantification at the outermost scope of a type (e.g.,  $\exists o.C < C < o >>$  and not  $C < \exists o.C < o >>$ ).

Omitting quantified contexts does not make Jo∃ incomplete because types involving quantifi-

cation of contexts are not denotable or expressible in  $Jo\exists$ ; no expression can be constructed that has a type containing  $\exists o.o.$ 

**Enforcing owners-as-dominators** In the type system of  $Jo\exists_{deep}$  we had to extend the usual restrictions found in ownership systems to enforce owners-as-dominators. Requiring context parameters to be outside an object's owner is standard, we needed to extend this to deal with quantified context variables and type parameters. The crucial observation is that, in enforcing owners-as-dominators, we always wish to show that a value is outside the object that refers to it. It is therefore conservative to use a lower bound on a value's owner rather than the value's owner itself. The additional premises in F-CLASS of  $Jo\exists_{deep}$  can thus deal with lower bounds on parameters. In the case of quantified context parameters this means that we can use their greatest lower bound. For type parameters we use the lower bound stored in  $\Psi$ ; this motivates using  $\Psi$  in Jo $\exists$  rather than just a set of type variables.

#### Restrictions on Jo∃

We have simplified Jo $\exists$  by omitting subclassing. We expect that adding subclassing to Jo $\exists$  would be easy and would not make the formal system or proofs more interesting, only longer. The syntax of class declarations would need to be extended with an 'extends' clause. Subtyping and the lookup functions would need to be extended accordingly. S-FULL would have to be modified to allow the quantified types (currently, both N) to be subtypes<sup>12</sup>.

Another restriction on  $Jo\exists$  is the lightweight treatment of type parameters. Adding existential quantification of type parameters is dealt with in Tame FJ (section 3.1). It may be useful to have both quantification of contexts and type parameters in a real language. Each form serves a different purpose and both have strong motivation, so there is no reason to believe a programmer would require only one or the other. Quantification of types does not complicate

<sup>&</sup>lt;sup>12</sup>In terms of the proofs we would require an extra few lemmas that show preservation of field type and method type under subclassing, these are usually easy to prove. We would also need to extend the lemmas that involve subtyping due to the extra rule, again, we expect this to be an easy modification. Corresponding to the change to S-FULL, some lemmas would need to be changed to conclude a subtyping relation between the quantified types, rather than equality.

the treatment of quantification of contexts, because contexts do not contain type information. Therefore, we leave investigation of a system with both forms of quantification to further work.

Jo $\exists$  could also be extended by allowing bounds on formal type variables. In the current system this could be done by allowing the programmer to specify the bounds in  $\Psi$ . We do not expect this extension to cause any significant changes since most of the machinery to deal with bounds in  $\Psi$  is already in place.

If we had also added subclassing, then we could add proper type bounds, in the same way as in Tame FJ. This would require some changes to the formalism and proofs. In particular, we expect to require the separation of subtyping into subclassing, extended subclassing, and subtyping, as in Tame FJ (see section 3.1.2). Again, we hope the orthogonality of ownership and subclassing would mean that all significant issues have been covered in the context of Tame FJ.

Using type bounds subsumes bounds on the owner since types include ownership information. For example, if X has the upper bound C<o1> then we can instantiate X to any subclass of C. Due to the invariance of context parameters<sup>13</sup>, this class has to be owned by o1. By using an existential type as an upper bound<sup>14</sup>, we can specify a range of possible context parameters. For example, if the upper bound of X is  $\exists o \rightarrow [o2 \bigcirc].C<o>$  then any instantiation of X must have an owner bounded by o2 and  $\bigcirc$ .

#### 4.4.1 An Application — Effects

To show how  $Jo\exists$  can be used to improve reasoning about programs we show how it could be combined with effects [48, 32]. Adding an effects system to  $Jo\exists$  is future work, but we imagine how existential types in  $Jo\exists$  could be used with effects. We give an example of a  $Jo\exists$  program with effects in figure 4.14.

 $<sup>^{13}</sup>$ Assuming that the owner of a superclass must be the owner of the subclass, as in [32].

<sup>&</sup>lt;sup>14</sup>This is similar to Clarke's use of a fresh context variable as the owner of the upper bound [30]. It is also hinted at in OGJ, where implicitly bound context parameters of bounding types are described as being similar to wildcard types [78].

```
C<owner> {
      . . .
      void m() {
                                         //effect: owner
}
D<owner, o2> \{
      \exists o \rightarrow [\bot \text{ this}].C < o > a;
      \exists o \rightarrow [owner \bigcirc].C < o > b;
      \exists o \rightarrow [\perp o2].C < o > c;
      void m2() {
            a.m();
                                           //effect: [\perp this]
            b.m();
                                           //effect: [owner ()]
            c.m();
                                           //effect: [\perp o2]
      }
}
```

Figure 4.14: An example of a  $Jo\exists$  program with effects.

Expressions and methods have a single<sup>15</sup> effect which denotes the area of the heap that may be read or written during execution of the code. An effect **a** means that only objects owned by **a** may have been read or written. An effect [**a b**] means that objects owned by contexts that are inside **b** and outside **a** may have been read or written.

In the example (fig 4.14), the method m has the effect owner which means that only objects owned by the owner of the receiver may be accessed in calls to m. Since all the fields of D have existential type, this means that the calls to m in m2 have range effects. This is still useful information, however: since this and owner must be distinct, the range effects of a.m() and b.m() must be disjoint and so these two calls could be reordered or parallelised. Since we have no information about o2 we do not know if the effect of c.m() is disjoint from the other effects and so we cannot reorder this expression.

# 4.5 Wildcards-Style Existential Types

An alternative way to design a system with owner variance would be to perform packing and unpacking implicitly in the subtype and type rules, rather than explicitly using open and close expressions. Such an approach is taken in the Tame FJ formalisation of Java wildcards

 $<sup>^{15}\</sup>mathrm{Most}$  real effects systems separate read and write effects

(section 3.1). In the context of variant ownership, we believe this approach is almost equivalent to using explicit packing and unpacking (in contrast to wildcards where explicit packing and unpacking is less expressive than the implicit approach, see section 3.3.2). In this section we will outline  $Jo\exists_{wild}$ , which uses *implicit* packing and unpacking to implement ownership variance.

 $Jo\exists_{wild}$  is closer to a usable language, at the expense of being more complicated and less transparent than Jo∃. The type rules of  $Jo\exists_{wild}$  are larger and more complex than Jo∃ (although there are less of them). The presence of context parameter inference further obfuscates the type system. The connection from  $Jo\exists_{wild}$  to classical existential types systems [26, 27, 64, 74, 75] is weaker than from Jo∃. For these reasons we have so far formalised ownership variance using *explicit* packing and unpacking. In particular, proof of soundness and owners-as-dominators is simpler in Jo∃ than  $Jo\exists_{wild}$ , thanks to the absence of context parameter inference and simpler type rules.

We imagine  $Jo\exists_{wild}$  is an intermediate language between  $Jo\exists$ , a convenient formalisation, and a real programming language. We outline a straightforward translation from  $Jo\exists_{wild}$  to  $Jo\exists$ and argue that we have investigated all the novel features of  $Jo\exists_{wild}$  (relative to  $Jo\exists$ ) in the context of Tame FJ. Therefore, we do not give a full formalisation or proofs of soundness and owners-as-dominators for  $Jo\exists_{wild}$ .

## **4.5.1** $Jo\exists_{wild}$

We give the syntax of  $Jo\exists_{wild}$  in figure 4.15, the elided syntax of environments and identifiers remains unchanged from  $Jo\exists$ . The differences are the removal of open and close expressions, the addition of  $\star$  to denote context parameters to be inferred during method invocation, and the addition of syntactic category **p** (and its runtime counterpart, **q**). Also, in order to simplify the syntax of types, we require that all types are quantified (possibly by the empty set).

From Jo $\exists$ , we remove the type and reduction rules that handle open and close expressions. The well-formedness rules for contexts, types, and environments, the inside relation, and the operational semantics remain mostly unchanged.

е	::=	$\begin{array}{l} \mbox{null} \mid \mathbf{x} \mid \gamma. \mathbf{f} \mid \gamma. \mathbf{f} = \mathbf{e} \mid \gamma. < \overline{\mathbf{p}}, \\ \mbox{new } \mathbb{C} < \overline{\mathbf{a}}, \ \overline{\mathbf{T}} > \mid \iota \mid \mbox{err} \end{array}$	Ī>m(ē)   expressions
Q W	::= ::=	class C< $\Delta$ , $\overline{X}$ > { $\overline{Tf}$ ; $\overline{W}$ } < $\Delta$ , $\overline{X}$ > Tm( $\overline{Tx}$ ) {return e;}	class declarations method declarations
V	::=	$\iota \mid \texttt{null} \mid \texttt{err}$	values
Ν	::=	C<ā, T>	class types
R	::=	$C < \overline{r}, \overline{T} >$	runtime types
М	::=	N   X	non-existential types
Т	::=	$\exists \Delta . N \mid \exists \emptyset . X$	types
a	::=	$\circ \mid x \mid \bigcirc \mid \iota$	actual owners
r	::=	$\bigcirc \mid \iota$	runtime owners
b	::=	$a \mid \perp$	bounds
р	::=	a   *	owner parameters for methods
q	::=	r   *	runtime owner parameters for methods

Figure 4.15: Syntax of  $Jo\exists_{wild}$ .

The type rules are changed to unpack sub-expressions before they are used in other premises and to pack the result of type checking to prevent free-variable escape. These changes follow Tame FJ (see section 3.1) closely (including adding guarding environments and modifying the subsumption rule); there are no interesting differences in dealing with contexts rather than types.

Actual context parameters to a method invocation that are marked with  $\star$ , must be inferred. This is simpler than in Tame FJ, due to the separation of parameters (contexts) from types (for example, we don't need a *sift* function). We give rules for method invocation and the necessary auxiliary functions in figure 4.16. Compared to Jo $\exists$ , we must do more work in T-INVK and less in *mType*, this is because we use the types of the formal parameters to infer actual context parameters. Inference of context parameters is done by the *match* function. Compared with Tame FJ, matching is simpler in Jo $\exists_{wild}$  since we do not have subclassing. Otherwise, matching and method invocation follows Tame FJ fairly closely.

The subtyping rule S-FULL in Jo∃ is replaced with the following rule, adapted from Tame FJ:

 $\begin{array}{c|c} \texttt{class } \mathbb{C} \overline{\langle \mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u]}, \ \overline{\mathsf{X}} > \ \{ \overline{\mathsf{U}'' \mathsf{f}}; \ \overline{\mathsf{W}} \} & \overline{\langle \mathsf{o}' \rightarrow [\mathsf{b}'_l \ \mathsf{b}'_u]}, \ \overline{\mathsf{X}'} > \ \mathtt{Tm}(\overline{\mathsf{Tx}}) \ \{\texttt{return } \mathsf{e}; \} \in \overline{\mathsf{W}} \\ \hline mType(\mathsf{m}, \gamma, \mathbb{C} \overline{\mathsf{a}}, \ \overline{\mathsf{U}} \overline{\mathsf{v}}) = \ [\overline{\mathsf{a}} \overline{\mathsf{o}}, \ \overline{\mathsf{U}} \overline{\mathsf{X}}, \ \gamma/\texttt{this}](\overline{\mathsf{o}'} \rightarrow [\mathsf{b}'_l \ \mathsf{b}'_u]} \overline{\mathsf{T}} \rightarrow \mathsf{T}) \end{array}$ 

$$\begin{array}{l} \forall j \ where \ \mathbf{p}_{j} = \star : \mathbf{o}_{j} \in fv(\overline{\mathbf{M}'}) & \forall i \ where \ \mathbf{p}_{i} \neq \star : \mathbf{a}_{i} = \mathbf{p}_{i} \\ \hline \mathbf{M} = [\overline{\mathbf{a}/\mathbf{o}'}, \overline{\mathbf{a}'/\mathbf{o}}] \mathbf{M'} \\ \hline dom(\overline{\Delta}) = \overline{\mathbf{o}} & fv(\overline{\mathbf{a}}, \overline{\mathbf{a}'}) \cap \overline{\mathbf{o}}, \overline{\mathbf{o}'} = \emptyset \\ \hline match(\overline{\mathbf{M}}, \overline{\exists \Delta} . \mathbf{M'}, \overline{\mathbf{p}}, \overline{\mathbf{o}'}, \overline{\mathbf{a}}) \end{array}$$

$$\begin{split} \Psi; \Delta; \Gamma \vdash \gamma : \exists \Delta'. \mathbb{N} & \Psi; \Delta; \Gamma \vdash \overline{\mathbf{e}} : \exists \Delta. \overline{\mathbb{M}} \\ \Delta; \Gamma \vdash \overline{p} \text{ OK} & \Psi; \Delta; \Gamma \vdash \overline{\mathbb{U}} \text{ OK} \\ mType(\mathbf{m}, \mathbb{N}) = \langle \overline{\mathbf{o}} \rightarrow [\overline{\mathbf{b}}_l \ \overline{\mathbf{b}}_u], \ \overline{\mathbf{X}} \\ \overline{\mathbf{T}} \rightarrow \mathbf{T} \\ match(\overline{\mathbb{M}}, \overline{\mathbf{T}}, \overline{\mathbf{p}}, \overline{\mathbf{o}}, \overline{\mathbf{a}}) & \Delta, \Delta', \overline{\Delta}; \Gamma \vdash \overline{\mathbb{M}} <: [\overline{\mathbf{a}}/\overline{\mathbf{o}}, \ \overline{\mathbf{U}}/\overline{\mathbf{X}}] \\ \overline{\Delta}, \Delta', \overline{\Delta}; \Gamma \vdash \mathbf{a} \preceq [\overline{\mathbf{a}}/\overline{\mathbf{o}}] \mathbf{b}_u & \Delta, \Delta', \overline{\Delta}; \Gamma \vdash [\overline{\mathbf{a}}/\overline{\mathbf{o}}] \mathbf{b}_l \preceq \mathbf{a} \\ \overline{\Psi}; \Delta; \Gamma \vdash \gamma. \langle \overline{\mathbf{p}}, \ \overline{\mathbb{U}} \\ > \mathbb{m}(\overline{\mathbf{e}}) : [\overline{\mathbf{a}}/\overline{\mathbf{o}}, \ \overline{\mathbf{U}}/\overline{\mathbf{X}}] \\ T \mid \Delta', \overline{\Delta} \end{split}$$

$$(\mathrm{T-INVK}) \end{split}$$

Figure 4.16:  $Jo\exists_{wild}$  rules for method invocation.

$$dom(\Delta') \cap fv(\exists \overline{\mathsf{o}} \to [\overline{\mathsf{b}_l \ \mathsf{b}_u}] . \mathbb{N}) = \emptyset \qquad fv(\overline{\mathsf{a}}) \subseteq dom(\Delta, \Delta')$$
$$\underline{\Delta, \Delta' \vdash [\overline{\mathsf{a}}/\mathsf{o}] \mathfrak{b}_l \preceq \mathsf{a} \qquad \Delta, \Delta' \vdash \mathsf{a} \preceq [\overline{\mathsf{a}}/\mathsf{o}] \mathfrak{b}_u}$$
$$\underline{\Delta; \Gamma \vdash \exists \Delta'. [\overline{\mathsf{a}}/\mathsf{o}] \mathbb{N} <: \exists \overline{\mathsf{o}} \to [\overline{\mathsf{b}_l \ \mathsf{b}_u}] . \mathbb{N}}$$
$$(S-ENV)$$

This is nearly identical to the Tame FJ version given in figure 3.2; the difference is that here we deal with contexts rather than types, both in the substitutions and quantification. We compare contexts using the inside relation rather than subtyping. S-ENV subsumes S-FULL by allowing subtyping between existential types (by using  $\overline{\mathbf{a}} = \overline{\mathbf{o}} = dom(\Delta')$ ). In addition, by using  $\Delta' = \emptyset$ , it allows for the packing of a type by finding a supertype. For example, we can pack C<o1> to its supertype,  $\exists \mathbf{o}.C<o>$ . This can encode any close expression that can be written in Jo $\exists$ .

In terms of well-formed contexts and the inside relationship, the only change from  $Jo\exists$  would be to lift the restriction in F-VAR and I-OWNER that variables must have non-existential type. We expect that we could allow a variable to be a well-formed context if it is defined in  $\Gamma$ , whatever its type. If an object's declared owner is existentially quantified, then we could take upper bounds until we find an un-quantified owner (similarly to the *glb* function (figure 4.12), but taking upper, rather than lower, bounds). For example, if x is mapped to  $\exists o \rightarrow [\bot \text{ this}].C<o>$ , we could derive that x is inside this.

Although heaps have the same syntax in  $Jo\exists_{wild}$  and  $Jo\exists$ , the different syntax of values affects the structure of heaps. In  $Jo\exists$ , close expressions can appear in the heap, referenced by fields with existential type. Fields in  $Jo\exists_{wild}$  point only to addresses. It is simple to translate heaps between the two systems. A  $Jo\exists_{wild}$  heap can be derived from a  $Jo\exists$  heap by replacing all occurrence of close  $\iota$  ... with  $\iota$ . In the other direction, a  $Jo\exists$  heap can be found by wrapping addresses with close expressions wherever a field has existential type. This is slightly more complex since the type of the address must be found and the contexts to be hidden in the close expression must be identified.

Since  $\text{Jo}\exists_{wild}$  heaps do not involve close values, the definition of *add* becomes trivial and  $own_{\mathcal{H}}$  much simpler. Proof of owners-as-dominators and other encapsulation properties should follow easily from the proofs for  $\text{Jo}\exists$ .

### **4.5.2** Programs in $Jo\exists_{wild}$ and $Jo\exists$

There is a simple translation from  $Jo\exists_{wild}$  programs to  $Jo\exists$  programs: implicit unpacking using capture conversion is translated to explicit unpacking using an open expression, implicit packing due to S-ENV is translated to explicit packing using a close expression. Unlike the wildcards case, implicit packing cannot take place in well-formedness checks, and so *all* occurrences of implicit packing can be translated (see sections 3.3.2 and 5.1). The translation is type preserving: the type of a translated expression is the same as the original expression. It is also structure preserving: no extra methods or method calls are created. The *only* effect of translation is inserting open and close expressions. The inverse translation (removing open and close expressions) works in most cases. However, there are some instances where a more complex translation is needed. We will now discuss some examples to show how programs can be translated between  $Jo\exists$  and  $Jo\exists_{wild}$ . We will assume the class declaration of C,

```
class C<oc> {
    C<oc> f;
    <om> void m(C<om> x1, C<om> x2) {...}
}
```

The  $Jo\exists_{wild}$  program is on the left and the equivalent  $Jo\exists$  program is on the right:

Similarly to the correspondence between wildcards and existential types (section 2.4.1), the Jo $\exists$  program must wrap the field access  $\mathbf{x}.\mathbf{f}$  in open and close expressions. Since  $\mathbf{x}$  has existential type, in Jo $\exists$  we must unpack it before its field can be accessed. This is done implicitly in Jo $\exists_{wild}$ . The result of the field access contains a free context variable (labelled o1 in the Jo $\exists$  program). Thus, the result of field access must be re-packed, either explicitly in Jo $\exists$  using close or implicitly in Jo $\exists_{wild}$ . Field assignment and most method calls are translated similarly.

Explicit open expressions (in Jo $\exists$ ) allow for the scope of an unpacked variable to be explicitly defined; implicit unpacking (in Jo $\exists_{wild}$ ) limits this scope to a single sub-expression. Therefore, Jo $\exists$  can express some programs more easily than Jo $\exists_{wild}$ . For example, the following program cannot be translated to Jo $\exists_{wild}$  by erasing the open expression, because each occurrence of **x** in the parameter list of the method invocation would be treated as unique.

```
void eg2(∃o.C<o> x) {
    open x as y,o1 in
        y.<o1>m(y, y);
}
```

Such programs can be translated to  $Jo\exists_{wild}$ , but a more complex translation is required. Explicit unpacking is encoded as a method call to eg2Aux. This method body gives the same scope as the body of the open expression in the Jo∃ program.

```
void eg2Aux(C y) {
    y.m(y, y);
}
void eg2(∃o.C<o> x) {
    this.<*>eg2Aux(x);
}
```

# 4.6 Chapter Summary

In this chapter we have presented, motivated, and discussed  $Jo\exists$ , a novel approach for incorporating subtype variance into a programming language with ownership types. Explicit existential quantification is used to denote variance; type parameterisation is combined with existential quantification of contexts to make a very flexible and expressive system. Explicit packing and unpacking of existential types are used to simplify the formal system and proofs.

We have proved Jo∃ type sound. We have also presented Jo∃<sub>deep</sub> an extension to Jo∃ that satisfies the owners-as-dominators property. The key to stating owners-as-dominators in a language with packed values is to ignore wrapping close expressions so that we include all references, even if they are to packed objects. To satisfy owners-as-dominators, we require that the lower bound on variant contexts are outside the object in which they are used, this ensures that the owner of a hidden object is outside the object that references the hidden object. We have presented Jo∃<sub>wild</sub> a variation of Jo∃ using implicit packing and unpacking that is closer to a usable (as opposed to formal) language.

# Chapter 5

# **Comparisons and Discussion**

In this chapter we compare the systems developed in this thesis with each other and with related work. We compare Tame FJ with Jo $\exists$  in section 5.1. We compare Tame FJ with other models of wildcards in section 5.2, and Jo $\exists$  with ownership systems that support existential types or variance in section 5.3. We compare the two systems in terms of: the use of quantification and the entities that are quantified, the techniques used to formalise the systems, the complexity of the formalisms, and the size and complexity of their proofs.

We show that the use of existential types for subtype variance involves similar concepts, even though the quantified variables are of different kinds. There are, however, some important differences; chiefly that in Tame FJ, but not  $Jo\exists$ , the parameters of types can be used as types themselves.

We show that Tame FJ is closely related to other formalisations of wildcard types and existential types.

We show that  $Jo\exists$  can be used to compare and encode ownership systems that support subtype variance or existential types. Existing mechanisms for supporting subtype variance have the same behaviour as existentially quantified contexts in  $Jo\exists$  and can be easily encoded. In the case of universes [66, 67, 39], using context parameters and existential types rather than universe annotations, give us a clearer picture of the underlying mechanisms used in type checking.  $Jo\exists$ 

can also be used to model existing kinds of existential types in ownership systems. In particular, explicit quantification and unpacking explain clearly the behaviour of "existential owners" in existential downcasting [98].

# 5.1 Comparison of Tame FJ and Jo∃

The use of existential types in Tame FJ and Jo $\exists$  is almost identical. The key difference is that Tame FJ supports quantification of type variables whereas Jo $\exists$  supports quantification of context variables<sup>1</sup>. This is significant because contexts are a separate set of variables from types. Therefore, there is a strict separation between the quantifying entities and types in Jo $\exists$ , whereas in Tame FJ the quantifying variables may appear as types.

Another difference is that packing and unpacking of existential types is implicit in Tame FJ and explicit in Jo $\exists$ . This is a difference in the formalisations rather than a fundamental difference between the two systems. Explicit packing and unpacking allows for a simpler formalism and simpler proofs. However, they cannot be used in Tame FJ, because packing is required in well-formedness checks which cannot accommodate an explicit close expression, see section 3.3.2. The separation of contexts and types in Jo $\exists$  means that well-formedness checks do not require packing (the inside relation is used, rather than subtyping), and so Jo $\exists$  can be formalised using explicit open and close expressions. Jo $\exists$  could also be modelled using implicit packing and unpacking, as discussed in section 4.5.

Explicit packing and unpacking simplify both the formalism and the proofs of soundness and the owners-as-dominators property. Simplifications in the formalism include not having to infer type or context parameters using *match*, not requiring guarding environments to keep track of unpacked variables, and simplified subtyping (S-FULL is a simpler rule than XS-ENV).

In  $Jo\exists_{wild}$ , which supports implicit packing and unpacking, the separation of contexts and types makes parameter inference at method calls simpler. In Tame FJ, parameter inference was a difficult part of the system to formalise. Care had to be taken not to cause problems

<sup>&</sup>lt;sup>1</sup>Although Jo $\exists$  supports type variables, these cannot be existentially quantified

with subject reduction by inferring types rather than type parameters. For example, inferring the value of X in calls to a method with signature <X>void m(X x) (X appears as a type), as opposed to calls to <X>void m(C<X> x) (X appears as a type *parameter*);

In  $Jo\exists_{wild}$ , inference of contexts is simpler; although the *match* rule is similar in both systems, there is no need for *sift* because types cannot be inferred, only contexts.

The quantifying variables of Tame FJ are types and therefore wholly static entities. Contexts in  $Jo\exists$  are treated statically, but represent dynamic entities. Since parameterisation is a per-type, rather then per-class, operation, and since contexts are treated statically, there is surprisingly little difference caused by parameterising by static or dynamic entities.

Both types and contexts are related by hierarchical relations: subtyping and the inside relation, respectively. As discussed in section 2.1.1, subtyping involves inclusion polymorphism, C represents objects with type C or a subtype of C. With contexts, there is no equivalent inclusion, a context, c, means exactly c. However, since contexts only appear as type parameters<sup>2</sup>, this is actually the same situation as with types.

Neither language supports existential quantification of variables: Tame FJ does not support  $\exists x.x$  and  $Jo\exists$  does not support  $\exists o.o.$  It would be possible to add such quantification to Tame FJ by relaxing the syntax of existential types. However, we encountered problems with these types in earlier versions of Tame FJ and it is possible (though unlikely) that there could be problems in the version of Tame FJ presented in this thesis. In  $Jo\exists$ , it is not so easy to add quantification of context variables, because contexts cannot be used as types,  $Jo\exists$  would have to be extended in several ways to support quantified contexts, as discussed in section 4.4.

There are many superficial differences between the two languages:  $Jo\exists$  has two kinds of parameters (types and contexts) and Tame FJ has only type parameterisation;  $Jo\exists$  can be used to enforce owners-as-dominators, there is no equivalent property in Tame FJ<sup>3</sup>; Tame FJ is functional and  $Jo\exists$  imperative;  $Jo\exists$  is more heavily simplified than Tame FJ; the lack of sub-

<sup>&</sup>lt;sup>2</sup>Although some contexts can be used as variables in expressions, where there is no inclusion polymorphism, this is a very different situation to being used as a type.

 $<sup>^{3}</sup>$ It is possible to use type parameterisation to enforce ownership properties [78]. It would be interesting future work to investigate the use of explicit existential types in such a system.

classing in Jo∃ means that the separation of subtyping into subclassing, extended subclassing, and subtyping in Tame FJ is not required and method and field lookup and checking is simplified. None of these differences concern existential quantification, nor are they fundamental differences between the systems.

### 5.1.1 Comparison of Proofs

The proof of soundness for Jo $\exists$  is longer (because the system is imperative and includes more environments, expressions and rules), but more straightforward (because of the simplifications discussed above), than that for Tame FJ. Both sets of proofs (given in appendices A and B) include the usual lemmas for weakening and substitution. In Tame FJ, we only have to deal with substitution of types for type variables and expressions for variables, whilst in Jo $\exists$  we must deal with substitution of types for type variables, contexts for context variables, and values for variables, and cannot substitute expressions for variables, since variables may appear (as contexts) in types and we wish to avoid dependent typing. Since subclassing is omitted from Jo $\exists$ , many lemmas dealing with the class hierarchy or method overriding are omitted or simplified. Tame FJ also requires lemmas relating the different kinds of subtyping, which are unnecessary in Jo $\exists$ . On the other hand, Jo $\exists$  has several lemmas dealing with the inside relation (for example, weakening, substitution, and well-formedness).

The most interesting differences are due to packing and unpacking. Since these operations are performed by expressions in Jo $\exists$ , reasoning about packing and unpacking is done in the proofs of lemmas about expression typing. In Tame FJ similar reasoning is done, but it is done in separate lemmas, such as those concerning the *match* relation and the relationship between subclassing and extended subclassing. Since the subsumption rule in Tame FJ also deals with packing, the inversion lemmas in Tame FJ are more complex than those of Jo $\exists$ .

Tame FJ supports F-bounds on type variables;  $Jo\exists$  has no equivalent of F-bounds for context variables (see section 4.4). This makes the treatment of environments in  $Jo\exists$  simpler than in Tame FJ. There is no need to use subclassing and the *uBound* function in F-ENV. It also means an environment can be separated without fear of variables going out of scope. That is,

in Jo $\exists$  if  $\Delta \vdash \Delta_1, \Delta_2$  ok then  $\Delta \vdash \Delta_1$  ok and  $\Delta, \Delta_1 \vdash \Delta_2$  ok. This is not true in Tame FJ where  $\Delta_1$  may contain references to variables defined in  $\Delta_2$ .

# 5.2 Related Work — Wildcards

In this section we compare Tame FJ with related work.

#### 5.2.1 Wild FJ

Wild FJ [60] (section 2.4.6) is a full model for Java wildcards, but lacks a soundness proof. Tame FJ is a refinement of Wild FJ, as the name suggests. The two systems are very similar: they are both extensions of FGJ [53], they share the same philosophy of modelling wildcards with implicitly handled existential types, and they have a similar form of subtyping. In particular, XS-ENV (see figure 3.2) is taken from Wild FJ with only minor adjustments. This is a significant design decision, in contrast to using close expressions and S-FULL (section 2.3.2) as in  $\exists J$  [24] (section 3.3.2) or two subtype rules as in Pizza [70] (section 2.2.2) and Java GI [94]. Wild FJ and Tame FJ are similar in assigning only existential types to expressions (Wild FJ never gives expressions wildcard types, available to the programmer in the surface syntax) and using existential types for subtyping. Existential types in both systems are quantified by an environment rather than single variables, and only non-existential types may be quantified (i.e., existential types cannot be further quantified).

The main differences between Tame FJ and Wild FJ are the way existential types occur, packing and unpacking, and type parameter inference.

Wildcards are assumed to have been translated to existential types in Tame FJ, whereas in Wild FJ they are translated within the calculus. This makes Tame FJ significantly simpler.

Existential packing in both systems takes place in ENV subtyping rules. In Wild FJ existential types are also packed *manually* in the conclusions of type rules. By "manual packing" we mean

that an environment and a type are explicitly combined in a type rule to give an existential type; for example,  $\Delta$  and N can be manually packed to  $\exists \Delta . N$ . Manual packing does not occur in Tame FJ; packing *only* occurs in XS-ENV. Manual packing is simpler than the corresponding rules in Tame FJ system; however, we avoid it because it can produce types of the form  $\exists X . X^4$  and requires extra steps to ensure that existential types are in the normal form described in section 3.3.

Unpacking in Wild FJ and Tame FJ is performed manually. By "manual unpacking" we mean that an existential type is explicitly split into an environment and an un-quantified type. For example,  $\exists \Delta . \mathbb{N}$  can be manually unpacked to  $\Delta$  and  $\mathbb{N}$ . Tracking the unpacked environment ( $\Delta$ ) is done implicitly in Wild FJ,  $\Delta$  is simply packed in the rule's conclusion. In Tame FJ,  $\Delta$ is tracked in the guarding environment and is re-packed (or safely forgotten by subsumption to an upper bound) by T-SUBS and XS-ENV.

Type parameter inference is performed by the *capture* relation in Wild FJ and the *match* relation in Tame FJ. These two relations fulfil the same purpose and produce the same results, but operate differently. Both relations infer missing type parameters to a method call. Both relations use the actual and formal type parameters and the types of actual and formal parameters of a method and both relations take subclassing into account. For example, given the method signature  $\langle X, Y \rangle$  void  $m(C \langle X \rangle x, C \langle Y \rangle y)$  and the method call  $\langle \star, \star \rangle m(a, b)$ , where a has type C $\langle$ Shape> and b has type  $\exists Z.C \langle Z \rangle$  both schemes infer Shape and Z as actual type parameters. In Tame FJ the *match* relation

$$match(\langle \{C < Shape >, C < Z > \}, \{C < X >, C < Y > \}\rangle, \{\star, \star\}, \{X, Y\}, \{Shape, Z\})$$

is checked. The interesting premises in the derivation are fulfilled by

#### $\vdash$ C<Shape> $\boxplus$ : [Shape/X,Z/Y]C<X>and $\vdash$ C<Z> $\boxplus$ : [Shape/X,Z/Y]C<Y>

<sup>&</sup>lt;sup>4</sup>We avoid these types because where the lower bound on X is  $\perp$ , they have equivalent behaviour to their upper bound because of implicit subsumption; where the lower bound is not  $\perp$ , they can neither be expressed nor denoted in Java.

, the substitution in these premises gives the inferred parameters **Shape** and **Z**. In Wild FJ, the *capture* function is called twice:

 $capture_{\Delta}(\star, X, \{C<X>, C<Y>\}, \{C<Shape>, C<Z>\}) and capture_{\Delta}(\star, Y, \{C<X>, C<Y>\}, \{C<Shape>, C<Z>\})$ 

. Looking in detail at the first instance, the *capture* function identifies one of C<X> and C<Y> that contains X and the position in which it occurs — the first parameter of C<X> (the first formal parameter type). It then picks the corresponding parameter from the corresponding actual parameter type — Shape.

More formally, and adjusting the notation of Wild FJ to match Tame FJ, we conjecture<sup>5</sup> that:

$$\forall \Delta : match(\langle \overline{\mathbf{R}}, \overline{\mathbf{P}} \rangle, \overline{\mathbf{U}}, \overline{\mathbf{X}}, \overline{\mathbf{T}}) \Leftrightarrow capture_{\Delta}(\overline{\mathbf{P}}, \overline{\mathbf{X}}, \overline{\mathbf{U}}, \overline{\mathbf{R}}) = \overline{\mathbf{T}}$$

Despite some effort, Wild FJ has not seen a soundness proof, however, we have not found any essentially unsound features. We found several reasons why Wild FJ was not amenable to a soundness proof, and these correspond with the major differences between Wild FJ and Tame FJ. It is difficult to ensure that *capture* produces types that are consistent under reduction and that the uses of *capture* in the type and reduction rules give corresponding results. We thus replaced *capture* with *match*. Having to deal with wildcard types and existential types in the formalism is difficult; furthermore, the use of *snap*, WS-ENV, and manual packing makes reasoning about packing in the type rules complex. Uniformly using existential types and restricting packing to XS-ENV addresses these problems in Tame FJ. In both Tame FJ and Wild FJ, there is no obvious relationship between subtyping and field and method types. This is in comparison with FGJ [53] where field and method types are invariant with respect to subtyping. In Tame FJ we use the subclassing relation when dealing with field and method type lookup; field and method types are invariant with respect to subclassing. Finally, Wild FJ is quite large and complicated: the auxiliary functions for translating wildcard to existential types and bound lookup require 16 derivation rules<sup>6</sup>; performing type translation within the

<sup>&</sup>lt;sup>5</sup>It is not very interesting to prove this (which would be non-trivial) because there is no formal link between Wild FJ and Java.

<sup>&</sup>lt;sup>6</sup>Tame FJ does not translate wildcard types to existential types and has only one rule (with two cases) for bounds lookup.

system, and supporting optional bounds on wildcards all make the system large and complex. This introduces more scope for complexity and thus errors in the proofs.

### 5.2.2 Variant Parametric Types

Variant parametric types [54] (section 2.2.2) can be thought of as a partial model for Java wildcards. The differences between variant parametric types and wildcards are described in section 2.4.5. The formalisation of variant parametric types is similar to Tame FJ in that it performs packing and unpacking implicitly. However, existential types are implicit in the variant parametric types system and the mechanisms for packing and unpacking are different to those in Tame FJ. There are many differences between the two calculi which reflect the differences between variant parametric types and wildcards; we do not describe these here, they are mostly straightforward and are discussed informally in section 2.4.5.

The close operation in variant parametric types,  $T \Downarrow_{\Delta} T'$  (see section 2.3.7), eliminates free type variables from T either by packing T or finding a supertype of T with no free variables. In Tame FJ, the same operation is done by T-SUBS and XS-ENV (figures 3.5 and 3.2). Guarding environments are used to facilitate this; the  $\Delta$  that parameterises the close operation on the result type of a rule in variant parametric types corresponds with the guarding environment on that rule in Tame FJ. The close operation in variant parametric types and subsumption in Tame FJ cannot create types of the form  $\exists \Delta . X$ , in contrast with manual packing in Wild FJ.

