A Statically Typed Object-Oriented Language with Runtime Type-Argument Inference

Xinjie Xu

Supervised by **Fabian Muehlboeck**

Australian
National
University

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

October 2026

© Xinjie Xu 2026

Declaration:

I declare that this work:

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

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

October (2026), Xinjie Xu

Acknowledgements

I am deeply grateful to my supervisor, Fabian Muehlboeck, whom I met in my first semester at the Australian National University and who has guided me for almost four years since. He introduced me to programming languages research and inspired much of my interest in the field. Working with him taught me a great deal about language design, as well as about compiler and runtime implementation and optimisation. He was consistently generous with his time, often beyond his working hours, whenever I needed guidance. His patience, encouragement and trust shaped this thesis, and it would not have been possible without him.

I would also like to express my sincere appreciation to Carlo Zancanaro, one of Fabian's PhD candidates, who took part in my supervision meetings during the year. He helped me explore the boundaries of the type system developed in this thesis, probing its corner cases and searching for counterexamples, and several of its design decisions arose from our discussions.

I am indebted to my parents, Tao and Ke, for their love and support at every stage of my studies. They gave me every opportunity to pursue my education, and their belief in me has been a constant source of strength. This thesis is, in large part, theirs as well.

To my fiancée, Hechen: thank you for your love, patience and companionship throughout the writing of this thesis.

Finally, I thank my friends for their company and good cheer along the way.

Abstract

Statically typed object-oriented languages infer the type arguments of a generic call from the static types of its arguments. When a caller passes a value through a base class or interface, the call cannot use the more precise type that the value carries during execution. This thesis presents the design and implementation of G#, a statically typed nominal object-oriented language in which a generic call may omit its type arguments and have them inferred from runtime type information about the actual argument values, refining static typing. Several assignments of type arguments can satisfy the checked requirements of one call, so a unique optimal one must exist, not merely a type-safe one. Signature checking decides, from each generic declaration alone and independently of any call, whether calls to it may omit their type arguments: they may only if it establishes that runtime inference will compute a unique optimal assignment for every well-typed call that omits them, every permitted runtime input, every valid instantiation of a generic caller and every later addition of type declarations; otherwise the programmer must specify them explicitly. Every such call therefore has one static type, determined by its arguments alone and never by the type its context expects, and every execution of it has a unique optimal assignment, which runtime inference computes. We formalise the language and its inference rules, which combine constraints by joins and meets, and develop restrictions on type declarations that make these operations computable by algorithms the compiler and runtime share. A port of a sequence-query library keeps its parameter types, with function objects in place of delegates, and shows that they support inference: the calls that its benchmark methods make omit the type arguments of every operator except those whose type arguments no argument determines. Whole programs whose generic calls use runtime inference execute less than one per cent more instructions than their explicitly instantiated versions.

Contents

Acknowledgements **v**

Abstract **vii**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Contributions . . . 3

2 Background **5**
2.1 Generic types and methods . . . 5
2.2 Nominal subtyping, top and bottom . . . 6
2.3 Typing judgments and checking contexts . . . 7
2.4 Joins, meets and intersections . . . 7
2.5 Variance and generic views . . . 9
2.5.1 Declaration-site variance . . . 9
2.5.2 Use-site projections . . . 10
2.5.3 Variance and inference constraints . . . 11
2.6 Reification and runtime observations . . . 11
2.7 Bounds and recursive constraints . . . 12
2.8 Open-world assumptions . . . 14
2.9 Compile-time and runtime inference . . . 15

3 The G# Language and Its Type System **17**
3.1 Syntax . . . 17
3.2 Values and runtime types . . . 20
3.3 Subtyping . . . 21
3.3.1 Projections as intervals . . . 22
3.3.2 Principal instantiation . . . 22
3.3.3 Subtyping rules . . . 26
3.4 Valid types . . . 27
3.5 Joins, meets and canonical types . . . 28
3.6 Expression typing . . . 29
3.6.1 Members . . . 29
3.6.2 Checking and type tests . . . 32
3.6.3 Explicit generic calls . . . 32
3.7 Valid declarations and programs . . . 33
3.7.1 Valid declarations . . . 33

3.7.2 Valid programs . . . 36

4 Runtime Type-Argument Inference **37**
4.1 The need for a unique optimum . . . 37
4.2 Processing an RI call . . . 38
4.3 Reading type parameters from arguments . . . 39
4.4 Admissible and optimal assignments . . . 45
4.5 Eligibility . . . 47
4.6 Typing an RI call . . . 51
4.7 Intersections and inference . . . 52
4.8 Inferring the assignment at run time . . . 53
4.8.1 Folding reads . . . 54
4.8.2 Bounds and coupled parameters . . . 55
4.9 Execution of an RI call . . . 55
4.10 Optimality and soundness . . . 56
4.11 Implementation . . . 59

5 Evaluation **61**
5.1 Programming capabilities . . . 61
5.1.1 Collection materialisation . . . 61
5.1.2 Typed event delivery . . . 61
5.1.3 Typed alternatives . . . 63
5.2 Applicability to existing generic code . . . 63
5.3 Cost of runtime inference . . . 65
5.3.1 Whole programs . . . 65
5.3.2 Cost of one RI call . . . 66

6 Related Work **69**
6.1 Inference at run time . . . 69
6.2 Inference at compile time . . . 72
6.3 Runtimes that retain type arguments . . . 74
6.4 Variance and subtyping . . . 75
6.5 Summary . . . 76

7 Concluding Remarks **77**
7.1 Limitations . . . 77
7.2 Future work . . . 78
7.3 Conclusion . . . 78

A Supporting Rules **81**
A.1 Declarations and parents . . . 81
A.2 Valid types and bounds . . . 82
A.3 Expressions and statements . . . 83
A.4 Valid declarations and programs . . . 85
A.5 Coupled parameters . . . 87

A.6 The static assignment . . . 88
A.7 Bounds the runtime can satisfy . . . 89

B Complete G# Examples **91**
B.1 ToList . . . 91
B.2 Running example . . . 92
B.3 Typed event delivery . . . 93

C Typed Event-Delivery Alternatives **95**
C.1 C#: an adapter retaining T . . . 95
C.2 Java: an adapter with a class token . . . 96
C.3 Kotlin: reified evidence at registration . . . 96

D Evaluation Programs **97**
D.1 The LINQ suites . . . 97
D.2 The MonNom ports . . . 100
D.3 The smaller programs . . . 101

Bibliography **103**

List of Figures

1.1 The three stages of the guarantee . . . 1
1.2 Copy through a supertype view . . . 2
1.3 Explicit and inferred copy calls . . . 2

2.1 Representative types in a hierarchy . . . 6
2.2 Running hierarchy . . . 8
2.3 Join and meet in G# and with unions . . . 8
2.4 Covariant interfaces of ToList . . . 10
2.5 Contravariant handler interface . . . 10
2.6 Views as argument intervals . . . 11
2.7 Successively tighter common upper bounds . . . 13
2.8 Self-referential comparison contract . . . 14
2.9 Growth is downward . . . 14

3.1 Core syntax . . . 18
3.2 The running example . . . 19
3.3 An empty sequence at Never . . . 21
3.4 Principal instantiations . . . 24
3.5 Principal instantiation . . . 24
3.6 Members through projections . . . 25
3.7 Subtyping rules . . . 26
3.8 Joins of argument types . . . 28
3.9 Methods through a receiver . . . 30
3.10 Combining a member of two heads . . . 31
3.11 A member of an open intersection . . . 31
3.12 Type tests and their bindings . . . 32
3.13 Explicit calls . . . 33
3.14 Rejected and accepted declarations . . . 33
3.15 Usage graphs . . . 34

4.1 A call with no optimum . . . 38
4.2 A call whose optimum differs from the static assignment . . . 38
4.3 Processing an RI call . . . 39
4.4 Reads . . . 40
4.5 A read two levels deep . . . 41
4.6 A parameter read at both variances . . . 42
4.7 Coupling in Pair . . . 43

4.8 A bound met by a join . . . 43
4.9 The admissible set of a ToList call . . . 46
4.10 Eligibility . . . 48
4.11 Eligibility of example signatures . . . 50
4.12 Typing an RI call . . . 51
4.13 Selection rules . . . 54
4.14 Folding two reads . . . 54
4.15 Entry to and return from a generic call . . . 56
4.16 Derivation of the ToList call . . . 56
4.17 Descriptor links for the ToList source . . . 59

5.1 Body of ToList . . . 62
5.2 Typed event delivery . . . 62
5.3 Handlers reached by each call form . . . 63
5.4 Whole-program cost of omitting type arguments . . . 66
5.5 Cost of one RI call . . . 67

A.1 Declaration lookup . . . 81
A.2 Parents . . . 82
A.3 Valid types . . . 82
A.4 Valid bounds . . . 83
A.5 Types read through a projection . . . 83
A.6 Fields and static methods . . . 83
A.7 Checking, construction and field reads . . . 84
A.8 Type tests . . . 84
A.9 Statements . . . 85
A.10 Refinement . . . 85
A.11 Consistent inheritance . . . 86
A.12 Class bodies . . . 86
A.13 Valid programs . . . 87
A.14 Coupling adjustments . . . 88
A.15 Static assignment rules . . . 88
A.16 Bounds the runtime can satisfy . . . 89

D.1 Interfaces and operators of the LINQ library . . . 98
D.2 Slot storage of the LINQ library . . . 98
D.3 The list class of the LINQ library . . . 99
D.4 Call sites of the two LINQ suites . . . 99
D.5 Calls in the MonNom ports . . . 101
D.6 Calls in the smaller programs . . . 101

List of Tables

2.1 Direct and generic-type observations 12

3.1 Combining one slot of two instantiations 23

4.1 Directions from one occurrence 42
4.2 Signatures that cannot guarantee an optimum 47
4.3 Bounds the runtime can satisfy 49
4.4 Intersections and inference 52

5.1 Typed alternatives to inference 63
5.2 Classes of LINQ operator 64
5.3 Type-argument sites on the workloads 64

6.1 Guarantee properties by approach 76

D.1 Whole-program execution cost 100

Introduction

Generics let one operation work with different types, while subtyping lets a caller use a value through a base class or interface [Cardelli and Wegner, 1985, Section 1.3]. Conventional compile-time inference uses only static types, chiefly those of the argument expressions [Microsoft, 2024; Pierce and Turner, 2000]. At run time an argument value can have a more precise type than its argument expression: in a language with reified generics, such as C#, a list created as `List<Cat>` still records `Cat` when it is used as an `IEnumerable<Animal>` [Kennedy and Syme, 2001, Section 2.2]. *Runtime type-argument inference*, or runtime inference for short, lets a generic call obtain its type arguments from the runtime types of its argument values. The chosen arguments shape the call's result and objects that other code may modify, so they must satisfy the caller's checked requirements. Beyond safety, a unique optimal choice must exist, and most languages do not guarantee one (in C#, the copying call of Section 1.1 has none); runtime inference must compute that choice.

This thesis presents the design and implementation of G#, a statically typed nominal object-oriented language with runtime inference. Its guarantee is that if a program compiles, no execution halts because of a type error. A generic call that omits its type arguments is a *runtime-inference call*, or *RI call*. G# guarantees the optimum where other languages do not: it permits an RI call only for an *eligible* method, one for which signature checking establishes, from the declaration alone, that every well-typed RI call of it has a unique optimal assignment of type arguments and that runtime inference computes it. The guarantee covers every value the static argument types allow, every valid instantiation of the enclosing type parameters and every later addition of type declarations, such as classes that loaded or generated code introduces. Calls of other generic methods must give their type arguments. Figure 1.1 shows the three stages that establish this guarantee, and Chapter 4 formalises them.

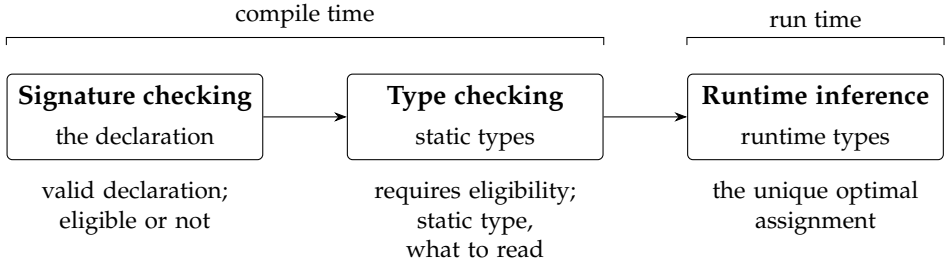


Figure 1.1: The three stages of the guarantee: what each examines (boxed) and establishes (below).

1.1 Motivation

`ToList` illustrates how the inferred type argument affects a generic operation's result; consider the C# program in Figure 1.2, where `Cat` is a subclass of `Animal`:

```
var cats = new List<Cat>();
cats.Add(new Cat());
IEnumerable<Animal> view = cats;
var copy = view.ToList(); // List<Animal>
```

Figure 1.2: Compile-time inference copies a list of cats through its `Animal`-typed view.

Although `view` refers to the original `List<Cat>`, the variable's static type, the *view* through which the caller uses the list, is `IEnumerable<Animal>`. Compile-time inference therefore chooses `Animal` for `ToList`, which creates a new `List<Animal>` [Microsoft, 2026a, 2024]. The copy cannot be passed to a consumer expecting `IEnumerable<Cat>`, even though the original list can.

A C# caller that expects cats can still obtain a `List<Cat>`, but only by naming `Cat`: the test `view is IEnumerable<Cat> c` succeeds, since the list holds cats, and `c.ToList()` then infers `Cat` from the static type of `c`. A generic method that copies whatever `IEnumerable<T>` it receives cannot write such a test, because it does not know which element type its input has at run time. Runtime inference gives such code a copy with the input's actual element type without naming that type.

Figure 1.3 shows the same operation in G#. Here T is the method's type parameter. Square brackets enclose type parameters and arguments. The result type `List[out T]` is a read-only view of the list, a *projection* (Section 3.3.1): elements can be read from it as T, but none can be added. The first call gives `Animal` explicitly; the second omits the argument, leaving it to runtime inference. G# has no `null`, so the list is constructed with a fill value for its unused slots (Section 3.2).

```
static List[out T] ToList[T](IEnumerable[T] source) { ... }

List[Cat] cats = new List[Cat](new Cat());   // fill value
IEnumerable[Animal] view = cats;
// constructs a List[Animal]
List[out Animal] explicitCopy = Program.ToList[Animal](view);
// constructs a List[Cat]
List[out Animal] runtimeCopy = Program.ToList(view);
```

Figure 1.3: The same copying operation with explicit and inferred type arguments. Both copies have the static type `List[out Animal]`; the comments give the list each call constructs.

The two copies have the same contents and the same static type, and differ in the instantiation constructed: the explicit call constructs a `List[Animal]`, as the C# call does, while the RI call recovers `Cat` from the source and constructs a `List[Cat]`.

The source is an `IEnumerable[T]`, which is covariant (Section 2.5), so a source of type `IEnumerable[S]` fits every T above S. Type checking sees only the static type of `view`, `IEnumerable[Animal]`, and takes the smallest such T, `Animal`, which gives the call the static type `List[out Animal]`. At run time the value is a `List[Cat]`, which fits every T above `Cat`, and the runtime choice must agree with the static type, which keeps T below `Animal`. A smaller T gives a more precise copy, so the smallest, `Cat`, is optimal (Section 4.4 defines the comparison). The caller still receives `List[out Animal]`. The copying body appears in Figure 5.1, and the supporting program is given in Appendix B.

This comparison depends on the projected result. In C#, `ToList` returns the invariant `List<T>`, and a list of cats is also an `IEnumerable<Animal>`, so both `Cat` and `Animal` are valid type arguments for it. Their results `List<Cat>` and `List<Animal>` are unrelated, neither a subtype of the other, so neither choice is better; C# fixes one by its inference rules. In G#, the result `List[out Cat]` is a subtype of `List[out Animal]`, so the choices are ordered and a best one can exist. More precise input information alone therefore cannot justify choosing a more specific type argument: the result and its write permissions constrain the choice too. With several inputs or bounds relating type parameters, inference must also compute joins and meets, the least upper and greatest lower bounds of types. Their existence and computability become part of the language design. Other generic operations face the same constraints and gain the same precision, for example delivering an event to the handlers that accept its runtime type and chaining sequence queries whose element types are recovered from the source (Chapter 5).

The projection is a trade-off. The C# `List<Animal>` lets its holder add any animal, but giving an inferred copy that type would be unsound, since the copy may be a `List<Cat>` and adding a `Dog` would need a check that can fail at run time. G#'s `ToList` therefore projects its result. Had it returned the exact `List[T]`, its invariant result would admit only the type argument given by the static types, and the call would behave as if `Animal` were given explicitly.

This work builds on compile-time inference [Pierce and Turner, 2000; Dolan and Mycroft, 2017], runtime recovery of type arguments [Cameron et al., 2008; Park et al., 2019] and, most directly, Muehlboeck's principally inferable generic methods, whose invocations infer their type arguments at run time [Muehlboeck, 2019, Chapter 8]; Chapter 6 compares them with G#.

1.2 Contributions

The thesis makes four contributions:

- The design of G#, in which a generic call may omit its type arguments and obtain them from the runtime types of its argument values, while the checked

requirements of its arguments, result and mutable objects still hold (Chapter 3).

- Signature checking, which decides from a method's declaration alone whether its calls may omit their type arguments: only when every such call has a principal typing (Section 2.9) and a unique optimal assignment for every permitted runtime input, also after new type declarations are added (Section 4.5).
- A formalisation of signature checking, type checking of RI calls and runtime inference that states this guarantee as properties of the rules, including that an RI call never halts an execution because of a type error, and an implementation in which the compiler and runtime share the algorithms for the type operations (Chapter 4).
- An evaluation that shows the capabilities inference enables, ports a LINQ-style library to see which calls can omit their type arguments, and measures the runtime cost of inference, in whole programs and for one call (Chapter 5).

Chapter 2 supplies the prerequisites, Chapter 3 defines G#'s type system, Chapter 4 formalises and implements runtime inference, Chapter 5 evaluates it, Chapter 6 compares related approaches and Chapter 7 concludes.

Background

Runtime inference chooses the type arguments of a statically checked generic call from the runtime types of its argument values. This chapter introduces what that choice depends on, in the order in which Chapters 3 and 4 use it: generic types and subtyping, typing judgments, joins and meets, variance, what execution can observe about a value, bounds, which facts remain valid when new type declarations are added, and finally type-argument inference itself. It uses the `ToList` example of Chapter 1 and the animal hierarchy of Figure 2.2 throughout.

2.1 Generic types and methods

A generic declaration abstracts over types. In `List[T]`, T is a *type parameter*; in `List[Cat]`, `Cat` is its *type argument*. Writing `List[Cat]` forms an instantiation. The *head* of a type is its declared class or interface name, and a *generic head* is a head with type parameters: `List[Cat]` and `List[Animal]` have the same generic head, `List`, but different arguments. A *slot* is a position in a generic declaration's type-parameter list. Featherweight Generic Java formalises generic classes and methods in these terms [Igarashi et al., 2001, Section 3].

A method can introduce its own type parameters. The argument for T in `ToList[T]` is chosen for a call; the separate T declared by the class `List[T]` determines a list's instantiation. A *substitution* maps these parameters to arguments; for a call of a generic method, an *assignment* gives one type argument to each of the method's type parameters. Substituting `Cat` for T turns `List[T]` into `List[Cat]`. A type expression containing a free parameter is *open*; one with no free parameters is *ground*. A *closing substitution* replaces all free parameters with ground type arguments. A type parameter may also declare a *bound* that restricts its type arguments, as in `T : IComparable[T]`, which allows only types comparable with themselves (Section 2.7). A *valid* type argument satisfies the language's type rules, and a closing substitution is *valid* when its type arguments are valid and satisfy the bounds. Since a bound may mention type parameters, it is checked after substitution: `Cat` is a valid type argument for this T only if `Cat` satisfies `IComparable[Cat]`. A generic method body is checked once, against its type parameters and bounds rather than for each substitution [Igarashi et al., 2001, Section 3.2], so the type arguments chosen at a call must be valid.

A method's *signature* specifies its type parameters and bounds, parameter types and result type. The declared type of a parameter is its *formal parameter type*, and

the expression passed for it at a call is its *argument*. *Signature checking* checks a declaration against the language rules, and the body is then checked against the signature. A valid signature does not by itself let calls omit their type arguments: in G# signature checking also decides, from the signature alone, whether the method is eligible, that is, whether every RI call of it meets the inference guarantee of Chapter 1. The decision depends neither on the static types at a particular call nor on the type its context expects, so it is the same for every call (Section 4.5).

2.2 Nominal subtyping, top and bottom

Generic and non-generic types are related by subtyping, which inference also uses to express constraints on type arguments. We write $A <: B$ when A is a subtype of B: an A value can be used through a B view [Cardelli and Wegner, 1985, Section 1.3]. Subtyping is reflexive; we also say that A is *below* B and B *above* A, in this inclusive sense. In a nominal system, subtyping between named classes and interfaces is given by the ancestry each declaration states [Kennedy and Pierce, 2007, Section 2], not by matching member signatures. We write $A \equiv B$ for type equivalence, meaning that both $A <: B$ and $B <: A$ hold.

G# has a greatest type, `object`, and a least type, `Never`. We write them $\top$ (top) and $\bot$ (bottom), so $\bot <: T <: \top$ for every type T. `Never` has no values, which is why it can lie below every type.

The built-in types `int`, `bool` and `void` are called *primitives*; no declaration can extend them. G# also provides the generic type `Array[T]` (abbreviated `T[]`), whose rules are in Chapter 3; all of them are below `object`. Unlike `Never`, `void` has a value: a normally completing void method returns its single value, *Unit*. In Figure 2.1, T stands for any valid type argument.

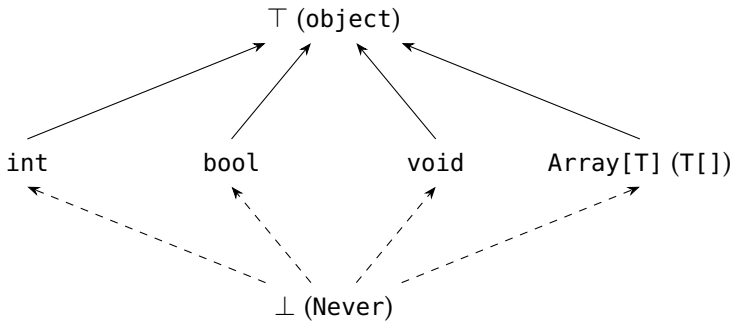


Figure 2.1: Representative types below $\top$ and above $\bot$. Dashed arrows express the universal bottom rule (Sub-Bot, Section 3.3.3), not declared inheritance. Arrows denote subtyping.

2.3 Typing judgments and checking contexts

A *typing judgment* states that an expression has a type under some assumptions. We write $\mathcal{D} \mid \Delta \mid \Gamma \vdash e : T$, where $\mathcal{D}$ records class and interface declarations, Δ the type parameters in scope and their bounds, and Γ the types of variables in scope. A rule places its premises above a line and its conclusion below it:

$$\frac{\Gamma(x) = T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash x \Rightarrow T}\ \textsc{T-Var}.$$

This rule, named T-Var, reads: if x has type T in Γ, then the expression x has type T. A rule's name combines a prefix for its judgment, here T for typing, with the case it covers. Every judgment in this thesis states the environments it consults, separated by bars.

Bidirectional typing distinguishes *synthesis*, which obtains a type from an expression, from *checking*, which verifies it against an expected type [Dunfield and Krishnaswami, 2021]. We write $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T$ and $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow T$ respectively. In the core of G#, checking has a single rule: an expression checks at T exactly when its synthesised type is a subtype of T.

2.4 Joins, meets and intersections

Several arguments can constrain the same type parameter, and joins and meets combine their constraints. A common *upper bound* of two types lies above both; their *least upper bound* (lub) lies below every other common upper bound. Dually, their *greatest lower bound* (glb) lies below both and above every other common lower bound. Both are unique up to equivalence when they exist.

We write $A \sqcup B$ for the lub, or *join*, and $A \sqcap B$ for the glb, or *meet*. A source *union type*, `A | B`, accepts values belonging to either type; an *intersection type*, `A & B`, requires both. Read as sets of values, they are set union and intersection [Frisch et al., 2002; Muehlboeck and Tate, 2018]. A join must be a type the language can express; G# has intersection types but no union types; the next example shows the precision this loses.

G# permits one class base and multiple interfaces. In Figure 2.2, `Cat` and `Dog` are sibling subclasses of `Animal`. `Kitten` extends `Cat` and implements the interfaces `IPet` and `INamed`; `Puppy` extends `Dog` and implements `IPet`. Later chapters reuse this hierarchy.

Figure 2.3 compares the join of `Kitten` and `Puppy`, and the meet of `IPet` and `INamed`, in G# and in a hypothetical language that also supports union types. Follow arrows upward to find common upper bounds and downward to find common lower bounds.

In panel (a), `Kitten` and `Puppy` share the class `Animal` and the interface `IPet`, so their least common upper bound is an intersection, and the meet of two interfaces is

```
class Animal { }
interface IPet { }
interface INamed { }
class Cat : Animal { }
class Dog : Animal { }
class Kitten : Cat, IPet, INamed { }
class Puppy : Dog, IPet { }
```

Figure 2.2: Two class branches under a common superclass, with interfaces implemented by their leaves.

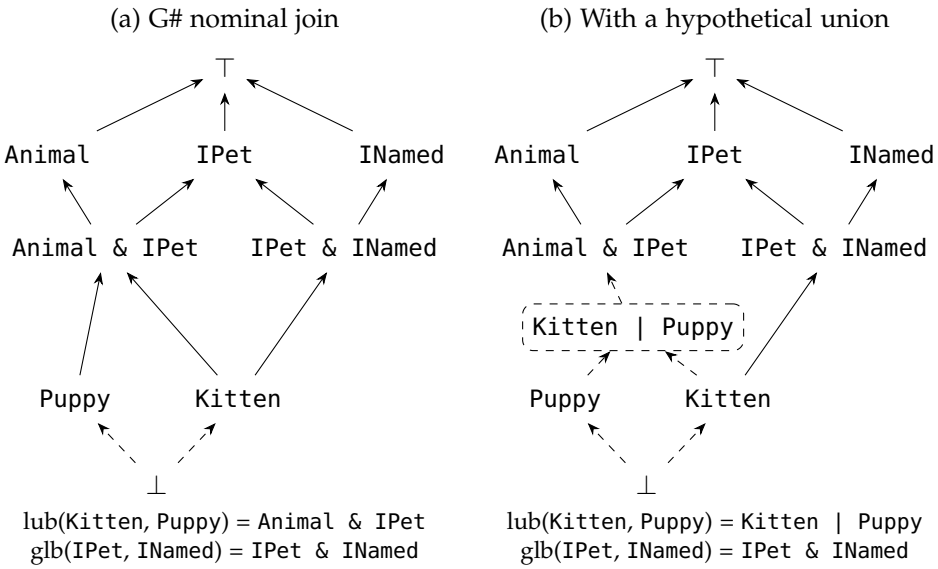


Figure 2.3: Join of `Kitten` and `Puppy` and meet of `IPet` and `INamed` in G# and with hypothetical union types; `Cat` and `Dog` are omitted. Arrows point to supertypes; dashed edges are hypothetical union edges or the universal bottom rule.

likewise an intersection:

$$\texttt{Kitten} \sqcup \texttt{Puppy} = \texttt{Animal \& IPet}, \qquad \texttt{IPet} \sqcap \texttt{INamed} = \texttt{IPet \& INamed}.$$

`Kitten` implements both interfaces, so it is below this meet but is not the meet, since a later class may implement both as well. Two classes unrelated by inheritance are disjoint, since a class has one class base: $\texttt{Cat} \sqcap \texttt{Dog} = \bot$. An intersection also exposes the members of both constituents; when both declare the same member, one implementation must meet the merged contract (Section 3.6.1).

The intersection is what makes this join exist. `Animal` and `IPet` are both common upper bounds of `Kitten` and `Puppy`, and neither is below the other. Without intersections no type below both `Animal` and `IPet` would be expressible, so each would only be a *minimal* upper bound, one with no smaller upper bound. No least

upper bound would exist, and inference would have to choose one arbitrarily. The same distinction governs inference: runtime inference must choose an assignment at least as good as every alternative, not merely one that no alternative improves on (Section 2.9). Subtyping alone does not guarantee a least upper bound; some Java wildcard types have none [Tate et al., 2011, Section 5.1].

In panel (b), `Kitten | Puppy` is a tighter join because it contains only the two classes, whereas `Animal & IPet` contains any pet animal. Each is the least upper bound among the types its system can express; adding union syntax also requires integrating its subtype rules with the rest of the system [Muehlboeck and Tate, 2018, Sections 3–4].

For generic types the join must also combine type arguments according to variance [Greenman et al., 2014; Muehlboeck and Tate, 2018]; Section 3.5 defines it.

Top and bottom give the identities

$$T \sqcup \bot = T, \qquad T \sqcap \bot = \bot, \qquad T \sqcup \top = \top, \qquad T \sqcap \top = T.$$

2.5 Variance and generic views

`Cat` is below `Animal` in the preceding hierarchy, but `List[Cat]` is not a subtype of `List[Animal]`. *Variance* describes how a subtype relation between type arguments affects the corresponding generic types. The safe relationship depends on how the generic declaration uses its parameter [Igarashi and Viroli, 2006; Altidor et al., 2011].