The open operation of variant parametric types corresponds with manual unpacking in Tame FJ. The only differences are due to the differences between variant parametric types and wildcards.

## 5.2.3 Pizza and $\mathcal{EX}_{upto}$

An interesting comparison can be made between the XS-ENV rule of Tame FJ (figure 3.2) and the PIZZA-S- $\exists$ - $\geq$  and PIZZA-S- $\exists$ - $\leq$  rules of Pizza [70] (section 2.3.7), these are adapted in Java GI [94], and  $\mathcal{EX}_{impl}$  and  $\mathcal{EX}_{upto}$  [95]. We give and adapt these rules and F-S-KERNEL (section 2.3.2) for easy comparison in figure 5.1.

$$\frac{\Delta, \Delta' \vdash \mathbf{T} <: \mathbf{T}' \quad dom(\Delta') \cap fv(\mathbf{T}') = \emptyset}{\Delta \vdash \exists \Delta'. \mathbf{T} <: \mathbf{T}'} \qquad \underbrace{\Delta \vdash \overline{[\mathbf{T}/\mathbf{X}]} \mathbf{B}_{l} <: \mathbf{T} \quad \Delta \vdash \overline{\mathbf{T}} <: [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{B}_{u}}{\Delta \vdash [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U} <: \exists \overline{\mathbf{X}} \rightarrow [\mathbf{B}_{l} \ \mathbf{B}_{u}] . \mathbf{U}} \\ (S-OPEN) \qquad (S-CLOSE)$$

$$\frac{dom(\Delta') \cap fv(\exists \overline{\mathbf{X}} \rightarrow [\mathbf{B}_{l} \ \mathbf{B}_{u}] . \mathbf{U}) = \emptyset}{\Delta, \Delta' \vdash \overline{[\mathbf{T}/\mathbf{X}]} \mathbf{B}_{l} <: \mathbf{T} \quad \Delta, \Delta' \vdash \overline{\mathbf{T}} <: [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{B}_{u}} \\ \Delta \vdash \exists \Delta'. [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U} <: \exists \overline{\mathbf{X}} \rightarrow [\mathbf{B}_{l} \ \mathbf{B}_{u}] . \mathbf{U} \qquad \underbrace{\Delta, \Delta' \vdash \mathbf{T} <: \mathbf{T}'}{\Delta \vdash \exists \Delta'. \mathbf{T} <: \exists \Delta'. \mathbf{T}'} \\ (S-ENV) \qquad (S-KERNEL)$$

Figure 5.1: Pizza and Tame FJ type rules for existential types.

S-OPEN unpacks existential types and allows subtyping between existential types with the same quantifying environment. S-CLOSE packs existential types. S-ENV performs both packing and unpacking. S-KERNEL gives subtyping between existential types with the same environment. We show that the two pairs of rules are equivalent when dealing with existential types, i.e., deriving  $\Delta \vdash \exists \Delta_1 . T_1 <: \exists \Delta_2 . T_2$ .

We can encode S-OPEN in terms of S-ENV and S-KERNEL; however, the right-hand side of the conclusion must be quantified by the empty environment:

$$\frac{\Delta, \Delta' \vdash \mathsf{T} <: \mathsf{T}'}{\Delta \vdash \exists \Delta'. \mathsf{T} <: \exists \Delta'. \mathsf{T}'} \qquad \frac{dom(\Delta') \cap fv(\exists \emptyset. \mathsf{T}') = \emptyset}{\Delta \vdash \exists \Delta'. [\emptyset] \mathsf{T}' <: \exists \emptyset. \mathsf{T}'} \\ \xrightarrow{(\mathsf{S}\text{-KERNEL})} \qquad \xrightarrow{(\mathsf{S}\text{-ENV})} \\
\Delta \vdash \exists \Delta'. \mathsf{T} <: \exists \emptyset. \mathsf{T}' \\ \xrightarrow{(\mathsf{S}\text{-TRANS})}$$

Likewise, we can encode S-CLOSE using S-ENV; in this derivation the first premise is trivially true, the second and third are the premises of S-CLOSE:

$$\frac{dom(\emptyset) \cap fv(\exists \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]} . \mathbf{U}) = \emptyset}{\Delta, \emptyset \vdash [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{B}_l <: \mathbf{T} \qquad \Delta, \emptyset \vdash \overline{\mathbf{T} <: [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{B}_u}}{\Delta \vdash \exists \emptyset. [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U} <: \exists \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]} . \mathbf{U}}_{(S-ENV)}$$

On the other hand, S-ENV and S-FULL can be encoded using S-OPEN and S-CLOSE:

The substituion in S-CLOSE in the second derivation is [X/X]; this can be made formal by using different alphabets for bound and free variables. The second premise of S-OPEN in the last derivation is tautological if  $\Delta'$  is well-formed.

Although we use S-KERNEL to discuss the relationship between the two approaches to subtyping, S-KERNEL is not present in Tame FJ or Wild FJ; instead, it is refactored into the other subtype rules. In Tame FJ this only affects XS-SUB-CLASS, where the conclusion is  $\Delta \vdash \exists \Delta . C < \overline{T} > <: \exists \Delta . [\overline{T/X}] N$ , rather than  $\Delta \vdash C < \overline{T} > <: [\overline{T/X}] N$  which would be used if S-KERNEL were present. If we included  $\exists \Delta . X$ , we would have to expand S-BOUND in a similar way, as is done in Wild FJ.

If we consider T different from  $\exists \emptyset.T$ , then S-OPEN and S-CLOSE would be slightly more flexible than S-ENV and S-KERNEL. We can close this small gap in expressivity by using the axioms  $\Delta \vdash T <: \exists \emptyset.T$  and  $\Delta \vdash \exists \emptyset.T <: T$ , or, as in Tame FJ, by defining  $\exists \emptyset.T$  as equivalent to T.

We use S-ENV in Tame FJ because the formulation of S-OPEN seems incompatible with our soundness proof, in particular, it does not seem compatible with lemma 17 (see section 3.1.2). In Tame FJ (which uses S-ENV), this lemma is used to convert a subtyping relationship to an extended subclassing relationship. It requires that  $\Delta$ , the environment that judges the subtype relationship, is well-formed. If we try to prove this lemma for Tame FJ with S-OPEN rather than with S-ENV, then there would be a problem in the S-OPEN case. Namely, to apply the inductive hypothesis to the premise of S-OPEN, we need to know that  $(\Delta, \Delta')$  is well-formed. This is only the case if  $\Delta'$  is well-formed, but this cannot be required in the premises of the lemma because it could not then be applied in the transitivity case.

# 5.3 Related Work — Ownership

In this section we compare  $Jo\exists$  with ownership types systems that support variance or existential types.

#### 5.3.1 Variance

Variant ownership types Variant ownership types [57] support use-site variance annotations [54] to give variant subtyping (section 2.5.3). The correspondence between variance annotations and existential types is described in section 2.3.7; this correspondence also applies to variant ownership types. Variant ownership types could, therefore, be encoded in a restriction of Jo $\exists$ .

There are some differences between type checking in variant ownership types and type checking in Jo $\exists$ . In variant ownership types, subtyping is defined using a variance relation that defines when contexts may be treated variantly. In the type rules a limited form of unpacking converts variant contexts into "abstract contexts" (denoted  $+_{?}a$ , where **a** is some context). A type parameterised by an abstract context only has subtypes by subclassing, not by variance.

An abstract context corresponds to an unpacked context with minimal scope, similar to a captured wildcard in Java. In  $Jo\exists_{wild}$ , a field with a type that involves an unpacked context cannot be assigned to, and a method with an unpacked context in any of its parameter types cannot be called because the unpacked context parameter cannot be named in any other type. Jo $\exists$  is more flexible because the scope of unpacking using open expressions can be larger, therefore unpacked context parameters can be named in more than one expressions (although they must be unpacked from a single source). This flexibility can be regained in  $Jo\exists_{wild}$  by using

owner-polymorphic methods and capture conversion. There is no support for these features in variant ownership types.

Types in Jo $\exists$  are more expressive than variant ownership types: Jo $\exists$  supports lower and upper bounds on contexts (as opposed to upper *or* lower bounds), type parameters, and explicit quantification (to express types such as  $\exists o.C < o, o >$ ); see section 4.1 for more details.

**MOJO** MOJO [23] (section 2.5.3) uses ? to denote an unknown context parameter. This corresponds to an existentially quantified context bounded by  $\perp$  and  $\bigcirc$  in Jo∃. In MOJO, ? may be used as an actual context parameter, there is no unpacking.

In the case of field access, substitution of ? (not found in other systems such as Wild FJ [60]) produces a similar behaviour to existential types in Jo∃. If an unpacked context variable appears in the type of a field in Jo∃ it must be re-packed. The effect is the same as substituting ?. For example (omitting bounds), given a class declaration class C<oo> { D<oo> f; } and an expression, e, with type C<?> (∃o.C<o> in Jo∃), e.f has type D<?> (∃o.D<o> in Jo∃). The derivation in MOJO is:

...  $\vdash \mathbf{e} : \mathsf{C} < ?> \qquad fType(\mathbf{f}, \mathsf{C} < ?>) = \mathsf{D} < ?>$ ...  $\vdash \mathbf{e} \cdot \mathbf{f} : \mathsf{D} < ?>$ (MOJO-T-FIELD)

In Jo $\exists$  we must perform explicit packing and unpacking and so the MOJO expressions e.f corresponds to the expression open e as x,o in close x.f with o hiding o in Jo $\exists$ . The typing derivation for this expression is:

$\Psi; \Delta, o; \Gamma, \mathtt{x}: \mathtt{C} \boldsymbol{<} o \boldsymbol{>} \vdash \mathtt{x}: \mathtt{C} \boldsymbol{<} o \boldsymbol{>} \qquad fType(\mathtt{f}, \mathtt{C} \boldsymbol{<} o \boldsymbol{>}) = \mathtt{C} \boldsymbol{<} o \boldsymbol{>}$		
$\Psi; \Delta, o; \Gamma, \mathtt{x:C} \vdash \mathtt{x.f}: \mathtt{C}$		
(T-FIELD)		
$\Psi; \Delta, o; \Gamma, \mathtt{x:C} \vdash \mathtt{close} \ \mathtt{x.f} \ \mathtt{with} \ \mathtt{o} \ \mathtt{hiding} \ \mathtt{o}: \exists \mathtt{o.C}$		
(T-CLOSE)		
$\Psi;\Delta;\Gamma\vdash$ open e as x,o in close x.f with o hiding o: $\exists$ o.C <o></o>		
(T-Open)		

In MOJO, *strict* method and field lookup are used to prevent field assignment and method call where ? would appear as a type parameter by substitution (but not where ? is written in the type). Likewise in Jo∃, it is impossible to type check a field assignment or method call where the receiver has existential type<sup>7</sup>. For example, consider  $\mathbf{x} \cdot \mathbf{f} = \mathbf{e}$  where the type of  $\mathbf{f}$  in  $\mathbf{x}$  is T,  $\mathbf{e}$  must have type T. If T involves an unpacked context variable (which will be the case if  $\mathbf{x}$  has existential type), then  $\mathbf{e}$  cannot have type T since the unpacked context is fresh<sup>8</sup>.

In a strict lookup, *strict substitution* is used to substitute actual for formal context parameters.  $[\overline{Q/p}]^{strict}T$  is defined as  $[\overline{Q/p}]T$  if  $? \in Q_i \Rightarrow p_i \notin T$  and undefined otherwise, where Q is a set of multiple actual context parameters and p is a formal context parameter. The strict versions of method and field lookup are used for field assignment and method call, but not field access. Variant types in MOJO are, therefore, treated in the same way as unbounded existential types in Jo∃.

Universes Universes [66, 67] (section 2.5.3) permit limited subtype variance with the any annotations. Universe types can be given corresponding types in Jo∃: any C corresponds to  $\exists o \rightarrow [\bot \bigcirc]$ . C<o>, peer C corresponds to C<o> (where o is the owner of the class declaration in which the type appears), and rep C corresponds to C<this>. The viewpoint adaptation<sup>9</sup> rules of universes correspond with the treatment of packing and unpacking in Jo∃. Generic universes [39] can be described using this correspondence and Jo∃'s type parameterisation. Using parametric ownership annotations and existential types to encode universe types gives a more transparent relationship between the types and their behaviour; for example, without requiring ad-hoc conversion tables for viewpoint adaptation. This could be beneficial in designing systems derived from or related to universes, or to reason about universe type systems.

The generic universes list is repeated from section 2.5.5 in figure 5.2. The assignments in m all require viewpoint adaptation: in assignment 1, the peer of a rep object is also a rep object.

<sup>&</sup>lt;sup>7</sup>To simplify the discussion we consider  $Jo\exists_{wild}$ .  $Jo\exists$  is more flexible and allows the scope of unpacking to include multiple subexpressions. See section 4.5.2 for examples of the difference between the two languages.

<sup>&</sup>lt;sup>8</sup>Even if e' is null this is true; the type of null must be well-formed (by T-NULL, figure 4.5), since only closed types are well-formed, a fresh context variable cannot occur in the type of null.

<sup>&</sup>lt;sup>9</sup>Viewpoint adaptation is the change in universe annotation when considering a type in a different context from the one in which it was declared.

The type parameter must also be adapted — the peer of a peer is a peer. In assignment 2, x.f is in the representation of a peer of this, this cannot be denoted in universes and so is adapted to any. If the type of the receiver is an any object, as in assignment 3, then the result type is also annotated with any. Finally in assignment 4, xF.datum has type peer Shape because type variables are not adapted by viewpoint. This type is a subtype of any Shape because of variant subtyping of any types.

```
class GUList<X> {
    peer GUList<X> next;
    X datum;
}
class C {
    rep GUList<peer Shape> f;
    void m(peer C x) {
        rep GUList<peer Shape> iNext = f.next;
                                                       //1
        any GUList<peer Shape> xF = x.f;
                                                       //2
        any GUList<peer Shape> xNext = xF.next;
                                                       //3
        any Shape xDatum = xF.datum;
                                                       //4
    }
}
```

Figure 5.2: Usage and definition of a list in the universes system.

Although these viewpoint adaptations are intuitively sensible, they are specified as a lookup table with no formal justification and so are somewhat arbitrary [39]. In the Jo $\exists$  version (given in figure 5.3), universe annotations are replaced by context parameters; viewpoint adaptation is accomplished by substitution of context parameters. Packing is used to introduce existential types (which represent the **any** annotation).

The context owner denotes the owner of this and is used to encode peer. The this context is used as the owner of rep references. In assignment 1, we substitute the actual context this for the formal context owner and Shape<owner> for X, the result is GUList<this, Shape<owner>> and corresponds to the universes version. In assignment 2, after substitution, x.f has the owner x, this is abstracted using a close expression to give the existential type of xF. In universes, the viewpoint is adapted to any because there is no annotation that corresponds to an object owned by x. Thus, viewpoint adaptation corresponds to substitution and abstraction.

Since xF has existential type it must be unpacked in Jo∃ before being accessed in assignments 3 and 4. In assignment 3, the owner of the right-hand side is oo, a context variable introduced

```
class GUList<owner, X> {
    GUList<owner, X> next;
    X datum;
}
class C<owner> {
    GUList<this, Shape<owner>> f;
    void m(C<owner> x) {
        GUList<this, Shape<owner>> iNext = f.next;
                                                                           //1
                                                                           //2
        ∃o.GUList<o, Shape<owner>> xF = close x.f with o hiding x;
        open xF as y,oo in {
            ∃o.GUList<o, Shape<owner>> xNext = close y.next with o hiding oo;
                                                                           //3
            \existso.Shape<o> xDatum = close y.datum with o hiding owner;
                                                                           //4
        }
    }
}
```

Figure 5.3: The universes list in  $Jo\exists$ .

by unpacking, a close expression must be used to abstract this context to ensure that no free variables escape the body of the open expression. Again, this abstraction corresponds to viewpoint adaptation. In assignment 4, packing corresponds with subtyping in universes. The hidden context (owner) can be represented using peer in universes.

The universes type system enforces the owners-as-modifiers discipline, rather than owners-asdominators. Extending Jo∃ to satisfy owners-as-modifiers would be interesting future work.

An alternative way to formalise the universe type system is using a lost modifier [36]. This modifier works in the same way as the unknown context in effective ownership (see below) or the abstract contexts of variant ownership (see above). A type annotated with lost corresponds to an unpacked existential type in Jo∃. In the example in figure 5.3, y could be given the type lost GUList<peer Shape>.

Effective Ownership The any context is also used to facilitate variance in effective ownership [57] (section 2.5.3). During field and method type lookup, all substitutions of any for x are replaced with substitutions of unknown for x. A type parameterised by unknown has no variant subtypes, it is similar to the abstract contexts of variant ownership types [57], see above. This system gives the same results as strict lookup in MOJO and, in the same way as MOJO, corresponds to the existential types of Jo∃. Similarly to variant ownership types and MOJO, it should be possible to encode the ownership structure of effective ownership in Jo $\exists$ . Effective owners (per-method owners) are currently beyond the scope of Jo $\exists$ . An effective owner cannot be **any**, and so there is no variance aspect to these owners.

### 5.3.2 Existential Types

System  $\mathbf{F}_{own}$  Existential quantification of ownership domains in System  $\mathbf{F}_{own}$  [56] (section 2.5.4) allows domains to be passed around without having to be able to name them. System  $\mathbf{F}_{own}$  does not have subtyping and so existential quantification does not lead to variance. System  $\mathbf{F}_{own}$  also supports existential quantification of types, absent in Jo $\exists$ .

System  $F_{own}$  is a core functional calculus and a formalisation of ownership domains, and has different design goals from Jo $\exists$ . Therefore, it is difficult to make a direct comparison between the two languages. Existential quantification in Jo $\exists$  can be used to pass around objects whose types include unnameable contexts. This is less useful in Jo $\exists$  because contexts cannot be created by expressions. Existential types in System  $F_{own}$  include information about the domain in which they were created. This is not required in Jo $\exists$ , but might be useful if we added support for ownership domains and their associated encapsulation properties.

Infinitary ownership types Infinitary ownership types [30] (section 2.5.4) use existential types to abstract contexts which cannot be named and therefore avoid dependent typing. Existential types in  $Jo\exists$  can be used in the same way. However, since contexts cannot be dynamically created in  $Jo\exists$ , abstraction is not necessary to avoid dependent typing.

Jo $\exists$  allows the use of quantified contexts as the owner of a type, whilst this is forbidden by Clarke [30]. Owners-as-dominators is satisfied because Jo $\exists$  supports lower bounds on quantified contexts. Lower bounds are used to ensure that existential quantification cannot be used to violate owners-as-dominators.
**Dynamic downcasting** Existential owners can be used in dynamic casts [98] (section 2.5.4). Casts are not supported in Jo $\exists$ , but they should be straightforward to add. Existential down-casting could then be encoded in Jo $\exists$  by casting using an existential type. To express the behaviour of dynamic downcasting in Jo $\exists$ , the expression being cast should be unpacked at the outermost scope of the method in which the cast appears. For example, the following method:

```
void m(Object<this> x) {
   List<this, d> 1 = (List<this, d>)x;
   Object<d> = l.get(0);
}
Would be encoded as:
```

```
void m(Object<this> x) {
    open (∃dd.List<this, dd>)x as xx,d in {
        List<this, d> l = xx;
        Object<d> = l.get(0);
    }
}
```

The encoding of existential downcasting in  $Jo\exists$  shows clearly the relationship to existential types and unpacking, and precisely defines the scope of introduced context variables.

## 5.4 Chapter Summary

In this chapter we have compared our treatment of Java wildcards with our treatment of variance in an ownership system. We have compared Tame FJ and  $Jo\exists$ — our formal models for Java wildcards and variant ownership; including a comparison of proofs of soundness for these systems. The two systems were more similar than different; the main differences stem from the separation of contexts from types in  $Jo\exists$ , this simplifies the underlying system and allows us to use explicit packing and unpacking to formalise it more simply. We have presented related work: alternative models for Java with wildcards and alternative approaches to variance and existential types in the ownership world. In particular, we have shown how  $Jo\exists$  can be used to examine existing systems that include these features. By using  $Jo\exists$  to encode dynamic

downcasting we make explicit where existential types are used and how they are unpacked; by encoding generic universes we show how the universe annotations and viewpoint adaption relate to ownership annotations, substitution, and packing and unpacking.

# Chapter 6

# Conclusions

Subtype variance increases the flexibility of parametric type systems, improves reuse, and reduces the use of casts and thus runtime type errors. Existential types are a good fit for modelling subtype variance. Existential types are well-understood and their typing properties closely match those of systems with use-site subtype variance.

We have investigated the relationship between existential types and subtype variance in the contexts of generics and ownership types. We have presented Tame FJ, a formal model for Java with wildcards, and Jo∃, an expressive model for subtype variance in ownership types languages.

To the best of our knowledge, Tame FJ is the first type sound model for Java that includes all the relevant features of wildcard types. We have shown through discussion and a formal translation, that Tame FJ is a satisfactory model for Java wildcards.

Jo $\exists$  is an ownership language with expressive, uniform, and general subtype variance. Jo $\exists$  is more expressive than previous work because of the use of existential quantification of contexts and their combination with type parameterisation. Jo $\exists$  uses existential types to support context variance in a uniform and transparent fashion. We have extended Jo $\exists$  to support owners-asdominators and proved both versions sound.

We have compared Tame FJ to Jo∃ to show the similarities and differences in using existential

types for variance in different contexts. Jo $\exists$  can be used to compare and encode ownership systems with existential types or different kinds of variance. Existing mechanisms for supporting context variance have the same behaviour as existential types in Jo $\exists$  and can be easily encoded (even if other language features cannot). Explicit existential types can give us a clearer picture of the underlying mechanisms used in type checking. Jo $\exists$  can also be used to encode existing kinds of existential types in ownership systems with similar benefits.

#### Critique

We have not *proved* that Tame FJ is an accurate model for Java with wildcards. Although the translation from Java to Tame FJ is straightforward, it is non-trivial and it is possible, although, we believe, unlikely, not to be type preserving. It would be preferable if Tame FJ were a strict subset of Java, like Wild FJ [60], so that a translation would be unnecessary.

Tame FJ is an abstraction of Java. It follows the standard approach for such models [53, 60], and we are satisfied that it contains all the features that are relevant to type checking. But of course, it does not support all the features of Java; it is possible that some missing feature could cause an unsoundness in the type system. Unfortunately, it is not currently practicable to prove properties for entire programming languages, so the accusation of missing features could be applied to any model. Finally, the soundness proof of Tame FJ is long and complex; it is possible, even probable, that there are errors in the proof. A machine checked proof would address this criticism and is planned for future work.

Tame FJ does not *feel* like an optimal model, it has a few rough edges. It could probably be made neater and more elegant. Packing using guarding environments and the treatment of well-formed type environments (which requires distinguishing between extended subclassing and subtyping) are areas that cry out for a better solution.

Jo∃ could be criticised for omitting subclassing, since it is considered a key part of objectoriented languages. However, subclassing is unrelated to ownership and variance, the focus of Tame FJ, and so we believe eliding it is a justified abstraction. On the other hand, the treatment of bounds on type variables could be improved. Allowing user-specified, type based bounds would be preferable to the current system of generating bounds on the owners of type variables. It would be preferable because it is not much more complex and more realistic. It would also address the slightly objectionable presence of  $\Psi$  (an environment recording the bounds on the owners of type variables) in the type system.

## 6.1 Further Work

Further work for this thesis follows the directions of its major strands. Tame FJ could be extended and improved, and Jo∃ could be further investigated and more widely applied. We also have an idea for the less conservative type checking of wildcard types.

#### 6.1.1 Tame FJ

Tame FJ could be extended in many ways to be a more complete model for Java. Possible extensions include imperative features, casts, interfaces (and the associated multiple inheritance), intersection types for bounds, and raw types.

It would be interesting to prove properties other than type soundness for Tame FJ and Java with wildcards. The most important is probably decidability (or undecidability, as it may turn out). Soundness and completeness for the translation from Java to Tame FJ, completeness of Tame FJ subtyping, and that an erasure translation of Tame FJ is sound would also be interesting. Mechanising the proof of soundness for Tame FJ is highly desirable.

The wildcards approach to existential types contains several features not present in the traditional approach, which is not expressive enough to model Java with wildcards. It would be interesting to formulate a minimal calculus, with a similar scope to System  $F_{<:}$ , that includes the key features of the wildcards approach, such as implicit packing and unpacking, quantification of multiple variables, F-bounds, and type parameter inference.

#### 6.1.2 Improving Wildcards in Java

There are some programs in Java that are safe, but which are rejected by the type checker. In terms of existential types, the scope of unpacking is too small. Java limits the scope of unpacking to the smallest enclosing sub-expression. If this were extended to a larger scope, for example, an entire method, then more programs could be accepted with no change to the syntax or other parts of the type system and without compromising soundness. For example,

```
class C<X> {
    C<X> f;
    void m(C<?> x) {
        x.f = x.f;
    }
}
```

In Java, m does not type check because the two x.f sub-expressions are unpacked separately. If the scope of unpacking were extended to the whole method, then this expression would type check.

This could be formalised using a dot notation. In the example,  $\mathbf{x}$  would still have type  $\exists \mathbf{X} . C < \mathbf{X} >$ , but it would be unpacked to  $C < \mathbf{x} . \mathbf{X} >$  rather than  $C < \mathbf{X} >$ , that is, the source of the type variable is remembered. By doing this, a fresh type variable is not needed each time  $\mathbf{x}$  is unpacked, in effect, it is only being unpacked once.

This approach could run into difficulties when alpha renaming of existential types is considered. Furthermore, it appears unstable in the presence of concurrency: in the example, if a different object is assigned to  $\mathbf{x}$  during the assignment, then this could result in a type error. We extend  $\mathbf{m}$  to show how:

We use | to denote two expressions executing in parallel. If y = x is executed after the righthand side of y.f = y.f, but before the left-hand side, then x.f (and thus d.f) could be set to an Object. This violates type safety because d is expected to have a Dog in f, not an Object. There are certainly solutions to this problem, such as using locks or requiring variables used before the dot to be constant, but there is much work to be done.

#### **6.1.3** Jo∃

Jo $\exists$  is restricted in several ways; we described how these restrictions could be lifted in section 4.4. We would like to extend and apply Jo $\exists$  in several other directions: support for the owners-as-modifiers property, which would facilitate modelling universe types; integration with OGJ, which could lead to a smaller and more practical language; and application to multiple ownership [23] and ownership domains [6].

We give an example of how existential quantification could be combined with multiple ownership in a possible extension to the MOJO language [23]. Class declarations in MOJO are similar to those in standard ownership languages and Jo∃. Given a class C that is declared to take a single context parameter, we can create types such as C<a> and C<a & b> in MOJO (where a and b are final variables). Using existential types we could create types such as  $\exists o$  intersects a.C<o> and  $\exists o$  intersects a & b.C<o><sup>1</sup>, which correspond to C<a & ?> and C<a & b & ?> respectively in MOJO. We could also write  $\exists o$  disjoint a.C<o>, to denote objects owned by some context that does not intersect with a. This type has no equivalent in MOJO. It would be interesting to consider if C< $\emptyset$ > in MOJO has a behavioural equivalent to some existential type (perhaps  $\exists o.C<o>$ ).

The requirement for variance is much more common in MOJO than in single ownership systems. This is because users of a class are often only concerned with one of several owners. For example, an object might be required to be owned by **this** along with some other objects. Extending this concept to collections requires that each element in the collection may have different owners as well as the specified one. In MOJO, this variance must be hard coded; for example, a (variant) list in MOJO would be declared as:

<sup>&</sup>lt;sup>1</sup>Which are probably equivalent to  $\exists o.C < o \& a > and \exists o.C < o \& a \& b >$ .

```
class MOJOList<o, d> {
    Object<d & ?> datum;
    MOJOList<o, d> next;
}
```

If this class is instantiated as MOJOList<this, a>, then items in the list could be owned by a and some other objects. Using List<this, a & ?> (see section 2.5) does not accomplish this because it requires that all the data in the list have the same owners.

As well as increased expressivity and cleaner type checking, using existential quantification and type parameterisation in MOJO would mean that classes can be written in *exactly* the same way as classes in single ownership systems. By using the GenericList class (see section 4.1.2) with existential types (for example, GenericList<this,  $\exists o \text{ intersects } a.Object<o>>$ ), the single ownership list can be used in MOJO without modification. This fits nicely with the philosophy of MOJO, that only the users of classes should be aware of the multiplicity of ownership.

It would be useful to do some large scale case studies of how variant ownership would actually be used. We suspect that use cases in the ownership world would differ significantly from those of parametric types. Such case studies would motivate further work on variance in ownership languages.

We would like to develop an expressive and user-friendly syntax to accompany Jo $\exists$ . Such a syntax would either involve some indicator of the scope of quantification<sup>2</sup>, or be unable to denote some types, as in Java wildcards. Case studies should help identify which types can be relegated to being expressible, but not denotable.

<sup>&</sup>lt;sup>2</sup>For example, in  $\exists X.C < C < X >>$  and  $C < \exists X.C < X >>$ , " $\exists X$ ." marks the scope of quantification and X makes the position of the quantified variable. In C < C <?>>, ? marks the position of the quantified variable, but the scope of quantification is fixed by convention.

# Appendix A

# Proofs of properties of Tame FJ

For all lemmas and theorems we require the additional premise that the program is well-formed, i.e., for all class declarations, Q, in the program,  $\vdash Q$  OK. Throughout, we assume the Barendregt convention, i.e., bound and free variables are distinct.

To use the premises of a judgement in a proof where we have the conclusion, an inversion lemma is required. However, where a judgement is syntax directed we reduce trivial overhead by using the inversion of the judgment directly in the proof.

Full proofs of all lemmas can be downloaded from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron\_ecoop08\_full.pdf

## A.1 Outline of proofs

The lemmas in the next section are sequenced so that they only use earlier lemmas; this ensures non-circularity. Most of the early lemmas prove "common-sense" properties about Tame FJ: lemmas 1 to 6 and 18 to 21 and 25 and 26 prove that the various relations and judgements are preserved under substitution of types and expressions, lemma 7 proves that guarding environments behave properly under alpha renaming, lemmas 8 to 11 prove weakening, lemmas 12 to 14 prove "obvious" properties about type environments, lemma 15 is a simple property of substitution, lemmas 27 to 30 prove that the results of various operations are well-formed, and lemmas 31 to 33 are standard inversion lemmas. These lemmas are used throughout the proofs and are mostly straightforward to prove.

Lemmas 16 and 22 to 24 prove that the subclassing relation preserves properties of types; these are all fairly simple lemmas, reflecting the simplicity of the subclassing relation. Lemmas 36 and 37 relate subclassing to the *match* relation and are more complicated. Lemma 17 relates subtyping to extended subclassing, lemma 34 relates subclassing to extended subclassing, and lemma 35 relates extended subclassing to subclassing. Lemmas 38 to 40 relate method bodies and types, and the lookup functions for fields; these are simple lemmas.

The most work is done in the proofs of the theorems themselves and of lemmas 2–4, 17, 21, 25, 29, and 35–37. At the highest level, in each case of the proof of subject-reduction, the standard

steps are followed at first (such as using the inversion lemmas, examining the premises of the type rules), then various 'big' lemmas are used to reduce the complexity of the case (that is, decompose complicated relations involving subtyping and *match* etc. into easier relations, familiar from FGJ [53] etc., and involving subclassing), the properties of subclassing are used to show some preservation of properties of types which are used to give the result. The hardest case is method invocation due to the *match* and *sift* relations and inheritance; field access is also complex due to inheritance.