2.5.1 Declaration-site variance

Invariance `List[T]` accepts T arguments through `Add` and returns T elements when read. If `List[Cat]` could be used as `List[Animal]`, a caller could insert a `Dog`, which is not a `Cat`. Reading it through the original `List[Cat]` reference would then violate its type. The reverse subtype relationship would be unsafe too: a `List[Animal]` may already contain a `Dog`. These exact list types are therefore incomparable.

The restriction concerns using the list through a different generic type; `Add` on `List[Animal]` still accepts a `Cat` argument by ordinary subtyping. Invariance is G#'s default, and `Array[T]` is invariant too.

Covariance (out) An `IEnumerable[Cat]` only produces `Cat` elements. Every such element can be used as `Animal`, so the enumerable can be used as `IEnumerable[Animal]`. The subtype direction is preserved. The enumeration interfaces in Figure 2.4 use T only in results [Microsoft, n.d.].

```
interface IEnumerator[out T] {
    bool MoveNext();
    T Current();
}
interface IEnumerable[out T] {
    IEnumerator[T] GetEnumerator();
}
```

Figure 2.4: Covariant interfaces used by `ToList`. The element parameter appears only in result positions.

Contravariance (in) An `IHandler[Animal]` accepts every `Animal` value, including `Cat` values. It can therefore serve a caller that needs an `IHandler[Cat]`. Here the subtype direction reverses (Figure 2.5) [Igarashi and Viroli, 2006; Emir et al., 2006].

```
interface IHandler[in E] {
    void Handle(E e);
}
```

Figure 2.5: A contravariant handler consumes E through a method parameter, so its subtype direction reverses: `IHandler[Animal]` can stand where `IHandler[Cat]` is needed.

2.5.2 Use-site projections

Projected views An invariant list can be used through a view with fewer permissions. `List[out Animal]` permits reading elements as `Animal`, but not passing an element to a T-typed input. Both `List[Cat]` and `List[Animal]` provide this view. Conversely, `List[in Cat]` accepts `Cat` arguments and reads elements as `object`. A view restricts access to the same object; it does not change its exact instantiation.

The `?` projection hides the argument, permitting reads as `object` but no element input. The `out`/`in` annotations on `IEnumerable` and `IHandler` are *declaration-site variance*; those on a `List` view are *use-site variance* [Torgersen et al., 2004; Altidor et al., 2011; Tate, 2013]. Argument intervals explain both their subtype relations and their permissions.

Views as argument intervals A projected view describes a range of possible exact type arguments [Igarashi and Viroli, 2006; Craciun et al., 2009]. Write $[lo, hi]$ for the interval of types X satisfying $lo <: X <: hi$.

For an invariant generic class C without bounds, the views have the following meanings (Figure 2.6). The element operations shown have T directly as their input or result type:

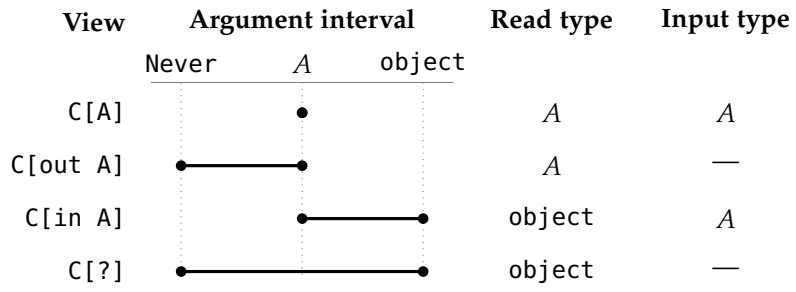


Figure 2.6: Views of an invariant generic class C. Bars mark argument intervals $[lo, hi]$; a point marks an exact argument. A dash means no element input is permitted.

The upper end is a safe read type for every instantiation in the interval, and an input must lie below the lower end to suit all of them. When the lower end is `Never`, no value can be supplied, which is why `out` and `?` views permit no element input. One view is a subtype of another when its interval is contained in the other's (Section 3.3.3). Without bounds, `List[?]`, `List[out object]` and `List[in Never]` have the same full interval.

2.5.3 Variance and inference constraints

Variance also determines what an input tells us about an unknown argument. Passing an `IEnumerable[Cat]` argument at the formal parameter type `IEnumerable[T]` requires `Cat` $<:$ T. Passing an `IHandler[Animal]` argument at `IHandler[T]` instead requires T $<:$ `Animal`. An exact `List[Cat]` at a formal parameter type `List[T]` forces $T \equiv$ `Cat`. Thus covariant, contravariant and invariant occurrences contribute lower-bound, upper-bound and equality constraints on T, respectively.

Directions compose along nested type arguments. Two in positions reverse the relation twice, preserving the original direction [Altidor et al., 2011]:

`IHandler[IHandler[Cat]]` $<:$ `IHandler[IHandler[Animal]]`.

The `?` projection leaves the exact argument unconstrained.

2.6 Reification and runtime observations

Runtime inference reads type arguments from values. A value's *exact runtime type* is the most specific type it has during execution, and languages differ in how much of it they retain. *Erasure* removes type arguments from the runtime representation of generic values. In Java, for example, execution can identify a value's class but cannot generally test whether a list has the argument `Cat` rather than `Animal` [Oracle, 2025b, Section 4.7]. Class files retain the generic signatures of declarations, but erased instances do not record their arguments [Goetz, 2020].

Reification retains a runtime representation of type arguments. The CLR can distinguish `List<Cat>` from `List<Animal>` [Kennedy and Syme, 2001, Section 2.2], so the source in the `ToList` example still carries `Cat` even when the caller sees `IEnumerable<Animal>`.

To read a type argument at a generic head, a type needs one entry for that head. When several inheritance paths reach the head with different arguments, the *principal instantiation* is the single entry that combines them. Ceylon defines it as the principal instantiation of the intersection of all the inherited instantiations, and calls inheritance through it *principal-instantiation inheritance* [King, 2017, Sections 3.7.3–3.7.4]; in the intersection of two instantiations, the arguments at a contravariant slot are joined and those at a covariant slot met [Muehlboeck and Tate, 2018, Section 6.3]. G# defines its rule in Section 3.3.2. For the covariant enumerable head,

$$\texttt{Cat} <: \texttt{Animal},$$
$$\texttt{IEnumerable[Animal]} \,\&\, \texttt{IEnumerable[Cat]} \equiv \texttt{IEnumerable[Cat]}.$$

Reading type arguments from values An *observation* is a fact about a type argument that inference reads from an input value's runtime type. The formal parameter type determines which observation is read. In `T x`, T is itself the formal parameter type, a *direct occurrence*; inference reads the input value's exact runtime type. In `IEnumerable[T] xs`, T occurs beneath a generic head, an occurrence within a generic type: inference finds the input's `IEnumerable` entry and reads its argument. `ToList` uses this second form. The same input can therefore supply different observations (Table 2.1).

In `IEnumerable[List[T]]`, T is an occurrence within a generic type, here beneath two generic heads. Reading T follows further retained type arguments (Section 4.3). An observation supplies a constraint, not the final assignment; the compiler records which observations to read and how to combine them.

Table 2.1: Observations for parameters used directly or within a generic type. Which of the two observations a call may use is fixed by the parameter's occurrence in the called method's signature.

Formal parameter	Input's runtime type	Observation for T
`T x`	`List[Cat]`	`List[Cat]` itself
`IEnumerable[T] xs`	`List[Cat]`	`Cat` at the `IEnumerable` head

2.7 Bounds and recursive constraints

Observations alone do not determine whether an assignment satisfies a generic declaration. A declared upper bound restricts the arguments that may substitute for a type parameter: `T : Animal` permits `Animal` itself and its subtypes. A bound can

also relate parameters, as in `U : T`, which requires the selected U to be a subtype of the selected T, possibly T itself. Unlike a fixed `Animal` endpoint, this bound changes when the substitution for T changes.

An *F-bound* constrains a parameter in terms of itself [Canning et al., 1989]. For example, `T : IComparable[T]` requires comparison with another value of the same T. A class `Item` implements `IComparable[Item]`, whose `CompareTo` method accepts another `Item`.

Self-reference becomes a difficulty when inference combines types. To isolate it, take two classes `Item` and `OtherItem` outside the animal hierarchy, so that `Animal` cannot mask the join. Each inherits only `object` and a comparison at its own type. This adapts Greenman et al.'s example [Greenman et al., 2014, Section 5.3]:

$$\texttt{Item} <: \texttt{IComparable[Item]}, \qquad \texttt{OtherItem} <: \texttt{IComparable[OtherItem]}.$$

For now, allow `IComparable` as an ordinary interface type. Both classes are below `IComparable[?]`. Because their type arguments are themselves below that bound, both are also below `IComparable[out IComparable[?]]`. Repeating this step adds another layer and gives a tighter common upper bound each time (Figure 2.7).

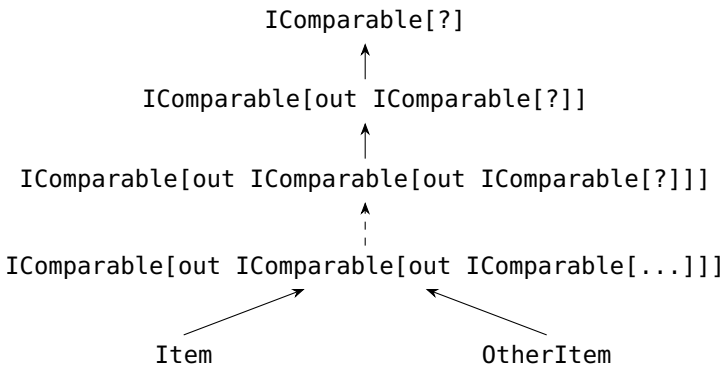


Figure 2.7: Common upper bounds become tighter downward; arrows point to supertypes. The ellipsis stands for further finite nesting.

Greenman et al. show the same absence of a least common upper bound for `Integer` and `Float` under a self-referential `Summable` [Greenman et al., 2014, Section 5.3]; the unrestricted `Item` example has the same structure, and any finite approximation loses precision. Java's `lub()` may instead produce an infinite type, represented with cyclic data structures and distinct from a recursive type [Oracle, 2025a, Section 4.10.4]. Runtime inference needs joins that it can compute during execution.

Material–shape separation preserves self-referential contracts while restricting where their heads may occur. Materials may be used as value types and type arguments. Shapes are nominal declarations used in inheritance and upper bounds, but may not themselves be value types or type arguments. In the separated system the comparison interface is a shape, so none of the `IComparable[...]` types in the figure is a material type, while `Item` and `OtherItem` remain materials. These two

otherwise unrelated materials therefore join at `object`. Greenman et al.'s Theorem 2 establishes computable joins for their material types with intersections [Greenman et al., 2014].

G# adopts this separation through explicit shape declarations (Figure 2.8), made precise in Section 3.7.1. `Item` can be a variable type or a `List` argument. Its comparison capability is expressed through the inherited shape, rather than through a shape-typed value.

```
shape interface IComparable[T] {
    int CompareTo(T other);
}
class Item : IComparable[Item] {
    /* comparison implementation omitted */
}
```

Figure 2.8: A G# shape declaration and a material class using it to constrain comparison.

2.8 Open-world assumptions

An *open world* permits new type declarations, including types introduced through loaded or generated code. G# fixes the supertypes and member contracts of every existing declaration: new classes may inherit existing classes or implement existing interfaces, but cannot change an old type's ancestry. This differs from an open type (Section 2.1).

Eligibility must remain valid beyond the currently known types. Figure 2.9 illustrates growth in the running hierarchy: a class such as `Parrot`, extending `Animal` and implementing `IPet` and `INamed`, may be added later.

So the ancestors, subtype relations and principal instantiations of existing types never change, but their set of implementations can grow, and enumerating today's implementations cannot establish all future candidates of an inference problem.

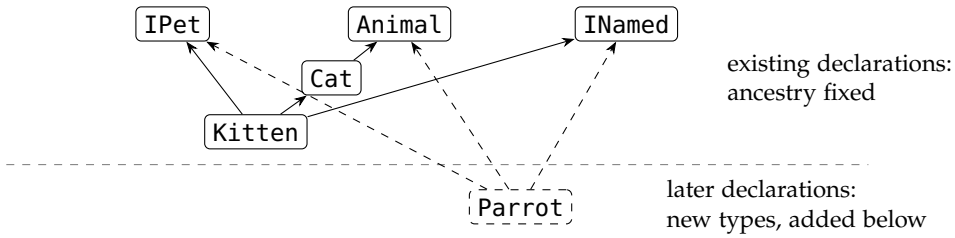


Figure 2.9: Growth is downward: `Parrot` may be added later without changing any existing edge. Arrows point to supertypes.

2.9 Compile-time and runtime inference

When a generic call omits its type arguments, inference must find an assignment that satisfies the call's typing requirements. Conventional *compile-time inference* uses the *static types* that type checking gives the argument expressions: local type inference collects subtype constraints from a call and chooses the substitution that minimises the instantiated result type [Pierce and Turner, 2000, Section 3.5], and C# fixes each unknown from bounds collected from the arguments [ECMA, 2023, Section 12.6.3]. G#'s type checking of an RI call computes its static type arguments, the *static assignment*, in the same way (Section 4.3). Some systems also use the expected type of the checking context [Odersky et al., 2001; Dunfield and Krishnaswami, 2021], so that the type arguments of a call depend on where it appears.

Admissible and optimal assignments Runtime inference chooses among the assignments that one execution of a call permits. An assignment is *admissible* when its type arguments are valid and satisfy their bounds, the argument values belong to the instantiated parameter types, and the instantiated result type stays within the result type that type checking gave the call, which is all the caller relies on (Section 4.4). Usually several assignments are admissible: for the `ToList` call on a list of cats, both `Cat` and `Animal` are. Each type parameter has a direction of preference, given by how the arguments constrain it (Section 2.5): a parameter bounded from below prefers a lower, more precise type, as `Cat` over `Animal`, one bounded from above prefers a higher one, and a fixed parameter, such as one that occurs as the argument of an invariant slot, as `T` in `List[T]`, prefers none of its type arguments to another, so it has an optimum only when a single type is admissible. An admissible assignment is *optimal*, the *optimum*, when it is at least as good as every other admissible assignment in every type parameter at once: assignments are compared parameter by parameter, in the *product order* of these preferences, with no parameter taking priority over another. This is stronger than being minimal, which only requires that no alternative is better (Section 2.4): two incomparable assignments can both be minimal, and then there is no optimum. *Selection* is the step of runtime inference that chooses the optimum.