## A.2 Proofs

Lemma 1. (Substitution preserves subclassing)

*If:*  **a.**  $\vdash \mathbb{R} \boxplus \mathbb{R}'$  *then:*  $\vdash [\overline{T/X}] \mathbb{R} \boxplus [\overline{T/X}] \mathbb{R}'$ 

Proof is by structural induction on the derivation of  $\vdash R \boxplus : R'$ 

#### Lemma 2. (Substitution preserves matching)

If:

**a.**  $match(\overline{\mathbb{R}}, \overline{\exists \Delta . \mathbb{R}'}, \overline{\mathbb{P}}, \overline{\mathbb{Y}}, \overline{\mathbb{U}})$  **b.**  $(\overline{\mathbb{X}} \cup fv(\overline{\mathbb{T}})) \cap \overline{\mathbb{Y}}) = \emptyset$ then:

 $match([\overline{T/X}]R, [\overline{T/X}] \exists \Delta . R', [\overline{T/X}]P, \overline{Y}, [\overline{T/X}]U)$ 



Lemma 3. (Substitution on  $\overline{U}$  preserves sift)

If:

**a.** 
$$sift(\overline{\mathbf{R}}, \overline{\mathbf{U}}, \overline{\mathbf{Y}}) = (\overline{\mathbf{R}_r}, \overline{\mathbf{T}_r})$$
  
**b.**  $(fv(\overline{\mathbf{T}}) \cup \overline{\mathbf{X}}) \cap \overline{\mathbf{Y}} = \emptyset$   
then:  
 $sift(\overline{\mathbf{R}}, \overline{[\overline{\mathbf{T}/\mathbf{X}}]\mathbf{U}}, \overline{\mathbf{Y}}) = (\overline{\mathbf{R}_r}, \overline{[\overline{\mathbf{T}/\mathbf{X}}]\mathbf{T}_r})$ 

Proof is by structural induction on the derivation of  $sift(\overline{R}, \overline{U}, \overline{Y}) = (\overline{R_r}, \overline{T_r})$  with a case analysis on the last step:

 $def\,sift$ 

subst

Case 1.  $\overline{U} = \emptyset$ 

trivial

Case 2.  $\overline{U} = \exists \Delta . N, \overline{U'}$ 

**Case 3.** 
$$\overline{U} = \exists \emptyset . Z, \overline{U'} \land Z \notin \overline{Y}$$

1.	$\overline{R} = R, \overline{R'}$	hay definit
2.	$(\overline{\mathtt{R}_r},\overline{\mathtt{T}_r})=(\mathtt{R},\overline{\mathtt{R}''},\ \exists \emptyset . \mathtt{Z},\overline{\mathtt{U}''})$	$\int \partial y  a e_j  s_{ij}  t$
3.	$sift(\overline{\mathbf{R}'}, \ \overline{\mathbf{U}'}, \ \overline{\mathbf{Y}}) = (\overline{\mathbf{R}''}, \ \overline{\mathbf{U}''})$	$by \ premise \ sift$
4.	$\overline{[\overline{T/X}]U} = [\overline{T/X}] \exists \emptyset.Z, \overline{[\overline{T/X}]U'}$	by def subst
5.	$sift(\overline{\mathbf{R}'}, \ \overline{[\overline{\mathbf{T}/\mathbf{X}}]\mathbf{U}'}, \ \overline{\mathbf{Y}}) = (\overline{\mathbf{R}''}, \ \overline{[\overline{\mathbf{T}/\mathbf{X}}]\mathbf{U}''})$	by $3, \mathbf{b}, ind hyp$

Case analysis on Z:

Case 1.  $Z \notin \overline{X}$ 

1.1.
$$[\overline{T/X}]U = \exists \emptyset.Z, [\overline{T/X}]U'$$
by 41.2. $sift(\overline{\mathbb{R}}, [\overline{T/X}]U, \overline{\mathbb{Y}}) =$ by 5, 1.1, 1, sift $(\mathbb{R}, \overline{\mathbb{R}''}, \exists \emptyset.Z, [\overline{T/X}]U'')$ by 5, 1.1, 1, sift1.3. $sift(\overline{\mathbb{R}}, [\overline{T/X}]U, \overline{\mathbb{Y}}) = (\overline{\mathbb{R}_r}, [\overline{T/X}]T_r)$ by 1.2, 2

Case 2.  $Z \in \overline{X}$ 

$\mathtt{Z}=\mathtt{X}_i$	
$[\overline{T/X}] \exists \emptyset. Z = T_i$	by <b>2.1</b> , $def$ subst
$\mathtt{T}_i = \exists \emptyset  .  \mathtt{Z}' \land \mathtt{Z}'  ot\in \overline{\mathtt{Y}} \lor \mathtt{T}_i = \exists \Delta  .  \mathtt{N}$	by <b>b</b>
$sift(\overline{\mathtt{R}},\overline{[\overline{\mathtt{T/X}}]\mathtt{U}},\overline{\mathtt{Y}}) =$	by <b>5</b> , <b>2.3</b> , <b>2.2</b> , <b>4</b> , <b>1</b> , sift
$(\mathbf{R}, \overline{\mathbf{R}''}, \mathbf{T}_i, \overline{[\overline{\mathbf{T}}/\mathbf{X}]}\mathbf{U''})$	
$sift(\overline{\mathtt{R}},\overline{[\overline{\mathtt{T/X}}]\mathtt{U}},\overline{\mathtt{Y}})=(\overline{\mathtt{R}_r},\overline{[\overline{\mathtt{T/X}}]\mathtt{T}_r})$	<i>by</i> <b>2.3</b> , <b>2.2</b> , <b>2</b>
	$\begin{split} & \mathbf{Z} = \mathbf{X}_i \\ & [\overline{\mathbf{T}/\mathbf{X}}] \exists \emptyset . \mathbf{Z} = \mathbf{T}_i \\ & \mathbf{T}_i = \exists \emptyset . \mathbf{Z}' \land \mathbf{Z}' \notin \overline{\mathbf{Y}} \lor \mathbf{T}_i = \exists \Delta . \mathbb{N} \\ & sift(\overline{\mathbf{R}}, [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U}, \overline{\mathbf{Y}}) = \\ & (\mathbf{R}, \overline{\mathbf{R}''}, \ \mathbf{T}_i, [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U}'') \\ & sift(\overline{\mathbf{R}}, [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{U}, \overline{\mathbf{Y}}) = (\overline{\mathbf{R}_r}, [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{T}_r) \end{split}$

Case 4.  $\overline{U} = \exists \emptyset . Z, \overline{U'} \land Z \in \overline{Y}$ 

1.	$\overline{\mathtt{R}} = \mathtt{R}, \overline{\mathtt{R}'}$	bu dof ai ft
2.	$(\overline{\mathtt{R}_r},\overline{\mathtt{T}_r})=(\overline{\mathtt{R}''},\ \overline{\mathtt{U}''})$	} by aef sift
3.	$\overline{[\overline{T/X}]U} = [\overline{T/X}] \exists \emptyset.Z, \overline{[\overline{T/X}]U'}$	by def subst
4.	$\overline{[\overline{T/X}]U} = \exists \emptyset.Z, \overline{[\overline{T/X}]U'}$	by 3, b
5.	$sift(\overline{\mathtt{R}}, \overline{[\overline{\mathtt{T/X}}]\mathtt{U}}, \overline{\mathtt{Y}}) = (\overline{\mathtt{R''}}, \overline{[\overline{\mathtt{T/X}}]\mathtt{U''}})$	by 4, 1, $sift$
6.	done	by 5, 2

Lemma 4. (Substitution on  $\overline{R}$  preserves *sift*)

If:

 $sift(\overline{R}, \overline{U}, \overline{Y}) = (\overline{R_r}, \overline{T_r})$ a. f is a mapping from and to types in the syntactic category R. b. then:

$$sift(\overline{f(\mathbf{R})}, \overline{\mathbf{U}}, \overline{\mathbf{Y}}) = (\overline{f(\mathbf{R}_r)}, \overline{\mathbf{T}_r})$$

Proof is by structural induction on the derivation of  $sift(\overline{R}, \overline{U}, \overline{Y}) = (\overline{R_r}, \overline{T_r})$  with a case analysis on the last step:

by def sift

Case 1.  $\overline{\mathtt{U}} = \emptyset$ 

trivial

Case 2.  $\overline{U} = \exists \Delta . N, \overline{U'}$ 

1.	$\overline{\mathtt{R}}=\mathtt{R},\overline{\mathtt{R}'}$	bu dof aift
2.	$(\overline{\mathtt{R}_r},\overline{\mathtt{T}_r})=(\mathtt{R},\overline{\mathtt{R}''},\ \exists\Delta.\mathtt{N},\overline{\mathtt{U}''})$	$\int by a e f sifi$
3.	$sift(\overline{\mathtt{R}'},\ \overline{\mathtt{U}'},\ \overline{\mathtt{Y}}) = (\overline{\mathtt{R}''},\ \overline{\mathtt{U}''})$	$by \ premise \ sift$
4.	$\overline{f(\mathtt{R})} = f(\mathtt{R}), \overline{f(\mathtt{R}')}$	by 1, c
5.	$sift(\overline{f(\mathbf{R}')}, \ \overline{\mathbf{U}'}, \ \overline{\mathbf{Y}}) = (\overline{f(\mathbf{R}'')}, \ \overline{\mathbf{U}''})$	by $3$ , ind hyp
6.	$sift(\overline{f(\mathtt{R})},\overline{\mathtt{U}},\overline{\mathtt{V}}) =$	by <b>5</b> , <b>4</b> ,  sift
	$(f(\mathbf{R}), \overline{f(\mathbf{R}'')}, \exists \Delta.\mathbf{N}, \overline{\mathbf{U}''})$	
7.	$sift(\overline{f(\mathtt{R})},\overline{\mathtt{U}},\overline{\mathtt{Y}}) = (\overline{f(\mathtt{R}_r)},\overline{\mathtt{T}_r})$	by <b>6</b> , <b>2</b>

Case 3.  $\overline{U} = \exists \emptyset . Z, \overline{U'} \land Z \notin \overline{Y}$ 

Case 4.  $\overline{U} = \exists \emptyset . Z, \overline{U'} \land Z \in \overline{Y}$ 

1.	$\overline{\mathtt{R}}=\mathtt{R},\overline{\mathtt{R}'}$	ha dof oift
2.	$(\overline{\mathtt{R}_r},\overline{\mathtt{T}_r})=(\overline{\mathtt{R}''},\ \overline{\mathtt{U}''})$	
3.	$\overline{f(\mathtt{R})} = f(\mathtt{R}), \overline{f(\mathtt{R}')}$	by 1, c
4.	$sift(\overline{f(\mathtt{R})}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}) = (\overline{f(\mathtt{R}'')}, \ \overline{\mathtt{U}''})$	by 3, $sift$
5.	done	by 4, 2

Lemma 5. (Substitution preserves field type)

If: **a.**  $fType(f, C<\overline{U}>) = U$ then:

 $fType(f, C < [\overline{T/X}]\overline{U} >) = [\overline{T/X}]U$ 

Proof is by induction on the derivation of  $fType(f, C < \overline{U} >) = U$ 

Lemma 6. (Substitution preserves method type)

If: **a.**  $mType(\mathbf{m}, \mathbb{C} < \overline{\mathbb{U}} >) = <\overline{\mathbb{X}' \lhd \mathbb{T}'} > \overline{\mathbb{U}'} \to \mathbb{U}$ then:  $mType(\mathbf{m}, \mathbb{C} < [\overline{\mathbb{T}/\mathbb{X}}] \overline{\mathbb{U}} >) = [\overline{\mathbb{T}/\mathbb{X}}] (<\overline{\mathbb{X}' \lhd \mathbb{T}'} > \overline{\mathbb{U}'} \to \mathbb{U})$ 

Proof is by induction on the derivation of  $mType(\mathbf{m}, \mathbb{C} < \overline{\mathbb{U}} >) = \langle \overline{\mathbb{X}' \triangleleft \mathbb{T}'} > \overline{\mathbb{U}'} \rightarrow \mathbb{U}$ 

Lemma 7. (Alpha renaming of guarding environments)

*If:*  **a.**  $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} \mid \overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]}$  **b.**  $\overline{\mathbf{Y}} \text{ are fresh}$  *then:*  $\Delta; \Gamma \vdash \mathbf{e} : [\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{T} \mid \overline{\mathbf{Y} \rightarrow [[\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}_l \ [\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}_u]}$ 

Proof is by structural induction on the derivation of  $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} | \overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]}$  with a case analysis on the last step:

Case 1. (T-VAR)

trivial since  $\overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]} = \emptyset$ Case 2. (T-NEW)

trivial since  $\overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]} = \emptyset$ Case 3. (T-FIELD)

1. 
$$\mathbf{e} = \mathbf{e}' \cdot \mathbf{f}$$
  
2.  $\Delta; \Gamma \vdash \mathbf{e}' : \exists \overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]} \cdot \mathbb{N} \mid \emptyset$   
3.  $fType(\mathbf{f}, \mathbb{N}) = \mathbf{T}$ 

by def T-FIELD by premises T-FIELD

 $\Delta; \Gamma \vdash \mathbf{e}' : \exists \mathbf{Y} \rightarrow [[\overline{\mathbf{Y}}/\mathbf{X}] \mathbf{B}_l \ [\overline{\mathbf{Y}}/\mathbf{X}] \mathbf{B}_u] . [\overline{\mathbf{Y}}/\mathbf{X}] \mathbb{N} \mid \emptyset \quad by \ \mathbf{2}, \mathbf{b}, alpha \ conversion$ 4. 5. fType(f, [Y/X]N) = [Y/X]Tby  $\mathbf{3}$ , lemma 5 6.  $\Delta; \Gamma \vdash \mathsf{e}'.\mathsf{f} : [\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{T} \mid \mathsf{Y} \rightarrow [[\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{B}_l \ [\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{B}_u]$ by 4, 5, T-FIELD 7. doneby 6, 1 Case 4. (T-INVK) 1.  $e = e'. \langle \overline{P} \rangle m(\overline{e})$ 2.  $\overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]} = \Delta', \overline{\Delta}$ by def T-INVK  $T = [\overline{T/Z}]U$ 3.  $\Delta; \Gamma \vdash \mathbf{e}' : \exists \Delta' . \mathbb{N} \mid \emptyset$ 4. 5.  $mType(\mathbf{m}, \mathbf{N}) = \langle \overline{\mathbf{Z} \triangleleft} \ \overline{\mathbf{B}} \rangle \overline{\mathbf{U}} \to \mathbf{U}$ 6.  $\Delta; \Gamma \vdash \mathbf{e} : \exists \Delta. \mathbf{R} \mid \emptyset$ 7.  $match(sift(\overline{R},\overline{U},\overline{Z}),\overline{P},\overline{Z},\overline{T})$ by premises T-INVK  $\Delta \vdash \overline{P}$  ок 8. 9.  $\Delta, \Delta', \overline{\Delta} \vdash \mathtt{T} <: [\mathtt{T}/\mathtt{Z}] \mathtt{B}$  $\Delta, \Delta', \overline{\Delta} \vdash \exists \emptyset. R <: [\overline{T/Z}] U$ 10.  $let \ \Delta' = \overline{\mathtt{X}' \to [\mathtt{B}'_l \ \mathtt{B}'_u]}$ 11.  $let \ \overline{\Delta} = \overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]}$ 12.  $\overline{X} = \overline{X'}, \overline{X}$ 13. by 11, 12, 2 14. let  $\Delta'' = \mathbf{Y}' \rightarrow [[\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}'_l \ [\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}'_u]$  $let \ \overline{\Delta'} = \mathbf{Y} \rightarrow [[\overline{\mathbf{Y}/\mathbf{X}}] \mathbf{B}_l \ [\overline{\mathbf{Y}/\mathbf{X}}] \mathbf{B}_u]$ 15. 16.  $\mathbf{Y} \rightarrow [[\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}_l \ [\overline{\mathbf{Y}/\mathbf{X}}]\mathbf{B}_u] = \Delta'', \overline{\Delta'}$ by 14, 15, 13 17.  $\Delta; \Gamma \vdash \mathbf{e}' : \exists \Delta'' \, . \, [\mathtt{Y}/\mathtt{X}] \, \mathtt{N} \, | \, \emptyset$ by **4**, alpha conversion 18.  $\Delta; \Gamma \vdash e : \exists \Delta' . [\overline{Y/X}] R \mid \emptyset$ by 6, alpha conversion 19.  $mType(\mathtt{m}, [\mathtt{Y}/\mathtt{X}]\mathtt{N}) = [\mathtt{Y}/\mathtt{X}] < \overline{\mathtt{Z} \triangleleft} \ \mathtt{B} > \overline{\mathtt{U}} \to \mathtt{U}$ by  $\mathbf{5}$ , lemma  $\mathbf{6}$  $match(sift([\overline{Y/X}]R, [\overline{Y/X}]U, \overline{Z}), [\overline{Y/X}]P, \overline{Z}, [\overline{Y/X}]T)$ 20. by 7, lemma 2  $[\overline{Y/X}]P = \overline{P}$ 21. by 822.  $match(sift([\overline{Y/X}]R, [\overline{Y/X}]U, \overline{Z}), \overline{P}, \overline{Z}, [\overline{Y/X}]T)$ by 20, 21  $\Delta, \Delta'', \overline{\Delta'} \vdash [\overline{Y/X}] T <: [\overline{Y/X}] [\overline{T/Z}] B$ 23. by 9, b, alpha conversion 24.  $\Delta, \Delta'', \overline{\Delta'} \vdash \exists \emptyset. [\overline{Y/X}] R <: [\overline{Y/X}] [\overline{T/Z}] U$ by 10, b, alpha conversion 25.  $\Delta, \Delta'', \overline{\Delta'} \vdash [\overline{Y/X}]T <: [[\overline{Y/X}]T/Z][\overline{Y/X}]B$ by 23, distinctness of formal type variables 26.  $\Delta, \Delta'', \overline{\Delta'} \vdash \exists \emptyset. [\overline{Y/X}] R <: [[\overline{Y/X}] T/Z] [\overline{Y/X}] U$ by 24, distinctness of formal type variables  $\Delta; \Gamma \vdash \mathsf{e}'. \langle \overline{\mathsf{P}} \rangle_{\mathsf{m}}(\overline{\mathsf{e}}) : [[\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{T}/\mathsf{Z}] [\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{U} | \mathsf{Y} \rightarrow [[\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{B}_l \ [\overline{\mathsf{Y}/\mathsf{X}}]\mathsf{B}_u]$ 27. by 17, 19, 18, 22, 8, 25, 26, T-INVK  $\Delta; \Gamma \vdash \mathbf{e}' \cdot \langle \overline{\mathbf{P}} \rangle \mathbf{m}(\overline{\mathbf{e}}) : [\overline{\mathbf{Y}/\mathbf{X}}] [\overline{\mathbf{T}/\mathbf{Z}}] \mathbf{U} | \mathbf{Y} \rightarrow [[\overline{\mathbf{Y}/\mathbf{X}}] \mathbf{B}_l \ [\overline{\mathbf{Y}/\mathbf{X}}] \mathbf{B}_{l'}]$ 28. by 27, distinctness of formal type variables 29. doneby 28, 1

Case 5. (T-SUBS)

trivial since  $\overline{X \to [B_l \ B_u]} = \emptyset$ Lemma 8. (Weakening of *uBound*)

If:  
**a.** 
$$uBound_{\Delta,\Delta'}(B) = B'$$
  
**b.**  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$   
then:  
 $uBound_{\Delta,\Delta'',\Delta'}(B) = B'$ 

Proof is by structural induction on the derivation of  $uBound_{\Delta,\Delta'}(B) = B'$ 

### Lemma 9. (Weakening of subtyping)

 $\begin{array}{ll} If: & & \\ \mathbf{a.} & dom(\Delta,\Delta') \cap dom(\Delta'') = \emptyset \\ and \ if: & \\ \mathbf{b.} & \Delta, \Delta' \vdash \mathbf{B} \sqsubset: \mathbf{B}' \\ then: & \\ \mathbf{c.} & \Delta, \Delta'', \Delta' \vdash \mathbf{B} \sqsubset: \mathbf{B}' \\ and \ if: & \\ \mathbf{d.} & \Delta, \Delta' \vdash \mathbf{B} <: \mathbf{B}' \\ then: & \\ & \Delta, \Delta'', \Delta' \vdash \mathbf{B} <: \mathbf{B}' \end{array}$ 

Proof is by structural induction on  $\Delta, \Delta' \vdash B \ll B'$  where  $\Delta \vdash B \ll B'$  is defined to hold if either  $\Delta \vdash B \sqsubset B'$  or  $\Delta \vdash B <: B'$  holds. There is a case analysis on the last step:

#### Lemma 10. (Weakening of well-formedness)

 $\begin{array}{ll} \textit{If:} & & \\ \textbf{a.} & dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset \\ \textbf{b.} & \Delta, \Delta' \vdash \psi \text{ OK} \\ \textit{and if:} & \\ \textbf{c.} & \psi = \Delta''' \ \textit{then } dom(\Delta, \Delta', \Delta'''') \cap dom(\Delta'') = \emptyset \\ \textit{where:} & \\ \textbf{d.} & \psi ::= \Delta''' \mid \mathsf{B} \mid \mathsf{R} \mid \star \\ \textit{then:} & \\ & \Delta, \Delta'', \Delta' \vdash \psi \text{ OK} \end{array}$ 

*Proof.* structural induction on the derivation of  $\Delta, \Delta' \vdash \psi$  OK

### Lemma 11. (Weakening of Typing)

 $\begin{array}{ll} \textit{If:} & & \\ \mathbf{a.} & dom(\Delta, \Delta', \Delta''') \cap dom(\Delta'') = \emptyset \\ & \\ \mathbf{b.} & dom(\Gamma, \Gamma'') \cap dom(\Gamma') = \emptyset \\ & \\ \mathbf{c.} & \Delta, \Delta'; \Gamma, \Gamma'' \vdash \mathbf{e} : \mathsf{T} \mid \Delta''' \\ \textit{then:} \\ & \Delta, \Delta'', \Delta'; \Gamma, \Gamma', \Gamma'' \vdash \mathbf{e} : \mathsf{T} \mid \Delta''' \end{array}$ 

Proof is by structural induction on the derivation of  $\Delta, \Delta'; \Gamma, \Gamma'' \vdash \mathbf{e} : \mathbf{T} \mid \Delta'''$ 

### Lemma 12. (Well-formed type environments are disjoint)

If: **a.**  $\Delta \vdash \Delta' \text{ ok}$ then:  $dom(\Delta) \cap dom(\Delta') = \emptyset$ 

Proof is by structural induction on the derivation of  $\Delta \vdash \Delta'$  or

Lemma 13. (Extension of type environments preserves well-formedness)

If: **a.**  $\Delta \vdash \Delta'$  OK **b.**  $\Delta, \Delta' \vdash \Delta''$  OK then:  $\Delta \vdash \Delta', \Delta''$  OK

Proof is by structural induction on the derivation of  $\Delta \vdash \Delta'$  ok

Lemma 14. (Concatenation of type environments preserves well-formedness)

If: **a.**  $\Delta \vdash \Delta' \text{ OK}$  **b.**  $\Delta \vdash \Delta'' \text{ OK}$  **c.**  $dom(\Delta') \cap dom(\Delta'') = \emptyset$ then:  $\Delta \vdash \Delta', \Delta'' \text{ OK}$ 

Proof is by induction on the size of  $\Delta'$ 

Lemma 15. (Limited commutativity of substitution)

If: a.  $[\overline{U/X}] [\overline{U'/X'}] T = T'$ b.  $\overline{X} \cap fv(\overline{U'}) = \emptyset$ c.  $\overline{X'} \cap fv(\overline{U}) = \emptyset$ d.  $\overline{X} \cap \overline{X'} = \emptyset$ then:  $[\overline{U'/X'}] [\overline{U/X}] T = T'$ 

Proof is by structural induction on the form of T:

Lemma 16. (Subclassing preserves class type)

If: **a.**  $\vdash R \square: N$ then: R = N' Proof is by structural induction on the derivation of  $\vdash \mathbf{R} \boxplus: \mathbf{N}$ 

#### Lemma 17. (*uBound* refines subtyping)

If: **a.**  $\vdash \Delta$  OK and if: **b.**  $\Delta \vdash T \sqsubset: T'$ or: **c.**  $\Delta \vdash T <: T'$ then:  $\Delta \vdash uBound_{\Delta}(T) \sqsubset: uBound_{\Delta}(T')$ 

Proof is by structural induction on  $\Delta \vdash T \ll T'$  where  $\Delta \vdash T \ll T'$  is defined to hold if either  $\Delta \vdash T \sqsubset T'$  or  $\Delta \vdash T \lt: T'$  holds. There is a case analysis on the last step:

Case 1. (XS-Reflex)

trivial

Case 2. (XS-SUB-CLASS, XS-ENV)

easy since  $T = \exists \Delta' . N$  and  $T' = \exists \Delta'' . N'$  and  $\forall \exists \Delta' . N : uBound_{\Delta}(\exists \Delta' . N) = \exists \Delta' . N$ Case 3. (XS-BOTTOM)

N/A

Case 4. (S-SC)

easy, by ind hyp.

Case 5. S-BOUND upper bound

1.  $T = \exists \emptyset . X$ 2.  $T' = B_u$ 3.  $\Delta(X) = X \rightarrow [B_l \ B_u]$ 4.  $uBound_{\Delta}(X) = uBound_{\Delta}(B_u)$ 5. done

Case 6. S-BOUND lower bound

1.  $T = B_l$ 2.  $T' = \exists \emptyset . X$ 3.  $\Delta(X) = X \rightarrow [B_l \ B_u]$ 4.  $uBound_{\Delta}(X) = uBound_{\Delta}(B_u)$ 5.  $\Delta \vdash uBound_{\Delta}(B_l) \sqsubset uBound_{\Delta}(B_u)$ 6. done

Case 7. (XS-TRANS)

1.  $\Delta \vdash \mathsf{T} \sqsubset : \mathsf{T}''$ 2.  $\Delta \vdash \mathsf{T}'' \sqsubset : \mathsf{T}'$  → by def S-Bound

by premise of S-BOUND by def uBound, **3** by **4**, XS-REFLEX

by def S-Bound by premise of S-BOUND

*by def uBound*, **3** *by* **3**, **a**, *def F-Env by* **5**, **4**, **2**, **1** SC-REFLEX

by premises of XS-TRANS/S-TRANS

3.  $\Delta \vdash uBound_{\Delta}(\mathsf{T}) \sqsubset uBound_{\Delta}(\mathsf{T}'')$ 

 $\Delta \vdash uBound_{\Delta}(\mathbf{T}'') \sqsubset : uBound_{\Delta}(\mathbf{T}')$ 4. 5.

 $\Delta \vdash uBound_{\Delta}(\mathsf{T}) \sqsubset : uBound_{\Delta}(\mathsf{T}')$ 

Case 8. (S-TRANS)

similar to case XS-TRANS

Corollary If  $\Delta \vdash \exists \Delta' . \mathbb{N} \lt: \exists \Delta'' . \mathbb{N}'$  and  $\vdash \Delta$  ok then  $\Delta \vdash \exists \Delta' . \mathbb{N} \sqsubset: \exists \Delta'' . \mathbb{N}'$ .

by 1, a, ind hyp

by **2**, **a**, ind hyp

by 3, 4, XS-TRANS

#### Lemma 18. (Substitution preserves subtyping)

If:  $\Delta_1 \vdash \mathsf{T} <: [\mathsf{T}/\mathsf{X}]\mathsf{B}_u$ a.  $\Delta_1 \vdash [\overline{T/X}] B_l <: T$ b.  $\Delta = \Delta_1, \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}, \Delta_2$ c.  $\Delta' = \Delta_1, [\overline{T/X}] \Delta_2$ d.  $\overline{\mathbf{X}} \cap fv(\Delta_1) = \emptyset$ e. f.  $fv(\overline{T}) \subseteq dom(\Delta')$ and if:  $\Delta \vdash B <: B'$ g. then:  $\Delta' \vdash [\overline{T/X}]B <: [\overline{T/X}]B'$ and if: h.  $\Delta \vdash \mathsf{B} \sqsubseteq: \mathsf{B}'$ then:  $\Delta' \vdash [\overline{T/X}] B \sqsubset : [\overline{T/X}] B'$ 

Proof is by structural induction on  $\Delta \vdash B \ll B'$  where  $\Delta \vdash B \ll B'$  is defined to hold if either  $\Delta \vdash B \sqsubset B' \text{ or } \Delta \vdash B \lt B' \text{ holds.}$  There is a case analysis on the last step:

#### Lemma 19. (Substitution preserves well-formedness)

If:

 $\Delta \vdash \psi$  ок a.  $\Delta_1 \vdash \mathsf{T} <: [\overline{\mathsf{T}/\mathsf{X}}] \mathsf{B}_u$ b.  $\Delta_1 \vdash [\overline{T/X}] B_l <: T$ c.  $\Delta = \Delta_1, \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}, \Delta_2$ d.  $\Delta' = \Delta_1, [T/X] \Delta_2$ e. f.  $\overline{\mathbf{X}} \cap fv(\Delta_1) = \emptyset$  $\Delta_1 \vdash \overline{\mathtt{T}} \text{ ok}$ g.  $\emptyset \vdash \Delta'$ ок h. where:  $\psi ::= \Delta_p \mid \mathbf{B} \mid \mathbf{R} \mid \star$ i. then:  $\Delta' \vdash [\overline{T/X}] \psi$  ok

Proof is by structural induction on the derivation of  $\Delta \vdash \psi$  OK

#### Lemma 20. (Corollary to lemma 19)

```
If:
                              \Delta \vdash \psi ок
            a.
                              \Delta_1 \vdash \mathsf{T} <: [\overline{\mathsf{T}/\mathsf{X}}] \mathsf{B}_u
            b.
                              \Delta_1 \vdash [\overline{T/X}] B_l <: T
            c.
                              \Delta = \Delta_1, \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}, \Delta_2
            d.
                              \Delta' = \Delta_1, [\overline{T/X}] \Delta_2
            e.
                              \overline{\mathbf{X}} \cap fv(\Delta_1) = \emptyset
            f.
                              \Delta_1 \vdash \overline{\mathtt{T}} \text{ ok}
            g.
                              \emptyset \vdash \Delta_1 ок
            h.
                              \Delta_1, \overline{X \rightarrow [B_l \ B_u]} \vdash \Delta_2 \text{ ok}
            i.
where:
                              \psi ::= \Delta_p \mid \mathsf{B} \mid \mathsf{R} \mid \star
            j.
then:
            \Delta' \vdash [\overline{T/X}] \psi ok
```

#### Lemma 21. (Substitution preserves typing)

If:

 $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} \mid \Delta''$ a.  $\Delta_1 \vdash \mathsf{T} <: [\overline{\mathsf{T/X}}] \mathsf{B}_u$ b.  $\Delta_1 \vdash [\overline{\mathtt{T/X}}] \mathtt{B}_l <: \mathtt{T}$ c.  $\Delta = \Delta_1, \overline{\mathbf{X} \to [\mathbf{B}_l \ \mathbf{B}_u]}, \Delta_2$ d.  $\Delta' = \Delta_1, [\overline{T/X}] \Delta_2$ e.  $\overline{\mathbf{X}} \cap fv(\Delta_1) = \emptyset$ f.  $\Delta_1 \vdash \overline{\mathtt{T}}$  ok g.  $\emptyset \vdash \Delta_1$  ок h.  $\Delta_1, \overline{X \rightarrow [B_l \ B_u]} \vdash \Delta_2 \text{ ok}$ i. then:  $\Delta'; [\overline{\mathsf{T/X}}]\Gamma \vdash [\overline{\mathsf{T/X}}]e : [\overline{\mathsf{T/X}}]T \mid [\overline{\mathsf{T/X}}]\Delta''$ 

Proof is by structural induction on the derivation of  $\Delta; \Gamma \vdash \mathbf{e} : \mathbf{T} \mid \Delta''$ 

#### Lemma 22. (Superclasses are well-formed)

*If:*  **a.**  $\vdash \mathbb{R} \boxplus \mathbb{R}'$  **b.**  $\Delta \vdash \mathbb{R} \text{ OK}$  **c.**  $\emptyset \vdash \Delta \text{ OK}$  *then:*  $\Delta \vdash \mathbb{R}' \text{ OK}$  Proof is by structural induction on the derivation of  $\vdash R \boxplus : R'$ 

### Lemma 23. (Subclassing preserves field types)

If: **a.**  $\vdash \mathbb{N} \boxplus : \mathbb{N}'$  **b.**  $fType(\mathbf{f}, \mathbb{N}') = \mathbb{T}$ then:  $fType(\mathbf{f}, \mathbb{N}) = \mathbb{T}$ 

Proof is by structural induction on the derivation of  $\vdash \mathbb{N} \boxplus: \mathbb{N}'$ 

Lemma 24. (Subclassing preserves method return type)

*If:*  **a.**  $\vdash \mathbb{N}_1 \boxplus \mathbb{N}_2$  **b.**  $mType(\mathbb{m}, \mathbb{N}_2) = \langle \overline{Y \triangleleft T_u} \rangle \overline{T} \rightarrow T$  *then:*  $mType(\mathbb{m}, \mathbb{N}_1) = \langle \overline{Y \triangleleft T_u} \rangle \overline{T} \rightarrow T$ 

Proof is by structural induction on the derivation of  $\vdash N_1 \boxplus : N_2$ 

### Lemma 25. (Expression substitution preserves typing)

If: **a.**  $\Delta; \Gamma, \mathbf{x} : \mathbf{U} \vdash \mathbf{e} : \mathbf{T} \mid \Delta'$  **b.**  $\Delta; \Gamma \vdash \mathbf{e}' : \mathbf{U}' \mid \emptyset$  **c.**  $\Delta \vdash \mathbf{U}' <: \mathbf{U}$  **d.**  $\Delta \vdash \mathbf{U} \text{ OK}$ then:  $\Delta; \Gamma \vdash [\mathbf{e}'/\mathbf{x}]\mathbf{e} : \mathbf{T} \mid \Delta'$ 

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma$ ,  $\mathbf{x}$ :  $\mathbf{U} \vdash \mathbf{e}$ :  $\mathbf{T} \mid \Delta'$ 

### Lemma 26. (Corollary to lemma 25)

```
Lemma 27. (fType gives well-formed types)
```

If:

a.

b.

c.

d.

 $\Delta, \overline{\Delta} \vdash \overline{T}$  ok

then:

If: **a.**  $fType(\mathbf{f}, \mathbb{C} < \overline{\mathbb{U}} >) = \mathbb{T}$  **b.**  $\emptyset \vdash \Delta \text{ OK}$  **c.**  $\Delta \vdash \exists \Delta' . \mathbb{C} < \overline{\mathbb{U}} > \text{ OK}$ then:  $\Delta, \Delta' \vdash \mathbb{T} \text{ OK}$ 

Proof is by induction on the derivation of  $fType(f, C<\overline{U}>) = T$ 

Lemma 28. (*mType* gives well-formed types)

If: **a.**  $mType(\mathbf{m}, \mathbb{C} < \overline{\mathbb{U}} >) = <\overline{\mathbb{Y} \lhd \mathbb{T}_u} > \overline{\mathbb{T}} \rightarrow \mathbb{T}$  **b.**  $\emptyset \vdash \Delta \text{ OK}$  **c.**  $\Delta \vdash \exists \Delta' . \mathbb{C} < \overline{\mathbb{U}} > \text{ OK}$ then:  $\Delta, \Delta', \overline{\mathbb{Y} \rightarrow [\bot \mathbb{T}_u]} \vdash \overline{\mathbb{T}} \text{ OK}$   $\Delta, \Delta', \overline{\mathbb{Y} \rightarrow [\bot \mathbb{T}_u]} \vdash \mathbb{T} \text{ OK}$  $\Delta, \Delta', \overline{\mathbb{Y} \rightarrow [\bot \mathbb{T}_u]} \vdash \overline{\mathbb{T}} \text{ OK}$ 

Proof is by induction on the derivation of  $mType(\mathbf{m}, \mathbb{C} < \overline{\mathbb{U}} >) = <\overline{\mathbb{Y}} < \overline{\mathbb{T}}_u > \overline{\mathbb{T}} \to \mathbb{T}$ 

#### Lemma 29. (match gives well-formed types)

 $match(\overline{R}, \overline{\exists \Delta' . R'}, \overline{P}, \overline{Y}, \overline{T})$ 

 $\Delta \vdash \overline{\mathtt{P}}$  ок

 $\emptyset \vdash \Delta$  ок

 $\Delta \vdash \overline{\exists \Delta . R}$  οκ

1. 
$$\forall i \text{ where } P_i \neq \star : T_i = P_i$$
  
2.  $\forall j \text{ where } P_j = \star : Y_j \in fv(\overline{R'})$   
3.  $\vdash \overline{R} \boxplus : [\overline{T/Y}, \overline{T'/X}]R'$   
4.  $dom(\overline{\Delta}) = \overline{X}$   
5.  $fv(\overline{T}, \overline{T'}) \cap \overline{Y}, \overline{X} = \emptyset$   
6.  $\Delta, \overline{\Delta} \vdash \overline{R} \text{ OK}$   
7.  $\Delta \vdash \overline{\Delta} \text{ OK}$   
8.  $\emptyset \vdash \Delta, \overline{\Delta} \text{ OK}$   
9.  $\Delta, \overline{\Delta} \vdash [\overline{T/Y}, \overline{T'/X}]R' \text{ OK}$ 

Case analysis on *each*  $T_i \in \overline{T}$ :

Case 1.  $P_i \neq \star$ 

 $by \mathbf{d}, def match$ 

by **b**, def F-EXIST

by 7, c, lemma 13 by 3, 6, 8, lemma 22 If:

1.1.	$\Delta \vdash P_i$ ok	$by \mathbf{a}$
1.2.	$\Delta, \overline{\Delta} \vdash P_i$ ok	by <b>1.1</b> , <b>8</b> , lemma 10
1.3.	$\Delta, \overline{\Delta} \vdash \mathtt{T}_i$ ok	<i>by</i> <b>1.2</b> , <b>1</b>
Case 2.	$P_i = \star$	
2.1.	$\mathbf{Y}_i \in fv(\overline{\mathbf{N}'})$	by $2$
2.2.	$let \ \overline{\mathbb{N}'} = \overline{\mathbb{C} \langle \overline{\mathbb{U}} \rangle}$	•
2.3.	$[\overline{\mathrm{T/Y}},\overline{\mathrm{T'/X}}]$ $\mathbb{N}'=\overline{\mathrm{C<}[\overline{\mathrm{T/Y}},\overline{\mathrm{T'/X}}]\overline{\mathrm{U}}}$	by <b>2.2</b> , $def$ subst
2.4.	$\exists \mathbf{N}'_i \text{ such that } [\overline{\mathbf{T/Y}}, \overline{\mathbf{T'/X}}] \mathbf{N}'_i = \mathbf{C}_j < \ldots, \mathbf{T}_i, \ldots >$	<i>by</i> <b>2.1</b> , <b>2.3</b>
2.5.	$\Delta, \overline{\Delta} \vdash \mathtt{T}_i \text{ ok}$	by 2.4, 9, def F-CLASS

#### Lemma 30. (Typing gives well-formed types)

 $\Delta; \Gamma \vdash \mathsf{e} : \mathsf{T} \mid \Delta'$ a.  $\emptyset \vdash \Delta$  ок b.  $\forall \mathbf{x} \in dom(\Gamma) : \Delta \vdash \Gamma(\mathbf{x}) \text{ ok}$ c. then:  $\Delta, \Delta' \vdash \mathbf{T}$  ok

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma \vdash \mathbf{e} : \mathbf{T} \mid \Delta'$ 

### Lemma 31. (Inversion Lemma (object creation))

```
If:
                            \Delta; \Gamma \vdash \text{new } C < \overline{T} > (\overline{e}) : T \mid \Delta'
           a.
then:
            \Delta' = \emptyset
           \Delta \vdash C < \overline{T} > OK
            fields(C) = \overline{f}
           fType(f, C<\overline{T}>) = U
           \Delta; \Gamma \vdash \mathbf{e} : \mathbf{U} \mid \emptyset
           \Delta \vdash \exists \emptyset. C < \overline{T} > <: T
```

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma \vdash \text{new } C < \overline{T} > (\overline{e}) : T \mid \Delta'$ 