Principal typing Three notions of principality are easily confused, and the guarantee of this thesis concerns the third. A *principal type scheme*, as in ML, is a polymorphic type of an expression of which every type it can be given is an instance [Hindley, 1969; Damas and Milner, 1982]. A language has *principal types*, also called minimal types [Ghelli and Pierce, 1998] in a sense unrelated to the minimal upper bounds of Section 2.4, when every typable expression can be given a most precise type, a subtype of every other type it can be given. Precision is that of the whole type, not of its type arguments: since `IHandler[Animal]` $<:$ `IHandler[Cat]`, the more precise type of a handler has the more general type argument. Principal types are not type schemes [Muehlboeck, 2019, Section 8.2.3]. With type-argument inference, a generic call has a principal type only if its inferred type arguments yield one. Muehlboeck calls a generic method whose invocations can always be given a principal type *prin-*

cipally inferable, a property of how the method is declared rather than of how it is used, and obtains *principal typing*, a principal type for every invocation, by checking that property at declarations [Muehlboeck, 2019, Section 8.2.3]. Jim uses the name principal typings for a different, stronger property, in which the types of a term's free variables are inferred as well [Jim, 1996, Section 1]; this thesis uses the term in Muehlboeck's sense.

G# ensures principal typing for RI calls. Signature checking decides from the declaration whether a method is eligible (Section 2.1), the counterpart of principal inferability, which in G# also requires a unique optimum at run time (Section 4.5); and an RI call's type is built from its arguments' types, never from the type its context expects. `Empty[T]()`, where no argument determines T and only an expected type could, is therefore not eligible, and every call gives its type argument explicitly (Section 4.6).

Chapter 3

The G# Language and Its Type System

This chapter defines the language over which the guarantee of Chapter 1 is stated, by following what runtime inference needs. In the running example, `ToList` receives a `List[Cat]` that its caller sees as an `IEnumerable[Animal]`, and must answer `Cat`. To read that answer, the value must carry its exact type, and no value may be missing or of type `Never` (Section 3.2). The runtime must find, in `List[Cat]`, the type argument at `IEnumerable` even when several inheritance paths lead there, and compare generic types whose arguments are projected (Section 3.3). A copy built at `Cat` behind the caller's `List[out Animal]` must not accept a `Dog`, so a projection restricts the members it exposes (Section 3.6). Several observations, a `Cat` and a `Dog`, must combine into one best type, their join (Section 3.5). Finally, these operations must terminate and their answers stay valid when new declarations are added, which restricts the declarations a program may contain (Section 3.7). The chapter gives in full the rules on which runtime inference builds, with an example before each, and describes the others, whose figures are in Appendix A; Chapter 4 gives the rules for the RI call.

3.1 Syntax

G# is a class-based language in the style of C#; Figure 3.1 gives the core that the rules cover. An overline denotes a possibly empty list, $\overline{T_i}$ for $T_1, \ldots, T_n$, and a part without the index is shared by every element, as the substitution in $\overline{T_i[\overline{X_j \mapsto T'_j}]}$. An overlined premise stands for one premise per index, lists that share an index have the same length, $|\overline{T_i}|$ is the length of a list, and a list is read as a set where one is expected, as in $X \in \overline{X_j}$. In a quantified formula such as $\forall x \in S.\ \ldots$, the dot ends the bound variables and their ranges, and the body after it extends as far to the right as possible. We omit empty type-argument and type-parameter brackets, the colon of an empty supertype list and an empty `: base()`, and we drop `object` from a class's supertype list, where it may be written as the class base but adds nothing, and a class that declares no constructor has the constructor C`() {}`.

A program $\mathcal{D}$ is a sequence of class and interface declarations, and execution starts at a static method `main`. An interface declared with `shape` is a *shape*, which may only be inherited and used in bounds; every other class or interface is a *material* (Section 3.7.1). A type parameter always has a bound; one written without a bound,

Class name C	Interface name I	Type parameter X
Field name f	Method name m	Variable x Integer literal z

Program	$\mathcal{D} ::= \overline{D_i}$
Declaration	$D ::= \texttt{class}\ C\texttt{[}\overline{P_i}\texttt{]}\ \texttt{:}\ \overline{T_j}\ \texttt{\{}\ \overline{M_l}\ \texttt{\}} \mid \texttt{interface}\ I\texttt{[}\overline{P_i}\texttt{]}\ \texttt{:}\ \overline{T_j}\ \texttt{\{}\ \overline{S_l}\ \texttt{\}}$ $\mid \texttt{shape interface}\ I\texttt{[}\overline{P_i}\texttt{]}\ \texttt{:}\ \overline{T_j}\ \texttt{\{}\ \overline{S_l}\ \texttt{\}}$
Type parameter	$P ::= X\ \texttt{:}\ T \mid \texttt{out}\ X\ \texttt{:}\ T \mid \texttt{in}\ X\ \texttt{:}\ T$
Method header	$H ::= T\ m\texttt{[}\overline{X_j\ \texttt{:}\ T_j^b}\texttt{]}\texttt{(}\overline{T_i\ x_i}\texttt{)}$
Member	$M ::= T\ f\texttt{;} \mid T\ f\ \texttt{=}\ e\texttt{;} \mid C\texttt{(}\overline{T_i\ x_i}\texttt{)}\ \texttt{:}\ \texttt{base(}\overline{e_j}\texttt{)}\ \texttt{\{}\ \overline{s_l}\ \texttt{\}}$ $\mid H\ \texttt{\{}\ \overline{s_i}\ \texttt{\}} \mid \texttt{static}\ H\ \texttt{\{}\ \overline{s_i}\ \texttt{\}}$
Interface member	$S ::= H\texttt{;}$
Head	$N ::= C \mid I$
Type	$T ::= X \mid N\texttt{[}\overline{a_i}\texttt{]} \mid T\ \texttt{\&}\ T \mid \texttt{int} \mid \texttt{bool} \mid \texttt{void} \mid \texttt{object} \mid \texttt{Never}$
Type argument	$a ::= T \mid \texttt{out}\ T \mid \texttt{in}\ T \mid \texttt{?}$
Variance	$v ::= \texttt{out} \mid \texttt{in} \mid \texttt{inv}$
Expression	$e ::= x \mid \texttt{this} \mid z \mid \texttt{true} \mid \texttt{false} \mid e\texttt{.}f \mid e\ \texttt{is}\ T \mid e\ \texttt{is}\ T\ x \mid \texttt{!}e$ $\mid e\ \texttt{\&\&}\ e \mid e\ \texttt{||}\ e$ $\mid r\texttt{.}m\texttt{[}\overline{T_j}\texttt{]}\texttt{(}\overline{e_i}\texttt{)} \mid r\texttt{.}m\texttt{(}\overline{e_i}\texttt{)} \mid \texttt{new}\ C\texttt{[}\overline{T_j}\texttt{]}\texttt{(}\overline{e_i}\texttt{)}$
Receiver	$r ::= e \mid C\texttt{[}\overline{T_j}\texttt{]}$
Receiver type	$::= T \mid \texttt{static}\ C\texttt{[}\overline{T_j}\texttt{]}$
Statement	$s ::= T\ x\ \texttt{=}\ e\texttt{;} \mid x\ \texttt{=}\ e\texttt{;} \mid e\texttt{.}f\ \texttt{=}\ e\texttt{;} \mid e\texttt{;}$ $\mid \texttt{\{}\ \overline{s_i}\ \texttt{\}} \mid \texttt{if (}e\texttt{)}\ s \mid \texttt{if (}e\texttt{)}\ s\ \texttt{else}\ s$ $\mid \texttt{while (}e\texttt{)}\ s \mid \texttt{return}\ e\texttt{;} \mid \texttt{throw}\ e\texttt{;}$
Type-parameter context	$\Delta ::= \emptyset \mid \Delta, X\ \texttt{:}\ T$
Variable context	$\Gamma, \Psi ::= \emptyset \mid \Gamma, x\ \texttt{:}\ T$
Closing	$\eta ::= \overline{X_j \mapsto T_j}$
Signature	$\sigma ::= \texttt{[}\overline{X_j : T_j^b}\texttt{]}\texttt{(}\overline{T_i}\texttt{)} \to T_r$

Figure 3.1: Core syntax of G#, with the contexts, closings and signatures used by its judgments.

as in `List[T]`, has the bound `object`. A class declares a method as a header with a body, and an interface as a header alone. A `static` method is called through its class rather than an object, and its body has no `this`. A head N names a class or an interface, including the built-in class `Array`, which every program contains implicitly and whose single parameter is invariant. We write `T[]` for `Array[T]`; `new T[](`e_1`,` e_2`)` constructs an array of length e_1 whose elements are all e_2, and e_1`[`e_2`]` indexes an array. The types `object` and `Never` are the top and bottom types of Section 2.2, written $\top$ and $\bot$. A type argument may be *projected* with `out`, `in` or `?`, as in `List[out Animal]` (Section 3.3.1); an unprojected type argument a, a type T, is *exact*, written $\mathrm{exact}(a)$, so a type argument that is not exact is projected, and an instantiation whose type arguments are all exact is exact. Type arguments are enclosed in square brackets; `<` and `>` are only comparisons.

A call names its method through a *receiver*: an expression, or a class for a static method, as in `Program.ToList(view)`; inside its own class, a static method may also

be called without the class name, which abbreviates the call through the class. An *explicit* call gives its type arguments, as in `Program.ToList[Animal](view)`, and an *RI call* omits them, as in `Program.ToList(view)`; a constructor call always gives them. A method without type parameters has an empty type-argument list, which our convention omits; hence $r.m(\overline{e_i})$ is an RI call only when m has type parameters. G# also has the array construction and indexing above, strings, arithmetic, comparison and equality operators, the Boolean connectives && and `||`, `for` loops, and `return;`, which leaves a `void` method early; they play no part in runtime inference and are left out of the figure. G# has no exception handling: `throw e;` halts the program, and e only supplies a message.

The running example Figure 3.2 is the program we use throughout the chapter. The call `Program.ToList(view)` is the copying call of Chapter 1: the caller sees `view` as an `IEnumerable[Animal]`, but the value it holds is a `List[Cat]`. `Shelter` handles cats and dogs through the contravariant `IHandler` of Figure 2.5; it shows a class that inherits one interface at two type arguments (Section 3.3.2).

```
class Shelter : IHandler[Cat], IHandler[Dog] {
    void Handle(Animal e) { /* handle e */ }
}
class Program {
    static List[out T] ToList[T](IEnumerable[T] source)
        { /* copies source into a new List[T] */ }
    static int main() {
        List[Cat] cats = new List[Cat](new Cat());  // fill
        cats.Add(new Cat());
        IEnumerable[Animal] view = cats;
        List[out Animal] copy = Program.ToList(view);
        bool exact = copy is List[Cat];       // true
        IHandler[Animal] h = new Shelter();
        return 0;
    }
}
```

Figure 3.2: The running example. The hierarchy and `IHandler` are those of Chapter 2; the complete program, with `List`, is in Appendix B.2.

Notation Primes, subscripts and superscripts distinguish uses of a metavariable, and a letter names a role: T^b is a bound, T^a an explicit type argument and T_r a result type. A method header $H = T_r\ m[\overline{X_j : T_j^b}](\overline{T_i\ x_i})$ has the signature $\sigma = \mathsf{sig}(H) = [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r$ (Figure 3.1) and the name $\mathsf{name}(H) = m$, and $\mathsf{name}(D)$ is the name that a declaration D declares. $T[\overline{X_j \mapsto T'_j}]$ is substitution, also of a type argument a_j for an X_j that occurs only as a type argument, and $\mathsf{ftv}(T)$, the *free type variables* of T, is the set of type parameters that occur in T. In a signature or a call,

j indexes the type parameters or type arguments and i the formal parameters or arguments, as in $r.m[\overline{T_j}](\overline{e_i})$.

Environments and judgments $\mathcal{D}$ is the program, Δ the type parameters in scope with their bounds and Γ the types of the variables in scope; we write $\mathsf{dom}(\Delta)$ for the type parameters in Δ, $\Delta(X)$ for the bound of X and $\Gamma(x)$ for the type of x. A closing η (Section 2.1) maps each type parameter in scope to a ground type. At run time it holds the type arguments with which the enclosing generic code runs, so applying it to a static type such as `List[T]` gives the ground type, such as `List[Cat]`, that the static type stands for in that execution (Section 3.4). A judgment lists, separated by bars, only the environments its rules consult: expression typing has the form $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T$, while subtyping and validity never read a variable and so omit Γ, as in $\mathcal{D} \mid \Delta \vdash T_1 <: T_2$. Operations defined from the declarations use the $\mathcal{D}$ and Δ of their judgment. Every judgment names its program $\mathcal{D}$ rather than fixing one program throughout, because the guarantee also covers the program after new declarations are added (Section 3.7.2), so each judgment must say in which program it holds.

Declarations Rules consult the program $\mathcal{D}$ only through the lookup judgments of Figure A.1. Let k range over the two kinds of head, mat for a material and shape for a shape (Section 3.7.1). We write $\mathcal{D} \vdash k\ N[\overline{P_i}] : \overline{T_j}\ \{\,\overline{B_l}\,\}$ when $\mathcal{D}$ declares the head N of kind k with type parameters $\overline{P_i}$, supertypes $\overline{T_j}$ and members $\overline{B_l}$, where B ranges over class members M and interface members S, and a rule writes only the parts it uses; $\mathcal{D} \vdash k\ N$ states only that $\mathcal{D}$ declares N of kind k. We write a type-parameter declaration P_i as $v_i\, X_i : T_i^b$, where v_i is inv when P_i carries no variance, and we write $\mathsf{vars}(N) = \overline{v_i}$ for the declared variances of N's slots. The index i names a slot throughout: X_i is the parameter declared for slot i and a_i the type argument at slot i of an instantiation $N[\overline{a_i}]$, which may be projected; we write $N[\overline{T_i}]$ when every argument is exact. We write $N = C$ or $N = I$ when the head N is a class or an interface. We write $\mathcal{D} \vdash N \ni m : \sigma$ when N declares a method m with signature σ. $\mathsf{arity}(\sigma) = |\overline{T_i}|$ is the number of formal parameters of σ.

3.2 Values and runtime types

A value is an integer, a Boolean, the Unit value of `void`, or a reference to an object, and every value has an exact runtime type (Section 2.6). An object created by `new C[`$\overline{T_j}$`](`$\overline{e_i}$`)` has the exact runtime type `C[`$\overline{T_j}$`]` whatever static type later refers to it, and runtime inference reads type arguments from these types (Section 4.11).

Non-null types Every type is non-null: a `Cat` variable holds an instance of `Cat` or of a subclass, since G# has no `null` value. Every argument that inference inspects therefore has a runtime type to read.

Initialisation Inference reads the runtime type of every argument it inspects, so no variable, field or array element may be read before it holds a value of its declared type. Locals and array elements receive a value when they are created, and definite initialisation, a flow analysis like C#'s definite assignment [ECMA, 2023, Section 9.4], ensures that every field of an object holds a value before it is read.

Keeping `Never` empty Because `Never` is below every type, a value of type `Never` could be used at any type, so G# must never produce one. Every construction therefore supplies its contents. A constructor that initialises a field of type T needs a T value, and no expression creates one from nothing, so such a class cannot be built at $T = \texttt{Never}$; an array construction supplies a fill value even for length zero, so no `Never[]` can be built either. A class whose constructor needs no T value can be built at `Never`, such as an empty sequence implementing `IEnumerable[Never]`. `Never` is thus a valid type argument, and runtime inference may return it, for instance $T = \texttt{Never}$ for a source that is such an empty sequence. Figure 3.3 gives such a sequence.

```
class NoItems : IEnumerator[Never] {
    bool MoveNext() { return false; }
    Never Current() { throw "no element"; }
}
class Nothing : IEnumerable[Never] {
    IEnumerator[Never] GetEnumerator() { return new
  NoItems(); }
}
IEnumerable[Animal] none = new Nothing();   // by covariance
```

Figure 3.3: An empty sequence at `Never`: no value of type `Never` is created, since `Current` can only halt.

3.3 Subtyping

The value of `view` is a `List[Cat]`, which the call sees as an `IEnumerable[Animal]`. Type checking must show `List[Cat]` $<:$ `IEnumerable[Animal]`, and runtime inference must read `Cat` at `IEnumerable`; both ask which type arguments `List[Cat]` guarantees at `IEnumerable`. We write $\mathcal{D} \mid \Delta \vdash T_1 <: T_2$ when T_1 is a subtype of T_2 under the type parameters Δ; for ground types, Δ is empty. Generic types are compared through their principal instantiations, whose type arguments are intervals, so we first define intervals (Section 3.3.1) and principal instantiations (Section 3.3.2) and then give the rules (Section 3.3.3). These definitions, together with disjointness and canonicalisation (Section 3.5), are mutually recursive, because intervals and principal instantiations compare their ends by subtyping. Properties of subtyping, such as the existence of the principal instantiations of an instantiation of a declaration, hold

only in a valid program (Section 3.7), and validation in turn uses subtyping to check bounds and member signatures.

3.3.1 Projections as intervals

We now make the interval account of Section 2.5 precise (Figure 2.6). A type argument a denotes an interval $[lo_a, hi_a]$ of exact type arguments, and p ranges over such intervals, with ends $p = [lo_p, hi_p]$: T denotes $[T, T]$, `out` T denotes $[\bot, T]$, `in` T denotes $[T, \top]$ and `?` denotes $[\bot, \top]$. The type arguments that a allows at a slot also depend on the slot's declared variance v, which makes the slot *covariant* (out), *contravariant* (in) or *invariant* (inv). A covariant slot keeps only the written upper end, since covariance accepts every exact type argument below it, a contravariant slot keeps only the written lower end, and an invariant slot keeps both. Hence `IEnumerable[out Cat]` and `IEnumerable[Cat]` denote the same type, and so do `IHandler[?]` and `IHandler[Never]`. The resulting *argument interval* $\mathsf{den}_v(a)$ is

$$\mathsf{den}_{\mathsf{out}}(a) = [\bot, hi_a], \qquad \mathsf{den}_{\mathsf{in}}(a) = [lo_a, \top], \qquad \mathsf{den}_{\mathsf{inv}}(a) = [lo_a, hi_a].$$

An interval p lies within an interval p' when its lower end is above, and its upper end below, the corresponding end of p':

$$\mathcal{D} \mid \Delta \vdash p \subseteq p' \iff \mathcal{D} \mid \Delta \vdash lo_{p'} <: lo_p \text{ and } \mathcal{D} \mid \Delta \vdash hi_p <: hi_{p'}.$$

Combining two instantiations of one head combines their intervals slot by slot. Table 3.1 combines the intervals p and p' of one slot according to its declared variance: the join is the smallest interval that contains both, and the meet keeps only the type arguments that both allow. For a head N, join_N and meet_N apply the two columns slot by slot to two lists of intervals of N, each slot by its declared variance, and to a set of such lists by combining all of its elements. The ends of the intervals are combined with the join $\sqcup$ and meet $\sqcap$ of types (Section 2.4), and Section 3.5 shows when they exist and how G# computes them. The Meet column joins lower ends, and under Δ the join of two ends that mention type parameters can be above the join at a closing: `X` $\sqcup$ `Y` is `object`, while at `X` $=$ `Cat` and `Y` $=$ `Animal` it is `Animal`. We call two types *stable* under Δ, $\mathsf{stable}_\Delta(T, T')$, when both are ground or one is a subtype of the other under Δ; their join is then the same at every closing. The meet of two intervals is therefore defined only when the ends it joins are stable; otherwise meet_N, and with it any inst (Section 3.3.2) that needs it, is undefined; join_N is defined whenever the joins of its ends are. An interval p contains T, written $T \in p$, when $lo_p <: T <: hi_p$; it is *empty* when $lo_p \not<: hi_p$, and it can be written as one type argument when it is a point $[T, T]$ (the type argument T), $[\bot, T]$ (`out` T), $[T, \top]$ (`in` T) or $[\bot, \top]$ (`?`).

3.3.2 Principal instantiation

Runtime inference must read one type argument at a generic head, even when several inheritance paths reach the head with different type arguments (Section 2.6).

Table 3.1: Combining one slot of two instantiations, with intervals p and p'.

Declared variance	Join	Meet
out	$[\bot,\ hi_p \sqcup hi_{p'}]$	$[\bot,\ hi_p \sqcap hi_{p'}]$
in	$[lo_p \sqcap lo_{p'},\ \top]$	$[lo_p \sqcup lo_{p'},\ \top]$
inv	$[lo_p \sqcap lo_{p'},\ hi_p \sqcup hi_{p'}]$	$[lo_p \sqcup lo_{p'},\ hi_p \sqcap hi_{p'}]$

Choosing one path would make the answer path-dependent, so G# combines them.

The principal instantiation answers one question: given a type T and a generic head N above it, which type arguments does T guarantee at N? For $T =$ `List[Cat]` and $N =$ `IEnumerable` the answer is `Cat`, since a `List[Cat]` is an `IEnumerable[Cat]`; as `IEnumerable` is covariant, the answer is the interval $[\bot, \texttt{Cat}]$ of type arguments it admits. N is a head, a class or interface name, not a type: the head of T itself or one that it inherits, so that T lies below some instantiation of N. When T is itself an instantiation, as `List[Cat]` is, we write N' for its own head, here `List`, to keep it apart from N. Figures A.2 and 3.5 define the answer through two judgments.

The *parents* of T, written $\mathsf{parents}(T) = \mathcal{T}$ for a set $\mathcal{T}$ of types (Figure A.2), are the types one step above it. For an exact instantiation $N'[\overline{T_i}]$, they are the declared supertypes of N' with its type arguments substituted for its parameters (Par-Inst); for example, `class List[T] : IEnumerable[T]` gives $\mathsf{parents}(\texttt{List[Cat]}) = \{\texttt{IEnumerable[Cat]}\}$. For an intersection, they are its conjuncts, since it lies below each of them (Par-And). For a type parameter X, the only parent is its bound $\Delta(X)$, which is all that is known above X (Sub-Param, Par-Param); under `T : IEnumerable[Cat]`, the parent of `T` is `IEnumerable[Cat]`. A built-in type has none (Par-Base).

The *heads* of a head N' are N' itself and the heads it inherits through its declared supertypes, and the heads of a type T are those of every instantiation reachable from T through parents:

$$\mathsf{heads}(N') = \{N'\} \cup \bigcup\{\mathsf{heads}(N'') \mid \mathcal{D} \vdash k\ N' : \overline{T'_j},\ T'_j = N''[\overline{a_l}]\}$$
$$\mathsf{heads}(T) = \bigcup\{\mathsf{heads}(N') \mid N'[\overline{a_i}] \in \mathsf{parents}^*(T)\},$$

where $\mathsf{parents}^*(T)$, the types reachable from T, contains T and $\mathsf{parents}^*(T'')$ for every T'' with $T'' \in \mathcal{T}$ and $\mathcal{D} \mid \Delta \vdash \mathsf{parents}(T) = \mathcal{T}$. The heads do not depend on type arguments: parents is not defined for a projected instantiation, so $\mathsf{parents}^*$ of it is the instantiation alone and its heads are those of its own head; an intersection has the heads of its conjuncts, a type parameter those of its bound, and a built-in type none.

For $N \in \mathsf{heads}(T)$, the *principal instantiation* of T at N is the narrowest instantiation $N[\overline{p_i}]$ that T guarantees, written $\mathsf{inst}(T, N) = \overline{p_i}$ (Figure 3.5). T is a subtype of $N[\overline{a_i}]$ when each p_i lies within the argument interval $\mathsf{den}_{v_i}(a_i)$, where $\mathsf{vars}(N) = \overline{v_i}$ (Section 3.3.3). The p_i are intervals rather than type arguments because projections and their combinations can give an interval, such as $[\texttt{Cat}, \texttt{Animal}]$ for `List[out`

`Animal] & List[in Cat]`, that no single type argument can express [Muehlboeck and Tate, 2018, Section 6.3].

The form of T selects the rule for inst. When T is itself an instantiation of N, Inst-Self reads its type arguments as intervals. When T is an instantiation of another head N' with a projected argument, Inst-Proj takes the join over the exact instantiations it covers. Otherwise Inst-Up walks up: each parent of T that reaches N supplies the intervals of its inheritance paths, and meet_N combines them slot by slot; a valid declaration needs the result to be non-empty at every slot (Section 3.7). For example, `List[Cat]` reaches `IEnumerable` through its supertype `IEnumerable[Cat]`, so $\mathsf{inst}(\texttt{List[Cat]}, \texttt{IEnumerable}) = [\bot, \texttt{Cat}]$, and `Shelter` combines its paths to `IHandler[Cat]` and `IHandler[Dog]` into $\mathsf{inst}(\texttt{Shelter}, \texttt{IHandler}) = [\texttt{Animal}, \top]$. Figure 3.4 draws both.

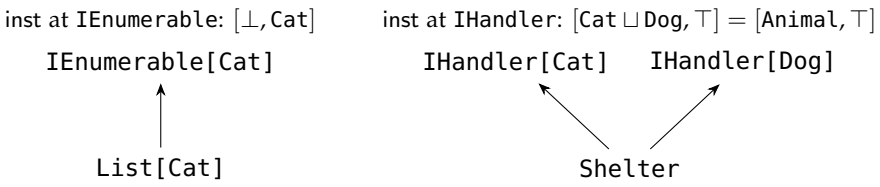


Figure 3.4: Principal instantiations. Left: one path to a covariant head gives an interval below the type argument. Right: two paths to the contravariant `IHandler` combine by Inst-Up; the meet at a contravariant slot keeps the join of the lower ends, giving the argument interval of `IHandler[Animal]`.

$\boxed{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N) = \overline{p_i}}$

$$\frac{\mathsf{vars}(N) = \overline{v_i}}{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(N[\overline{a_i}], N) = \overline{\mathsf{den}_{v_i}(a_i)}}\ \textsc{Inst-Self}$$

$$\frac{\begin{array}{c} N' \neq N \qquad \mathsf{vars}(N') = \overline{v_i} \qquad \exists i.\ \neg\mathsf{exact}(a_i) \\ \mathcal{S} = \{\overline{p_l} \mid \overline{T_i \in \mathsf{den}_{v_i}(a_i)},\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(N'[\overline{T_i}], N) = \overline{p_l}\} \\ \overline{\forall T_i \in \mathsf{den}_{v_i}(a_i).\ \exists \overline{p_l}.\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(N'[\overline{T_i}], N) = \overline{p_l}} \end{array}}{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(N'[\overline{a_i}], N) = \mathsf{join}_N\, \mathcal{S}}\ \textsc{Inst-Proj}$$

$$\frac{\begin{array}{c} N \in \mathsf{heads}(T) \qquad T \neq N[\overline{a_i}] \qquad T = N'[\overline{a_i}] \implies \forall i.\ \mathsf{exact}(a_i) \\ \mathcal{D} \mid \Delta \vdash \mathsf{parents}(T) = \mathcal{T} \\ \mathcal{S} = \{\overline{p_l} \mid T' \in \mathcal{T},\ N \in \mathsf{heads}(T'),\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(T', N) = \overline{p_l}\} \\ \forall T' \in \mathcal{T}.\ N \in \mathsf{heads}(T') \implies \exists \overline{p_l}.\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(T', N) = \overline{p_l} \end{array}}{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N) = \mathsf{meet}_N\, \mathcal{S}}\ \textsc{Inst-Up}$$

Figure 3.5: Principal instantiation of a type T at a head $N \in \mathsf{heads}(T)$; N' is the own head of an instantiation and $\mathcal{S}$ ranges over sets of lists of intervals.

Inst-Proj can give an interval that is not a point. Take a rack of lists, `class Rack[T] : List[List[T]]`, seen through the view `Rack[out Animal]`. The view

covers `Rack[Cat]`, `Rack[Dog]` and every other `Rack[S]` with $S <:$ `Animal`, whose `List` slots hold `List[Cat]`, `List[Dog]` and so on. No single type argument is held by all of them, so Inst-Proj gives the interval $[\bot, \texttt{List[out Animal]}]$ at `List`. The join is computed without enumerating the covered instantiations: each parameter of N' is replaced by the interval of its argument, and each supertype is then read with those intervals, written as a projection where the result is not a single type.

Signatures at a principal instantiation A method declared in N has a signature that mentions the type parameters of N, which we write X'_i for slot i. Called through a type T, they stand for the type arguments that T guarantees at N, the intervals $\overline{p_i}$ with $\mathsf{inst}(T, N) = \overline{p_i}$. When every p_i is a point, the signature is read by substitution: through `List[Cat]`, `Add` takes a `Cat`. An interval from a projection holds several type arguments, and the signature must be safe for each of them [King, 2017, Section 3.6.2]. A result may have any of them, so it is read at the upper end hi_{p_i}; a parameter must accept a value for each, so it is read at the lower end lo_{p_i}. Through `List[out Animal]`, whose interval is $[\bot, \texttt{Animal}]$, `GetEnumerator` returns an `IEnumerator[Animal]` and `Add` needs a `Never`, so it cannot be called; through `List[in Cat]`, `Add` accepts a `Cat` and `GetEnumerator` returns an `IEnumerator[object]`. Figure 3.6 shows these uses, of which the type checker rejects the `Add` through `List[out Animal]`.

```
List[Cat] cats = new List[Cat](new Cat());
List[out Animal] animals = cats;
IEnumerator[Animal] e = animals.GetEnumerator(); // reads: Animal
animals.Add(new Dog());           // rejected: Add takes a Never
List[in Cat] sink = new List[Animal](new Dog());
sink.Add(new Kitten());           // writes accept a Cat
IEnumerator[object] o = sink.GetEnumerator();    // reads: object
```

Figure 3.6: Members through projections: a covariant projection allows reads and a contravariant one writes.