Lemma 32. (Inversion Lemma (field access))

```
If:
                         \Delta; \Gamma \vdash \texttt{e.f} : \texttt{T} \mid \Delta'
          a.
                         \Delta \vdash \Delta'ок
          b.
then:
          there exists \Delta_n
where:
          \Delta \vdash \Delta', \Delta_n ok
          \Delta; \Gamma \vdash \mathbf{e} : \exists \Delta', \Delta_n . \mathbb{N} \mid \emptyset
          \Delta, \Delta', \Delta_n \vdash fType(f, N) <: T
```

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma \vdash \texttt{e.f}$ :  $\texttt{T} \mid \Delta'$ 

#### Lemma 33. (Inversion Lemma (method invocation))

If:  $\Delta; \Gamma \vdash e. < \overline{P} > m(\overline{e}) : T \mid \Delta'$ a.  $\emptyset \vdash \Delta \text{ ok}$ b.  $\Delta \vdash \Delta'$  ок c. d.  $\forall \mathbf{x} \in dom(\Gamma) : \Delta \vdash \Gamma(\mathbf{x}) \text{ ok}$ then: there exists  $\Delta_n$ where:  $\Delta', \Delta_n = \Delta'', \overline{\Delta}$  $\Delta \vdash \Delta', \Delta_n$  ок  $\Delta; \Gamma \vdash \mathbf{e} : \exists \Delta'' . \mathbb{N} \mid \emptyset$  $mType(\mathbf{m}, \mathbf{N}) = \langle \overline{\mathbf{Y} \triangleleft} \ \overline{\mathbf{B}} \rangle \overline{\mathbf{U}} \to \mathbf{U}$  $\Delta; \Gamma \vdash e : \exists \Delta.R \mid \emptyset$  $match(sift(\overline{R},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$  $\Delta \vdash \overline{P} \text{ ok}$  $\Delta, \Delta'', \overline{\Delta} \vdash \mathsf{T} <: [\overline{\mathsf{T/Y}}]\mathsf{B}$  $\Delta, \Delta'', \overline{\Delta} \vdash \exists \emptyset. R <: [\overline{T/Y}] U$  $\Delta, \Delta'', \Delta_n \vdash [\overline{\mathsf{T/Y}}] \mathsf{U} <: \mathsf{T}$ 

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma \vdash e. \langle \overline{P} \rangle m(\overline{e}) : T \mid \Delta'$ 

#### Lemma 34. (Subclassing gives extended subclassing)

If: **a.**  $\vdash \mathbf{R}' \sqsubseteq : \mathbf{R}$ then:  $\Delta \vdash \exists \Delta' . \mathbf{R}' \sqsubset : \exists \Delta' . \mathbf{R}$ 

Proof is by structural induction on the derivation of  $\vdash \mathsf{R}' \sqsubseteq: \mathsf{R}$ 

### Lemma 35. (Extended subclassing gives subclassing)

```
\begin{array}{l} If: \\ \mathbf{a.} \qquad \Delta \vdash \exists \Delta' \,. \, \mathbf{R}' \sqsubseteq : \exists \overline{\mathbf{X}} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u] \,. \mathbf{R} \\ \mathbf{b.} \qquad \Delta \vdash \ \mathbf{OK} \\ then: \\ there \ exists \ \overline{\mathbf{T}} \\ where: \\ \vdash \mathbf{R}' \boxplus : \ [\overline{\mathbf{T}/\mathbf{X}}] \,\mathbf{R} \\ \Delta, \Delta' \vdash \ \overline{\mathbf{T} <: \ [\overline{\mathbf{T}/\mathbf{X}}] \mathbf{B}_u} \\ \Delta, \Delta' \vdash \ [\overline{\mathbf{T}/\mathbf{X}}] \,\mathbf{B}_l <: \mathbf{T} \\ fv(\overline{\mathbf{T}}) \subseteq dom(\Delta, \Delta') \end{array}
```

Proof is by structural induction on the derivation of  $\Delta \vdash \exists \Delta' . \mathbb{R}' \sqsubset : \exists \overline{X \rightarrow [B_l \ B_u]} . \mathbb{R}$  with a case analysis on the last step:

Case 1. (XS-Reflex)

Easy, using SC-REFLEX,  $\overline{T} = \overline{X}$  and S-BOUND. Case 2. (XS-TRANS)

1.	$\Delta \vdash \exists \Delta'  .  R' \sqsubset: B$
2.	$\Delta \vdash B \sqsubset: \exists \overline{X \to [B_l \ B_u]}$ .R
3.	$\mathtt{B}=\exists \mathtt{X}'  ightarrow \mathtt{[B}'_l \ \mathtt{B}'_l \mathtt{]} . \mathtt{R}''$
4.	$wlog \ assume \ \overline{X'} \ are \ fresh$
5.	there exists $U'$
6.	$\vdash \mathbf{R}' \sqsubseteq : [\mathbf{U}'/\mathbf{X}']\mathbf{R}''$
7.	$\Delta, \Delta' \vdash \underline{\mathbf{U}' <: [\overline{\mathbf{U}'/\mathbf{X}'}]  \mathbf{B}_u}$
8.	$\Delta, \Delta' \vdash [\overline{\mathtt{U}'/\mathtt{X}'}] \mathtt{B}_l <: \mathtt{U}'$
9.	$fv(\overline{\mathtt{U}'})\subseteq dom(\Delta,\Delta')$
10.	there exists $\overline{U}$
11.	$\vdash \mathbf{R}'' \square : [\overline{\mathbf{U}/\mathbf{X}}]\mathbf{R}$
12.	$\Delta, \overline{\mathbf{X}' \to [\mathbf{B}'_l \ \mathbf{B}'_u]} \vdash \underline{\mathbf{U} <: [\overline{\mathbf{U}/\mathbf{X}}] \mathbf{B}'_u}$
13.	$\Delta, \overline{\mathtt{X}' \to [\mathtt{B}'_l \ \mathtt{B}'_u]} \vdash [\overline{\mathtt{U}/\mathtt{X}}]  \mathtt{B}'_l <: \mathtt{U}$
14.	$fv(\overline{\mathtt{U}})\subseteq dom(\Delta), \mathtt{X}'$
15.	$\vdash [\overline{U'/X'}]R'' \boxplus : [\overline{U'/X'}][\overline{U/X}]R$
16.	$\vdash \mathbf{R}' \boxplus : [\overline{\mathbf{U}'/\mathbf{X}'}] [\overline{\mathbf{U}/\mathbf{X}}] \mathbf{R}$
17.	$\vdash \mathbf{R}' \blacksquare : [\overline{[\mathbf{U}'/\mathbf{X}']}\mathbf{U}/\mathbf{X}]\mathbf{R}$
18.	$\Delta, \Delta', \overline{\mathtt{X}' \to [\mathtt{B}'_l \ \mathtt{B}'_u]} \vdash \overline{\mathtt{U} <: [\overline{\mathtt{U}/\mathtt{X}}] \mathtt{B}_u}$
19.	$\Delta, \Delta', \overline{\mathtt{X}' \to [\mathtt{B}'_l \ \mathtt{B}'_u]} \vdash \overline{[\mathtt{U}/\mathtt{X}]} \mathtt{B}_l <: \mathtt{U}$
20.	$\Delta, \Delta' \vdash \overline{[\overline{\mathtt{U}'/\mathtt{X}'}]}\mathtt{U} <:  [\overline{\mathtt{U}'/\mathtt{X}'}][\overline{\mathtt{U}/\mathtt{X}}]\mathtt{B}_u$
21.	$\Delta, \Delta' \vdash \overline{[\overline{\mathbf{U}'/\mathbf{X}'}]\mathbf{U} <: [\overline{[\overline{\mathbf{U}'/\mathbf{X}'}]\mathbf{U}/\mathbf{X}}]\mathbf{B}_u}$
22.	$\Delta, \Delta' \vdash \overline{[\overline{\mathbf{U}'/\mathbf{X}'}]  [\overline{\mathbf{U}/\mathbf{X}}]  \mathbf{B}_l <: [\overline{\mathbf{U}'/\mathbf{X}'}]  \mathbf{U}}$
23.	$\Delta, \Delta' \vdash \overline{[\overline{[\overline{U'/X'}]U/X}]B_l <: [\overline{U'/X'}]U}$
24.	$fv(\overline{[\overline{U'/X'}]U}) \subseteq dom(\Delta, \Delta')$
25.	$let \ \overline{\mathtt{T}} = \overline{[\overline{\mathtt{U}'/\mathtt{X}'}]}  \mathtt{U}$
26.	done

Case 3. (XS-ENV)

1.	R = N	har def VC ENT
2.	$R' = [\overline{U/X}]N$	$\begin{cases} oy \ aej \ AS-ENV \end{cases}$
3.	$\Delta, \Delta' \vdash \overline{\mathtt{U} <: [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{B}_u}$	)
4.	$\Delta, \Delta' \vdash \overline{\overline{[\overline{\mathtt{U}/\mathtt{X}}]}} \mathtt{B}_l <: \mathtt{U}$	hu premises XS
5.	$dom(\Delta') \cap fv(\exists \overline{X \to [B_l \ B_u]}.N) = \emptyset$	( by premises Ac
6.	$fv(\overline{\mathtt{U}}) \subseteq dom(\Delta, \Delta')$	J
7.	$\vdash$ N $\boxplus$ : N	by SC-Reflex
8.	$\vdash [\overline{U/X}]$ N $\boxplus$ : $[\overline{U/X}]$ N	by 7, lemma 1
9.	$\vdash$ N' $\square$ : $[\overline{U/X}]$ N	by <b>8</b> , <b>2</b>
10.	$let \ \overline{\mathtt{T}} = \overline{\mathtt{U}}$	
11.	done	<i>by</i> <b>10</b> , <b>9</b> , <b>3</b> , <b>4</b> , <b>6</b>

} by premises XS-TRANS
by 1 gives B ≠⊥
by 3, Barendregt
}
by 1, 3, b, ind hyp
}
by 1, 3, b, ind hyp
}
by 2, 3, b, ind hyp
by 11, lemma 1
by 6, 15, SC-TRANS
by 16, 4
by 12, 4, lemma 9
by 13, 4, lemma 9
by 18, 7, 8, b, lemma 18
by 20, 4
by 19, 7, 8, b, lemma 18
by 22, 4
by 9, 14

 $by\ \mathbf{25},\ \mathbf{17},\ \mathbf{21},\ \mathbf{23},\ \mathbf{24}$ 

XS-Env

Case 4. (XS-SUB-CLASS)

1. 2. 3.	$\Delta' = \overline{\mathbf{X} \rightarrow [\mathbf{B}_l \ \mathbf{B}_u]}$ $\mathbf{R}' = \mathbf{C} < \overline{\mathbf{U}} >$ $\mathbf{R} = [\overline{\mathbf{U}/\mathbf{Y}}] \mathbf{N}''$	by def XS-Sub-Class
4.	class $C < \overline{Y} > \lhd N''$	by premise XS-Sub-Class
5.	$\vdash C < \overline{U} > \Box : [\overline{U/Y}] N''$	by 4, SC-SUB-CLASS
6.	$\vdash R' \boxplus : R$	by <b>5</b> , <b>2</b> , <b>3</b>
7.	$let \ \overline{\mathtt{T}} = \overline{\mathtt{X}}$	
8.	done	<i>by</i> <b>6</b> , <b>7</b> , S-BOUND, <b>1</b>

Case 5. (XS-BOTTOM)

#### N/A

#### Lemma 36. (Subclassing preserves matching (receiver))

#### If:

 $\begin{array}{ll} \mathbf{a.} & \Delta \vdash \exists \Delta_1 \,.\, \mathbb{N}_1 \sqsubseteq : \exists \Delta_2 \,.\, \mathbb{N}_2 \\ \mathbf{b.} & mType(m,\mathbb{N}_2) = < \overline{\mathbb{Y}_2 \to [\mathbb{B}_{2l} \ \mathbb{B}_{2u}]} > \overline{\mathbb{U}_2} \to \mathbb{U}_2 \\ \mathbf{c.} & mType(m,\mathbb{N}_1) = < \overline{\mathbb{Y}_1 \to [\mathbb{B}_{1l} \ \mathbb{B}_{1u}]} > \overline{\mathbb{U}_1} \to \mathbb{U}_1 \\ \mathbf{d.} & match(sift(\overline{\mathbb{R}},\overline{\mathbb{U}_2},\overline{\mathbb{Y}_2}),\overline{\mathbb{P}},\overline{\mathbb{Y}_2},\overline{\mathbb{T}}) \\ \mathbf{e.} & \emptyset \vdash \Delta \text{ OK} \\ \mathbf{f.} & \Delta, \Delta' \vdash \overline{\mathbb{T}} \text{ OK} \end{array}$ 

then:

 $match(sift(\overline{\mathtt{R}},\overline{\mathtt{U}_{1}},\overline{\mathtt{Y}_{1}}),\overline{\mathtt{P}},\overline{\mathtt{Y}_{1}},\overline{\mathtt{T}})$ 

17.	$\vdash [T'/X]R'' \boxplus : [T'/X][T/Y_2,T''/Z]R'$	by $14$ , lemm
18.	$\overline{\mathtt{X}} \cap fv(\overline{\mathtt{R}''}) = \emptyset$	by Barendre
19.	$\vdash \overline{R'' \sqsubseteq} : [\overline{T'/X}] [\overline{T/Y_2}, \overline{T''/Z}] R'$	<i>by</i> <b>17</b> , <b>18</b>
20.	$\overline{Z} \cap fv(\overline{T'}) = \emptyset$	by <b>15</b> , <b>11</b> , B
21.	$\vdash \mathbf{R}'' \sqsubseteq : [\overline{[\mathbf{T}'/\mathbf{X}]} \mathbf{T}/\mathbf{Y}_2], \overline{[\mathbf{T}'/\mathbf{X}]} \mathbf{T}''/\mathbf{Z}] [\overline{\mathbf{T}'/\mathbf{X}}] \mathbf{R}'$	by <b>19</b> , <b>6</b> , <b>20</b>
22.	$\forall j \ where \ \mathtt{P}_j = \star : \mathtt{Y}_{2j} \in fv([\overline{\mathtt{T}'/\mathtt{X}}]\mathtt{R}')$	<i>by</i> <b>13</b> , <b>5</b>
23.	$fv(\overline{[\mathbf{T}'/\mathbf{X}]}\mathbf{T},\overline{[\mathbf{T}'/\mathbf{X}]}\mathbf{T}'')\cap\overline{\mathbf{Y}_2},\overline{\mathbf{Z}}=\emptyset$	<i>by</i> <b>16,6,20</b>
24.	$match(\overline{\mathbf{R}''}, [\overline{\mathbf{T}'/\mathbf{X}}] \exists \Delta . \mathbf{R}', \overline{\mathbf{P}}, \overline{\mathbf{Y}_2}, [\overline{\mathbf{T}'/\mathbf{X}}] \mathbf{T})$	<i>by</i> <b>12</b> , <b>22</b> , <b>2</b>
25.	$sift(\overline{\mathtt{R}}, [\overline{\mathtt{T}'/\mathtt{X}}]\mathtt{U}_2, \overline{\mathtt{Y}_2}) = (\overline{\mathtt{R}''}, [\overline{\mathtt{T}'/\mathtt{X}}] \exists \Delta.\mathtt{R}')$	<i>by</i> <b>11</b> , <b>5</b> , <b>6</b> ,
26.	$sift(\overline{\mathtt{R}},\overline{\mathtt{U}_1},\overline{\mathtt{Y}_1})=(\overline{\mathtt{R}''},\overline{[\mathtt{T}'/\mathtt{X}]}\exists\Delta.\mathtt{R}')$	<i>by</i> <b>25</b> , <b>9</b> , <b>10</b>
27.	$match(sift(\overline{R}, \overline{U_1}, \overline{Y_1}), \overline{P}, \overline{Y_1}, \overline{[T'/X]T})$	by <b>24</b> , <b>26</b> , <b>9</b>
28.	$match(sift(\overline{\mathtt{R}},\overline{\mathtt{U}_{1}},\overline{\mathtt{Y}_{1}}),\overline{\mathtt{P}},\overline{\mathtt{Y}_{1}},\overline{\mathtt{T}})$	by <b>27</b> , <b>f</b> , <b>2</b> , <i>l</i>

by 14, lemma 1 by Barendregt by 17, 18 by 15, 11, Barendregt by 19, 6, 20 by 13, 5 by 16,6,20 by 12, 22, 21, 15, 23, def match by 11, 5, 6, lemma 3 by 25, 9, 10 by 24, 26, 9 by 27, f, 2, Barendregt

### Lemma 37. (Subclassing preserves matching (arguments))

If:

 $\Delta \vdash \overline{\exists \Delta_1 \, . \, \mathsf{R}_1 \sqsubset : \exists \Delta_2 \, . \, \mathsf{R}_2}$ a.  $match(sift(\overline{R_2},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$ b.  $fv(\overline{\mathtt{U}})\cap\overline{\mathtt{Z}}=\emptyset$ c.  $\overline{\Delta_2} = \overline{\mathsf{Z} \to [\mathsf{B}_l \ \mathsf{B}_u]}$ d.  $\emptyset \vdash \Delta$  ок e.  $\Delta \vdash \overline{\exists \Delta_1 \, . \, \mathbf{R}_1} \, \operatorname{ok}$ f.  $\Delta \vdash \overline{P}$  ок g. then: there exists  $\overline{\mathtt{U}'}$ 

where:

$$\begin{split} & match(sift(\overline{\mathbf{R}_{1}},\overline{\mathbf{U}},\overline{\mathbf{Y}}),\overline{\mathbf{P}},\overline{\mathbf{Y}},\overline{[\overline{\mathbf{U'/Z}}]\,\mathbf{T}}) \\ & \Delta,\overline{\Delta_{1}} \vdash \overline{\mathbf{U'} <: [\overline{\mathbf{U'/Z}}]\,\mathbf{B}_{u}} \\ & \Delta,\overline{\Delta_{1}} \vdash \overline{[\overline{\mathbf{U'/Z}}]\,\mathbf{B}_{l} <: \mathbf{U'}} \\ & \vdash \overline{\mathbf{R}_{1}} \boxplus: \overline{[\overline{\mathbf{U'/Z}}]\,\mathbf{R}_{2}} \\ & fv(\overline{\mathbf{U'}}) \subseteq \Delta,\overline{\Delta_{1}} \end{split}$$

13.  $\forall i \ where \ \mathsf{P}_i \neq \star : \mathsf{T}_i = \mathsf{P}_i$ **14**.  $\forall j \ where \ \mathsf{P}_j = \star : \mathsf{Y}_j \in fv(\overline{\mathsf{R}_3})$  $\vdash \mathbb{R}'_2 \boxplus : [\overline{\mathbb{T}/\mathbb{Y}}, \overline{\mathbb{T}'/\mathbb{X}}]\mathbb{R}_3$ 15. 16.  $dom(\overline{\Delta_3}) = \overline{\mathbf{X}}$  $fv(\overline{\mathtt{T}},\overline{\mathtt{T}'})\cap\overline{\mathtt{Y}},\overline{\mathtt{X}}=\emptyset$ 17.  $\vdash [\overline{\mathbf{U}'/\mathbf{Z}}] \mathbf{R}_2' \boxplus : [\overline{\mathbf{U}'/\mathbf{Z}}] [\overline{\mathbf{T}/\mathbf{Y}}, \overline{\mathbf{T}'/\mathbf{X}}] \mathbf{R}_3$ 18. 19.  $\vdash R'_1 \boxplus : [\overline{U'/Z}] [\overline{T/Y}, \overline{T'/X}] R_3$ 20.  $\vdash$  R'<sub>1</sub>  $\boxplus$ : [[U'/Z]T/Y, [U'/Z]T'/X]R<sub>3</sub> 21.  $\Delta, \overline{\Delta_1} \vdash \overline{\mathtt{R}}_1$  ок  $\Delta \vdash \overline{\Delta_1}$  ок 22.  $\emptyset \vdash \Delta, \overline{\Delta_1}$  ок 23.  $\Delta, \overline{\Delta_1} \vdash [\overline{\mathbf{U}'/\mathbf{Z}}] \mathbf{R}_2 \text{ ok}$ **24**.  $fv(\overline{\mathtt{U}'}) \cap \overline{\mathtt{X}} = \emptyset$ 25. 26.  $fv(\overline{\mathtt{U}'}) \cap \overline{\mathtt{Y}} = \emptyset$  $fv([\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{T}, [\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{T}') \cap \overline{\mathbf{Y}}, \overline{\mathbf{X}} = \emptyset$ 27. **28**.  $\forall i \ where \ \mathsf{P}_i \neq \star : [\mathsf{U}'/\mathsf{Z}] \ \mathsf{T}_i = [\mathsf{U}'/\mathsf{Z}] \ \mathsf{P}_i = \mathsf{P}_i$  $match(\overline{\mathbf{R}'_{1}}, \overline{\exists \Delta_{3} \cdot \mathbf{R}_{3}}, \overline{\mathbf{P}}, \overline{\mathbf{Y}}, [\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{T})$ 29.  $match(sift(\overline{R_1}, \overline{U}, \overline{Y}), \overline{P}, \overline{Y}, [\overline{U'/Z}]T)$ 30. 31. done

by b, 1, def match
by 15, lemma 1
by 7, 18, SC-TRANS
by 12, 19
by f, def F-EXIST
by 22, e, lemma 13
by 7, f, e, lemma 22
by 16, Barendregt
by 10
by 17, 25, 26
by 14, g, d, Barendregt
by 28, 14, 20, 16, 27, def match
by 29, 4
by 30, 8, 9, 7, 10

Lemma 38. (Method body is well typed)

If: **a.**  $\emptyset \vdash \Delta \text{ OK}$  **b.**  $\Delta \vdash C < \overline{T} > \text{ OK}$  **c.**  $mType(\mathbf{m}, C < \overline{T} >) = <\overline{Y} \lhd \overline{U_u} > \overline{U} \rightarrow U$  **d.**  $mBody(\mathbf{m}, C < \overline{T} >) = (\overline{\mathbf{x}}; \mathbf{e})$ then:  $\Delta, \overline{Y \rightarrow [\perp U_u]}; \overline{\mathbf{x}:U}, \text{this}: \exists \emptyset. C < \overline{T} > \vdash \mathbf{e}: U \mid \emptyset$ 

Proof is by induction on the derivation of  $mBody(m, C<\overline{T}>) = (\overline{x}; e)$ 

Lemma 39. (*mType* defined gives *mBody* defined)

If: **a.**  $mType(\mathbf{m}, \mathbb{C} < \overline{\mathbb{T}} >)$  defined then:  $mBody(\mathbf{m}, \mathbb{C} < \overline{\mathbb{T}} >)$  defined

Proof is by case analoys on the definition of  $mType(m, C<\overline{T}>)$ 

Lemma 40. (*fType* and *fields* related)

a.  $fType(f, C<\overline{T}>)$  defined b.  $fields(C) = \overline{f}$ then:  $f \in \overline{f}$  Proof is by induction on the derivation of fType(f, N)

#### Theorem (Subject Reduction)

*If:*  **a.**  $\emptyset; \emptyset \vdash \mathbf{e} : \mathbf{T} \mid \emptyset$  **b.**  $\mathbf{e} \rightsquigarrow \mathbf{e}'$  *then:*  $\emptyset; \emptyset \vdash \mathbf{e}' : \mathbf{T} \mid \emptyset$ 

Proof is by structural induction on the derivation of  $\mathbf{e} \rightsquigarrow \mathbf{e}'$  with a case analysis on the last step:

Case 1. (R-FIELD)

1.  $e = new C < \overline{T} > (\overline{v}) . f_i$ by def R-FIELD 2.  $e' = v_i$ 3.  $fields(C) = \overline{f}$ by premise of R-FIELD  $\emptyset \vdash \Delta'$ ок 4. 5.  $\emptyset; \emptyset \vdash \text{new } C < \overline{T} > (\overline{v}) : \exists \Delta' . N \mid \emptyset$ by 1, a, F-ENV-EMPTY, lemma 32 6. fType(f, N) = T' $\Delta' \vdash \mathsf{T}' <: \mathsf{T}$ 7. 8.  $\emptyset \vdash C < \overline{T} > OK$ 9.  $fType(f, C<\overline{T}>) = U$ by 5, lemma 31  $\emptyset; \emptyset \vdash \overline{v : U \mid \emptyset}$ 10.  $\emptyset \vdash \exists \emptyset. C < \overline{T} > <: \exists \Delta'. N$ 11. 12.  $\emptyset \vdash \exists \emptyset. C < \overline{T} > \Box : \exists \Delta'. N$ by 11, F-ENV-EMPTY, lemma 17  $let \ \Delta' = \overline{\mathsf{Z} \to [\mathsf{B}_l \ \mathsf{B}_u]}$ 13. 14. There exists  $\overline{T_s}$  $\vdash C < \overline{T} > \Box : [\overline{T_s/Z}] N$ 15. by 12, F-ENV-EMPTY, lemma 35  $\emptyset \vdash T_s <: [\overline{T_s/Z}] B_u$ **16**.  $\emptyset \vdash [\overline{T_s/Z}]B_l <: T_s$ 17.  $fv(\overline{T_s}) = \emptyset$ 18.  $\emptyset \vdash T$  ok 19. by **a**, F-ENV-EMPTY, lemma 30  $\mathbf{U}_i = fType(\mathbf{f}_i, [\overline{\mathbf{T}_s/\mathbf{Z}}]\mathbf{N})$ 20. by 15, 6, 9, lemma 23  $\mathbf{U}_i = [\overline{\mathbf{T}_s/\mathbf{Z}}] fType(\mathbf{f}_i, \mathbf{N})$ 21. *by* **20**, *lemma* 5 22.  $U_i = [\overline{T_s/Z}]T'$ by 21, 6  $\emptyset \vdash [\overline{T_s/Z}]T' <: [\overline{T_s/Z}]T$ 23. by 7, 13, 16, 17, 18, lemma 18  $\emptyset \vdash U_i <: T$ 24. by 23, 19, 22 25.  $\emptyset; \emptyset \vdash \mathbf{v}_i : \mathbf{T} \mid \emptyset$ by 10, 24, F-ENV-EMPTY, 19, T-SUBS Case 2. (R-INVK)  $e = v.\langle \overline{P} \rangle(\overline{v})$ 1. by def R-INVK  $e' = [\overline{T/Y}, \overline{v/x}, v/this]e_0$ 2.  $v = new C < \overline{T'} > (\overline{v'})$ 3.  $v = new N(\overline{v''})$ 4. by premises R-INVK  $mBody(\mathbf{m}, C < \overline{T'} >) = (\overline{\mathbf{x}}; \mathbf{e}_0)$ 5.  $mType(\mathbf{m}, \mathbb{C}\langle \overline{\mathbf{T}'} \rangle) = \langle \overline{\mathbf{Y} \triangleleft \mathbf{B}} \rangle \overline{\mathbf{U}} \rightarrow \mathbf{U}$ 6. 7.  $match(sift(\overline{N},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$ 

8. 
$$\emptyset \vdash \Delta', \overline{\Delta}$$
 or  
9.  $\emptyset, \emptyset \vdash v : \exists \Delta', \mathbb{N} \mid \emptyset$   
10.  $mType(\underline{n}, \mathbb{N}) = \langle \overline{Y' \triangleleft S'} \mid \overline{S'} \mid \overline{V'} \rightarrow U''$   
11.  $\emptyset, \emptyset \vdash v : \exists \Delta', \mathbb{N} \mid [\emptyset]$   
12.  $match(siff(\overline{\mathbb{R}}, \overline{Y'}, \overline{Y}, \overline{Y}, \overline{Y'}, \overline{Y'})$   
13.  $\emptyset \vdash \overline{P}$  or  
14.  $\Delta', \overline{\Delta} \vdash \overline{T'' \land Y'} \mid \overline{U''}$   
15.  $\Delta', \overline{\Delta} \vdash \overline{U'' \land Y'} \mid \overline{U''}$   
16.  $\Delta', \overline{\Delta} \vdash \overline{U'' \land Y'} \mid \overline{U''}$   
17.  $\emptyset \vdash \exists \Delta', \mathbb{N}$  or  
19.  $fidds(\underline{C} = \overline{1}$   
20.  $fType(\underline{f}, \mathbb{C}, \overline{T'}) = U'$   
21.  $\emptyset, \emptyset \vdash \overline{V' : U'' \mid \emptyset}$   
22.  $\emptyset \vdash \overline{N} \circ \mathbb{C}^{T'} > \mathbb{C} : \exists \Delta', \mathbb{N}$   
23.  $\emptyset \vdash \overline{N} \circ \mathbb{C}^{T'} > \mathbb{C} : \exists \Delta', \mathbb{N}$   
24.  $\emptyset \vdash \overline{N} \circ \mathbb{C}^{T'} > \mathbb{C} : \exists \Delta', \mathbb{N}$   
25.  $ftdds(\mathbb{N}) = \overline{1}$   
26.  $fType(\underline{f}, \mathbb{C}, \overline{T'}) = U'$   
27.  $\emptyset \vdash v'' : U' \mid \overline{0}$   
28.  $\emptyset \vdash \overline{2} \Delta, \mathbb{R}$  or  
29.  $\emptyset \vdash T \circ \mathbb{K}$   
29.  $\emptyset \vdash T \circ \mathbb{K}$   
29.  $\emptyset \vdash T \circ \mathbb{K}$   
20.  $\emptyset \vdash \overline{2} \Delta, \mathbb{R}$  or  
21.  $\widehat{\Delta} \downarrow \widehat{N} \wedge \mathbb{C} : \exists \Delta, \mathbb{R}$   
29.  $\emptyset \vdash T \circ \mathbb{K}$   
20.  $\emptyset \vdash \overline{1} \Delta, \mathbb{R}$  or  
21.  $\widehat{\Delta} \vdash \widehat{N} \wedge \mathbb{C} : \exists \Delta, \mathbb{R}$   
23.  $\widehat{\Delta} \vdash \overline{T'} \otimes \mathbb{C} : \widehat{D}, \mathbb{K}$   
34.  $there exists \overline{N}_{fresh} : \overline{\mathbb{R}} = \overline{N}_{fresh}$   
35.  $\widehat{\Theta} \vdash \overline{2} \Lambda, \mathbb{C} : \overline{\mathbb{N}} \setminus \mathbb{K}, \overline{\mathbb{K}}, \overline{\mathbb{T}'}, \overline{\mathbb{T}'}$   
36.  $tat \Delta' = \overline{\chi}_{-} - \overline{\mathbb{R}_{H}} \mathbb{B}_{H^{-1}}$   
37.  $There exists \overline{U},$   
38.  $\vdash \mathbb{C}(\overline{U}, (\overline{X}, \mathbb{R})) = \overline{U}_{J}/\overline{X}, \mathbb{R} \setminus \overline{Y'} \in \mathbb{R}^{J} \to \mathbb{R}^{J}$   
37.  $There exists \overline{U},$   
38.  $\vdash \mathbb{C}(\overline{U}, \overline{X}, \mathbb{R}) = \overline{U} = \overline{U}, \overline{X}, \mathbb{R} \setminus \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus \mathbb{R}^{J}$   
37.  $There exists \overline{U},$   
38.  $\vdash \mathbb{C}(\overline{U}, \overline{X}, \mathbb{R}) = \overline{U} = \overline{U}, \overline{X}, \mathbb{R} \setminus \overline{Y'} \in \mathbb{R}^{J} \cup \overline{U'} \to U''$   
37.  $There exists \overline{U},$   
38.  $\vdash \mathbb{C}(\overline{U}, \overline{X}, \mathbb{R}) = \overline{U} = \overline{U} \circ \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus \mathbb{R} \in \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus \mathbb{R}^{J} = \overline{\mathbb{R}} \in \mathbb{R}^{J} = \overline{\mathbb{R}} \in \mathbb{R}^{J} = \overline{\mathbb{R}} \in \mathbb{R}^{J} = \overline{\mathbb{R}} \setminus$ 