Formally, the variance of a position composes the variances of the levels above it: out is neutral, $\mathsf{out} \circ v = v \circ \mathsf{out} = v$; two in levels cancel, $\mathsf{in} \circ \mathsf{in} = \mathsf{out}$; and inv absorbs, $\mathsf{inv} \circ v = v \circ \mathsf{inv} = \mathsf{inv}$. A type argument a at a slot of declared variance v contributes the *level variance* $\mathsf{lv}(a, v)$: the declared variance unless a projection overrides it,

$$\mathsf{lv}(T, v) = \mathsf{lv}(\texttt{?}, v) = v \qquad \mathsf{lv}(\texttt{out}\ T, v) = \mathsf{out} \qquad \mathsf{lv}(\texttt{in}\ T, v) = \mathsf{in}.$$

A position of variance v reads an interval p at the end

$$\mathsf{end}_{\mathsf{out}}(p) = hi_p \qquad \mathsf{end}_{\mathsf{in}}(p) = lo_p \qquad \mathsf{end}_{\mathsf{inv}}(p) = lo_p \ \text{ if } lo_p \equiv hi_p.$$

We write $\mathsf{at}_v^{\overline{p_i}}(T)$ for a type T read *at* a position of variance v through the list $\overline{p_i}$ of the intervals of N's slots: each X'_i in T is replaced by $\mathsf{end}_{v'}(p_i)$, where v' is the variance

of its own position. A signature reads its bounds and parameter types at in and its result at out, $\mathsf{view}(\sigma, \overline{p_i})$, as Ceylon reads a member at the principal instantiation of the type through which it is reached [King, 2017, Section 3.7.7]; Figure A.5 in Appendix A defines both. An occurrence at inv, such as the type `T` of a field, is both read and written, so unless $lo_{p_i} \equiv hi_{p_i}$, $\mathsf{at}^{\overline{p_i}}_{\mathsf{inv}}(\cdot)$ is undefined there and the member is unavailable; a field of type T_f reads as $\mathsf{at}^{\overline{p_i}}_{\mathsf{inv}}(T_f)$.

3.3.3 Subtyping rules

Figure 3.7 defines subtyping. Reflexivity, transitivity, top and bottom have their standard rules. We write $\mathsf{disjoint}(T)$ when the conjuncts of an intersection T can have no common subtype other than `Never`, in the program or after any addition of declarations; the test is sufficient, not complete, and holds when two of them are instantiations of classes neither of which inherits the other, as in `Cat & Dog`, since a class has one class base; or two different primitive types among `int`, `bool` and `void`, or a primitive type and a class or interface type; or two of them are ground and, at a common head N, meet_N of their principal instantiations has an empty interval at some slot, as for `List[Cat] & List[Dog]`. In the running example, $\mathsf{inst}(\texttt{List[Cat]}, \texttt{IEnumerable}) = [\bot, \texttt{Cat}]$, and $[\bot, \texttt{Cat}] \subseteq [\bot, \texttt{Animal}]$ gives `List[Cat] <: IEnumerable[Animal]` by Sub-Inst; likewise `List[Cat] <: List[out Animal]`, as $[\texttt{Cat}, \texttt{Cat}] \subseteq [\bot, \texttt{Animal}]$.

$\boxed{\mathcal{D} \mid \Delta \vdash T_1 <: T_2}$

$$\frac{}{\mathcal{D} \mid \Delta \vdash T <: T}\ \textsc{Sub-Refl} \qquad \frac{}{\mathcal{D} \mid \Delta \vdash T <: \top}\ \textsc{Sub-Top} \qquad \frac{}{\mathcal{D} \mid \Delta \vdash \bot <: T}\ \textsc{Sub-Bot}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T_1 <: T_2 \quad \mathcal{D} \mid \Delta \vdash T_2 <: T_3}{\mathcal{D} \mid \Delta \vdash T_1 <: T_3}\ \textsc{Sub-Trans} \qquad \frac{X \in \mathsf{dom}(\Delta)}{\mathcal{D} \mid \Delta \vdash X <: \Delta(X)}\ \textsc{Sub-Param}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T <: T_1 \quad \mathcal{D} \mid \Delta \vdash T <: T_2}{\mathcal{D} \mid \Delta \vdash T <: T_1 \,\&\, T_2}\ \textsc{Sub-And-R} \qquad \frac{i \in \{1, 2\}}{\mathcal{D} \mid \Delta \vdash T_1 \,\&\, T_2 <: T_i}\ \textsc{Sub-And-L}$$

$$\frac{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N) = \overline{p_i} \quad \mathsf{vars}(N) = \overline{v_i} \quad \overline{\mathcal{D} \mid \Delta \vdash p_i \subseteq \mathsf{den}_{v_i}(a_i)}}{\mathcal{D} \mid \Delta \vdash T <: N[\overline{a_i}]}\ \textsc{Sub-Inst}$$

$$\frac{\mathsf{disjoint}(T)}{\mathcal{D} \mid \Delta \vdash T <: \bot}\ \textsc{Sub-Disjoint}$$

Figure 3.7: Subtyping.

By Sub-Param, a type parameter is below its bound [Igarashi et al., 2001, Section 3.2]; by Sub-And-L, an intersection is below each conjunct; by Sub-And-R, a type below both conjuncts is below their intersection; and by Sub-Disjoint, an intersection of disjoint types is `Never`. Their meet is then $\bot$, as $\texttt{Cat} \sqcap \texttt{Dog} = \bot$ in Section 2.4: Sub-Disjoint gives $T <: \bot$ and Sub-Bot gives $\bot <: T$, so `Cat & Dog` is

the bottom type itself, not a type below it. The rule specific to G# is Sub-Inst, which compares a type with a class or interface type $N[\overline{a_i}]$, where $\overline{a_i}$ is empty when N is not generic. It looks up the principal instantiation of the source at N (Section 3.3.2) and requires each of its intervals to lie within the argument interval of the corresponding a_i (Section 3.3.1).

We write $\mathcal{D} \mid \Delta \vdash T_1 \equiv T_2$ when $T_1 <: T_2$ and $T_2 <: T_1$.

3.4 Valid types

A type that a program writes must be valid, and so must every type argument that runtime inference computes. Figure A.3 defines when a type is valid, $\mathcal{D} \mid \Delta \vdash T$, and when a type argument is valid at a slot of declared variance v, $\mathcal{D} \mid \Delta \vdash a\ \mathsf{ok}_v$. Type parameters in scope, the built-in types and intersections of valid types are valid. An intersection may be open (Section 2.1), as `T & IA` is. Inference may need an open intersection as its answer: for a parameter of type `IEnumerable[E]` and an argument of type `IEnumerable[A] & IEnumerable[B]`, with type parameters `A` and `B`, the only principal type argument is `A & B` [Muehlboeck and Tate, 2018, Section 2]; in G# it is the meet of the two covariant upper ends, which is always defined. An intersection written in a bound or a supertype, at any depth, must be ground, $\mathsf{gi}(T)$ (Valid-Decl and Valid-Sig): `U : IA & IB` is allowed, while `U : T & IA` and `class C[T] : D[T & IA]` are not. Bounds and supertypes fix what principal instantiation and the runtime read at every closing, and an open intersection there would canonicalise differently at each closing (Section 3.5). An instantiation $N[\overline{a_i}]$ is valid (Type-Inst) when N is a material with one parameter for each argument, each a_i is valid at the variance of its slot, and, if every a_i is exact, each lies below its bound with the type arguments substituted. An instantiation with a projected type argument is not checked against the bounds: every value has an exact runtime type, checked where it was constructed, so a covered instantiation that violates the bounds has no values and only makes the projection less precise.

A projection is valid only where the declared variance can express it: an exact type argument and `?` are valid at every slot, an `out` projection, whose interval reaches $\bot$, not at a contravariant slot, and an `in` projection not at a covariant one, so `IEnumerable[in Cat]` is invalid. Only an instantiation whose type arguments are all exact can be constructed. A shape (Section 3.1) is neither the type of a value nor a type argument; it may appear only as a bound or a supertype, $\mathcal{D} \mid \Delta \vdash T\ \mathsf{bound}$ (Figure A.4), which also accepts every valid type and intersections of bounds, and checks a shape instantiation as Type-Inst checks a material.

Valid closings A closing η of Δ (Section 3.1), applied to a type T by substitution as ηT, is *valid* in $\mathcal{D}$ when $\mathcal{D} \mid \varnothing \vdash \eta X$ and $\mathcal{D} \mid \varnothing \vdash \eta X <: \eta\, \Delta(X)$ for every X. A check on an open type, such as a member signature in a generic class, must hold at every valid closing.

3.5 Joins, meets and canonical types

When several arguments constrain one type parameter, inference combines them with the join $\sqcup$ or the meet $\sqcap$ of Section 2.4. G# computes both slot by slot on principal instantiations and represents the result as a canonical intersection. In Figure 3.8, both arguments of `Two` constrain `T`, which takes the join of their types.

```
static List[out T] Two[T](T a, T b) { ... }
List[out Animal] x = Program.Two(new Cat(), new Dog());
                                // T = Animal, the join
List[out Animal & IPet] y = Program.Two(new Kitten(), new Puppy());
                                // T = Animal & IPet
```

Figure 3.8: Joins of argument types, with `Puppy : Dog, IPet`: two unrelated classes join at their common class, and two pets at an intersection.

Canonicalisation, canon, rewrites an intersection into one representation, so that equivalent types are spelled alike and a unique assignment has one spelling; it also canonicalises the type arguments of every type. It flattens nested intersections and orders the conjuncts by a fixed total order, so `Animal & IPet` and `IPet & Animal` are one type; an intersection left with one conjunct is that conjunct, and one with none is $\top$. It merges conjuncts with the same generic head, combining their type arguments by the Meet column, whenever each combined interval can be written as one type argument, and otherwise keeps them apart: `IHandler[Cat] & IHandler[Dog]` becomes `IHandler[Animal]`, since a contravariant slot takes the join, while `List[out Animal] & List[in Cat]`, whose combined interval $[\texttt{Cat}, \texttt{Animal}]$ is not one type argument, stays as it is. It drops a conjunct above another, so `Cat & Animal` is `Cat`. Finally, it rewrites an intersection of disjoint types, such as `Cat & Dog` or `List[Cat] & List[Dog]`, to $\bot$ (Section 3.3.3). An open intersection is canonicalised in the same way, with each type parameter related only to itself, `object` and `Never`, so its canonical form does not depend on Δ; conjuncts whose meet is undefined stay apart, and a closing canonicalises the result again: `T & IA` is `Cat & IA` at `T` $=$ `Cat` and `IA` at `T` $=$ `IA`. Types that are equivalent only under Δ, such as `T & Cat` and `T` for `T` $<:$ `Cat`, keep two spellings, so for open types a unique assignment is unique up to $\equiv$ under Δ; ground types, and so every answer of runtime inference, have one spelling.

The meet is $T_1 \sqcap T_2 = \mathsf{canon}(T_1 \,\&\, T_2)$. The join takes every material head N that both T_1 and T_2 have, since a shape is never the type of a value, with $\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T_1, N) = \overline{p_i}$ and $\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T_2, N) = \overline{p'_i}$, combines the two lists by the Join column, and canonicalises the intersection of the results, and is undefined if one of these principal instantiations is undefined:

$$T_1 \sqcup T_2 = \mathsf{canon}\Big(\mathop{\&}_N N[\mathsf{join}_N(\overline{p_i}, \overline{p'_i})]\Big),$$

where an intersection with no conjuncts is $\top$, and $T_1 \sqcup T_2$ is the larger of the two when one is a subtype of the other. If every interval can be written as one type argument, the head gives one conjunct. Otherwise it gives two: in one, each such slot p is written $\texttt{out}\, hi_p$, and in the other $\texttt{in}\, lo_p$, so $\texttt{List[Cat]} \sqcup \texttt{List[Animal]} = \texttt{List[out Animal] \& List[in Cat]}$.

For the invariant `List`, the join of the points `Cat` and `Dog` is $[\bot, \texttt{Animal}]$, since $\texttt{Cat} \sqcap \texttt{Dog} = \bot$, so $\texttt{List[Cat]} \sqcup \texttt{List[Dog]} = \texttt{List[out Animal]}$; the common head `IEnumerable` contributes `IEnumerable[Animal]`, which canonicalisation drops as redundant.

The meet of valid types always exists and is valid, since conjuncts whose meet is undefined stay apart. Their join exists and is valid whenever the principal instantiations it combines are defined, which holds for every ground type and every instantiation of a declaration. A type parameter is read, as in Greenman et al., as an abstract class below its bound [Greenman et al., 2014, Section 5.4], so under Δ the join is least among the types that Δ can express; a join of types that are not stable (Section 3.3.1) is used only where a larger type is safe. G# has no union types, because a union has no single head at which the runtime could read a type argument, as `X f[X](Int | X x)` applied to an `Int` shows (Section 6.2); a join can therefore be less precise than a union (Figure 2.3), and since existing types never gain supertypes, added declarations cannot change it. Joins and meets are unique up to equivalence, commutative and associative, so $\bigsqcup$ and $\bigsqcap$ over a set of types are well defined. Disjointness must likewise survive added declarations, so it is sound but incomplete: the meet of two types is $\bot$ only when no valid new declaration can be below both. Two non-generic interfaces are never disjoint, since a new class may implement both (Figure 2.9).

3.6 Expression typing

`ToList` may build a `List[Cat]` behind the caller's `List[out Animal]`, so the caller must not be able to add a `Dog` to it: the members that a type exposes must depend on its projections. G# is checked bidirectionally with the judgments $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T$ and $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow T$ of Section 2.3; an expression is *well typed* when one of them derives a type for it, and *ill typed* otherwise, while types, declarations and programs are *valid* (Sections 3.4 and 3.7). This section gives the rules on which inference depends; the other forms, such as blocks, which extend Γ with their locals, and `return e`, which checks `e` against $\Gamma(\mathsf{ret})$, have their usual rules (Figure A.9).

3.6.1 Members

A call or field access is typed by its member as seen through the receiver's type. We write $\mathcal{D} \mid \Delta \vdash T.m : \sigma$ when a method m has the signature σ through a receiver of type T. A method can be declared by several heads of T, since a class that overrides or implements a method declares it again; the declaration that counts is the nearest one, which refines those above it (Section 3.7.1), so a call through a class uses the

class's own declaration of a method rather than that of an interface it implements. For a method name m and a number n of parameters, let $\mathcal{N}_{m,n}$ be the heads of T that declare it; near keeps those that no other head in $\mathcal{N}_{m,n}$ inherits, where $N \in \mathsf{heads}(N')$ means that N' inherits N:

$$\mathcal{N}_{m,n} = \{N \in \mathsf{heads}(T) \mid \exists \sigma.\ \mathcal{D} \vdash N \ni m : \sigma \wedge \mathsf{arity}(\sigma) = n\}$$
$$\mathsf{near}(T, m, n) = \{N \in \mathcal{N}_{m,n} \mid \nexists N' \in \mathcal{N}_{m,n}.\ N' \neq N \wedge N \in \mathsf{heads}(N')\}.$$

If a class C implements an interface I and both declare m, then $\mathcal{N}_{m,n} = \{C, I\}$ and C inherits I, so near $= \{C\}$; for `IPet & INamed`, if both interfaces declared a method `Greet`, both would stay, since neither inherits the other.

A head N that declares m with signature σ declares it with the type parameters of N; through T, these stand for the intervals $\mathsf{inst}(T, N) = \overline{p_i}$, so the member has the signature $\mathsf{view}(\sigma, \overline{p_i})$ of Section 3.3.2. When one head is nearest, Mem-View (Figure 3.9) gives the member that signature. Through inst, an inherited member has one signature and a type parameter has the members of its bound. For example, through `List[out Animal]`, Mem-View gives `Add` the parameter type `Never`, so it cannot be called, and through `List[Cat]` a `Cat`.

$\boxed{\mathcal{D} \mid \Delta \vdash T.m : \sigma}$

$$\frac{\mathcal{D} \vdash N \ni m : \sigma \qquad \mathsf{near}(T, m, \mathsf{arity}(\sigma)) = \{N\} \qquad \mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N) = \overline{p_i}}{\mathcal{D} \mid \Delta \vdash T.m : \mathsf{view}(\sigma, \overline{p_i})}\ \textsc{Mem-View}$$

$$\frac{\begin{array}{c}\mathsf{near}(T, m, n) = \{\overline{N_l}\} \qquad |\overline{N_l}| \geq 2 \\ \forall l.\ \mathcal{D} \vdash N_l \ni m : \sigma_l \ \wedge\ \mathsf{arity}(\sigma_l) = n \ \wedge\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N_l) = \overline{p_{l,h}} \\ \wedge\ \mathsf{view}(\sigma_l, \overline{p_{l,h}}) = [\overline{X_j : T^b_{l,j}}](\overline{T_{l,i}}) \to T_{l,r} \\ \forall l, l', i, j.\ \mathsf{stable}_{\Delta'}(T_{l,i}, T_{l',i}) \wedge \mathsf{stable}_{\Delta'}(T^b_{l,j}, T^b_{l',j})\end{array}}{\mathcal{D} \mid \Delta \vdash T.m : [\overline{X_j : \bigsqcup_l T^b_{l,j}}](\overline{\bigsqcup_l T_{l,i}}) \to \bigsqcap_l T_{l,r}}\ \textsc{Mem-Combine}$$

Figure 3.9: Methods through a receiver; Mem-Combine renames the type parameters of every σ_l to one list $\overline{X_j}$, and Δ' is Δ extended with them, bounded as in σ_1.

When several heads, none inheriting another, declare a method with the same name and number of parameters, as the conjuncts of `IPet & INamed` may, the member must still have one type, or its type would depend on which head was chosen. Inference itself produces such receivers: the join of two types can be an intersection such as `Animal & IPet` (Section 3.5). Mem-Combine joins their parameter types and method bounds and meets their results, under Δ', Δ extended with the renamed type parameters $\overline{X_j}$, so heads that declare different numbers of type parameters do not combine. Since a class implementing all the heads gives each method the bounds of every declaration (Refine-Iface), the joined bounds are equal wherever an implementing class exists. This is safe because G# has neither overloading on

parameter types nor explicit interface implementation, so a class implementing all of them supplies one body, as validation checks (Section 3.7): its parameter lies above every declared parameter type, and so above their join, and its result lies below every declared result. For example, if `IPet` declared `void Greet(Cat c)` and `INamed` declared `void Greet(Dog d)`, a receiver of type `IPet & INamed` would accept any `Animal` for `Greet`, since `Cat` ⊔ `Dog` = `Animal`. Figure 3.10 shows the declarations and such a call. A field is looked up in the same way by Mem-Field, read at inv, so a field whose type mentions a projected slot is unavailable through that projection.

```
interface IPet { void Greet(Cat c); }
interface INamed { void Greet(Dog d); }
static void Visit(IPet & INamed p) {
    p.Greet(new Animal());   // takes Cat join Dog = Animal
}
```

Figure 3.10: A member declared by two heads, in a variant of `IPet` and `INamed`: through their intersection, Mem-Combine joins the parameter types.

A join of types that are not stable is not safe. Take the receiver `IP[X] & IN[Y]`, whose heads declare `Greet(X x)` and `Greet(Y y)`: the join of `X` and `Y` under Δ is `object`, so a combined `Greet` would accept any object. At `X` = `Cat` and `Y` = `Dog`, however, the receiver may be a `Pet`, which implements `IP[Cat]` and `IN[Dog]` with the one body `Greet(Animal a)`, and an object that is not an `Animal` would reach it (Figure 3.11). The member is type-checked once, for every closing, so its parameter type would have to lie below the join at each closing; `X` and `Y` both do, but neither is below the other, and taking one would make typing choose. Mem-Combine therefore joins only stable types (Section 3.3.1), and `Greet` is not available through `IP[X] & IN[Y]`; a caller can still view the receiver as an `IP[X]` and pass an `X`.

```
interface IP[X] { void Greet(X x); }
interface IN[Y] { void Greet(Y y); }
class Pet : IP[Cat], IN[Dog] {
    public void Greet(Animal a) { ... }
}
static void Visit2[X, Y](IP[X] & IN[Y] p, X x, object o) {
    p.Greet(o);              // rejected: no Greet through p
    IP[X] q = p;
    q.Greet(x);              // accepted
}
```

Figure 3.11: A member declared by two heads with type parameters: `X` and `Y` are not stable, so `Greet` is not available through `IP[X] & IN[Y]`, but is through `IP[X]`.

3.6.2 Checking and type tests

An expression is checked by T-Sub against every supertype of its synthesised type, and field reads and constructions have the usual rules (Figures A.6 and A.7). A class named as a receiver, as in `Program.ToList(view)`, has the receiver type static $C[\overline{T_j}]$ (Recv-Class) and reaches only the static methods of C (Mem-Static); every other member rule finds only instance methods.

A *test* is an expression `e is T`, `e is T x`, `!e`, `e && e` or `e || e`. It also records the bindings Ψ available where it is true and Ψ' where it is false, $\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi'$ (Figure A.8). `e is T` compares the value's runtime type with T under the running code's type arguments and binds nothing (T-Is), any other Boolean expression has no bindings (T-Cond), `e is T x`, the only downcast in G#, binds $x{:}T$ where it succeeds (T-Bind), `!` exchanges the two sets (T-Not), and a test used as a value drops them (T-Test). `if` checks each branch under the bindings of its outcome and `while` its body under the true bindings (Figure A.9), so the `else` branch of `if (!(a is Cat c))` sees `c`; `&&` and `||` pass bindings on (T-And, T-Or) as in occurrence typing [Tobin-Hochstadt and Felleisen, 2010, Sections 3.2 and 4.2] and C# [ECMA, 2023, Sections 9.4.4.26–9.4.4.28 and 9.4.4.34]. Figure 3.12 shows these cases on the running example.

```
Animal a = new Kitten();
if (a is Cat c) { cats.Add(c); }       // c : Cat where the test holds
if (!(a is Dog d)) { } else { h.Handle(d); }  // else sees d
bool b = a is Cat k;                   // a value binds nothing
```

Figure 3.12: Type tests and their bindings, with `cats` and `h` of Figure 3.2.

The other statements have the usual rules: a local declaration has a valid type, a binding may shadow an outer variable but not reuse a name already bound on the same path, a restriction we state here rather than as a rule, and `return e;` checks e against the method's result type.

3.6.3 Explicit generic calls

T-Call-Explicit (Figure 3.13) types a call $r.m[\overline{T_j^a}](\overline{e_i})$ that gives its type arguments. It types the receiver r and looks up the signature $[\overline{X_j : T_j^b}](\overline{T_i}) \to T_r$ of m through the receiver's type (Section 3.6.1). With the type arguments substituted for $\overline{X_j}$, each T_j^a must be valid and below its bound, which may name any of the parameters, each e_i checks against its formal parameter type, and the result type is the call's type. For `Program.ToList[Animal](view)`, with `List[out T] ToList[T](IEnumerable[T] source)`, `Animal` is valid and below the bound `object`, `view` checks against `IEnumerable[Animal]`, and the call has type `List[out Animal]`. If a premise fails, the call is ill typed: `Program.ToList[Cat](view)` is, since `view`, an `IEnumerable[Animal]`, does not check against `IEnumerable[Cat]`.

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash r \Rightarrow T \qquad \mathcal{D} \mid \Delta \vdash T.m : [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r \qquad \overline{\mathcal{D} \mid \Delta \vdash T_j^a} \qquad \overline{\mathcal{D} \mid \Delta \vdash T_j^a <: T_j^b[\overline{X_l \mapsto T_l^a}]} \qquad \overline{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_i \Leftarrow T_i[\overline{X_j \mapsto T_j^a}]}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash r.m[\overline{T_j^a}](\overline{e_i}) \Rightarrow T_r[\overline{X_j \mapsto T_j^a}]}\ \text{T-Call-Explicit}$$

Figure 3.13: Explicit generic calls.

An RI call $r.m(\overline{e_i})$ omits $\overline{T_j^a}$; its static assignment (Sections 2.9 and 4.3) takes their place in typing, and Chapter 4 gives its rule.

3.7 Valid declarations and programs

The operations of this chapter are defined for every type, but some declarations would make them fail: a type that grows without end as its supertypes are taken, so that an operation never terminates; a class that reaches one generic interface at incompatible type arguments, so that its principal instantiation is empty; a member that contradicts the declared variance of its interface; or a class with two class bases. Figure 3.14 shows one declaration of each kind, together with the accepted `Shelter`. *Validation* rejects, independently of any call, every declaration that breaks a rule of this section: a program is valid when every declared head meets the conditions of Section 3.7.1 and the bodies of its classes are well typed by the rules of Section 3.6 (Section 3.7.2).

```
shape interface D[out T] { }
interface IBox[T] { }
class C[T] : D[C[C[T]]] { }          // rejected: a growing cycle
interface IOut[out T] { void Put(T x); }
                                     // rejected: out T in a parameter
class Pet : Cat, Dog { }             // rejected: two class bases
class TwoBoxes : IBox[Cat], IBox[Dog] { }
                                     // rejected: empty at IBox
class Shelter : IHandler[Cat], IHandler[Dog] { ... }
                                     // accepted: [Animal, object]
```

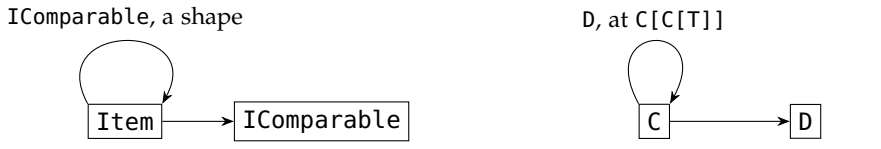
Figure 3.14: Declarations that validation rejects, by the growing-cycle condition (Section 3.7.1), the variance check (Section 3.7.1) and consistent inheritance (Valid-Inh), and one that it accepts.

3.7.1 Valid declarations

A head must respect its bounds, supertypes and declared variances.

Bounds A bound, as in `U : IEnumerable[T]` or `T : IComparable[T]`, restricts a parameter's type arguments to subtypes of the bound [Cardelli and Wegner, 1985, Section 6.2]; an F-bound (Section 2.7) mentions the parameter itself. Dependency cycles among bounds, such as `[T : U, U : T]`, are rejected: bounds are *acyclic*, $\mathsf{acyclic}(\overline{X_i : T_i^b})$, when the relation that links X_i to each $X_j \neq X_i$ in $\mathsf{ftv}(T_i^b)$ has no cycle.

Materials, shapes and cycles ($\mathsf{usage}(\mathcal{D})$) G# adopts Material–Shape Separation [Greenman et al., 2014] (Section 2.7). Two conditions make type operations terminate. The *usage graph* $\mathsf{usage}(\mathcal{D})$ records the inheritance and type arguments that these operations walk: its nodes are heads, with an edge $N \to N'$ when $\mathcal{D} \vdash k\ N[\overline{P_i}] : \overline{T_j}$ and $T_j = N'[\overline{a_l}]$ for some j, and an edge $N \xrightarrow{N'} N''$, labelled N', when moreover N'' is the head of an instantiation that occurs in $\overline{a_l}$. A type parameter is also a node, whose bound counts as its inheritance clause and which counts as a head where it occurs in $\overline{a_l}$ [Greenman et al., 2014, Section 5.4]. The first condition is the separation: the graph has no cycle once the edges labelled by shapes are removed, which makes both walks terminate [Greenman et al., 2014, Sections 3.2–3.3 and 5.1]. So a bound mentions its own parameter only under a shape, as in `T : IComparable[T]`: `class C[T : C[T]]` gives an edge from `T` to itself labelled by the material `C`. Type arguments can still grow along inheritance: with a shape `D`, `class C[T] : D[C[C[T]]]` passes the separation, yet `C[T]` reaches `C[C[T]]` and so on. The second condition rejects a cycle with a *growing* edge, one whose instantiation has an argument other than a type parameter or a head without arguments, which makes the *inheritance closure* of finitely many types, the types reachable by taking supertypes and type arguments, finite [Kennedy and Pierce, 2007, Theorem 9]. The graph is *finitary*, $\mathsf{finitary}(\mathsf{usage}(\mathcal{D}))$, when it meets both conditions. Figure 3.15 shows the usage graphs of `Item` and `C`.



`class Item : IComparable[Item]`
accepted: the cycle is labelled by a shape

`class C[T] : D[C[C[T]]]`
rejected: the cycle grows

Figure 3.15: Usage graphs. Both cycles pass the separation, since their labels are shapes; the second has a growing edge, whose instantiation `C[C[T]]` has an argument other than a type parameter.