59.	$\Delta', \overline{\mathbf{Y}' \to [\bot \ \mathbf{B}']} \vdash \mathbf{U}'' \ \mathbf{OK}$	)
60.	$\Delta', \underline{\underline{\mathbf{Y}' \to [\bot \ \mathbf{B}']}} \vdash \underline{\underline{\mathbf{U}''}}$ ok	<i>by</i> <b>10</b> , <b>17</b> , F
61.	$\Delta', \Upsilon' \rightarrow [\bot B'] \vdash B' OK$	J
62.	$\Delta' \vdash \boxed{[\overline{\mathtt{U}_s/\mathtt{X}_s}]\mathtt{T}'' <:  [\overline{\mathtt{U}_s/\mathtt{X}_s}][\overline{\mathtt{T}''/\mathtt{Y}'}]\mathtt{B}'}$	by 14, 55, 56, 58,
63.	$\Delta' \vdash \underbrace{[\overline{\mathbf{U}_s}/\mathbf{X}_s] \mathbf{T}'' <: [\overline{[\overline{\mathbf{U}_s}/\mathbf{X}_s] \mathbf{T}''/\mathbf{Y}'}] \mathbf{B}'$	by <b>62</b> , <b>61</b>
64.	$\Delta' \vdash \mathtt{T} <:  [\overline{\mathtt{T/Y}}]  \mathtt{B}'$	by <b>63</b> , <b>54</b> , <b>45</b>
65.	$\emptyset \vdash \overline{[\overline{\mathbf{U}_{x}}/\mathbf{X}_{x}]} \mathtt{T} <: \overline{[\overline{\mathbf{U}_{x}}/\mathbf{X}_{x}]} \overline{[\mathbf{T}/\mathbf{Y}]} \mathtt{B}'$	by 64, 39, 40, 41,
66.	$\emptyset \vdash \overline{\mathtt{T} <: [\overline{\mathtt{T}/\mathtt{Y}}] [\overline{\mathtt{U}_x/\mathtt{X}_x}] \mathtt{B}'}$	<i>by</i> <b>65</b> , <b>29</b> , <b>6</b> , <b>36</b> ,
67.	$\emptyset \vdash \overline{T <: [\overline{T/Y}]B}$	<i>by</i> <b>66</b> , <b>48</b>
68.	$\Delta' \vdash \exists \emptyset. [\overline{\mathtt{U}_s/\mathtt{X}_s}] \mathtt{R} <: [\overline{\mathtt{U}_s/\mathtt{X}_s}] [\overline{\mathtt{T}''/\mathtt{Y}'}] \mathtt{U}''$	by 15, 55, 56, 58,
69.	$\Delta' \vdash \exists \emptyset. [\overline{\mathbb{U}_s/\mathbb{X}_s}] \mathbb{R} <: [\overline{[\mathbb{U}_s/\mathbb{X}_s}] \mathbb{T}''/\mathbb{Y}'] \mathbb{U}''$	by <b>63</b> , <b>60</b>
70.	$\Delta' \vdash \exists \emptyset. [\overline{\mathtt{U}_s/\mathtt{X}_s}] \mathtt{R} <: [\overline{\mathtt{T}/\mathtt{Y}}] \mathtt{U}''$	<i>by</i> <b>69</b> , <b>54</b> , <b>45</b>
71.	$\emptyset \vdash \exists \emptyset. [\overline{U_x/X_x}] [\overline{U_s/X_s}] R <: [\overline{U_x/X_x}] [\overline{T/Y}] U''$	by <b>70</b> , <b>39</b> , <b>40</b> , <b>41</b> ,
72.	$\emptyset \vdash \overline{\exists \emptyset. [\overline{U_x/X_x}] [\overline{U_s/X_s}] R <: [\overline{T/Y}] [\overline{U_x/X_x}] U''}$	<i>by</i> <b>71</b> , <b>29</b> , <b>6</b> , <b>36</b> ,
73.	$\emptyset \vdash \overline{\exists \emptyset  .  [\overline{\mathrm{U}_x / \mathrm{X}_x}]  [\overline{\mathrm{U}_s / \mathrm{X}_s}]  \mathrm{R} <:  [\overline{\mathrm{T} / \mathrm{Y}}]  \mathrm{U}}$	<i>by</i> <b>72</b> , <b>46</b>
74.	$\emptyset \vdash \overline{\exists \emptyset. \mathbb{N} \sqsubseteq: \exists \emptyset. [\overline{\mathbb{U}_s / \mathbb{X}_s}] \mathbb{R}}$	by <b>57</b> , lemma 34
75.	$\emptyset \vdash \overline{\exists \emptyset. [\overline{U_x/X_x}] N \sqsubset} : \exists \emptyset. [\overline{U_x/X_x}] [\overline{U_s/X_s}] R$	by <b>74</b> , <b>39</b> , <b>40</b> , <b>41</b> ,
76.	$\emptyset \vdash \exists \emptyset. [\overline{U_x/X_x}] N <: [\overline{T/Y}] U$	<i>by</i> <b>73</b> , <b>75</b> , XS-TE
77.	$\emptyset \vdash \exists \emptyset. \mathbb{N} <: [\overline{\mathbb{T}/\mathbb{Y}}] \mathbb{U}$	by <b>76</b> , <b>24</b>
78.	$\Delta' \vdash [\overline{U_s/X_s}] [\overline{T''/Y'}] U'' <: [\overline{U_s/X_s}] T$	<i>by</i> <b>16</b> , <b>55</b> , <b>56</b> , <b>58</b> ,
<b>79</b> .	$\Delta' \vdash [\overline{\mathbf{U}_s/\mathbf{X}_s}] [\mathbf{T}''/\mathbf{Y}'] \mathbf{U}'' <: \mathbf{T}$	by <b>78</b> , <b>29</b>
80.	$\Delta' \vdash [[\overline{U_s/X_s}]T''/Y']U' <: T$	by <b>79</b> , <b>59</b>
81.	$\Delta' \vdash [\overline{T/Y}] U'' <: T$	by <b>80</b> , <b>54</b> , <b>45</b>
82.	$\emptyset \vdash [\mathbf{U}_x/\mathbf{X}_x] [\mathbf{T}/\mathbf{Y}] \mathbf{U}'' <: [\mathbf{U}_x/\mathbf{X}_x] \mathbf{T}$	<i>by</i> <b>81</b> , <b>39</b> , <b>40</b> , <b>41</b> ,
83.	$\emptyset \vdash [U_x/X_x] [T/Y] U <: T$	<i>by</i> <b>82</b> , <b>29</b>
84.	$\emptyset \vdash [\underline{T}/\underline{Y}] [\underline{U}_x / X_x] U <: T$	by 83, 29, 6, 36,
85. 86	$ \begin{array}{c} \emptyset \vdash [1/Y] \cup <: 1 \\ \hline \hline$	0y 84, 47
80. 87	$Y \rightarrow [\bot B]; x: 0, this: C \vdash e_0: 0 \mid \emptyset$ $\emptyset \vdash \overline{T} \cap V$	by <b>F</b> -ENV-EMPT
01.	ØT I OK	$F_{\rm ENV}$
	$(h, \overline{\pi, [\pi/N]})$ + high $[\pi/N]$ ( $(\overline{\pi/N})$	hu 86 67 XS B
00.	$\emptyset, \mathbf{X} : [\mathbf{I} / \mathbf{I}] 0, \text{ cmis} : [\mathbf{I} / \mathbf{I}] 0 \in \mathbf{I} / \mathbf{V}$	87 lemma 91
80	$\emptyset_{1} \xrightarrow{[1]{1}} [\overline{1}, \overline{1}] = 0  [1]{1} $	ba 99 19
09. 00	$\emptyset, X: [1/1]0, \text{cms:} C<1 \ge [1/1]e_0: [1/1]0   \emptyset$ $\emptyset: \emptyset \vdash w: C<\overline{T'} \setminus \emptyset$	by 18, 10, 20, 21
90. 01	$\emptyset, \emptyset \vdash \overline{\forall : C C I \neq   \emptyset}$ $\emptyset, \emptyset \vdash \overline{w} : \exists \emptyset   N = I \emptyset$	by 10, 19, 20, 21, by 24, 25, 26, 27
91.	$\psi, \psi \vdash \nabla \cdot \exists \psi. N \downarrow \psi$	by 24, 23, 20, 21,
92.	$\emptyset \vdash [1/Y] \cup OK$	0y 50, 87, F-ENV 67 lamma 10
93.	$\emptyset \cdot \emptyset \vdash [\overline{T/Y}, \overline{y/x}, y/this]e_0 \cdot [\overline{T/Y}]U \mid \emptyset$	bu 89 90 91 18
	ν, ν. ει, <b>ι, ι, ι, ι, ι, ι</b> , οι οι οι τι τι ο   φ	<b>92</b> , <b>77</b> , <i>lemma</i> 25
94.	$\emptyset; \emptyset \vdash [\overline{\mathtt{T/Y}}, \ \overline{\mathtt{v/x}}, \ \mathtt{v/this}] \mathtt{e}_0 : \mathtt{T} \mid \emptyset$	<i>by</i> <b>93</b> , <b>85</b> , F-Env

Case 3. (RC-FIELD)

1.  $\mathbf{e} = \mathbf{e}_r \cdot \mathbf{f}$ 2.  $\mathbf{e}' = \mathbf{e}'_r \cdot \mathbf{f}$ 3.  $\mathbf{e}_r \rightsquigarrow \mathbf{e}'_r$ 4.  $\exists \Delta_n :$ 5.  $\emptyset \vdash \Delta_n \text{ OK}$ 6.  $\emptyset; \emptyset \vdash \mathbf{e}_r : \exists \Delta_n \cdot \mathbb{N} \mid \emptyset$ 7.  $\Delta_n \vdash fType(\mathbf{f}, \mathbb{N}) <: \mathbb{T}$  '-ENV-EMPTY, lemma 28 lemma 18 lemma 18 lemma 15  $lemma \ 18$ lemma 18 lemma 15 lemma 18 RANS, S-SC lemma 18 lemma 18 lemma 15 Y, 18, 5, 6, lemma 38 ST, **7**, lemma 29 ITM, F-ENV-EMPTY T-NEW T-NEW и-Емрту, XS-Вттм, -Empty, T-Subs

by def RC-FIELD
by premise RC-FIELD

by 1, a, F-ENV-EMPTY, lemma 32

8.  $\emptyset; \emptyset \vdash \mathbf{e}'_r : \exists \Delta_n . \mathbb{N} \mid \emptyset$ 9.  $\emptyset; \emptyset \vdash \mathbf{e}'_r . \mathbf{f} : fType(\mathbf{f}, \mathbb{N}) \mid \Delta_n$ 10.  $\emptyset \vdash \mathsf{T} \text{ OK}$ 11.  $\emptyset; \emptyset \vdash \mathbf{e}'_r . \mathbf{f} : \mathbb{T} \mid \emptyset$ 12. done

Case 4. (RC-NEW-ARG)

1.  $e = new C < \overline{T} > (\overline{e})$ 2.  $e' = new C < \overline{T} > (\overline{e'})$ 3.  $\overline{e} = \ldots e_i \ldots$  $e' = \ldots e'_i \ldots$ 4.  $e_i \sim e'_i$ 5.  $\emptyset \vdash \mathbb{C} < \overline{\mathbb{T}} > OK$ 6. 7.  $fields(C) = \overline{f}$  $fType(f, C<\overline{T}>) = U$ 8.  $\emptyset; \emptyset \vdash \overline{\mathbf{e} : \mathbf{U} \mid \emptyset}$ 9. 10.  $\emptyset \vdash \exists \emptyset. C < \overline{T} > <: T$ 11.  $\emptyset; \emptyset \vdash \mathbf{e}'_i : \exists \Delta_i . \mathbf{R}_i \mid \emptyset$  $\emptyset; \emptyset \vdash \text{new } C < \overline{T} > (\overline{e'}) : \exists \emptyset. C < \overline{T} > | \emptyset$ 12. 13.  $\emptyset \vdash T \text{ ok}$  $\emptyset; \emptyset \vdash \text{new } C < \overline{T} > (e') : T \mid \emptyset$ 14. 15. done

Case 5. (RC-INVK-RECV)

1.  $e = e_r . \langle \overline{P} \rangle m(\overline{e})$  $e' = e'_r . < \overline{P} > m(\overline{e})$ 2. 3.  $e_r \rightsquigarrow e'_r$  $\Delta_n = \Delta'', \overline{\Delta}$ 4. 5.  $\emptyset \vdash \Delta_n$ ок  $\emptyset; \emptyset \vdash \mathbf{e}_r : \exists \Delta''. \mathbb{N} \mid \emptyset$ 6. 7.  $mType(\mathbf{m}, \mathbf{N}) = \langle \overline{\mathbf{Y} \triangleleft} \ \overline{\mathbf{B}} \rangle \overline{\mathbf{U}} \to \mathbf{U}$ 8.  $\emptyset; \emptyset \vdash \mathbf{e} : \exists \Delta . \mathbf{R} \mid \emptyset$ 9.  $match(sift(\overline{R},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$ 10.  $\emptyset \vdash \overline{P} \text{ ok}$ 11.  $\Delta'', \overline{\Delta} \vdash \mathtt{T} <: [\overline{\mathtt{T/Y}}]\mathtt{B}$  $\Delta'', \overline{\Delta} \vdash \exists \emptyset. \mathbb{R} <: [\overline{T/Y}] \mathbb{U}$ 12.  $\Delta'', \Delta_n \vdash [\overline{\mathbf{T/Y}}] \mathbf{U} <: \mathbf{T}$ 13.  $\emptyset; \emptyset \vdash \mathbf{e}'_r : \exists \Delta'' . \mathbb{N} \mid \emptyset$ 14.  $\emptyset; \emptyset \vdash \mathbf{e}'_r. \langle \overline{\mathsf{P}} \rangle m(\overline{\mathbf{e}}) : [\overline{\mathsf{T/Y}}] \mathsf{U} \mid \Delta'', \overline{\Delta}$ 15. 16.  $\emptyset \vdash T \text{ ok}$  $\emptyset; \emptyset \vdash \mathbf{e}'_r . < \overline{P} > m(\overline{\mathbf{e}}) : T \mid \emptyset$ 17. 18. done

Case 6. (RC-INVK-ARG)

1.  $e = e_r . < \overline{P} > m(\overline{e})$ 2.  $e' = e_r . < \overline{P} > m(\overline{e'})$ 3.  $\overline{e} = ... e_i ...$ 4.  $\overline{e'} = ... e'_i ...$ 5.  $e_i \sim e'_i$  by 3, 6, ind hyp by 8, T-FIELD by a, c, lemma 30 by 9, 7, 5, 10, T-SUBS by 11, 2

by def RC-New-Arg

by premise RC-NEW-ARG

by **1**, **a**, *lemma 31* 

by 5, 9, ind hyp by 6, 7, 8, 9, 11, T-NEW by a, F-ENV-EMPTY, lemma 30 by 12, 10, F-ENV-EMPTY, 13, T-SUBS by 14, 2

by def RC-INVK-RECV
by premise RC-INVK-RECV

by 1, a, F-ENV-EMPTY, lemma 33

by 3, 6, ind hyp by 14, 7, 8, 9, 10, 11, 12, T-INVK by a, F-ENV-EMPTY, lemma 30 by 15, 13, 4, 5, 16, T-SUBS by 17, 2

> by def RC-Invk-Arg

by premise RC-INVK-ARG

 $\Delta_n = \Delta'', \overline{\Delta}$ 6. 7.  $\emptyset \vdash \Delta_n$ ок  $\emptyset; \emptyset \vdash \mathbf{e}_r : \exists \Delta''. \mathtt{N} \mid \emptyset$ 8. 9.  $mType(\mathtt{m},\,\mathtt{N}) = <\overline{\mathtt{Y} \lhd \, \mathtt{B}} > \overline{\mathtt{U}} \to \mathtt{U}$ 10.  $\emptyset; \emptyset \vdash \mathbf{e} : \exists \Delta . \mathbf{R} \mid \emptyset$  $match(sift(\overline{R},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$ 11. 12.  $\emptyset \vdash \overline{P}$  ок  $\Delta, \Delta'', \overline{\Delta} \vdash \overline{\mathtt{T} <: [\overline{\mathtt{T/Y}}]\mathtt{B}}$ 13.  $\Delta, \Delta'', \overline{\Delta} \vdash \exists \emptyset. R <: [\overline{T/Y}] U$ 14.  $\Delta'', \overline{\Delta} \vdash [\overline{T/Y}] U <: T$ 15.  $\emptyset; \emptyset \vdash \mathbf{e}'_i : \exists \Delta_i . \mathbf{R}_i \mid \emptyset$ 16. 17.  $\emptyset; \emptyset \vdash \mathbf{e}_r \,.\, \langle \overline{\mathsf{P}} \rangle \mathtt{m}(\overline{\mathsf{e}'}) \,:\, [\overline{\mathsf{T}/\mathsf{Y}}] \, \mathtt{U} \,|\, \Delta'', \overline{\Delta}$ 18.  $\emptyset \vdash T$  ok  $\emptyset; \emptyset \vdash \mathbf{e}_r . < \overline{\mathbf{P}} > \mathfrak{m}(\overline{\mathbf{e}'}) : \mathbf{T} \mid \emptyset$ 19. 20. done

by 1, a, F-ENV-EMPTY, lemma 33 by 5, 10, ind hyp by 8, 9, 10, 16, 11, 12, 13, 14, T-INVK by a, F-ENV-EMPTY, lemma 30

by 17, 15, 6, 7, 18, T-SUBS

by 19, 2

#### Theorem (Progress)

If: **a.**  $\emptyset; \emptyset \vdash \mathbf{e} : \mathbf{T} \mid \Delta$ then:  $\mathbf{e} \rightsquigarrow \mathbf{e}'$ or: there exists  $\mathbf{v}$  such that  $\mathbf{e} = \mathbf{v}$ 

Proof is by structural induction on the derivation of  $\emptyset$ ;  $\emptyset \vdash \mathbf{e} : \mathsf{T} \mid \Delta$  with a case analysis on the last step:

Case 1. (T-VAR)

1.	$\mathtt{T}=\Gamma(x)$	$by \ def \ T-VAR$
2.	$\Gamma = \emptyset$	$by \mathbf{a}$
3.	case N/A	by contradiction, $1, 2$

Case 2. (T-NEW)

1.	$e = new N(\overline{e})$	by def T-New
2.	$\emptyset; \emptyset \vdash \overline{e: U \mid \emptyset}$	by premise of T-NEW
3.	$\forall \mathbf{e}_i \in \overline{\mathbf{e}} : \text{ there exists } \mathbf{v}_i \text{ with } \mathbf{e}_i = \mathbf{v}_i \text{ or there}$	$e \ exists \ \mathbf{e}_i \in \overline{\mathbf{e}} \ with \ \mathbf{e}_i \rightsquigarrow \mathbf{e}'_i$
		<b>2</b> , ind hyp

#### Case analysis on $\overline{e}$ :

Case 1.  $\forall e_i \in \overline{e} : \exists v_i . e_i = v_i$ 

1.1.	$e = new N(\overline{v})$	$by \ 1$
1.2.	done	by 1.1

Case 2.  $\exists e_i \in \overline{e} : e_i \rightsquigarrow e'_i$ 

**2.1.** *done* 

by RC-NEW

by def T-FIELD

by  $\mathbf{2}$ , ind hyp

by RC-Field

by premises T-Field

Case 3. (T-FIELD)

- 1.  $e = e_r . f$
- **2.**  $\emptyset; \emptyset \vdash \mathbf{e}_r : \exists \Delta . \mathbb{N} \mid \emptyset$
- $3. \qquad fType(\mathtt{f},\mathtt{N})=\mathtt{T}$
- 4.  $\mathbf{e}_r \sim \mathbf{e}'_r$  or there exists  $\mathbf{v}_r$  with  $\mathbf{e}_r = \mathbf{v}_r$

Case analysis on  $e_r$ :

Case 1.  $e_r \rightsquigarrow e'_r$ 

**1.1.** *done* 

Case 2. there exists  $v_r$  where  $e_r = v_r$ 

let  $v_r = \text{new } C < \overline{T} > (\overline{v}).f$ 2.1. 2.2.  $fields(C) = \overline{f}$  $\emptyset \vdash \exists \emptyset.C < \overline{T} > <: \exists \Delta.N$ 2.3.  $\emptyset \vdash \exists \emptyset. C < \overline{T} > \Box : \exists \Delta. N$ 2.4. 2.5.there exists  $\overline{\mathtt{U}}$ 2.6.  $dom(\Delta) = \overline{Z}$  $\vdash C < \overline{T} > \Box : [\overline{U/Z}] \mathbb{N}$ 2.7.  $fType(f, [\overline{U/Z}]N) = [\overline{U/Z}]T$ 2.8. 2.9.  $fType(f, C<\overline{T}>) = [\overline{U/Z}]T$ 2.10.  $f \in \overline{f}$ **2.11.** *done* 

by 2, 2.1, lemma 31
 by 2.3, F-ENV-EMPTY, lemma 17
 by 2.4, F-ENV-EMPTY, lemma 35
 by 3, lemma 5
 by 2.8, 2.7, lemma 23

*by* **2.1**, **2.2**, **2.10**, RC-FIELD

by premise of T-SUBS

by  $\mathbf{1}$ , ind hyp

by 2.2, 2.9, lemma 40

Case 4. (T-SUBS)

- $\mathbf{1.} \qquad \emptyset; \emptyset \vdash \mathbf{e} : \mathbf{U} \mid \Delta'$
- **2.** *done*

Case 5. (T-INVK)

 $e = e_r . \langle \overline{P} \rangle m(\overline{e})$ by def T-INVK 1.  $\emptyset; \emptyset \vdash \mathbf{e}_r : \exists \Delta' \, . \, \mathbf{N} \mid \emptyset$ 2.  $\emptyset; \emptyset \vdash \mathbf{e} : \exists \Delta . \mathbf{R} \mid \emptyset$ 3. by premises T-Invk  $mType(\mathbf{m}, \mathbf{N}) = \langle \overline{\mathbf{Y} \triangleleft} \ \overline{\mathbf{T}_u} \rangle \overline{\mathbf{U}} \rightarrow \mathbf{U}$ **4**. 5.  $match(sift(\overline{R},\overline{U},\overline{Y}),\overline{P},\overline{Y},\overline{T})$  $\emptyset \vdash \overline{P}$  ok 6. 7.  $\mathbf{e}_r \rightsquigarrow \mathbf{e}'_r$  or (there exists  $\mathbf{v}_r$  with  $\mathbf{e}_r = \mathbf{v}_r$ ) by  $\mathbf{2}$ , ind hyp 8.  $\forall \mathbf{e}_i \in \overline{\mathbf{e}}$ : (there exists  $\mathbf{v}_i$  with  $\mathbf{e}_i = \mathbf{v}_i$ ) or (there exists  $\mathbf{e}_i \in \overline{\mathbf{e}}$ :  $\mathbf{e}_i \rightsquigarrow \mathbf{e}'_i$ ) **3**, *ind hyp*  $\emptyset \vdash \exists \Delta' . \mathbb{N}$  ok 9. by 2, F-ENV-EMPTY, lemma 30 Case analysis on  $e_r, \overline{e}$ : Case 1.  $e_r \rightsquigarrow e'_r$ 1.1. doneby RC-INV-RECV Case 2. there exists  $e_i \in \overline{e}$  where  $e_i \sim e'_i$ 2.1. doneby RC-INV-ARG **Case 3. there exists**  $\mathbf{v}_r$  where  $\mathbf{e}_r = \mathbf{v}_r$  and  $\forall \mathbf{e}_i \in \overline{\mathbf{e}} : \exists \mathbf{v}_i \text{ where } \mathbf{e}_i = \mathbf{v}_i$ 3.1.  $let v_r = new N'(v')$ let  $\overline{\mathbf{v}}_r = \mathbf{new} \ \mathbf{N}'(\overline{\mathbf{v}''})$ 3.2.  $\emptyset \vdash \mathbb{N}'$  ok 3.3. by **3.1**, **2**, lemma 31  $\emptyset \vdash \exists \emptyset . \mathbb{N}' <: \exists \Delta' . \mathbb{N}$ 3.4. 3.5. $\emptyset \vdash \exists \emptyset . \mathbb{N}' \sqsubset : \exists \Delta' . \mathbb{N}$ by 3.4, F-ENV-EMPTY, lemma 17 3.6. let  $\Delta' = \overline{Z \rightarrow [B_l \ B_u]}$ 3.7.  $\vdash$  N'  $\square$ :  $[\overline{U'/Z}]$  N  $\emptyset \vdash U' <: [\overline{U'/Z}] B_u$ 3.8. by **3.5**, **3.6**, lemma 35 3.9.  $\emptyset \vdash [\overline{U'/Z}] B_l <: U'$  $fv(\overline{\mathbf{U}'}) \subseteq dom(\Delta)$ 3.10. **3.11.**  $mType(\mathbf{m}, [\overline{\mathbf{U}'/\mathbf{Z}}]\mathbf{N}) = [\overline{\mathbf{U}'/\mathbf{Z}}] < \overline{\mathbf{Y} \triangleleft \mathbf{T}_u} > \overline{\mathbf{U}} \rightarrow \mathbf{U}$ by 4, lemma 6 **3.12.**  $mType(\mathbf{m}, \mathbf{N}') = [\overline{\mathbf{U}'/\mathbf{Z}}] < \overline{\mathbf{Y} \triangleleft \mathbf{T}_u} > \overline{\mathbf{U}} \rightarrow \mathbf{U}$ by 3.11, 3.7, lemma 24 **3.13.**  $mBody(\mathbf{m}, \mathbf{N}')$  defined by **3.12**, lemma 39 **3.14.**  $\emptyset \vdash \overline{\exists \Delta . R}$  OK by 3, F-ENV-EMPTY, lemma 30 **3.15.**  $\overline{\Delta} \vdash \overline{\mathtt{T}}$  OK by 6, 3.14, F-ENV-EMPTY, 5, lemma 29 **3.16.**  $match(sift(\overline{R}, [\overline{U'/Z}]U, \overline{Y}), \overline{P}, \overline{Y}, \overline{T})$ by 5, 3.12, 4, 3.5, F-ENV-EMPTY, 3.15, lemma 36  $\emptyset \vdash \overline{\mathtt{N}'}$  ok 3.17. by 3.2, 3, lemma 31  $\emptyset \vdash \overline{\exists \emptyset . \mathbb{N}' <: \exists \Delta . \mathbb{R}}$ 3.18. **3.19.**  $\exists \overline{N_{fresh}} \text{ such that } \overline{R} = \overline{N_{fresh}}$ by 3.14 **3.20.**  $\emptyset \vdash \overline{\exists \emptyset . \mathbb{N}' \sqsubset : \exists \Delta . \mathbb{R}}$ by 3.18, 3.19, F-ENV-EMPTY, lemma 17 **3.21.**  $\emptyset \vdash \overline{\exists \emptyset . \mathbb{N}'}$  OK by 3.17, F-ENV-EMPTY, F-EXISTS **3.22.** let  $\overline{X_s \rightarrow [B_{sl} \ B_{su}]} = \overline{\Delta}$ **3.23.** wlog assume  $\overline{X_s}$  are fresh by **3.22**, Barendregt  $match(sift(\overline{N'}, [\overline{U'/Z}]U, \overline{Y}), \overline{P}, \overline{Y}, [\overline{U_s/X_s}]T)$ 3.24. by 3.20, 3.16, 3.23, 3.22, 3.25.  $\overline{\emptyset} \vdash \mathtt{U}_s <: [\overline{\mathtt{U}_s / \mathtt{X}_s}] \mathtt{B}_u$ F-ENV-EMPTY, 3.21, 6, lemma 37  $\overline{\emptyset} \vdash [\overline{U_s/X_s}] B_l <: U_s$ 3.26.  $\vdash \mathbb{N}' \square : [\overline{\mathbb{U}_s / \mathbb{X}_s}] \mathbb{R}$ 3.27. **3.28.** done by 3.12, 3.13, 3.24, R-INVK

# Appendix B

# Proofs of properties of Jo∃

In this appendix we list all lemmas, and the interesting proofs, used in the proof of soundness and owners-as-dominators for  $Jo\exists$  and  $Jo\exists_{deep}$ . As in the proofs for Tame FJ, we require that the program is well-formed in all lemmas, i.e., for all class declarations, Q, in the program,  $\vdash Q$  OK. Throughout, we assume the Barendregt convention, i.e., bound and free variables are distinct.

We use fv to find the free variables of an expression or type. Where we need to be precise we use  $fv_{\gamma}$ ,  $fv_{o}$ ,  $fv_{x}$  to find the free expression variables, owner variables, and type variables, respectively. Otherwise, we use just fv where it is clear from the context what kind of free variables we are concerned with.

Lemmas and proof steps that only apply to  $Jo\exists_{deep}$  and are used only in the proof of ownersas-dominators are marked in grey; those parts that only apply in  $Jo\exists$  without owners-asdominators are highlighted using a box.

Full proofs of all lemmas can be downloaded from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron\_joexists\_proofs.pdf

## **B.1** Outline of proofs

As in the proofs for Tame FJ, the lemmas in the next section are sequenced so that they only use earlier lemmas. Lemmas 1–2, 4–12, 32, and 52–59 are "common sense" lemmas about the system, including weakening and inversion lemmas. Lemmas 3 and 31 show which entities have existential and non-existential type. Lemmas 13–30, 33–41, and 43 are substitution lemmas for three varieties of substitution. These constitute the bulk of work in the proofs, but are fairly tedious (but fiddly) to prove. Lemmas 42, 44–49, and 58 concern well-formedness: when it can be expected and properties of well-formed types; some follow easily from the judgement rules, those concerning field and method types were surprisingly tricky due to the substitutions involved. Lemma 50 relates method types to method bodies. Lemma 51 states that reduction preserves relations that are judged using the heap. These two lemmas are straightforward. Lemmas 60–65 are properties of ownership which are used only to prove the

owners-as-dominators property. These are the most interesting lemmas to formulate and prove and required some fiddling with the system to get the existential quantification aspects correct.

In general, these proofs are much more straightforward than those for Tame FJ. Subject reduction is proved by a standard case analysis on the reduction rules which require the supporting lemmas. The owners-as-dominators property is proved by showing that reduction preserves this property of the heap; this is done as part of the subject reduction proofs. The only interesting case is field assignment where we must show that the new reference created by the assignment satisfies the owners-as-dominators property. This is done by showing that the owner of the object is outside the declared owner in the type of the field being assigned to. We also show that the owner in the field's type is outside the object that includes that field. By putting these two relationships together with transitivity we have the owners-as-dominators property for the updated reference.

## **B.2** Proofs

Lemma 1. (Reflexivity of subtyping)

If:

 $\Delta; \Gamma \vdash \mathtt{T} <: \mathtt{T}$ 

Proof is by case analysis on the structure of  ${\tt T}$ 

Lemma 2. (Transitivity of Subtyping)

```
\begin{array}{ll} \textit{If:} & & \\ \textbf{a.} & \Delta; \Gamma \vdash \mathtt{T}_1 <: \mathtt{T}_2 \\ \textbf{b.} & \Delta; \Gamma \vdash \mathtt{T}_2 <: \mathtt{T}_3 \\ \textit{then:} \\ \Delta; \Gamma \vdash \mathtt{T}_1 <: \mathtt{T}_3 \end{array}
```

Proof is by case analysis on the last step of each derivation of  $\Delta$ ;  $\Gamma \vdash T_1 <: T_2 \text{ and } \Delta$ ;  $\Gamma \vdash T_2 <: T_3$ 

### Lemma 3. (Subtyping preserves non-existential type)

```
If:

a. \Delta; \Gamma \vdash T <: M

then:

T = M

and if:

b. \Delta; \Gamma \vdash M <: T

then:

T = M
```
Proof is by case analysis on the last step of the derivation of  $\Delta; \Gamma \vdash T <: M \text{ or } \Delta; \Gamma \vdash M <: T:$ 

Lemma 4. (Weakening of well-formed owners)

If: **a.**  $\Delta, \Delta'; \Gamma, \Gamma' \vdash \mathbf{b} \text{ OK}$  **b.**  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$  **c.**  $dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset$ then:  $\Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash \mathbf{b} \text{ OK}$ 

Proof is by case analysis on  $\Delta, \Delta'; \Gamma, \Gamma' \vdash \mathbf{b}$  ok

```
Lemma 5. (Weakening of own)
```

If:

**a.**  $own_{\Psi,\Psi'}(\mathsf{T}) = \mathsf{b}$  **b.**  $dom(\Psi, \Psi') \cap dom(\Psi'') = \emptyset$ then:  $own_{\Psi,\Psi'',\Psi'}(\mathsf{T}) = \mathsf{b}$ 

Proof is by case analysis on T

### Lemma 6. (Weakening of the inside relation for owners)

If: **a.**  $\Delta, \Delta'; \Gamma, \Gamma' \vdash \mathbf{b}_1 \preceq \mathbf{b}_2$  **b.**  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$  **c.**  $dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset$ then:  $\Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash \mathbf{b}_1 \preceq \mathbf{b}_2$ 

Proof is by structural induction on the derivation of  $\Delta, \Delta'; \Gamma, \Gamma' \vdash b_1 \preceq b_2$ 

## Lemma 7. (Weakening of the inside relation for environments)

If: **a.**  $\Delta, \Delta'; \Gamma, \Gamma' \vdash \Delta_1 \preceq \Delta_2$  **b.**  $dom(\Delta, \Delta', \Delta_1, \Delta_2) \cap dom(\Delta'') = \emptyset$  **c.**  $dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset$ then:  $\Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash \Delta_1 \preceq \Delta_2$ 

Proof is by deduction

Lemma 8. (Weakening of subtyping)

If:

1). **a.**  $\Delta, \Delta'; \Gamma, \Gamma' \vdash T <: T'$  **b.**  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$  **c.**  $dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset$ then:  $\Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash T <: T'$ 

Proof is by case analysis on the last step of the derivation of  $\Delta, \Delta'; \Gamma, \Gamma' \vdash T <: T'$ 

## Lemma 9. (Weakening of well-formed types and environments)

If:  $\Psi, \Psi'; \Delta, \Delta'; \Gamma, \Gamma' \vdash \psi$  ok a.  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$ b.  $dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset$ c.  $dom(\Psi, \Psi') \cap dom(\Psi'') = \emptyset$ d. and if:  $\psi = \Delta'''$  then  $dom(\Delta, \Delta', \Delta_p) \cap dom(\Delta'') = \emptyset$ e. then:  $\Psi, \Psi'', \Psi'; \Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash \psi$  ok where:  $\psi ::= \Delta_p \mid \mathbf{T}$ 

Proof is by structural induction on the derivation of  $\Psi, \Psi'; \Delta, \Delta'; \Gamma, \Gamma' \vdash \psi$  OK

# Lemma 10. (Weakening of Typing)

 $\begin{array}{ll} \textit{If:} & \mathbf{a.} & \Psi, \Psi'; \Delta, \Delta'; \Gamma, \Gamma' \vdash \mathbf{e} : \mathtt{T} \\ & \mathbf{b.} & dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset \\ & \mathbf{c.} & dom(\Gamma, \Gamma') \cap dom(\Gamma'') = \emptyset \\ & \mathbf{d.} & dom(\Psi, \Psi') \cap dom(\Psi'') = \emptyset \\ & \textit{then:} \\ & \Psi, \Psi'', \Psi'; \Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma' \vdash \mathbf{e} : \mathtt{T} \end{array}$ 

Proof is by structural induction on the derivation of  $\Psi, \Psi'; \Delta, \Delta'; \Gamma, \Gamma' \vdash e: T$ 

# Lemma 11. (Weakening of well-formed heaps)

If: **a.**  $\Delta, \Delta' \vdash \mathcal{H} \text{ OK}$  **b.**  $dom(\Delta, \Delta') \cap dom(\Delta'') = \emptyset$ then:  $\Delta, \Delta'', \Delta' \vdash \mathcal{H} \text{ OK}$ 

Proof is by deduction

Lemma 12. (Inversion of subtyping)

If:  
**a.** 
$$\Delta; \Gamma \vdash \exists \overline{o \rightarrow [b_l \ b_u]} . \mathbb{N} <: \exists \overline{o' \rightarrow [b'_l \ b'_u]} . \mathbb{N}'$$
  
then:  
 $\overline{o} = \overline{o'}$   
 $\mathbb{N} = \mathbb{N}'$   
 $\Delta; \Gamma \vdash \overline{o \rightarrow [b_l \ b_u]} \preceq \overline{o' \rightarrow [b'_l \ b'_u]}$ 

Proof is by case analysis on the last step of the derivation of  $\Delta; \Gamma \vdash \exists \overline{\mathsf{o}} \to [\mathsf{b}_l \ \mathsf{b}_u]$ .  $\mathbb{N} <:$  $\exists \overline{\mathsf{o}'} \to [\mathsf{b}'_{l} \ \mathsf{b}'_{u}] \ . \mathbb{N}'$ 

Lemma 13. (Type substitution preserves well-formed owners)

If: a.  $\Delta; \Gamma_1, \Gamma_2 \vdash b \text{ ok}$ b.  $X \notin fv(\Gamma_1)$ 

then:

 $\Delta; \Gamma_1, [U/X] \Gamma_2 \vdash b \text{ OK}$ 

Proof is by case analysis on  $\Delta; \Gamma_1, \Gamma_2 \vdash b$  OK

Lemma 14. (Type substitution preserves the inside relation)

If:

 $\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{b} \preceq \mathbf{b}'$ a.  $X \notin fv(\Gamma_1)$ b. then:  $\Delta; \Gamma_1, [U/X] \Gamma_2 \vdash \mathbf{b} \preceq \mathbf{b}'$ 

Proof is by structural induction on the derivation of  $\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{b} \preceq \mathbf{b}'$ 

Lemma 15. (Type substitution preserves subtyping)

If:  $\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{T} <: \mathsf{T}'$ a.  $X \notin fv(\Gamma_1)$ b. then:  $\Delta; \Gamma_1, [U/X] \Gamma_2 \vdash [U/X] T <: [U/X] T'$ 

Proof is by case analysis on the last step of the derivation of  $\Delta$ ;  $\Gamma_1, \Gamma_2 \vdash T <: T'$ 

Lemma 16. (Type substitution preserves well-formed environments)

If:  $\Delta; \Gamma_1, \Gamma_2 \vdash \Delta'$  ок a.  $X \notin fv(\Gamma_1)$ b. then:

 $\Delta; \Gamma_1, [U/X] \Gamma_2 \vdash \Delta'$ ок

Proof is by structural induction on the derivation of  $\Delta; \Gamma_1, \Gamma_2 \vdash \Delta'$  OK

# Lemma 17. (Type substitution preserves well-formed types)

If: **a.**  $\Psi_1, X \rightarrow [b_l \bigcirc], \Psi_2; \Delta; \Gamma_1, \Gamma_2 \vdash T \text{ OK}$  **b.**  $\Psi_1; \Delta; \Gamma_1 \vdash U \text{ OK}$  **c.**  $X \notin fv(\Gamma_1)$  **d.**  $\Delta; \Gamma_1 \vdash b_l \preceq own_{\Psi_1}(U)$ then:  $\Delta; \Gamma_1, [U/X]\Gamma_2 \vdash [U/X]T \text{ OK}$ 

Proof is by structural induction on the derivation of  $\Psi_1, X \rightarrow [b_l \ b_u], \Psi_2; \Delta; \Gamma_1, \Gamma_2 \vdash T \text{ OK}$ 

## Lemma 18. (Substitution of types preserves field lookup)

If: **a.**  $fType(f, \gamma, C<\overline{a}, \overline{T}>) = T$ then:  $fType(f, \gamma, [T'/X']C<\overline{a}, \overline{T}>) = [T'/X']T$ 

Proof is by deduction

## Lemma 19. (Substitution of types preserves method lookup)

If:

a.  $mType_{\Delta;\Gamma_1,\Gamma_2}(\mathbf{m}<\overline{\mathbf{a}'}, \overline{\mathbf{T}'}>, \gamma, \mathbb{C}<\overline{\mathbf{a}}, \overline{\mathbf{T}''}>) = \overline{\mathbf{T}} \rightarrow \mathbb{T}$ b.  $\Psi_1; \Delta; \Gamma_1 \vdash \mathbb{U} \text{ OK}$ c.  $X \notin fv(\Gamma_1)$ then:  $mType_{\Delta:\Gamma_1,[\mathbb{U}/X]\Gamma_2}([\mathbb{U}'/\mathbb{Y}']\mathbb{m}<\overline{\mathbf{a}'}, \overline{\mathbf{T}'}>, \gamma, [\mathbb{U}'/\mathbb{Y}']\mathbb{C}<\overline{\mathbf{a}}, \overline{\mathbf{T}}>) = [\mathbb{U}'/\mathbb{Y}'](\overline{\mathbb{U}} \rightarrow \mathbb{U})$ 

Proof is by deduction

Lemma 20. (Type substitution preserves typing)

If: **a.**  $\Psi_1, \mathbb{X} \rightarrow [\mathbf{b}_l \bigcirc], \Psi_2; \Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{e} : \mathbb{T}$  **b.**  $\Psi_1; \Delta; \Gamma_1 \vdash \mathbb{U} \text{ OK}$  **c.**  $\mathbb{X} \notin fv(\Gamma_1)$  **d.**  $\Delta; \Gamma_1 \vdash \mathbf{b}_l \preceq own_{\Psi_1}(\mathbb{U})$ then:

 $\Psi_1,\Psi_2;\Delta;\Gamma_1,\,\texttt{[U/X]}\,\Gamma_2\vdash\,\texttt{[U/X]e}:\,\texttt{[U/X]T}$ 

Proof is by structural induction on the derivation of  $\Psi_1, X \rightarrow [b_l \bigcirc], \Psi_2; \Delta; \Gamma_1, \Gamma_2 \vdash e: T$ 

# Lemma 21. (Owner substitution preserves well-formed owners)

If:  $\Delta_1, \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{b} \text{ OK}$ a. b.  $\Delta_1; \Gamma_1 \vdash a \text{ OK}$  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash a \preceq [a/o]b_u$ c.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_l \preceq a$ d.  $o \notin fv(\Delta_1)$ e. f.  $o \notin fv(\Gamma_1)$ 

then:

 $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b$  OK

Proof is by case analysis on  $\Delta_1$ ,  $\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{b}$  OK

### Lemma 22. (Owner substitution preserves the inside relation)

If:  $\Delta_1, \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{b}_1 \preceq \mathbf{b}_2$ a.  $\Delta_1; \Gamma_1 \vdash a \text{ OK}$ b.  $\Delta_1$ , [a/o] $\Delta_2$ ;  $\Gamma_1$ , [a/o] $\Gamma_2 \vdash a \preceq [a/o]b_u$ c. d.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_l \preceq a$  $o \notin fv(\Delta_1)$ e. f.  $o \notin fv(\Gamma_1)$ then:  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_1 \preceq [a/o]b_2$ 

Proof is by structural induction on the derivation of  $\Delta_1$ ,  $\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{b}_1 \preceq \mathbf{b}_2$ 

## Lemma 23. (Corrollary to lemma 22)

If:  $\Delta_1, \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} \preceq \overline{\mathbf{o} \rightarrow [\mathbf{b}'_l \ \mathbf{b}'_u]}$ a.  $\Delta_1; \Gamma_1 \vdash a \text{ OK}$ b.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash a \preceq [a/o]b_u$ c.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_l \preceq a$ d.  $o \notin fv(\Delta_1)$ e. f.  $o \notin fv(\Gamma_1)$ then:

 $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash \overline{[a/o](o \rightarrow [b_l \ b_u])} \preceq \overline{[a/o](o \rightarrow [b'_l \ b'_u])}$ 

Proof is by deduction

### Lemma 24. (Owner substitution preserves subtyping)

If:

- $\Delta_1, \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{T}_1 <: \mathbf{T}_2$ a.
- $\Delta_1; \Gamma_1 \vdash \texttt{a } \text{OK}$ b.
- $\Delta_1$ , [a/o] $\Delta_2$ ;  $\Gamma_1$ , [a/o] $\Gamma_2 \vdash a \preceq [a/o]b_u$ c.
- d.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_l \preceq a$

```
e. o \notin fv(\Delta_1)

f. o \notin fv(\Gamma_1)

then:

\Delta_1, [a/o] \Delta_2; \Gamma, [a/o] \Gamma_2 \vdash [a/o] T_1 <: [a/o] T_2
```

Proof is by case analysis on the last step of the derivation of  $\Delta_1$ ,  $\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathbf{T}_1 <: \mathbf{T}_2$ 

Lemma 25. (Owner substitution preserves glb)

If:

a.  $glb_{\Delta_T}(\mathbf{b}) = \mathbf{b}'$ b.  $\Delta; \Gamma \vdash \mathbf{a} \ OK$ c.  $dom(\Delta) \cap dom(\Delta_T) = emptyset$ then:  $[\mathbf{a}/\mathbf{o}]\mathbf{b}' = glb_{[\mathbf{a}/\mathbf{o}]\Delta_T}([\mathbf{a}/\mathbf{o}]\mathbf{b})$ 

Proof is by structural induction on the derivation of  $glb_{\Delta_T}(\mathbf{b})$ 

Lemma 26. (Owner substitution preserves own)

If:

a.  $own_{\Psi,\Psi'}(\mathbf{T}) = \mathbf{b}$ b.  $\mathbf{o} \notin fv(\Psi)$ c.  $\Delta_1; \Gamma_1 \vdash \mathbf{a} \text{ OK}$ then:  $own_{\Psi, [\mathbf{a}/\mathbf{o}] \Psi'}([\mathbf{a}/\mathbf{o}]\mathbf{T}) = [\mathbf{a}/\mathbf{o}]\mathbf{b}$ 

Proof is by case analysis on T

### Lemma 27. (Owner substitution preserves well formedness)

If:

 $\Psi_1, \Psi_2; \Delta_1, \mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \psi \text{ ok}$ a.  $\Delta_1; \Gamma_1 \vdash a \text{ OK}$ b.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash a \preceq [a/o]b_u$ c.  $\Delta_1$ , [a/o] $\Delta_2$ ;  $\Gamma_1$ , [a/o] $\Gamma_2 \vdash$  [a/o] $b_l \preceq a$ d.  $o \notin fv(\Delta_1)$ e.  $o \notin fv(\Gamma_1)$ f.  $o \notin fv(\Psi_1)$ g. where: h.  $\psi ::= \Delta \mid \mathsf{T}$ then:  $\Psi_1$ ,  $[a/o]\Psi_2$ ;  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]\psi$  ok

Proof is by structural induction on the derivation of  $\Psi_1, \Psi_2; \Delta_1, o \rightarrow [b_l \ b_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \psi$  OK

Lemma 28. (Substitution of owners preserves field lookup)

 $\begin{array}{l} If: \\ \textbf{a.} \quad fType(\texttt{f},\gamma,\texttt{C}<\overline{\texttt{a}},\ \overline{\texttt{T}}>)=\texttt{T} \\ then: \\ \quad fType(\texttt{f},\gamma,\texttt{[a'/o']C<\overline{\texttt{a}},\ \overline{\texttt{T}}>)=\texttt{[a'/o']T} \end{array}$ 

 $Proof \ is \ by \ deduction$ 

## Lemma 29. (Substitution of owners preserves method lookup)

If:

a.  $mType_{\Delta_1, \mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u], \Delta_2; \Gamma_1, \Gamma_2}(\mathbf{m} < \overline{\mathbf{a}'}, \overline{\mathbf{T}'} >, \gamma, \mathbb{C} < \overline{\mathbf{a}}, \overline{\mathbf{T}} >) = \overline{\mathbf{T}} \to \mathbb{T}$ b.  $\Delta_1; \Gamma_1 \vdash \mathbf{a''} \ OK$ c.  $\Delta_1, [\mathbf{a''/o''}] \Delta_2; \Gamma_1, [\mathbf{a''/o''}] \Gamma_2 \vdash \mathbf{a''} \preceq [\mathbf{a''/o''}] \mathbf{b}_u$ d.  $\Delta_1, [\mathbf{a''/o''}] \Delta_2; \Gamma_1, [\mathbf{a''/o''}] \Gamma_2 \vdash [\mathbf{a''/o''}] \mathbf{b}_l \preceq \mathbf{a''}$ e.  $\mathbf{o} \notin fv(\Delta_1)$ f.  $\mathbf{o} \notin fv(\Gamma_1)$ then:

 $mType_{\Delta_1,[\mathbf{a}''/\mathbf{o}'']\Delta_2;\Gamma_1,[\mathbf{a}''/\mathbf{o}'']\Gamma_2}([\mathbf{a}''/\mathbf{o}'']\mathbf{m}<\overline{\mathbf{a}'}, \ \overline{\mathbf{T}'}>,\gamma,[\mathbf{a}''/\mathbf{o}'']\mathbf{C}<\overline{\mathbf{a}}, \ \overline{\mathbf{T}}>)=[\mathbf{a}''/\mathbf{o}''](\overline{\mathbf{U}}\rightarrow\mathbf{U})$ 

Proof is by deduction

## Lemma 30. (Owner substitution preserves typing)

If:  $\Psi_1, \Psi_2; \Delta_1, \mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash \mathsf{e} : \mathsf{T}$ a.  $\Delta_1; \Gamma_1 \vdash a \text{ OK}$ b.  $\Delta_1$ , [a/o] $\Delta_2$ ;  $\Gamma_1$ , [a/o] $\Gamma_2 \vdash a \preceq [a/o]b_u$ c.  $\Delta_1$ ,  $[a/o]\Delta_2$ ;  $\Gamma_1$ ,  $[a/o]\Gamma_2 \vdash [a/o]b_l \preceq a$ d.  $o \notin fv(\Delta_1)$ e. f.  $o \notin fv(\Gamma_1)$  $o \notin fv(\Psi_1)$ g. then:  $\Psi_1$ ,  $[a/o]\Psi_2$ ;  $\Delta_1$ ,  $[r/o]\Delta_2$ ;  $\Gamma_1$ ,  $[r/o]\Gamma_2 \vdash [r/o]e : [r/o]T$ 

Proof is by structural induction on the derivation of  $\Psi_1, \Psi_2; \Delta_1, o \rightarrow [b_l \ b_u], \Delta_2; \Gamma_1, \Gamma_2 \vdash e : T$ 

# Lemma 31. (Values with non-existential type are addresses)

If: **a.**  $\Psi; \Delta; \Gamma \vdash \mathbf{v} : \mathbf{M}$ then:  $\mathbf{v} = \iota$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash v : M$ 

Lemma 32. (Inversion lemma, address)

If:  $\Psi; \Delta; \Gamma \vdash \iota : \mathsf{T}$ a. then:  $\Delta; \Gamma \vdash \Gamma(\iota) <: \mathsf{T}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \iota : \mathsf{T}$ 

Lemma 33. (Value substitution preserves well-formed owners)

If:  $\Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{b}$  ok a.  $\Psi; \Delta; \Gamma \vdash v : U'$ b.  $\Delta; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{U}' <: \mathbf{U}$ c.  $\mathbf{x} \notin fv(\Delta)$ d. then:  $\Delta$ ,  $[v/x]\Delta'$ ;  $\Gamma$ ,  $[v/x]\Gamma' \vdash [v/x]b$  ok

Proof is by case analysis on  $\Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{b}$  OK

Lemma 34. (Value substitution preserves the inside relation)

If:  $\Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{b}_1 \leq \mathbf{b}_2$ a.  $\Psi; \Delta; \Gamma \vdash v : U'$ b.  $\Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U}$ c. d.  $\mathbf{x} \notin fv(\Gamma)$  $\mathbf{x} \notin fv(\Delta)$ e. then:

 $\Delta$ ,  $[v/x]\Delta'; \Gamma$ ,  $[v/x]\Gamma' \vdash [v/x]b_1 \preceq [v/x]b_2$ 

Proof is by structural induction on the derivation of  $\Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{b}_1 \leq \mathbf{b}_2$ 

Lemma 35. (Corrollary to lemma 34)

If:  $\Delta, \Delta'; \Gamma, \mathbf{x} : \mathbf{U}, \Gamma' \vdash \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} \preceq \overline{\mathbf{o} \rightarrow [\mathbf{b}'_l \ \mathbf{b}'_u]}$ a.  $\Psi; \Delta; \Gamma \vdash v : U'$ b.  $\Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U}$ c.  $\mathbf{x} \not\in fv(\Gamma)$ d.  $\mathbf{x} \notin fv(\Delta)$ e. then:  $\Delta, \llbracket \mathbf{v}/\mathbf{x} \rrbracket \Delta'; \Gamma, \llbracket \mathbf{v}/\mathbf{x} \rrbracket \Gamma' \vdash \overline{\llbracket \mathbf{v}/\mathbf{x} \rrbracket (\mathbf{o} \rightarrow \llbracket \mathbf{b}_l \ \mathbf{b}_u \rrbracket)} \preceq \overline{\llbracket \mathbf{v}/\mathbf{x} \rrbracket (\mathbf{o} \rightarrow \llbracket \mathbf{b}_l' \ \mathbf{b}_u' \rrbracket)}$ 

Proof is by deduction

Lemma 36. (Value substitution preserves subtyping)

 $\begin{array}{ll} \textit{If:} & \mathbf{a.} & \Delta, \Delta'; \Gamma, \mathbf{x} \colon \mathbf{U}, \Gamma' \vdash \mathbf{T}_1 <: \mathbf{T}_2 \\ & \mathbf{b.} & \Psi; \Delta; \Gamma \vdash \mathbf{v} : \mathbf{U}' \\ & \mathbf{c.} & \Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U} \\ & \mathbf{d.} & \mathbf{x} \not\in fv(\Gamma) \\ & \mathbf{e.} & \mathbf{x} \not\in fv(\Delta) \\ \textit{then:} \\ & \Delta, [\mathbf{v}/\mathbf{x}] \Delta'; \Gamma, [\mathbf{v}/\mathbf{x}] \Gamma' \vdash [\mathbf{v}/\mathbf{x}] \mathbf{T}_1 <: [\mathbf{v}/\mathbf{x}] \mathbf{T}_2 \end{array}$ 

Proof is by case analysis on the last step of the derivation of  $\Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \mathbf{T}_1 <: \mathbf{T}_2$ 

Lemma 37. (Value substitution preserves glb)

If: **a.**  $glb_{\Delta}(\mathbf{b}) = \mathbf{b}'$ then:  $[\mathbf{v}/\mathbf{x}]\mathbf{b}' = glb_{[\mathbf{v}/\mathbf{x}]\Delta}([\mathbf{v}/\mathbf{x}]\mathbf{b})$ 

Proof is by structural induction on the derivation of  $glb_{\Delta}(b)$ 

Lemma 38. (Value substitution preserves own)

If:

a.  $own_{\Psi,\Psi'}(\mathbf{T}) = \mathbf{b}$ b.  $\mathbf{x} \notin fv(\Psi)$ then:  $own_{\Psi, [\mathbf{v}/\mathbf{x}]\Psi'}([\mathbf{v}/\mathbf{x}]\mathbf{T}) = [\mathbf{v}/\mathbf{x}]\mathbf{b}$ 

Proof is by case analysis on T

Lemma 39. (Value substitution preserves well-formed types and environments)

If:  $\Psi, \Psi'; \Delta, \Delta'; \Gamma, \mathbf{x} : \mathbf{U}, \Gamma' \vdash \psi \text{ ok}$ a.  $\Psi; \Delta; \Gamma \vdash \mathtt{v} : \mathtt{U}'$ b.  $\Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U}$ c. d.  $\mathbf{x} \notin fv(\Gamma)$  $\mathbf{x} \not\in fv(\Delta)$ e.  $\mathbf{x} \notin fv(\Psi)$ f. where:  $\psi ::= \Delta'' \mid \mathsf{T}$ g. then: Ψ, [v/x]Ψ'; Δ, [v/x]Δ'; Γ,  $[v/x]Γ' \vdash [v/x]ψ$  ok

Proof is by structural induction on the derivation of  $\Psi, \Psi'; \Delta, \Delta'; \Gamma, \mathbf{x}: \mathbf{U}, \Gamma' \vdash \psi$  ok

Lemma 40. (Substitution of values preserves field lookup)

```
 \begin{array}{l} \textit{If:} \\ \textbf{a.} \quad fType(\texttt{f}_i, \gamma, \texttt{C} < \overline{\texttt{a}}, \ \overline{\texttt{T}} >) = \texttt{T} \\ \textit{then:} \\ \quad fType(\texttt{f}_i, \texttt{[v/x]}\gamma, \texttt{[v/x]}\texttt{C} < \overline{\texttt{a}}, \ \overline{\texttt{T}} >) = \texttt{[v/x]}\texttt{T} \end{array}
```

Proof is by deduction

## Lemma 41. (Substitution of values preserves method lookup)

If: a.  $mType_{\Delta,\Delta';\Gamma,\mathbf{x}:\mathbf{U},\Gamma'}(\mathbf{m}<\overline{\mathbf{a}'}, \overline{\mathbf{U}'}>, \gamma, \mathbb{C}<\overline{\mathbf{a}}, \overline{\mathbf{U}}>) = \overline{\mathbf{T}} \rightarrow \mathbb{T}$ b.  $\Psi; \Delta; \Gamma \vdash \mathbf{v}: \mathbf{U}'$ c.  $\Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U}$ d.  $\Psi; \Delta \vdash \Gamma \text{ OK}$ e.  $\mathbf{x} \notin fv(\Delta)$ f.  $\mathbf{x} \notin fv(\Psi)$ then:

 $mType_{\Delta,[v/x]\Delta';\Gamma,[v/x]\Gamma'}([v/x]m<\overline{a'}, \overline{U'}>, [v/x]\gamma, [v/x]C<\overline{a}, \overline{U}>) = [v/x](\overline{T}\rightarrow T)$ 

Proof is by deduction

# Lemma 42. (Type Checking values gives well-formed types)

If: **a.**  $\Psi; \Delta; \Gamma \vdash v : T$  **b.**  $\Psi; \Delta \vdash \Gamma$  ok then:  $\Psi; \Delta; \Gamma \vdash T$  ok

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash v : T$ 

Lemma 43. (Substitution of values preserves typing)

 $\begin{array}{ll} \textit{If:} & & \\ \mathbf{a.} & \Psi, \Psi'; \Delta, \Delta'; \Gamma, \mathbf{x} \colon \mathbf{U}, \Gamma' \vdash \mathbf{e} : \mathbf{T} \\ & \\ \mathbf{b.} & \Psi; \Delta; \Gamma \vdash \mathbf{v} : \mathbf{U}' \\ & \\ \mathbf{c.} & \Delta; \Gamma \vdash \mathbf{U}' <: \mathbf{U} \\ & \\ \mathbf{d.} & \Psi; \Delta \vdash \Gamma \text{ OK} \\ & \\ \mathbf{e.} & \mathbf{x} \not\in fv(\Delta) \\ & \\ & \\ \mathbf{f.} & \mathbf{x} \not\in fv(\Psi) \\ & \\ \textit{then:} \\ & \\ \Psi, [\mathbf{v}/\mathbf{x}] \Psi'; \Delta, [\mathbf{v}/\mathbf{x}] \Delta'; \Gamma, [\mathbf{v}/\mathbf{x}] \Gamma' \vdash [\mathbf{v}/\mathbf{x}] \mathbf{e} : [\mathbf{v}/\mathbf{x}] \mathbf{T} \end{array}$ 

 $\textit{Proof is by structural induction on the derivation of } \Psi, \Psi'; \Delta, \Delta'; \Gamma, x: U, \Gamma' \vdash e: T$ 

Lemma 44. (The inside relation gives well-formed owners)

If: a.  $\Delta; \Gamma \vdash \mathbf{b} \leq \mathbf{b}'$ b.  $\emptyset; \Gamma \vdash \Delta \text{ OK}$ c.  $\Psi; \Delta; \emptyset \vdash \Gamma \text{ OK}$ then: d.  $\Delta; \Gamma \vdash \mathbf{b}, \mathbf{b}' \text{ OK}$ or: e.  $\mathbf{b} = \mathbf{b}'$ 

Proof is by structural induction on the derivation of  $\Delta$ ;  $\Gamma \vdash \mathbf{b} \preceq \mathbf{b}'$  with a case analysis on the last step:

Case 1. (I-REFLEX, I-BOTTOM, I-WORLD)

trivial

Case 2. (I-TRANS)

1.	$\Delta; \Gamma \vdash b \preceq b''$	
2.	$\Delta; \Gamma \vdash {\tt b}'' \preceq {\tt b}'$	} 02
3.	$\Delta; \Gamma \vdash  blue,  blue''$ ОК	
4.	$or \mathbf{b} = \mathbf{b}''$	} <i>0</i> į
5.	$\Delta; \Gamma \vdash b'', b'$ ok	
6.	$or \mathbf{b}'' = \mathbf{b}'$	$\int -\partial t$
7.	$done - \mathbf{d}$	<i>by</i> <b>3</b> ,
8.	$done - \mathbf{d}$	<i>by</i> <b>4</b> ,
9.	$done - \mathbf{d}$	<i>by</i> <b>3</b> ,
10.	$done - \mathbf{e}$	<i>by</i> <b>4</b> ,

Case 3. (I-BOUND (UPPER BOUND CASE))

1. 
$$\Delta(o) = [b_l \ b_u]$$
  
2. 
$$b = o$$
  
3. 
$$b' = b_u$$
  
4. 
$$\Delta; \Gamma \vdash b \text{ OK}$$
  
5. 
$$\Delta; \Gamma \vdash b' \text{ OK}$$

Case 4. (I-Bound (lower bound case))

1. 
$$\Delta(o) = [b_l \ b_u]$$
  
2.  $b' = o$   
3.  $b = b_l$   
4.  $\Delta; \Gamma \vdash b' \text{ OK}$   
5.  $\Delta; \Gamma \vdash b \text{ OK}$ 

Case 5. (I-OWNER)

 $\begin{array}{ll} \mathbf{1.} & \Gamma(\gamma) = \mathbb{C} < \overline{\mathtt{a}} \,, \ \overline{\mathtt{T}} > \\ \mathbf{2.} & \mathtt{b} = \gamma \\ \mathbf{3.} & \mathtt{b}' = \mathtt{a}_0 \end{array}$ 

by premises I-TRANS
 by 1, b, c, ind hyp
 by 2, b, c, ind hyp
 by 3, 5
 by 4, 5
 by 3, 6
 by 4, 6

by premise I-BOUND by def I-BOUND by 2, 1, F-OWNER by 3, 1, b, def F-ENV

by premise I-BOUND by def I-BOUND by 2, 1, F-OWNER by 3, 1, b, def F-ENV

by premise I-Owner by def I-Owner

4.	$\Delta; \Gamma \vdash b$ ok	<i>by</i> <b>2</b> , <b>1</b> , F-VAR
5.	$\Delta; \Gamma \vdash \mathbf{b}' \text{ ok}$	<i>by</i> <b>3</b> , <b>1</b> , <b>c</b> , <i>def</i> F-GAMMA

Lemma 45. (fType gives well-formed types at runtime)

If: a.  $fType(f_i, \iota, C<\overline{a}, \overline{T}>) = T$ b.  $\emptyset; \Delta; \mathcal{H} \vdash C<\overline{a}, \overline{T}> \text{ OK}$ c.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : C<\overline{a}, \overline{T}>$ d.  $\emptyset; \emptyset; \emptyset \vdash \Delta \text{ OK}$ e.  $\Delta \vdash \mathcal{H} \text{ OK}$ then:  $\emptyset; \Delta; \mathcal{H} \vdash T \text{ OK}$ 

Proof is by deduction

Lemma 46. (mType gives well-formed types at runtime)

If:  $mType_{\Delta;\mathcal{H}}(\mathsf{m}{<}\overline{\mathsf{a}'}\text{, }\overline{\mathsf{T}'}\text{>},\iota,\mathsf{C}{<}\overline{\mathsf{a}}\text{, }\overline{\mathsf{T}}\text{>})=\overline{\mathtt{U}}{\rightarrow}\mathtt{U}$ a.  $\emptyset; \Delta; \mathcal{H} \vdash C < \overline{a}, \overline{T} > OK$ b.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : C < \overline{a}, \overline{T} >$ c. d.  $\emptyset, \emptyset \vdash \Delta \text{ ok}$  $\Delta \vdash \mathcal{H}$ ок e.  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathsf{a}'} \text{ ok}$ f.  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathsf{T}'}$  ок g. then:  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{U}$  ок

Proof is by deduction

Lemma 47. (Runtime type Checking gives well-formed types)

If: **a.**  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{e} : \mathbf{T}$  **b.**  $\Delta \vdash \mathcal{H} \text{ OK}$  **c.**  $\emptyset; \emptyset; \vdash \Delta \text{ OK}$ then:  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{T} \text{ OK}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash e: T$ 

Lemma 48. (Well-formed types are closed)

If: **a.**  $\Psi; \Delta; \Gamma \vdash T$  ok then:  $\forall \gamma \in fv_{\gamma}(\mathbf{T}) : \gamma \in dom(\Gamma)$  $\forall \mathsf{o} \in fv_{\mathsf{o}}(\mathsf{T}) : \mathsf{o} \in dom(\Delta)$  $\forall X \in fv_X(T) : X \in dom(\Psi)$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash T$  OK

Lemma 49. (Well-typed expressions are closed)

If:  $\Psi; \Delta; \Gamma \vdash \mathsf{e} : \mathsf{T}$ a. then:  $\forall \gamma \in fv_{\gamma}(\mathbf{e}) : \gamma \in dom(\Gamma)$  $\forall \mathsf{o} \in fv_\mathsf{o}(\mathsf{e}) : \mathsf{o} \in dom(\Delta)$  $\forall X \in fv_X(e) : X \in dom(\Psi)$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash e: T$ 

Lemma 50. (A method body has the method's return type)

 $mBody(m < \overline{r}, \overline{S} >, \iota, R) = (\overline{x}; e)$ a.  $mType_{\Delta;\mathcal{H}}(\mathsf{m}{<}\overline{\mathsf{r}}, \overline{\mathsf{S}}{>}, \iota, \mathsf{R}) = \overline{\mathsf{T}}{\rightarrow}\mathsf{T}$ b.  $\Delta; \mathcal{H} \vdash \overline{r} \text{ ok}$ c.  $\emptyset; \Delta; \mathcal{H} \vdash \overline{S} \text{ ok}$ d.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbb{R}$ e. f.  $\Delta \vdash \mathcal{H}$  ок  $\emptyset; \emptyset; \mathcal{H} \vdash \Delta$  ok g. then:  $\emptyset; \Delta; \mathcal{H}, \overline{\mathbf{x}: \mathbf{T}} \vdash \mathbf{e} : \mathbf{T}$ 

Proof is by deduction

If:

let  $R = C < \overline{r'}, \overline{S'} >$ 1. class C< $\overline{o' \to [b'_l \ b'_u]}$ ,  $\overline{X}$ > { $\overline{U \ f;} \ \overline{W}$ } 2. by  $\mathbf{a}$ , premises mBody  $\langle \overline{o'' 
ightarrow [b_l'' \ b_u'']}, \ \overline{X'} 
ightarrow U \ m(\overline{U'} \ x) \ \{ ext{return } e_o; \} \in \overline{W}$ 3.  $\mathbf{e} = [\overline{\mathbf{r}'/\mathbf{o}'}, \ \overline{\mathbf{r}/\mathbf{o}''}, \ \overline{\mathbf{S}'/\mathbf{X}}, \ \overline{\mathbf{S}/\mathbf{X}'}, \ \iota/\mathtt{this}]\mathbf{e}_0$  $by \; \mathbf{a}, \; def \; mBody$ **4**.  $\overline{T} = [\overline{r'/o'}, \ \overline{r/o''}, \ \overline{S'/X}, \ \overline{S/X'}, \ \iota/this]\overline{U'}$ 5. by **b**, def mType $\mathtt{T}=[\overline{\mathtt{r}'/\mathtt{o}'},\ \overline{\mathtt{r}/\mathtt{o}''},\ \overline{\mathtt{S}'/\mathtt{X}},\ \overline{\mathtt{S}/\mathtt{X}'},\ \iota/\mathtt{this}]\mathtt{U}$ 6.  $\Delta; \mathcal{H}, \texttt{this:C<}\overline{\mathbf{r'}}, \ \overline{\mathtt{X}} \succ \vdash \overline{\mathbf{r} \preceq [\overline{\mathbf{r'}/\mathbf{o'}}, \ \overline{\mathbf{r}/\mathbf{o''}}]\mathtt{b}_u''}$ 7. by  $\mathbf{b}$ , premises mType  $\Delta; \mathcal{H}, \texttt{this:C} < \overline{\mathbf{r}'}, \ \overline{\mathbf{X}} > \vdash \overline{[\overline{\mathbf{r}'/\mathbf{o}'}, \ \overline{\mathbf{r}/\mathbf{o}''}]\mathbf{b}_l'' \preceq \mathbf{r}}$ 8. let  $\mathbf{a}_l = \mathbf{o}_0'$  or  $|\perp|$ 9.  $\overline{\mathrm{X}\!\rightarrow\![\mathrm{a}_l~\bigcirc]}, \overline{\mathrm{X}'\rightarrow\![\bot~\bigcirc]}; \overline{\mathrm{o}'\rightarrow\![\mathrm{b}_l'~\mathrm{b}_u']}, \overline{\mathrm{o}''\rightarrow\![\mathrm{b}_l''~\mathrm{b}_u'']}; \texttt{this:C<}\overline{\mathrm{o}'}, \ \overline{\mathrm{X}}\!\!>\!\!, \overline{\mathrm{x}\!:\!\mathrm{U}'}\vdash\mathrm{e}_0:\mathrm{U}$ 10. by 2, 3, def T-CLASS, def T-METHOD 11. by **10**, lemma 10 12.  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{R} \text{ ok}$ by  $\mathbf{e}, \mathbf{f}, \mathbf{g}$ , lemma 47

13. 14. 15. 16.	$\begin{array}{l} \Delta; \mathcal{H} \vdash \overline{\mathbf{r}'} \text{ OK} \\ \emptyset; \Delta; \mathcal{H} \vdash \overline{\mathbf{S}'} \text{ OK} \\ \Delta; \mathcal{H}, \texttt{this:} \mathbb{C} < \overline{\mathbf{r}'}, \ \overline{\mathbf{X}} > \vdash \overline{[\mathbf{r}'/\mathbf{o}'] \mathbf{b}'_l \preceq \mathbf{r}'} \\ \Delta; \mathcal{H}, \texttt{this:} \mathbb{C} < \overline{\mathbf{r}'}, \ \overline{\mathbf{X}} > \vdash \overline{\mathbf{r}' \preceq [\mathbf{r}'/\mathbf{o}'] \mathbf{b}'_u} \end{array}$	by <b>12</b> , <b>1</b> , def F-Class
17.	$\forall \mathbf{S}'_i \in \overline{\mathbf{S}'} : \emptyset; \Delta; \mathcal{H} \vdash \mathbf{r}'_0 \preceq own_{\emptyset}(\mathbf{S}'_i)$	
18.	$\overline{\mathbf{X} \rightarrow [[\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}] \mathbf{a}_l \bigcirc]}, \overline{\mathbf{X}' \rightarrow [\bot \bigcirc]}; \Delta; \mathcal{H}, t:$ $[\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}] \mathbf{e}_0: [\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}] U$	his:C $\overline{\mathbf{r}'}$ , $\overline{\mathbf{X}}$ >, $\overline{\mathbf{x}}$ : $[\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}]$ U'
		by 11, c, 13, 7, 8, 1, 15, 16, f, g, lemma 30
19.	$\emptyset; \Delta; \mathcal{H} \vdash \overline{[\mathbf{r}'/\mathbf{o}', \mathbf{r}/\mathbf{o}'']} \mathbf{a}_l \preceq own_{\emptyset}(\mathbf{S}')$	by 9, 17 or $I - Bottom$
20.	$\emptyset; \Delta; \mathcal{H}, \texttt{this:R}, \overline{\mathbf{x}: [\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}, \overline{\mathbf{S}'/\mathbf{X}}, \overline{\mathbf{S}/\mathbf{X}'}] \mathbf{U}$	· · · · · · · · · · · · · · · · · · ·
	$[\overline{\mathbf{r}'/\mathbf{o}'}, \ \overline{\mathbf{r}/\mathbf{o}''}, \ \overline{\mathbf{S}'/\mathbf{X}}, \ \overline{\mathbf{S}/\mathbf{X}'}]\mathbf{e}_0: [\overline{\mathbf{r}'/\mathbf{o}'}, \ \overline{\mathbf{r}/\mathbf{o}''}]$	, $\overline{S'/X}$ , $\overline{S/X'}$ ]U
		by 18, d, 14, f, 19, I-BOTTOM, I-WORLD, lemma 20
21.	$\emptyset; \Delta; \mathcal{H}, \overline{\mathbf{x}: [\overline{\mathbf{r}'/\mathbf{o}'}, \overline{\mathbf{r}/\mathbf{o}''}, \overline{\mathbf{S}'/\mathbf{X}}, \overline{\mathbf{S}/\mathbf{X}'}, \iota/\text{this}]}$	$\overline{\mathbf{U}'} \vdash$
	$[\overline{\mathbf{r}'/\mathbf{o}'}, \ \overline{\mathbf{r}/\mathbf{o}''}, \ \overline{\mathbf{S}'/\mathbf{X}}, \ \overline{\mathbf{S}/\mathbf{X}'}, \ \iota/\mathtt{this}]\mathbf{e}_0: [\overline{\mathbf{S}'/\mathbf{X}}]$	$\overline{X}$ , $\overline{S/X'}$ , $\overline{r'/o'}$ , $\overline{r/o''}$ , $\iota/\text{this}]U$
		by <b>20</b> , <b>e</b> , <b>1</b> , lemma 1, <b>f</b> , <b>g</b> , lemma 43
22.	$\emptyset; \Delta; \mathcal{H}, \overline{\mathrm{x:T}} \vdash e: T$	<i>by</i> <b>21</b> , <b>5</b> , <b>4</b> , <b>6</b>

Lemma 51. (Reduction preserves heap judgements)

If: **a.**  $\mathbf{e}; \mathcal{H} \sim \mathbf{e}'; \mathcal{H}'$  **b.**  $\dots \mathcal{H} \dots \vdash \dots$  **c.**  $\mathbf{e}' \neq \mathbf{err}$ then:  $\dots \mathcal{H}' \dots \vdash \dots$ 

Proof is by structural induction on the derivation of  $\mathbf{e}; \mathcal{H} \rightsquigarrow \mathbf{e}'; \mathcal{H}'$ 

Lemma 52. (Inversion lemma, object creation)

 $\begin{array}{l} \textit{If:} \\ \textbf{a.} \quad \Psi; \Delta; \Gamma \vdash \texttt{new } \mathbb{C} < \overline{\texttt{a}}, \ \overline{\texttt{U}} > : \texttt{T} \\ \textit{then:} \\ \Delta; \Gamma \vdash \mathbb{C} < \overline{\texttt{a}}, \ \overline{\texttt{U}} > <: \texttt{T} \\ \Psi; \Delta; \Gamma \vdash \mathbb{C} < \overline{\texttt{a}}, \ \overline{\texttt{U}} > \text{OK} \end{array}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \text{new C<a}, \overline{U} > : T$ 

Lemma 53. (Inversion lemma, field access)

 $\begin{array}{l} \textit{If:} \\ \textbf{a.} \quad \Psi; \Delta; \Gamma \vdash \gamma \,.\, \textbf{f} : \textbf{T} \\ \textit{then:} \\ \Psi; \Delta; \Gamma \vdash \gamma : \textbf{N} \\ \Delta; \Gamma \vdash fType(\textbf{f}, \gamma, \textbf{N}) <: \textbf{T} \end{array}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \gamma.f: T$ 

Lemma 54. (Inversion lemma, field assignment)

 $\begin{array}{l} \textit{If:} \\ \textbf{a.} \quad \Psi; \Delta; \Gamma \vdash \gamma \, . \, \textbf{f} \; = \; \textbf{e} : \textbf{T} \\ \textit{then:} \\ \Psi; \Delta; \Gamma \vdash \gamma : \textbf{N} \\ \textit{fType}(\textbf{f}, \gamma, \textbf{N}) = \textbf{U} \\ \Psi; \Delta; \Gamma \vdash \textbf{e} : \textbf{U} \\ \Delta; \Gamma \vdash \textbf{U} <: \textbf{T} \end{array}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \gamma.f = e: T$ 

### Lemma 55. (Inversion lemma, method invocation)

```
If:

a. \Psi; \Delta; \Gamma \vdash \gamma . \langle \overline{a}, \overline{U} \rangle m(\overline{e}) : T

then:

\Psi; \Delta; \Gamma \vdash \gamma : \mathbb{N}

mType_{\Delta;\Gamma}(m\langle \overline{a}, \overline{U} \rangle, \gamma, \mathbb{N}) = \overline{T} \rightarrow T'

\Psi; \Delta; \Gamma \vdash \overline{e} : \overline{T}

\Delta; \Gamma \vdash \overline{a} \text{ OK}

\Psi; \Delta; \Gamma \vdash \overline{U} \text{ OK}

\Delta; \Gamma \vdash T' <: T
```

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \gamma. \langle \overline{a}, \overline{U} \rangle m(\overline{e}) : T$ 

Lemma 56. (Inversion lemma, unpacking)

If: **a.**  $\Psi; \Delta; \Gamma \vdash \text{open } \mathbf{e}_1 \text{ as } \mathbf{x}:\overline{\mathbf{o}} \text{ in } \mathbf{e}_2: \mathbf{T}$ then:  $\Psi; \Delta; \underline{\Gamma} \vdash \mathbf{e}_1 : \exists \overline{\mathbf{o}} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u] . \mathbb{N}$   $\Psi; \Delta, \overline{\mathbf{o}} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]; \Gamma, \mathbf{x}: \mathbb{N} \vdash \mathbf{e}_2: \mathbb{U}$   $\Psi; \Delta; \Gamma \vdash \mathbb{U} \text{ OK}$  $\Delta; \Gamma \vdash \mathbb{U} <: \mathbb{T}$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \text{open } e_1 \text{ as } x: \overline{o} \text{ in } e_2: T$ 

### Lemma 57. (Inversion lemma, packing)

If: **a.**  $\Psi; \Delta; \Gamma \vdash \text{close e with } \overline{o \rightarrow [b_l \ b_u]} \text{ hiding } \overline{a}: T$ then:  $\Delta; \Gamma \vdash \overline{[\overline{a/o}]b_l \preceq a}$   $\Delta; \Gamma \vdash \overline{a \preceq [\overline{a/o}]b_u}$  $\Delta; \Gamma \vdash \overline{a} \text{ OK}$  
$$\begin{split} \Psi; \Delta; \Gamma \vdash \mathbf{e} : \overline{[\mathbf{a}/\mathbf{o}]} \mathbb{N} \\ \Psi; \Delta; \Gamma \vdash \exists \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} . \mathbb{N} \text{ OK} \\ \Delta; \Gamma \vdash \exists \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} . \mathbb{N} <: \mathbb{T} \end{split}$$

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash \text{close e with } \overline{o \rightarrow [b_l \ b_u]}$  hiding  $\overline{r}$ : T

Lemma 58. (Well-typed values have addresses in the domain of the variable environment)

If: **a.**  $\Psi; \Delta; \Gamma \vdash \mathbf{v} : \mathbf{T}$  **b.**  $add(\mathbf{v})$  defined then:  $add(\mathbf{v}) \in dom(\Gamma)$ 

Proof is by structural induction on the derivation of  $\Psi; \Delta; \Gamma \vdash v : T$ 

# Lemma 59. (Canonical Forms)

If: **a.**  $\emptyset; \emptyset; \mathcal{H} \vdash \mathbf{v} : \exists \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} . \mathbb{N}$  **b.**  $\emptyset \vdash \mathcal{H} \text{ OK}$ then:  $\mathbf{v} = \text{close } \mathbf{v}' \text{ with } \overline{\mathbf{o} \rightarrow [\mathbf{b}'_l \ \mathbf{b}'_u]} \text{ hiding } \overline{\mathbf{r}}$ 

Proof is by structural induction on the derivation of  $\emptyset; \emptyset; \mathcal{H} \vdash v : \exists \overline{o \rightarrow [b_l \ b_u]}$ .N

Lemma 60. (Reduction preserves owners)

If: **a.**  $\mathbf{e}; \mathcal{H} \rightsquigarrow \mathbf{e}'; \mathcal{H}'$  **b.**  $own_{\mathcal{H}}(v) = \iota$  **c.**  $\mathbf{e}' \neq \mathbf{err}$ then:  $own_{\mathcal{H}'}(v) = \iota$ 

Proof is by structural induction on the derivation of  $e; \mathcal{H} \rightarrow e'; \mathcal{H}'$  with a case analysis on the last step:

*N/A by* **c Case 2.** (R-NEW) 1.  $\mathcal{H}' = \mathcal{H}, \iota \to ...$ 2.  $v \neq \iota$ 

**2.** $v \neq l$ **3.**done

Case 3. (R-FIELD, R-INVK, R-OPEN-CLOSE)

trivial, since  $\mathcal{H}' = \mathcal{H}$ 

Case 4. (R-Assign)

1.  $\mathbf{e} = \iota \cdot \mathbf{f}_1 = \mathbf{v}$ 2.  $\mathcal{H}(\iota) = \{ \mathsf{C} < \overline{\mathbf{a}}, \ \overline{\mathsf{T}} >; \ \overline{\mathbf{f} \to \mathbf{v}} \}$ 3.  $\mathcal{H}' = \mathcal{H}[\iota \mapsto \{ \mathsf{C} < \overline{\mathbf{a}}, \ \overline{\mathsf{T}} >; \ \overline{\mathbf{f} \to \mathbf{v}} [\mathbf{f}_i \mapsto \mathbf{v}] \} ]$  by def R-Assign
by premises R-Assign

by premise R-New

by  $\mathbf{b}$ , def own

by  $\mathbf{2}, \mathbf{1}, def own$ 

```
Case analysis on v:
```

Case 1.  $v \neq \iota$ 

easy, since the rest of the heap is unchanged, def own

Case 2.  $v = \iota$ 

2.1.	$own_{\mathcal{H}}(\mathtt{v}) = \mathtt{a}_0$	by $2$ , def own
2.2.	$own_{\mathcal{H}'}(\mathtt{v}) = \mathtt{a}_0$	by $3$ , def own
2.3.	done	by <b>2.2</b> , <b>2.3</b>

Case 5. (RC-Assign, RC-Invk, RC-Open, RC-Close)

easy, by ind hyp

Lemma 61. (An object is inside the owner of its fields)

If:

a.  $fType(\mathbf{f}_{i}, \iota, \mathbf{R}) = \mathbf{T}$ b.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbf{R}$ c.  $\Delta \vdash \mathcal{H} \text{ OK}$ d.  $\emptyset; \emptyset; \mathcal{H} \vdash \Delta \text{ OK}$ then:  $\Delta; \mathcal{H} \vdash \iota \preceq own(\mathbf{T})$ 

Proof is by deduction

1. class 
$$C < \overline{o} \rightarrow [b_l \ b_u]$$
,  $\overline{X} > \{\overline{U} \ f; \overline{W}\}$  by premise of  $fType$   
2.  $R = C < \overline{r}, \overline{T'} >$   
3.  $T = [\overline{r/o}, \overline{T'/X}, \iota/this]U_i$   $\}$  by def  $fType$   
4.  $\overline{X \rightarrow [o_0 \bigcirc]; \overline{o} \rightarrow [b_l \ b_u]}; this: C < \overline{o}, \overline{X} > \vdash U_i \ OK$   $\}$  by 1, wf prog, T-CLASS

6.  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{R}$  ok by  $\mathbf{b}$ ,  $\mathbf{c}$ ,  $\mathbf{d}$ , lemma 47  $\emptyset'; \Delta; \mathcal{H} \vdash \overline{\mathsf{T}'}$  ok 7. by 6, def F-CLASS  $\forall \mathbf{r}_i \in \overline{\mathbf{r}} : \Delta; \mathcal{H} \vdash \mathbf{r}_0 \preceq \mathbf{r}_i$ 8.  $\forall \mathsf{T}'_i \in \overline{\mathsf{T}'} : \Delta; \mathcal{H} \vdash \mathsf{r}_0 \preceq own(\mathsf{T}'_i)$ 9. Case analysis on  $U_i$ : Case 1.  $U_i = X_i$  $T = [\iota/this]T'_i$  $by \mathbf{3}$ 1.1. 1.2.  $T = T'_i$ by 1.1, 7 1.3.  $\Delta; \mathcal{H} \vdash \mathbf{r}_0 \preceq own(\mathbf{T})$ by 9, 1.2  $\Delta; \mathcal{H} \vdash \iota \preceq \mathbf{r}_0$ by b, 2, lemma 12, I-OWNER 1.4. 1.5. done *by* **1.4**, **1.3**, I-TRANS Case 2.  $U_i = D < \overline{a^T}, \overline{T^T} >$  $T = [\overline{r/o}, \overline{T'/X}, \iota/this]D < \overline{a^T}, \overline{T^T} >$ 2.1.  $by \mathbf{3}$  $own(\mathbf{T}) = [\overline{\mathbf{r}/\mathbf{o}}, \iota/\mathtt{this}]\mathbf{a}_0^T$ 2.2. by  $\mathbf{2.1}$ , def own by 4, def wf owners, syntax a 2.3.  $a_0^T \in \overline{o}, \texttt{this}, \bigcirc$  $own(\mathtt{T}) \in \overline{\mathtt{r}}, \iota, \bigcirc$ 2.4.by 2.2, 2.3 if  $own(T) = \iota \text{ or } \bigcirc$ , then done trivially 2.5. 2.6. wlog assume  $own(T) = r_i$ by 2.4, 2.5 2.7.  $\Delta; \mathcal{H} \vdash \mathbf{r}_0 \preceq own(\mathbf{T})$ by 2.6, 8 by b, 2, lemma 12, def own, I-OWNER 2.8.  $\Delta; \mathcal{H} \vdash \iota \preceq \mathbf{r}_0$ 2.9. doneby 2.7, 2.8, I-TRANS Case 3.  $U_i = \exists \Delta^T . D < \overline{a^T}, \overline{T^T} >$  $T = [\overline{r/o}, \overline{T'/X}, \iota/this] \exists \Delta^T.D < \overline{a^T, \overline{T^T}} >$ 3.1.  $by \mathbf{3}$ if  $[\overline{r/o}, \iota/\text{this}]a_0^T \notin dom(\Delta)$ ,  $own(T) = [\overline{r/o}, \iota/\text{this}]a_0^T$  and we follow the above case 3.2. by **3.1**, def own  $otherwise_{-}[\overline{r/o}, \iota/this]a_{0}^{T} = o$ 3.3. by def own where  $\Delta^T(\mathbf{o}) = [\mathbf{b}_l \ \mathbf{b}_u]$ 3.4. by premise own  $\underline{own(T) = glb_{[\overline{r/o}, \iota/this]\Delta^T}([\overline{r/o}, \iota/this]b_l)}$ 3.5. by 3.3, 3.4, def own  $\overline{\mathbf{X} \to [\mathbf{o}_0 \ \bigcirc]}; \overline{\mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u]}; \texttt{this:C<\overline{o}}, \overline{\mathbf{X}} \vdash \Delta_T \text{ orby } 4, def \text{ F-Exists}$ 3.6. 3.7.  $\overline{X \to [o_0 \bigcirc]}; \overline{o \to [b_l \ b_u]}; \text{this:} C < \overline{o}, \overline{X} > \vdash b_l \text{ OK } by \textbf{3.6}, \textbf{3.4}, def \text{ F-Env}$  $\mathbf{b}_l \in \overline{\mathbf{o}}, \mathtt{this}, \bigcirc, dom(\Delta^T)$ 3.8. by **3.7**, def wf owners, **5** 3.9. let  $b = [\overline{r/o}, \iota/this]b_l$ **3.10.**  $\mathbf{b} \in \overline{\mathbf{r}}, \iota, \bigcirc, dom(\Delta^T)$ by 3.8, 3.9 **3.11.**  $own(T) \in \overline{r}, \iota, \bigcirc$ by 3.5, 3.10, def glb **3.12.** if  $own(T) = \iota$  or  $\bigcirc$ , then done trivially by 3.11, 3.12 **3.13.** wlog assume  $own(T) = r_i$ **3.14.**  $\Delta$ ;  $\mathcal{H} \vdash \mathbf{r}_0 \preceq own(\mathbf{T})$ *by* **3.13**, **8 3.15.**  $\Delta$ ;  $\mathcal{H} \vdash \iota \preceq \mathbf{r}_0$ by b, 2, lemma 12, def own, I-OWNER **3.16.** done by 3.14, 3.15, I-TRANS

Lemma 62. (The *glb* function preserves the inside relation)

If:

**a.**  $\Delta; \Gamma \vdash \Delta'' \preceq \Delta'$  **b.**  $\emptyset; \Gamma \vdash \Delta \text{ oK}$  **c.**  $\Psi; \Delta; \emptyset \vdash \Gamma \text{ oK}$ then:  $\Delta; \Gamma \vdash glb_{\Delta'}(\mathbf{b}) \preceq glb_{\Delta''}(\mathbf{b})$ 

Proof is by case analysis on b

Case 1. b  $\notin dom(\Delta')$ 

1.	$dom(\Delta'') = dom(\Delta')$	$by \mathbf{a}, def \text{ I-Env}$
2.	$\mathtt{b}\not\in dom(\Delta'')$	by 1, def case
3.	$glb_{\Delta'}(\mathtt{b}) = \mathtt{b}$	by def case, def glb
4.	$glb_{\Delta^{\prime\prime}}({ t b})={ t b}$	$by \ 2, \ def \ glb$
5.	done	by I-REFLEX, 3, 4

Case 2.  $b \in dom(\Delta')$ 

1.	$dom(\Delta'') = dom(\Delta')$	$by \mathbf{a}, def \text{I-Env}$
2.	$\mathtt{b} \in dom(\Delta'')$	by 1, def case
3.	$let \ \Delta'(b) = [b_l \ b_u]$	by def case
4.	$let \ \Delta''(\mathbf{b}) = [\mathbf{b}'_l \ \mathbf{b}'_u]$	$by \ 2$
5.	$glb_{\Delta'}({ t b})=glb_{\Delta'}({ t b}_l)$	$by 3, \ def \ glb$
6.	$glb_{\Delta^{\prime\prime}}(\mathtt{b})=glb_{\Delta^{\prime\prime}}(\mathtt{b}_l')$	by 4, def glb
7.	$\Delta; \Gamma \vdash b_l \preceq b'_l$	$by \mathbf{a}, def \text{I-Env}$
8.	$\Psi;\Delta;\Gamma\vdash b_l,b_l'$ ОК	hav 7 b a lamma //
9.	$or \ \mathbf{b}_l = \mathbf{b}'_l$	$\int \partial y 1, \mathbf{b}, \mathbf{c}, iemma 44$
10.	$if \ case \ 9, \ then \ done \ by \ ind \ hyp, \ otherwise:$	
11.	$glb_{\Delta'}(\mathtt{b})=\mathtt{b}_l$	by 5, 8, $def$ $glb$
12.	$glb_{\Delta^{\prime\prime}}({ t b})={ t b}_l'$	by <b>6</b> , <b>8</b> , def glb
13.	done	<i>by</i> <b>7</b> , <b>11</b> , <b>12</b>

Lemma 63. (The owner of a subtype is outside the owner of the supertype)

If:

a.  $\Delta; \Gamma \vdash T <: T'$ b.  $\emptyset; \Gamma \vdash \Delta \text{ OK}$ c.  $\Psi; \Delta; \emptyset \vdash \Gamma \text{ OK}$ then:  $\Delta; \Gamma \vdash own_{\Psi}(T') \preceq own_{\Psi}(T)$ 

Proof is by case analysis on the last step of the derivation of  $\Delta; \Gamma \vdash T' <: T$ 

Case 1. (S-REFLEX)

trivial

Case 2. (S-FULL)

1.	$\mathtt{T}=\exists\Delta'.\mathtt{C}<\mathtt{\overline{a}},\ \mathtt{\overline{T}}>$	)	ha dof C Erri
2.	$\mathtt{T}'=\exists\Delta''.\mathtt{C}<\overline{\mathtt{a}},\ \overline{\mathtt{T}}>$	Ì	by all S-FULL

- **3.**  $\Delta; \Gamma \vdash \Delta' \preceq \Delta''$
- 4.  $own_{\Psi}(\mathtt{T}) = glb_{\Delta'}(\mathtt{a}_0)$
- 5.  $own_{\Psi}(\mathbf{T}') = glb_{\Delta''}(\mathbf{a}_0)$
- **6.**  $\Delta; \Gamma \vdash own_{\Psi}(\mathsf{T}') \preceq own_{\Psi}(\mathsf{T})$

by premise S-FULL by 1, def own by 2, def own by 3, 4, 5, b, c, lemma 62

Lemma 64. (the result of glb is inside the input)

If:  
**a.** for all **b**  
then:  

$$\Delta, \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} \vdash glb_{\overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]}}(\mathbf{b}) \preceq \mathbf{b}$$

Proof is by structural induction on the derivation of  $glb_{\overline{o} \to [b_l \ b_u]}(b)$  with a case analysis on the last step:

 $\mathbf{Case \ 1.} \ base \ case$ 

1.	$glb_{\overline{\mathbf{a}} \to \overline{[\mathbf{b}_l, \mathbf{b}_l]}}(\mathbf{b}) = \mathbf{b}$	$by \ def \ glb$
2.	done	by 1, I-Reflex

 $Case \ 2. \ inductive \ case$ 

1.	$glb_{\overline{\mathbf{o}} \to [\mathbf{b}_l, \mathbf{b}_n]}(\mathbf{b}) = glb_{\overline{\mathbf{o}} \to [\mathbf{b}_l, \mathbf{b}_n]}(\mathbf{b}_{li})$	by def alb
2.	$b = o_i$	$\int \frac{\partial g}{\partial x} \frac{\partial g}{\partial y} \frac{\partial g}{\partial y} dx$
3.	$\Delta, \overline{o \to [b_l \ b_u]} \vdash glb_{\overline{o \to [b_l \ b_u]}}(b_{li}) \preceq b_{li}$	by $1$ , ind hyp
4.	$\Delta, \overline{o \! \rightarrow \! [b_l \ b_u]} \vdash b_{li} \preceq o_i$	by I-Bound
5.	$\Delta, \overline{o \to [b_l \ b_u]} \vdash glb_{\overline{o \to [b_l \ b_u]}}(b_{li}) \preceq o_i$	<i>by</i> <b>3</b> , <b>4</b> , I-trans
6.	done	by 5, 1, 2

#### Lemma 65. ( A dynamic owner is outside the static owner )

If: **a.**  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{v} : \mathbf{T}$  **b.**  $add(\mathbf{v}) \ defined$  **c.**  $\emptyset; \mathcal{H} \vdash \Delta \ OK$  **d.**  $\Delta; \vdash \mathcal{H} \ OK$ then:  $\Delta; \mathcal{H} \vdash own_{\emptyset}(\mathbf{T}) \preceq own_{\mathcal{H}}(\mathbf{v})$ 

Proof is by structural induction on the derivation of  $\emptyset; \Delta; \mathcal{H} \vdash v : \exists \overline{o} \to [b_l \ b_u]$ . C<a,  $\overline{T}$ > with a case analysis on the last step:

Case 1. (T-SUBS)

1. 2.	$egin{array}{lll} \emptyset ; \Delta ; \mathcal{H} dash \mathtt{v} : \mathtt{T}' \ \Delta ; \mathcal{H} dash \mathtt{T}' <: \mathtt{T} \end{array}$	by premises T-Subs
3.	$\Delta; \mathcal{H} \vdash own_{\emptyset}(\mathtt{T}') \preceq own_{\mathcal{H}}(\mathtt{v})$	by $1$ , $\mathbf{b}$ , ind hyp
4.	$\Delta; \mathcal{H} \vdash own_{\emptyset}(\mathtt{T}) \preceq own_{\mathcal{H}}(\mathtt{T}')$	by <b>2</b> , <b>c</b> , <b>d</b> , lemma 63
5.	done	by 3, 4, I-TRANS

Case 2. (T-VAR)

1. 2. 3. 4. 5. 6. 7.	$ \begin{split} \mathbf{v} &= \gamma \\ \mathbf{T} &= \mathcal{H}(\gamma) \\ \mathcal{H}(\gamma) &= \mathbf{C} < \overline{\mathbf{r}} , \ \overline{\mathbf{T}} >, \dots \\ own_{\mathcal{H}}(\gamma) &= \mathbf{r}_0 \\ \mathbf{T} &= \mathbf{C} < \overline{\mathbf{r}} , \ \overline{\mathbf{T}} > \\ own_{\emptyset}(\mathbf{T}) &= \mathbf{r}_0 \\ done \end{split} $	by def T-VAR by def T-VAR, def H-T by def $\mathcal{H}$ by 3, def $own_{\mathcal{H}}$ by 2, 3 by 5, def $own$ by 4, 6, I-Reflex
Case 3. (7	$\Gamma$ -Close)	
1. 2.	$\begin{array}{l} \mathtt{v} = \mathtt{close} \ \mathtt{v}' \ \mathtt{with} \ \overline{\mathtt{o} \! \rightarrow \! \mathtt{[b}_l \ b_u]} \ \mathtt{hiding} \ \overline{\mathtt{r}'} \\ \mathtt{T} = \exists \overline{\mathtt{o} \! \rightarrow \! \mathtt{[b}_l \ b_u]}  . \mathtt{N} \end{array}$	$\bigg\} by def T-CLOSE$
3. 4. 5. 6. 7.	$\begin{array}{l} \Delta; \mathcal{H} \vdash \overline{[\overline{\mathbf{r}/\mathbf{o}}]  \mathbf{b}_l \preceq \mathbf{r}} \\ \Delta; \mathcal{H} \vdash \overline{\mathbf{r}} \preceq [\overline{\mathbf{r}/\mathbf{o}}]  \mathbf{b}_u \\ \Delta; \mathcal{H} \vdash \overline{\mathbf{r}} \ \mathrm{OK} \\ \emptyset; \Delta; \mathcal{H} \vdash \overline{\mathbf{r}} \ \mathrm{OK} \\ \end{bmatrix} \end{array}$	by premises T-CL
8. 9. 10.	$\begin{array}{l} \Delta; \mathcal{H} \vdash own_{\emptyset}(\lceil \overline{\texttt{r/o}} \rceil \texttt{N}) \preceq own_{\mathcal{H}}(\texttt{v}') \\ own_{\mathcal{H}}(\texttt{v}) = own_{\mathcal{H}}(\texttt{v}') \\ let \ \texttt{N} = \texttt{C} < \overline{\texttt{a}}, \ \overline{\texttt{T}} > \end{array}$	by $6$ , $\mathbf{b}$ , ind hyp by $1$ , def own <sub>H</sub>

Case analysis on  $a_0$ :

Case 1.  $a_0 \notin \overline{o}$ 

 $own_{\emptyset}(\mathtt{T}) = \mathtt{a}_0$ 1.1. 1.2.  $own_{\emptyset}([\overline{r/o}]N) = a_0$ 1.3.  $\Delta; \mathcal{H} \vdash \mathbf{a}_0 \preceq own_{\mathcal{H}}(\mathbf{v}')$ 1.4.  $\Delta; \mathcal{H} \vdash own(\mathbf{T}) \preceq own_{\mathcal{H}}(\mathbf{v}')$ 1.5. done

Case 2.  $\exists i : a_0 = o_i$ 

 $own_{\emptyset}(\mathbf{T}) = glb_{\overline{\mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u]}}(\mathbf{b}_{il})$ 2.1. by 2, 10, def own 2.2.  $own_{\emptyset}([\mathbf{r/o}]\mathbf{N}) = \mathbf{r}_i$ by 2.1, 10, def own  $\Delta, \mathcal{H} \vdash \mathbf{r}_i \preceq \underline{own}_{\mathcal{H}}(\mathbf{v})$ 2.3. by 8, 2.2, 9  $\Delta, \overline{\mathbf{o} \rightarrow [\mathbf{b}_{l} \quad \mathbf{b}_{u}]}; \mathcal{H} \vdash glb_{\overline{\mathbf{o} \rightarrow [\mathbf{b}_{l} \quad \mathbf{b}_{u}]}}(\mathbf{b}_{il}) \preceq \mathbf{b}_{il}$ 2.4. by lemma 64  $\begin{array}{l} \Delta; \mathcal{H} \vdash [\overline{\mathbf{r/o}}] glb_{\overline{\mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u]}}(\mathbf{b}_{il}) \stackrel{\scriptstyle }{\preceq} [\overline{\mathbf{r/o}}] \mathbf{b}_{il} \\ \overline{\mathbf{o}} \cap fv(glb_{\overline{\mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u]}}(\mathbf{b}_{il})) = \emptyset \end{array}$ 2.5. by 2.4, 5, 3, 4, c, d, lemma 22 2.6. by def glb 2.7.  $\Delta; \mathcal{H} \vdash glb_{\overline{\mathbf{o} \to [\mathbf{b}_l \ \mathbf{b}_u]}}(\mathbf{b}_{il}) \preceq [\mathbf{r/o}]\mathbf{b}_{il}$ by 2.5, 2.6  $\Delta; \mathcal{H} \vdash glb_{\overline{\mathbf{o}} \to [\underline{\mathbf{b}}_l \ \mathbf{b}_u]}(\mathbf{b}_{il}) \preceq \mathbf{r}_i$ 2.8. by 2.7, 3, I-TRANS 2.9.  $\Delta; \mathcal{H} \vdash own_{\emptyset}(\mathsf{T}) \preceq \mathsf{r}_{i}$ by 2.8, 2.1 **2.10.**  $\Delta$ ;  $\mathcal{H} \vdash own_{\emptyset}(T) \preceq own_{\mathcal{H}}(v)$ by 2.9, 2.3, I-TRANS

Case 4. (T-NULL, T-FIELD, T-ASSIGN, T-NEW, T-INVK, T-OPEN)

N/A

LOSE

by 2, def own, def glb

*by* **1.1**,**10**,  $a_0 \notin \overline{o}$ 

by 8, 1.2

by 1.4, 9

by 1.3, 1.1

#### Theorem (Subject Reduction)

If:

 $\emptyset; \Delta; \mathcal{H} \vdash \mathsf{e} : \mathsf{T}$ a.  $\mathsf{e};\mathcal{H} \leadsto \mathsf{e}';\mathcal{H}'$ b. c.  $\Delta; \mathcal{H} \vdash e \text{ ok}$  $\emptyset; \mathcal{H} \vdash \Delta$  ок d.  $\mathtt{e}' \neq \mathtt{err}$ e. then:  $\emptyset; \Delta; \mathcal{H}' \vdash e' : T$  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 

Proof is by structural induction on the derivation of  $\mathbf{e}; \mathcal{H} \sim \mathbf{e}'; \mathcal{H}'$  with a case analysis on the last step:

Case 1. (R-FIELD-NULL, R-ASSIGN-NULL, R-INVK-NULL, RC-ASSIGN-ERR, RC-INVK-ERR, RC-OPEN-ERR, RC-CLOSE-ERR)

N/A by e

Case 2. (R-FIELD)

1.  ${\tt e} = \iota . {\tt f}_i$ 2.  $e' = v_i$  $\mathcal{H}'=\mathcal{H}$ 3.  $\mathcal{H}(\iota) = \{ \mathtt{R}; \ \overline{\mathtt{f} \to \mathtt{v}} \}$ 4. 5.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbb{N}$ 6.  $fType(\mathbf{f}_i, \iota, \mathbf{N}) = \mathbf{T}'$  $\Delta; \mathcal{H} \vdash \mathsf{T}' <: \mathsf{T}$ 7. 8.  $\Delta; \mathcal{H} \vdash \mathbf{R} <: \mathbf{N}$  $\mathtt{R}=\mathtt{N}$ 9. 10.  $\Delta \vdash \mathcal{H}$  ок  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{v}_i : \mathbf{T}'$ 11. 12.  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{T}$  ok  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{v}_i : \mathbf{T}$ 13.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{e}' : \mathsf{T}$ 14. 15.  $\forall \iota \in fv(\mathbf{e}') : \iota \in dom(\mathcal{H})'$  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 16. 17. done

Case 3. (R-Assign)

1.  $e = \iota . f_i = v$ by def R-Assign e' = v2. 3.  $\mathcal{H}' = \mathcal{H}$  $\mathcal{H}(\iota) = \{\mathtt{R}; \ \overline{\mathtt{f} \rightarrow \mathtt{v}}\}$ 4.  $\mathcal{H}' = \mathcal{H}[\iota \mapsto \{\mathtt{R}; \overline{\mathtt{f} \to \mathtt{v}}[\mathtt{f}_i \mapsto \mathtt{v}]\}]$ 5.

by def R-Field by premises R-FIELD by **a**, **1**, lemma 53 by 5, lemma 32, 4, def H-T by 8, lemma 3 by  $\mathbf{c}$ , def F-Config by 10, 4, 6, 9, def F-HEAP by **a**, **d**, **10**, lemma 47 by 11, 7, 12, T-SUBS by 13, 2, 3 by 14, lemma 49 by 3, 10, 15 by 14, 16

by premises R-Assign

6.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbb{N}$ 7.  $fType(\mathbf{f}_i, \iota, \mathbf{N}) = \mathbf{T}'$ by  $\mathbf{a}, \mathbf{1}$ , lemma 54  $\emptyset; \Delta; \mathcal{H} \vdash v : T'$ 8.  $\Delta; \mathcal{H} \vdash \mathsf{T}' <: \mathsf{T}$ 9. 10.  $\Delta \vdash \mathcal{H}$  ок by c, def F-Config  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{T}$  ok by a, d, 10, lemma 47 11. 12.  $\emptyset; \Delta; \mathcal{H} \vdash v : T$ by 8, 9, 11  $\emptyset; \Delta; \mathcal{H}' \vdash v : T$ 13. by 12, b, e, lemma 51 14. if add(v) not defined, then steps**15–23** are unnecessary, therefore assume v defined 15.  $add(\mathbf{v} \in dom(\mathcal{H}'))$ by 14, 13, lemma 58 16. if add(v) not defined, goto 2417.  $\Delta; \mathcal{H} \vdash \mathbf{R} <: \mathbf{N}$ by 6, lemma 32, 4, def H-T 18.  $\mathbf{R} = \mathbf{N}$ by 17, lemma 3 19.  $\Delta, \mathcal{H} \vdash \iota \preceq own(\mathsf{T}')$ by 7, 6, 18, 10, d, lemma 61 20.  $\Delta, \mathcal{H} \vdash own(\mathbf{T}') \preceq own_{\mathcal{H}}(\mathbf{v})$ by 8, 16, d, 11, lemma 65 21.  $\Delta, \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\mathbf{v})$ by 19, 20, I-TRANS 22.  $\Delta, \mathcal{H}' \vdash \iota \preceq own_{\mathcal{H}}(\mathbf{v})$ by 21, b, lemma 51 23.  $\Delta, \mathcal{H}' \vdash \iota \preceq own_{\mathcal{H}'}(\mathbf{v})$ by 22, e, lemma 60 24.  $\overline{\Delta \vdash \mathcal{H} \text{ ok}'}$ 25.  $\forall \iota \in fv(\mathbf{e}') : \iota \in dom(\mathcal{H})'$ by **13**, lemma 49  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 26. by 24, 25 27. done by 13, 26 Case 4. (R-NEW)  $\texttt{e} = \texttt{new C} < \overline{\texttt{r}}, \ \overline{\texttt{T}} >$ 1. by def R-NEW 2.  $e' = \iota$ 3.  $\mathcal{H}(\iota)$  undefined 4.  $fields(C) = \overline{f}$ 

- 5.  $\mathcal{H}' = \mathcal{H}, \iota \to \{ C < \overline{r}, \overline{T} >; \overline{f} \rightarrow null \}$
- $\emptyset; \Delta; \mathcal{H} \vdash C < \overline{r}, \overline{T} > <: T$ 6.
- 7.  $\emptyset; \Delta; \mathcal{H} \vdash \mathsf{C} < \overline{\mathsf{r}}, \overline{\mathsf{T}} > \mathsf{OK}$
- 8.  $\emptyset; \Delta; \mathcal{H}' \vdash \iota : fst(\mathcal{H}'(\iota))$
- $\emptyset; \Delta; \mathcal{H}' \vdash \iota : C < \overline{r}, \overline{T} >$ 9.
- $\Delta; \mathcal{H}' \vdash C < \overline{r}, \overline{T} > <: T$ 10.
- 11.  $\Delta \vdash \mathcal{H}$  ок
- $\emptyset; \Delta; \mathcal{H} \vdash \mathsf{T}$  ок 12.
- 13.  $\emptyset; \Delta; \mathcal{H}' \vdash \iota : \mathsf{T}$
- 14. let  $fType(f, \iota, C < \overline{r}, \overline{T} >) = U$ 15.  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathbf{U}}$  ok  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\texttt{null} : \texttt{U}}$ 16.  $\Delta \vdash \mathcal{H}, \iota \rightarrow \{ \mathsf{C} < \overline{\mathsf{r}}, \overline{\mathsf{T}} >; \overline{\mathsf{f}} \rightarrow \mathsf{null} \}$  OK 17. 18. add(null) undefined 19.  $\Delta \vdash \mathcal{H}'$  ок 20.  $\forall \iota \in fv(\mathbf{e}') : \iota \in dom(\mathcal{H}')$  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 21. 22. done

Case 5. (R-INVK)

by 10, 5, 7, 8, 14, 15, 23 or 16, def F-Heap

by premises R-NEW by **1**, **a**, lemma 52 by T-VAR, H-T *by* **8**, **5** by 6, 5, lemma 8 by  $\mathbf{c}$ , def F-Config by a, d, 11, lemma 47 by 9, 10, 12, T-SUBS

by 14, 7, lemma 45 by 15, T-NULL by 11, 7, 14, 16, def F-HEAP by def add by 17, 5 by 2, 5 by 20, 19, F-CONFIG by 13, 21

 $e = \iota . < \overline{r}, \overline{U} > m(\overline{v})$ 1. by def R-Invk 2.  $e' = [v/x]e_0$ 3.  $\mathcal{H}' = \mathcal{H}$ **4**.  $\mathcal{H}(\iota) = \{\mathtt{R}; \ldots\}$ by premises R-INVK 5.  $mBody(m < \overline{r}, \overline{U} >, \iota, R) = (\overline{x}; e_0)$ 6.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbb{N}$ 7.  $mType_{\Delta;\mathcal{H}}(\mathsf{m}<\overline{\mathbf{r}}, \overline{\mathbf{U}}>, \iota, \mathbb{N}) = \overline{\mathbf{T}} \rightarrow \mathbf{T}'$  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathsf{e} : \mathsf{T}}$ 8. by 1, a, lemma 55  $\Delta; \mathcal{H} \vdash \overline{\mathbf{r}} \text{ ok}$ 9. 10.  $\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathbf{U}}$  ок 11.  $\Delta; \mathcal{H} \vdash \mathbf{T}' <: \mathbf{T}$ 12.  $\Delta; \mathcal{H} \vdash \mathbf{R} <: \mathbf{N}$ by 4, 6, lemma 32, def H-T 13.  $\mathtt{R}=\mathtt{N}$ *by* **12**, *lemma 3* 14.  $\Delta \vdash \mathcal{H}$  ок by  $\mathbf{c}$ , def F-Config  $\emptyset; \Delta; \mathcal{H}, \overline{\mathbf{x}: \mathbf{T}} \vdash \mathbf{e}_0 : \mathbf{T}'$ by 5, 7, 6, 13, 9, 10, d, 14, 15. lemma 50  $\emptyset; \Delta; \mathcal{H} \vdash [\overline{v/x}] e_0 : [\overline{v/x}] T$ 16. by 15, 6, lemma 1, 14, d, lemma 43  $\emptyset; \Delta; \mathcal{H} \vdash \mathsf{T}'$  ок 17. by 15, 11, lemma 47 18.  $\emptyset; \Delta; \mathcal{H} \vdash [\overline{v/x}] e_0 : T'$ by 16, 17  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{T}$  ok 19. by **a**, **d**, **14**, lemma 47 20.  $\emptyset; \Delta; \mathcal{H} \vdash [\overline{v/x}] e_0 : T$ by 18, 11, 19, T-SUBS  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{e}' : \mathtt{T}$ 21. by 20, 2, 3 22.  $\forall \iota \in fv(\overline{\mathbf{v}}) : \iota \in dom(\mathcal{H})$ by 1, c, def F-Config 23.  $\forall \iota \in fv(\overline{\mathbf{e}}_0) : \iota \in dom(\mathcal{H}), \overline{\mathbf{x}}$ by 15, lemma 49 24.  $\forall \iota \in fv([\overline{\mathbf{v}/\mathbf{x}}]\mathbf{e}_0) : \iota \in dom(\mathcal{H})$ by 22, 23 25.  $\forall \iota \in fv(\mathbf{e}') : \iota \in dom(\mathcal{H})$ by 24, 2  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 26. by 3, 14, 25 27. doneby 21, 26

Case 6. (R-OPEN-CLOSE)

$e = open$ (close v with $o \rightarrow [b_l \ b_u]$ hiding $e' = [\overline{r/o}, \ v/x]e$ $\mathcal{H}' = \mathcal{H}$	$\overline{\mathbf{r}}$ ) as x: $\overline{\mathbf{o}}$ in e by def R-OPEN-CLOSE
$ \begin{array}{l} \emptyset; \Delta; \mathcal{H} \vdash \texttt{close v with } \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} \text{ hiding } \overline{\mathbf{r}} \\ \emptyset; \Delta, \overline{\mathbf{o}' \rightarrow [\mathbf{b}'_l \ \mathbf{b}'_u]}; \mathcal{H}, \texttt{x}: \texttt{N} \vdash \texttt{e} : \texttt{U} \\ \emptyset; \Delta; \mathcal{H} \vdash \texttt{U} \text{ OK} \\ \Delta; \mathcal{H} \vdash \texttt{U} <: \texttt{T} \end{array} $	$\left. \overline{\mathbf{c}} : \exists \overline{\mathbf{o}' \to [\mathbf{b}'_l \ \mathbf{b}'_u]} \cdot \mathbf{N} \right\}  by \ 1, \ \mathbf{a}, \ lemma \ 56$
$\Delta; \mathcal{H} \vdash \overline{[\overline{\mathbf{r}/\mathbf{o}}]  \mathbf{b}_l \preceq \mathbf{r}}$	)
$\Delta; \mathcal{H} \vdash \overline{\mathbf{r} \preceq [\overline{\mathbf{r}/\mathbf{o}}] \mathbf{b}_u}$	
$\Delta; \mathcal{H} \vdash \overline{\mathbf{r}}$ ok	hu 4 lemma 57
$\emptyset; \Delta; \mathcal{H} \vdash \mathtt{v} : [\overline{\mathtt{r/o}}] \mathtt{N}'$	
$\emptyset; \Delta; \mathcal{H} \vdash \exists \overline{o \rightarrow [b_l \ b_u]} . \mathbb{N}' \text{ OK}$	
$\Delta; \mathcal{H} \vdash \exists \overline{o \rightarrow [b_l \ b_u]}  .  \mathbb{N}' <: \exists \overline{o' \rightarrow [b'_l \ b'_u]}  .  \mathbb{N}$	J
$\Delta \vdash \mathcal{H}$ ок	by $\mathbf{c}$ , def F-Config
$\emptyset; \Delta; \mathcal{H}, \mathtt{x}: [\overline{\mathtt{r/o}}] \mathtt{N} \vdash [\overline{\mathtt{r/o}}] \mathtt{e} : [\overline{\mathtt{r/o}}] \mathtt{U}$	by 5, 8, 9, 10, d, 14, lemma 30
$\mathtt{N}=\mathtt{N}'$	by <b>13</b> , lemma 12
$\Delta, \overline{o' \to [b'_l \ b'_u]}; \mathcal{H} \vdash \mathtt{N}' <: \mathtt{N}$	by <b>16</b> , <i>lemma 1</i>
$\Delta; \mathcal{H} \vdash [\overline{\mathbf{r/o}}]  \mathbb{N}' <: [\overline{\mathbf{r/o}}]  \mathbb{N}$	by <b>17</b> , <b>10</b> , <b>8</b> , <b>9</b> , <b>d</b> , <b>14</b> , lemma 24
$\emptyset; \Delta; \mathcal{H} \vdash [\overline{r/o}, v/x]e : [\overline{r/o}, v/x]U$	by 15, 11, 18, d, 14, lemma 43
$\emptyset;\Delta;\mathcal{H}\vdash [\overline{ t{r/o}},  t{v/x}] extbf{e}: extbf{U}$	<i>by</i> <b>19</b> , <b>6</b>
$\emptyset; \Delta; \mathcal{H} \vdash \mathtt{T}$ ok	$by \mathbf{a}, \mathbf{d}, 14, \ lemma \ 47$
	e = open (close v with $o \rightarrow [b_l \ b_u]$ fiding e' = $[\overline{r/o}, v/x]e$ $\mathcal{H}' = \mathcal{H}$ $\emptyset; \Delta; \mathcal{H} \vdash close v with \overline{o \rightarrow [b_l \ b_u]}$ hiding $\overline{r}$ $\emptyset; \Delta; \mathcal{H} \vdash v close v with \overline{o \rightarrow [b_l \ b_u]}$ hiding $\overline{r}$ $\emptyset; \Delta; \mathcal{H} \vdash v close v with \overline{o \rightarrow [b_l \ b_u]}$ hiding $\overline{r}$ $\emptyset; \Delta; \mathcal{H} \vdash v close v with \overline{o \rightarrow [b_l \ b_u]}$ hiding $\overline{r}$ $\Delta; \mathcal{H} \vdash v close v with \overline{o \rightarrow [b_l \ b_u]}$ $\Delta; \mathcal{H} \vdash \overline{r} close v with \overline{o \rightarrow [b_l \ b_u]}$ $\Delta; \mathcal{H} \vdash \overline{r} close v with \overline{o \rightarrow [b_l \ b_u]}$ $\emptyset; \Delta; \mathcal{H} \vdash \overline{so \rightarrow [b_l \ b_u]}$ . N' ok $\Delta; \mathcal{H} \vdash \overline{so \rightarrow [b_l \ b_u]}$ . N' ok $\Delta \vdash \mathcal{H} ck$ $\emptyset; \Delta; \mathcal{H}, x: [\overline{r/o}]N \vdash [\overline{r/o}]e: [\overline{r/o}]U$ N = N' $\Delta; \mathcal{H} \vdash [\overline{r/o}]N' <: [\overline{r/o}]N$ $\emptyset; \Delta; \mathcal{H} \vdash [\overline{r/o}]N' <: [\overline{r/o}]N$ $\emptyset; \Delta; \mathcal{H} \vdash [\overline{r/o}, v/x]e: [\overline{r/o}, v/x]U$ $\emptyset; \Delta; \mathcal{H} \vdash [\overline{r/o}, v/x]e: U$ $\emptyset; \Delta; \mathcal{H} \vdash T ck$

22.  $\emptyset; \Delta; \mathcal{H} \vdash [\overline{\mathbf{r}/\mathbf{o}}, \mathbf{v}/\mathbf{x}] \mathbf{e} : \mathbf{T}$ 23.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathbf{e}' : \mathbf{T}$ 24.  $\forall \iota \in fv(\mathbf{e}') : \iota \in dom(\mathcal{H})'$ 25.  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ OK}$ 26. done

Case 7. (RC-Assign)

1.  $e = \iota . f = e''$  $e' = \iota . f = e'''$ 2.  $e''; \mathcal{H} \rightsquigarrow e'''; \mathcal{H}'$ 3.  $\texttt{e}''' \neq \texttt{err}$ 4. 5.  $\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathbb{N}$ 6.  $fType(f, \iota, N) = U$  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{e}'' : \mathbf{U}$ 7.  $\Delta$ ;  $\mathcal{H} \vdash \mathbf{U} <: \mathbf{T}$ 8.  $\Delta; \mathcal{H} \vdash \mathbf{e}'' \text{ ok}$ 9. 10.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{e}''' : \mathsf{U}$  $\Delta; \mathcal{H}' \vdash \mathbf{e}'''$  ok 11. 12.  $\emptyset; \Delta; \mathcal{H}' \vdash \gamma.f = e''' : U$  $\Delta \vdash \mathcal{H}'$  ок 13.  $fv(\mathbf{e}''') \subseteq dom(\mathcal{H}')$ **14.**  $\emptyset; \Delta; \mathcal{H}' \vdash \mathbf{T}$  ok 15.  $\emptyset; \Delta; \mathcal{H}' \vdash \gamma.f = e''' : T$ 16.  $\emptyset; \Delta; \mathcal{H}' \vdash e' : T$ 17. 18.  $fv(\mathbf{e}') \subseteq dom(\mathcal{H}')$  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 19. 20. done

Case 8. (RC-INVK)

1.	$e = \iota. < \overline{r}, \ \overline{T} > m(\overline{v}, e_i, \overline{e})$
2.	$e' = \iota. < \overline{r}, \ \overline{T} > m(\overline{v}, e'_i, \overline{e})$
3.	$e_i;\mathcal{H}\rightsquigarrowe_i';\mathcal{H}'$
4.	$\mathbf{e}'_i  eq \mathtt{err}$
5.	$\emptyset; \Delta; \mathcal{H} \vdash \iota : \mathtt{N}$
6.	$mType_{\Delta;\mathcal{H}}(\mathtt{m}{<}\overline{\mathtt{r}},\overline{\mathtt{T}}{>},\iota,\mathtt{N})=\overline{\mathtt{U}}{\rightarrow}\mathtt{U}$
7.	$\emptyset; \Delta; \mathcal{H} \vdash (\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) : \overline{\mathtt{U}}$
8.	$\Delta; \mathcal{H} \vdash \overline{\mathbf{r}}$ ok
9.	$\emptyset; \Delta; \mathcal{H} \vdash \overline{\mathtt{T}}$ ок
10.	$\Delta; \mathcal{H} \vdash \mathtt{U} <: \mathtt{T}$
11.	$\Delta; \mathcal{H} \vdash \mathbf{e}_i$ ok
12.	$\emptyset;\Delta;\mathcal{H}'dash \mathbf{e}'_i:\mathtt{U}_i$
13.	$\Delta; \mathcal{H}' \vdash e'_i$ ok
14.	$\emptyset; \Delta; \mathcal{H}' \vdash \iota. < \overline{\mathtt{r}}, \overline{\mathtt{T}} > \mathtt{m}(\overline{\mathtt{v}}, \mathtt{e}'_i, \overline{\mathtt{e}}) : \mathtt{U}$
15.	$\emptyset;\Delta;\mathcal{H}'dash  extbf{e}': extbf{U}$
16.	$\Delta \vdash \mathcal{H}'$ ок
17.	$fv(\mathbf{e}'_i) \subseteq dom(\mathcal{H}')$
18.	$\emptyset; \Delta; \mathcal{H}' \vdash T$ ок
19.	$\emptyset;\Delta;\mathcal{H}'dash  extbf{e}': extbf{T}$
20.	$fv(\mathbf{e}') \subseteq dom(\mathcal{H}')$
21.	$\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$
22.	done

*by* **20**, **7**, **21**, T-SUBS *by* **22**, **2**, **3** 

by **23**, lemma 49 by **3**, **14**, **24** 

by 23, 25

} by def RC-Assign
by premises RC-Assign
} by 1, a, lemma 54
by c, 1, def F-CONFIG
} by 7, 3, 9, d, 4, ind hyp
by 5, 6, 10, T-Assign
} by 11, def F-CONFIG
by a, d, 13, lemma 47
by 12, 8, 15, T-SUBS
by 16, 2
by c, 1, 2, 14
by 13, 18, F-CONFIG
by 17, 19

} by def RC-INVK
by premises RC-INVK
} by 1, a, lemma 55
by d, 1, def F-CONFIG
} by 7, 3, 11, d, 4, ind hyp
by 5, 6, 7, 12, 8, 9, T-INVK
by 14, 2
} by 13, def F-CONFIG
by a, d, 16, lemma 47
by 15, 10, 18, T-SUBS
by c, 1, 2, 17
by 16, 20, F-CONFIG
by 19, 21

Case 9. (RC-OPEN)

1.  $e = open e_1 as x:\overline{o} in e_2$ 2.  $\mathbf{e}' = \mathbf{open} \ \mathbf{e}'_1$  as  $\mathbf{x}: \overline{\mathbf{o}} \ \mathbf{in} \ \mathbf{e}_2$  $e_1; \mathcal{H} \rightsquigarrow e_1'; \mathcal{H}'$ 3. **4**.  $\mathsf{e}_1'\neq \texttt{err}$  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{e}_1 : \exists \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} . \mathbb{N}$ 5. 6.  $\emptyset; \Delta, \overline{\mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u]}; \mathcal{H}, \mathtt{x}: \mathtt{N} \vdash \mathsf{e}_2 : \mathtt{U}$ 7.  $\emptyset; \Delta; \mathcal{H} \vdash \mathbf{U}$  ok  $\Delta; \mathcal{H} \vdash \mathbf{U} <: \mathbf{T}$ 8. 9.  $\Delta; \mathcal{H} \vdash \mathsf{e}_1 \text{ ok}$ 10.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathbf{e}'_1 : \exists \overline{\mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_u]} . \mathbb{N}$ 11.  $\Delta; \mathcal{H}' \vdash \mathbf{e}'_1 \text{ ok}$ 12.  $\emptyset; \Delta; \mathcal{H}' \vdash \texttt{open e}'_1 \texttt{ as } \texttt{x}: \overline{\texttt{o}} \texttt{ in e}_2: \texttt{U}$ 13.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{e}' : \mathsf{U}$  $\Delta \vdash \mathcal{H}'$  ок 14.  $fv(\mathbf{e}_1') \subseteq dom(\mathcal{H}')$ 15.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{T}$  ок 16.  $\emptyset; \Delta; \mathcal{H}' \vdash e' : T$ 17. 18.  $fv(\mathbf{e}') \subseteq dom(\mathcal{H}')$  $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 19. 20. doneCase 10. (RC-CLOSE)  $e = close \ e''$  with  $\overline{o \rightarrow [b_l \ b_u]}$  hiding  $\overline{r}$ 1.  $\mathsf{e}'=\mathsf{close}\ \mathsf{e}'''$  with  $\overline{\mathsf{o}\!\rightarrow\![\mathsf{b}_l\ \mathsf{b}_u]}$  hiding  $\overline{\mathsf{r}}$ 2. 3.  $e''; \mathcal{H} \rightsquigarrow e'''; \mathcal{H}'$ 4.  $\texttt{e}''' \neq \texttt{err}$ 5.  $\Delta; \mathcal{H} \vdash [\overline{a/o}] b_l \preceq a$  $\Delta; \mathcal{H} \vdash a \preceq [\overline{a}/o] b_u$ 6. 7.  $\Delta; \mathcal{H} \vdash \overline{a} \text{ OK}$ 

8.  $\emptyset; \Delta; \mathcal{H} \vdash e'' : [\overline{a/o}] \mathbb{N}$ 

 $\emptyset; \Delta; \mathcal{H} \vdash \exists \overline{\mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u]}$ .N ok 9.

 $\Delta; \mathcal{H} \vdash \exists \overline{\mathsf{o} \rightarrow [\mathsf{b}_l \ \mathsf{b}_u]} . \mathbb{N} <: \mathsf{T}$ 10.  $\Delta; \mathcal{H} \vdash \mathbf{e}'' \text{ ok}$ 11.

 $\emptyset; \Delta; \mathcal{H}' \vdash e''' : [\overline{a/o}] \mathbb{N}$ 12.

 $\Delta; \mathcal{H}' \vdash \mathbf{e}'''$  ok 13.

 $\emptyset; \Delta; \mathcal{H}' \vdash \texttt{close e}''' \text{ with } \overline{\mathsf{o} \! \rightarrow \! [\texttt{b}_l \ \texttt{b}_u]} \text{ hiding } \overline{\texttt{r}} : \exists \overline{\mathsf{o} \! \rightarrow \! [\texttt{b}_l \ \texttt{b}_u]} . \mathbb{N}$ 14. 15.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathbf{e}' : \exists \mathbf{o} \rightarrow [\mathbf{b}_l \ \mathbf{b}_n] . \mathbb{N}$  $\Delta \vdash \mathcal{H}'$  ok 16.  $fv(\mathbf{e}''') \subseteq dom(\mathcal{H}')$ 17.  $\emptyset; \Delta; \mathcal{H}' \vdash \mathsf{T}$  ok 18.  $\emptyset; \Delta; \mathcal{H}' \vdash e' : T$ 19. 20.  $fv(\mathbf{e}') \subseteq dom(\mathcal{H}')$ 

 $\Delta; \mathcal{H}' \vdash \mathbf{e}' \text{ ok}$ 21.

22. done

by def RC-Open by premises RC-Open by 1, a, lemma 56 by **b**, **1**, def F-Config by 5, 3, 9, d, 4, ind hyp by 5, 6, 7, 10, T-OPEN by 12, 2 by 11, def F-Config by a, d, 14, lemma 47 by 13, 8, 16, T-SUBS by c, 1, 2, 15 by 14, 18, F-CONFIG by 17, 19

by def RC-CLOSE by premises RC-CLOSE by 1, a, lemma 57 by d, 1, def F-Config by 7, 3, 11, d, 4, ind hyp by 5, 6, 7, 12, 8, 9, T-CLOSE *by* **14**, **2** by 13, def F-Config *by* **a**, **d**, **16**, *lemma* 47 by 15, 10, 18, T-SUBS by c, 1, 2, 17

by 16, 20, F-CONFIG *by* **19**, **21** 

Theorem (Progress)

If:

a. 
$$\emptyset; \emptyset; \mathcal{H} \vdash e : T$$
  
b.  $\emptyset \vdash \mathcal{H} \text{ OK}$   
then:  
 $e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}'$   
or:

there exists v such that e = v

Proof is by structural induction on the derivation of  $\emptyset; \emptyset; \mathcal{H} \vdash e : T$  with a case analysis on the last step:

Case 1. (T-VAR)

1.	$e=\gamma$	by def T-VAR
2.	$e = \iota$	$by \mathbf{a}, 1, \ lemma \ 49$
3.	$done, \mathbf{e} = \mathbf{v}$	$by \ 2$

Case 2. (T-SUBS)

 $\emptyset; \emptyset; \mathcal{H} \vdash e : T'$ by premise T-SUBS 1. 2. doneby  $\mathbf{1}, \mathbf{b}, ind hyp$ 

Case 3. (T-FIELD)

1.	$\mathtt{e}=\gamma\mathtt{.f}$	$by \ def \ { m T-Field}$
2.	$\emptyset; \emptyset; \mathcal{H} dash \gamma: \mathtt{N}$	by premise of T-FIELD
3.	$\gamma = \iota \in dom(\mathcal{H})$	by <b>2</b> , lemma 49
4.	done	by 1, 3, R-Field

Case 4. (T-Assign)

1.	$\mathbf{e}=\gamma.\mathbf{f}$ = $\mathbf{e}''$	by def T-Assign
2.	$\emptyset; \emptyset; \mathcal{H} dash \gamma: \mathtt{N}$	hu promises of T ASSICN
3.	$\emptyset; \emptyset; \mathcal{H} \vdash \mathbf{e}'': \mathtt{U}$	$\int \frac{\partial g}{\partial t} premises \frac{\partial g}{\partial t} 1 - \mathbf{ASSIGN}$
4.	$\gamma = \iota \in dom(\mathcal{H})$	by <b>2</b> , lemma 49
5.	$e'',\mathcal{H}\rightsquigarrowe''',\mathcal{H}' \ or \ \exists \mathtt{v}:\mathtt{e}'=\mathtt{v}$	by $3$ , $\mathbf{b}$ , ind hyp

Case analysis on e'':

 $\mathbf{Case \ 1.}\ e'', \mathcal{H} \rightsquigarrow e''', \mathcal{H}'$ 

1.1. done

Case 2.  $\exists v : e'' = v$ 

2.1. done by 1,4,RC-Assign or RC-Assign-Err

 $by \mathbf{1}, \mathbf{4}, \mathbf{e}'' = \mathbf{v}, \operatorname{R-Assign}$ 

1. 2. 3	$\begin{array}{l} \mathbf{e} = \gamma . < \overline{\mathbf{a}}, \ \overline{\mathbf{T}} > \mathtt{m}(\overline{\mathbf{e}}) \\ \emptyset; \emptyset; \mathcal{H} \vdash \gamma : \mathtt{N} \\ \emptyset: \emptyset: \mathcal{H} \vdash \overline{\mathbf{a}} : \overline{\mathtt{H}} \end{array}$	by def T-INVK
3. 4.	mType defined	$\int \frac{\partial g}{\partial t} premises \frac{\partial g}{\partial t} = 1 \text{ for } K$
5. 6. 7. 8.	$\begin{split} &\gamma = \iota \in dom(\mathcal{H}) \\ &fv(\overline{\mathbf{a}}) \subseteq dom(\mathcal{H}) \\ &\overline{\mathbf{a}} = \overline{\mathbf{r}} \\ &\forall \mathbf{e}_i \in \overline{\mathbf{e}} : \mathbf{e}_i, \mathcal{H} \rightsquigarrow \mathbf{e}'_i, \mathcal{H}' \text{ or } \exists \mathbf{v} : \mathbf{e}_i = \mathbf{v} \end{split}$	by <b>2</b> , lemma 49 by <b>1</b> , <b>a</b> , lemma 49 by <b>2</b> , def syntax <b>r</b> by <b>3</b> , <b>b</b> , ind hyp
Case	analysis on ē:	
Ca	se 1. $\exists e_i \in \overline{e} : e_i, \mathcal{H} \rightsquigarrow e'_i, \mathcal{H}'$	
1	1.1. done	by 1, 5, 7, RC-INVK or RC-INVK-ERR

2.1.  $mBody \ defined$ by 4, def mBody, mType2.2. done

Case 6. (T-NEW)

Case 2.  $\forall e_i \in \overline{e} : \exists v : e_i = v$ 

1.	$e = new C < \overline{a}, \overline{U} >$	by def T-NEW
2.	$\emptyset; \emptyset; \mathcal{H} \vdash C < \overline{a}, \overline{U} > OK$	by premise T-NEW
3.	$\emptyset; \emptyset; \mathcal{H} \vdash \overline{\mathbf{a}} \text{ OK}$	
4.	$\emptyset; \emptyset; \mathcal{H} \vdash \overline{U}$ ок	by 2, premises F-CLASS
5.	class C<> ${\overline{f}; \ldots}$	J
6.	$\overline{a} = \overline{r}$	by <b>3</b> , syntax of <b>a</b>
7.	$fields(\mathtt{C}) = \overline{\mathtt{f}}$	$by \ {f 5} \ def \ fields$
8.	done	<i>by</i> <b>1</b> <i>,</i> <b>7</b> <i>,</i> <b>6</b> <i>,</i> R-NEW

Case 7. (T-OPEN)

1.	$e = open e'' as x:\overline{o} in e'''$	by def T-Open
2.	$\emptyset; \emptyset; \mathcal{H} \vdash \mathbf{e}'' : \exists \overline{\mathbf{o}  ightarrow [\mathbf{b}_l \ \mathbf{b}_u]}$ .N	by premises T-Open
3.	$e'',\mathcal{H} \leadsto e''',\mathcal{H}' \ or \ \exists \mathtt{v}: e'' = \mathtt{v}$	by 2, b, ind hyp

#### Case analysis on e'':

 $\mathbf{Case \ 1.}\ e'', \mathcal{H} \rightsquigarrow e''', \mathcal{H}'$ 

**1.1.** *done* 

Case 2.  $\exists v : e'' = v$ 

 $e''=\texttt{close}~v'~\texttt{with}~\overline{o{\rightarrow}[b'_l~b'_u]}$  hiding  $\overline{\texttt{r}}$ 2.1. 2.2. done

by 1, 5, 7, 2.1, R-INVK

by 1, RC-OPEN or RC-OPEN-ERR

by e'' = v, 2, b, lemma 59by 1, 2.1, R-Open-Close Case 8. (T-CLOSE)

1. 2. 3. 4. 5.	$\begin{array}{l} \mathbf{e} = \mathtt{close} \ \mathbf{e}'' \ \mathtt{with} \ \overline{\mathbf{o} \rightarrow [\mathtt{b}_l \ \mathtt{b}_u]} \ \mathtt{hiding} \ \overline{\mathtt{a}} \\ fv(\overline{\mathtt{a}}) \subseteq dom(\mathcal{H}) \\ \overline{\mathtt{a}} = \overline{\mathtt{r}} \\ \emptyset; \emptyset; \mathcal{H} \vdash \mathtt{e}'': \mathtt{U} \\ \mathtt{e}'', \mathcal{H} \sim \mathtt{e}''', \mathcal{H}' \ or \ \exists \mathtt{v} : \mathtt{e}'' = \mathtt{v} \end{array}$	by def T-CLOSE by 1, a, lemma 49 by 2, def syntax r by premise T-CLOSE by 4, b, ind hyp	
Case analysis on e":			
Cas	e 1. e $'', \mathcal{H} \sim$ e $''', \mathcal{H}'$		

**1.1.** *done* 

by 1,3, RC-CLOSE or RC-CLOSE-ERR

**Case 2.**  $\exists v : e'' = v$ 

**2.1.** 
$$\exists v' : e = v'$$
 by **1**, **3**,  $e'' = v$ 

# Bibliography

- [1] ECMA. C# Language Specification, 2002.
- [2] International Standard ISO/IEC 14882. Programming Languages C++, 2nd edition, 2003.
- [3] Scala Language Specification the Scala language reference document., 2007.
- [4] Ada83 Language Reference Manual, 1983.
- [5] Marwan Abi-Antoun and Jonathan Aldrich. Ownership Domains in the Real World. In International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), 2008.
- [6] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In European Conference on Object Oriented Programming (ECOOP), 2004.
- [7] Paulo Sérgio Almeida. Balloon Types: Controlling Sharing of State in Data Types. In European Conference on Object Oriented Programming (ECOOP), 1997.
- [8] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)/European Conference on Object Oriented Programming (ECOOP), 1990.
- [9] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-Time Java Virtual Machine with Applications in Avionics. *Transactions on Embedded Computing Systems*, 7(1):1–49, 2007.
- [10] Henk Barendregt. The Lambda Calculus. North-Holland, revised edition, 1984.
- [11] Boyapati, Liskov, and Shrira. Ownership Types for Object Encapsulation. In *Principles* of *Programming Languages (POPL)*, 2003.
- [12] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.
- [13] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-free Java Programs. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2001.

- [14] Gilad Bracha. Generics in the Java Programming Language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
- [15] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1993.
- [16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.
- [17] Kim B. Bruce, Martin Odersky, and Philip Wadler. A Statically Safe Alternative to Virtual Types. In European Conference on Object Oriented Programming (ECOOP), 1998.
- [18] Alex Buckley. Ownership Types Restrict Aliasing. Master's thesis, Department of Computing, Imperial College London, 2000.
- [19] Nicholas Cameron and Sophia Drossopoulou. Variant Ownership with Existential Types. In International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), 2008.
- [20] Nicholas Cameron and Sophia Drossopoulou. Existential Quantification for Variant Ownership. In European Symposium on Programming Languages and Systems (ESOP), 2009.
- [21] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In European Conference on Object Oriented Programming (ECOOP), 2008.
- [22] Nicholas Cameron, Sophia Drossopoulou, and James Noble. More Expressive Ownership Types, 2008. http://www.doc.ic.ac.uk/~ncameron/papers/cameron\_proposal08.pdf.
- [23] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2007.
- [24] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an Existential Types Model for Java Wildcards. In Formal Techniques for Java-like Programs (FTfJP), 2007.
- [25] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-Bounded Quantification for Object-Oriented Programming. In *Functional Programming Languages* and Computer Architecture (FPCA), 1989.
- [26] Luca Cardelli and Xavier Leroy. Abstract Types and the Dot Notation. Research report 56, DEC Systems Research Center, 1990.
- [27] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4):471–522, 1985.
- [28] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A Flow-Based Approach for Variant Parametric Types. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2006.
- [29] Dave Clarke and Tobias Wrigstad. External Uniqueness. In Foundations of Object-Oriented Languages (FOOL), 2003.

- [30] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
- [31] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In European Conference on Object Oriented Programming (ECOOP), 2003.
- [32] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.
- [33] David G. Clarke, James Noble, and John M. Potter. Simple Ownership Types for Object Containment. In European Conference on Object Oriented Programming (ECOOP), 2001.
- [34] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.
- [35] W. R. Cook. A Proposal for Making Eiffel Type-safe. The Computer Journal, 32(4):305– 311, August 1989.
- [36] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)*, 2008.
- [37] Ferruccio Damiani, Elena Giachino, Paola Giannini, Nick Cameron, and Sophia Drossopoulou. A state abstraction for coordination in Java-like languages. In *Formal Techniques for Java-like Programs (FTfJP)*, 2006.
- [38] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1995.
- [39] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In European Conference on Object Oriented Programming (ECOOP), 2007.
- [40] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. Journal of Object Technology, 4(8):5–32, 2005.
- [41] Sophia Drossopoulou, David Clarke, James Noble, and Tobias Wrigstad. Tribe: More Types for Virtual Classes. In Aspect-Oriented Software Development (AOSD), 2007.
- [42] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In European Conference on Object Oriented Programming (ECOOP), 2006.
- [43] Nicu G. Fruja. Type Safety of Generics for the .NET Common Language Runtime. In *European Symposium on Programming, ESOP*, 2006.
- [44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison-Wesley, 1995.
- [45] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. Theoretical Computer Science, 193(1-2):75–96, 1998.

- [46] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. PhD thesis, University of Paris VII, 1972.
- [47] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Third Edition. Addison-Wesley, Boston, Mass., 2005.
- [48] Aaron Greenhouse and John Boyland. An Object-Oriented Effects System. In European Conference on Object Oriented Programming (ECOOP), 1999.
- [49] Dan Grossman. Existential Types for Imperative Languages. In European Symposium on Programming Languages and Systems (ESOP), 2002.
- [50] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1991.
- [51] Igarashi and Pierce. Foundations for Virtual Types. INFCTRL: Information and Computation (formerly Information and Control), 175, 2002.
- [52] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A Recipe for Raw Types. In Foundations of Object-Oriented Languages (FOOL), 2001.
- [53] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus For Java and GJ. ACM Trans. Program. Lang. Syst., 23(3):396–450, 2001. An earlier version of this work appeared at OOPSLA'99.
- [54] Atsushi Igarashi and Mirko Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *Transactions on Programming Languages and Systems*, 28(5):795– 847, 2006. An earlier version appeared as "On variance-based subtyping for parametric types" at European Conference on Object Oriented Programming (ECOOP) 2002.
- [55] Andrew J. Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance. In Foundations and Developments of Object-Oriented Languages (FOOL/WOOD), 2007.
- [56] Neel Krishnaswami and Jonathan Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In Programming Language Design and Implementation (PLDI), 2005.
- [57] Yi Lu and John Potter. On Ownership and Accessibility. In European Conference on Object Oriented Programming (ECOOP), 2006.
- [58] Yi Lu and John Potter. Protecting Representation with Effect Encapsulation. In *Principles* of *Programming Languages (POPL)*, 2006.
- [59] Yi Lu, John Potter, and Jingling Xue. Validity Invariants and Effects. In European Conference on Object Oriented Programming (ECOOP), 2007.
- [60] Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In Foundations of Object-Oriented Languages (FOOL), 2005.
- [61] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: a Powerful Mechanism in Object-Oriented Programming. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1989.

- [62] Karl Mazurak and Steve Zdancewic. Note on "Type Inference for Java 5: Wildcards, F-Bounds, and Undecidability", 2006. http://www.cis.upenn.edu/~stevez/note.html.
- [63] Naftaly Minsky. Towards Alias-Free Pointers. In European Conference on Object Oriented Programming (ECOOP), 1996.
- [64] John C. Mitchell and Gordon D. Plotkin. Abstract Types have Existential Type. Transactions on Programming Languages and Systems, 10(3):470–502, 1988.
- [65] Benoit Montagu and Didier Remy. Modeling Abstract Types in Modules with Open Existential Types. In *Principles of Programming Languages (POPL)*, 2009.
- [66] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.
- [67] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.
- [68] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, October 2006.
- [69] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In European Conference on Object Oriented Programming (ECOOP), 1998.
- [70] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In Principles of Programming. Languages (POPL), 1997.
- [71] Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005.
- [72] Johan Ostlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, Uniqueness, and Immutability. In Objects, Components, Models and Patterns, TOOLS-EUROPE, 2008.
- [73] Benjamin C. Pierce. Bounded quantification is undecidable. In Principles of Programming Languages (POPL), 1992.
- [74] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002.
- [75] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. Journal of Functional Programming, 4(2):207–247, 1994.
- [76] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Principles and Practice of Programming in Java (PPPJ), 2007.
- [77] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Ownership. In Formal Techniques for Java-like Programs (FTfJP), 2005.
- [78] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2006.
- [79] John Potter, James Noble, and David Clarke. The Ins and Outs of Objects. In Australian Software Engineering Conference (ASEC), 1998.

- [80] John C. Reynolds. Towards a theory of type structure. In Paris colloquium on programming, 1974.
- [81] Claudio Russo. First-Class Structures for Standard ML. In International Conference on Functional Programming (ICFP), 1999.
- [82] Claudio Russo. Non-Dependent Types for Standard ML Modules. In *Principles and Practice of Declarative Programming (PPDP)*, 1999.
- [83] Jan Schfer and Arnd Poetzsch-Heffter. Simple Loose Ownership Domains . In Formal Techniques for Java-like Programs (FTfJP), 2006.
- [84] Matthew Smith. A Model of Effects with an Application to Ownership Types. PhD thesis, Department of Computing, Imperial College London, 2007.
- [85] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), 2003.
- [86] Bjarne Stroustrup. The C++ Programming Language. Special Edition. Addison-Wesley, 2000.
- [87] Kresten Krab Thorup. Genericity in Java with Virtual Types. In European Conference on Object Oriented Programming (ECOOP), 1997.
- [88] Kresten Krab Thorup and Mads Torgersen. Unifying Genericity Combining the Benefits of Virtual Types and Parameterized Classes. In European Conference on Object Oriented Programming (ECOOP), 1999.
- [89] Mads Torgersen. Virtual Types are Statically Safe. In Foundations of Object-Oriented Languages (FOOL), 1998.
- [90] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding Wildcards to the Java Programming Language. *Journal of Object Technology*, 3(11):97–116, 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus.
- [91] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's Variable Convention in Rule Inductions. In *Conference on Automated Deduction (CADE)*, 2007.
- [92] Mirko Viroli and Giovanni Rimassa. On Access Restriction with Java Wildcards. Journal of Object Technology, 4(10):117–139, 2005. Special issue: OOPS track at SAC 2005, Santa Fe/New Mexico. An earlier version appeared as "Understanding access restriction of variant parametric types and Java wildcards" at SAC 2005.
- [93] Jan Vitek and Boris Bokowski. Confined Types. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1999.
- [94] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized Interfaces for Java. In European Conference on Object Oriented Programming (ECOOP), 2007.
- [95] Stefan Wehr and Peter Thiemann. Subtyping Existential Types. In Formal Techniques for Java-like Programs (FTfJP), 2008.

- [96] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. Information and Computation, 115(1):38–94, 1994.
- [97] Tobias Wrigstad. Ownership-Based Alias Managemant. PhD thesis, KTH, Sweden, 2006.
- [98] Tobias Wrigstad and Dave Clarke. Existential Owners for Ownership Types. Journal of Object Technology, 6(4), 2007.
- [99] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based Confinement. J. Funct. Program, 16(1):83–128, 2006.