Occurrences and variance ($\mathsf{respects}(N)$) Whether an occurrence of a type parameter is covariant, contravariant or invariant (Section 2.5) depends on the slots on the way to it. We record the way as an *occurrence path* ρ, a sequence of *levels* from the outside in: (N, i, v) enters slot i of an instantiation of N, where v is the slot's level variance (lv, Section 3.3.2), and $(\&, i, \mathsf{out})$ enters conjunct i of an intersection, which is covariant in

each conjunct; we write $\& \in \rho$ when ρ has such a level. ϵ is the empty path, and $\ell\rho$ puts the level ℓ in front of ρ. The variance of an occurrence composes the variances of its levels, and $\mathsf{occ}(T)$ is the set of triples (Y, ρ, v), one for each occurrence of a type parameter Y in T with path ρ and variance v:

$$\mathsf{occ}(X) = \{(X, \epsilon, \mathsf{out})\}$$
$$\mathsf{occ}(T_1 \,\&\, T_2) = \{(Y, (\&, i, \mathsf{out})\rho, v) \mid i \in \{1,2\},\ (Y, \rho, v) \in \mathsf{occ}(T_i)\}$$
$$\mathsf{occ}(N[\overline{a_i}]) = \{(Y, (N, i, v_i')\rho, v_i' \circ v) \mid (Y, \rho, v) \in \mathsf{occ}(a_i)\},$$

where $\mathsf{vars}(N) = \overline{v_i}$ and $v_i' = \mathsf{lv}(a_i, v_i)$, with $\mathsf{occ}(\texttt{out}\ T) = \mathsf{occ}(\texttt{in}\ T) = \mathsf{occ}(T)$, and occ empty for `?` and the built-in types. For example, $\mathsf{occ}(\texttt{T \& Cat}) = \{(\texttt{T}, (\&, 1, \mathsf{out}), \mathsf{out})\}$, and

$$\mathsf{occ}(\texttt{IEnumerable[List[T]]}) = \{(\texttt{T}, (\texttt{IEnumerable}, 1, \mathsf{out})(\texttt{List}, 1, \mathsf{inv}), \mathsf{inv})\}.$$

A type in a declaration also has a variance from its position, its *root variance* v_0: out for a result type or a supertype, in for a method parameter or a method bound, and inv for a field. $\mathsf{pos}(N)$ is the set of pairs (T, v_0) of the types of N's supertypes and instance members with their roots; constructor parameters and static members are not in it, since the class's type parameters cannot vary there [Emir et al., 2006, Section 2.3]. A head N with type parameters $\overline{v_i\, X_i}$ *respects* its declared variances when, composed with its root, every occurrence of a covariant parameter has variance out and every occurrence of a contravariant one variance in:

$$\mathsf{respects}(N) \iff \forall (T, v_0) \in \mathsf{pos}(N),\ (X_i, \rho, v) \in \mathsf{occ}(T).\ v_i \neq \mathsf{inv} \implies v_0 \circ v = v_i.$$

For example, T in a parameter of type `IEnumerable[T]` has variance $\mathsf{in} \circ \mathsf{out} = \mathsf{in}$, and in a result of type `List[T]` it has $\mathsf{out} \circ \mathsf{inv} = \mathsf{inv}$, so a covariant T may appear in neither.

Inheritance and members ($\mathcal{D} \mid \Delta \vdash N$ inherits) Without overloading on parameter types or explicit interface implementation, a class C supplies one body for each method name m and number n of parameters, the nearest along its class bases, whose signature read in C is $\mathsf{impl}(C, m, n)$. We write $\mathcal{D} \mid \Delta \vdash \sigma \leq_N \sigma'$ when σ may implement or override a signature σ' declared in N (Figure A.10). By Refine-Class, an override of a class method keeps its signature exactly. By Refine-Iface, a body implementing a declaration of an interface I that C inherits, read at the principal instantiation of I in C, may instead accept supertypes of its parameter types and return a subtype of its result type.

The inheritance of N is consistent, $\mathcal{D} \mid \Delta \vdash N$ inherits, by Valid-Inh (Figure A.11) when each supertype is a single instantiation, with exact type arguments, of a class or interface other than `Array`, so not a primitive type, `Never`, a type parameter or an intersection, while a class base `object` adds nothing and is dropped; when a class has at most one class base and any number of interfaces, and an interface inherits only in-

terfaces; when every principal instantiation $\mathsf{inst}(N[\overline{X_i}], N'')$ is defined and non-empty at each slot; and, for a class, when the body it supplies for each method refines every declaration of that method in its supertypes, read through view. Inheritance paths must therefore agree at every invariant slot: the contravariant slot of `IHandler` combines the paths of `Shelter` to `IHandler[Cat]` and `IHandler[Dog]` into $[\texttt{Animal}, \top]$, but a class implementing `IBox[Cat]` and `IBox[Dog]`, for an invariant `IBox[T]`, is rejected, since its paths meet in an empty interval. Paths that carry type parameters meet only at stable ends (Section 3.3.1): if `IA[T]` and `IB[U]` inherit the contravariant `ICons[T]` and `ICons[U]`, then `IC[T, U] : IA[T], IB[U]` is rejected, since `T` and `U` are not stable, while `ID[T, U : T] : IA[T], IB[U]` is accepted, since `U` lies below `T`. `Array` is excluded because the runtime creates arrays with their own contiguous storage and `Array` has no constructor that a subclass could call.

3.7.2 Valid programs

Bodies ($\mathcal{D} \mid \Delta \vdash C$ bodies) Valid-Body (Figure A.12) checks each member of a class C by the typing rules of Section 3.6, with RI calls typed by T-Call-RI (Chapter 4). An instance member is checked with `this` bound to $C[\overline{X_i}]$, and a method body also with its parameters and the pseudo-variable ret, its result type, against which `return e;` checks e; a `static` method has no `this`. A method with a result must end every path in a `return` or `throw`, $\mathsf{returns}(\bar{s})$, so `Never F() {}` is rejected. A constructor checks its `base` call as a construction of the class base with only its parameters in scope, since the object is not yet initialised, and its body as that of a `void` method. Definite initialisation, $\mathsf{init}(C)$, checks that every field is assigned before it is read (Section 3.2).

Programs and declarations ($\vdash \mathcal{D}$, $\mathcal{D} \vdash N$ valid) By Valid-Prog (Figure A.13), every head is valid, names are unique and the usage graph is finitary. Valid-Decl checks a head against Section 3.7.1, its signatures, static or not, by Valid-Sig, which requires acyclic, valid and ground-intersection bounds and valid parameter and result types; it also checks the types of its fields and constructor parameters and its bodies, and requires distinct members, $\mathsf{distinct}(\overline{B_l})$: no two methods share name and number of parameters, no two constructors their number of parameters and no two fields their name. For the open world of Section 2.8, we write $\mathcal{D} \sqsubseteq \mathcal{D}'$ when $\vdash \mathcal{D}$, $\vdash \mathcal{D}'$ and $\mathcal{D}'$ only adds fresh declarations to $\mathcal{D}$. As no existing supertype, variance or bound changes, every judgment of this chapter that holds in $\mathcal{D}$ holds in $\mathcal{D}'$, and new types never appear above existing ones (Figure 2.9), while constructing `List[Cat]` adds none.

Chapter 4

Runtime Type-Argument Inference

Chapter 3 typed the calls that give their type arguments (T-Call-Explicit) and left to this chapter the RI call, which omits them and whose type arguments runtime inference chooses. The three stages of Figure 1.1 establish that every RI call has a unique optimal assignment and that runtime inference finds it. Section 4.1 shows why a unique optimum must be required, and Section 4.2 follows one call through the stages. Section 4.3 and Section 4.4 define what the stages share: how a type parameter is read from an argument, the static assignment, admissible and optimal assignments, and the eligibility contract. The rules of each stage follow in turn: signature checking decides eligibility (Section 4.5), type checking types an RI call of an eligible method (Section 4.6), with the positions where an intersection blocks inference collected in Section 4.7, and runtime inference infers the assignment (Section 4.8) when the call executes (Section 4.9). Section 4.10 states what the rules guarantee and what they rely on, and Section 4.11 how the compiled code follows the rules.

4.1 The need for a unique optimum

An assignment $\overline{\theta_j}$ gives θ_j to each type parameter X_j; we also write θ_X for $X = X_j$, $T[\overline{\theta_j}]$ for $T[\overline{X_j \mapsto \theta_j}]$ and $\overline{\tau_j}$ for the static assignment. An RI call may have several admissible assignments (Section 2.9), and runtime inference is well defined only if one of them is the optimum; Section 4.4 makes these notions precise.

Apart from a fixed one, each type parameter prefers either a smaller or a larger type argument, according to how its formal parameter types use it (Section 2.9). A bound can tie together two parameters that prefer opposite directions. In `Link` (Figure 4.1), `T` occurs only in the source and prefers a smaller type argument, while `U` occurs only in the sink and prefers a larger one. For an `IEnumerable[Cat]` and an `IHandler[Animal]`, the assignments $(\texttt{Cat}, \texttt{Cat})$ and $(\texttt{Animal}, \texttt{Animal})$ are admissible and incomparable, and every assignment better than both, such as $(\texttt{Cat}, \texttt{Animal})$, violates the bound `U` <: `T`. This call has no optimum, so `Link`'s signature cannot guarantee one: `Link` is not eligible, and every call must specify its type arguments explicitly, as `Link[Cat, Cat]` or `Link[Animal, Animal]`.

The optimum may differ from the static assignment. In `Pair` (Figure 4.2), both sources are seen as `IEnumerable[Animal]`, so $\overline{\tau_j} = (\texttt{Animal}, \texttt{Animal})$, but at run time they are a `List[Cat]` and a `List[Dog]`. The bound `U` <: `T` forces `T` up to `Animal`, because `Dog` is not below `Cat`, while `U` can be `Dog`. The optimum $(\texttt{Animal}, \texttt{Dog})$ is strictly better than $\overline{\tau_j}$ in `U`.

```
static void Link[T, U : T](IEnumerable[T] src,
                           IHandler[U] sink) { ... }
IEnumerable[Cat] cats = ...;
IHandler[Animal] animals = ...;
Program.Link(cats, animals);        // not eligible
```

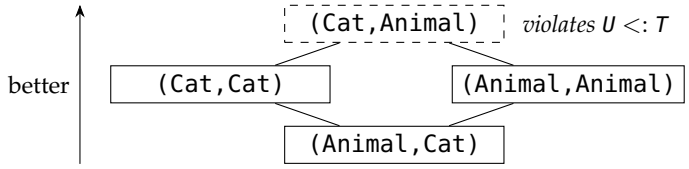


Figure 4.1: A call with no optimum, and its assignments (T, U) ordered by preference, better ones higher. Solid boxes are admissible and dashed ones are not; assignments that use other types lie below these. The two highest admissible assignments are incomparable, and the one above both violates U <: T.

```
interface IPair[out T, out U] { }
static IPair[T, U] Pair[T, U : T](IEnumerable[T] a,
                                  IEnumerable[U] b) { ... }
IEnumerable[Animal] a = ...;   // holds a List[Cat]
IEnumerable[Animal] b = ...;   // holds a List[Dog]
IPair[Animal, Animal] r = Program.Pair(a, b);
```

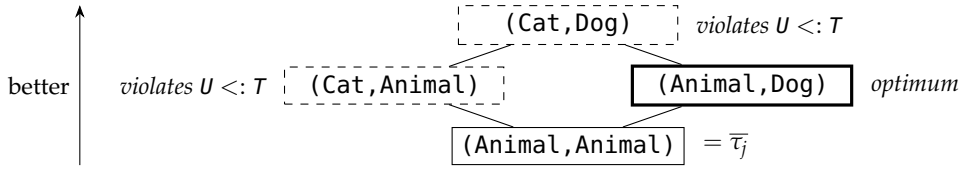


Figure 4.2: A call whose optimum differs from the static assignment $\overline{\tau_j}$, drawn as in Figure 4.1; the bold box is the optimum. The body runs with T = Animal and U = Dog, while the caller keeps the static type IPair[Animal, Animal].

4.2 Processing an RI call

An RI call is processed in three stages. Signature checking decides, from the declaration alone, whether a generic method is eligible (Section 4.5). A call of a method that is not eligible must give its type arguments explicitly. Type checking accepts an RI call only if its method is eligible. It reads each type parameter from the arguments' static types, at the parameter's occurrences in the formal parameter types, and gives each parameter the best type argument that these reads allow. This is the static assignment $\overline{\tau_j}$, which gives the call its static type $T_r[\overline{\tau_j}]$. Type checking also records, for each type parameter, an *interval* that an inferred type argument must lie in, and passes the runtime a plan (Section 4.8), from which it knows which parts of the arguments' runtime types to read and how to combine them. At run time,

runtime inference follows the plan: it reads the exact runtime types of the argument values and infers the assignment $\overline{\theta_j}$ with which the body runs. Figure 4.3 follows these stages for the running example.

In the running example (Figure 3.2), `ToList`'s `T` occurs in its parameter only at `out`, so a smaller `T` is better, and `ToList` is eligible. The call `Program.ToList(view)` passes a `view` of static type `IEnumerable[Animal]` that holds a `List[Cat]`. Reading `IEnumerable[Animal]` at the `out` occurrence of `T` gives `Animal`, a lower bound `Animal` <: `T`, and the join of the lower bounds is $\tau_\mathsf{T} = \texttt{Animal}$. At run time, reading the principal instantiation of the `List[Cat]` at `IEnumerable` gives `Cat`, the optimum. The body constructs a `List[Cat]` behind the promised `List[out Animal]`, so the caller's types remain correct. Section 4.9 ends with the formal derivation of this call.

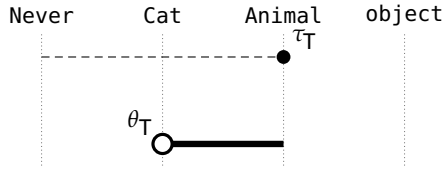


Figure 4.3: The RI call `ToList(view)` on the chain from `Never` to `object`. The lower bound comes from the argument: from `view`'s static type at compile time and from the value at run time; the upper end, τ_T, keeps the result within the call's static type. The dashed range is the interval $\mathcal{I}_\mathsf{T}$ that an inferred type argument must lie in, and the bold segment the admissible set, whose least element, circled, is the optimum.

4.3 Reading type parameters from arguments

To infer `T`, runtime inference must find it inside the argument's type: `ToList`'s `T` sits in the covariant slot of `IEnumerable[T]`, so a `List[Cat]` gives `Cat`. A parameter may sit deeper, as in `IEnumerable[IEnumerable[T]]`; it may occur at both `in` and `out` positions, as in `Feed` below; and a bound may tie two parameters together, as in `Pair` (Figure 4.2). This section defines how a parameter is read, the direction in which it prefers its type arguments, how coupled parameters are resolved, and the static assignment.

Throughout this chapter, m is a generic method with signature $\sigma = [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r$, as looked up through the receiver of a call (Section 3.6.1). As for assignments, a per-parameter quantity is indexed by position or by parameter: the bound of $X = X_j$ is T_j^b or T_X^b. Superscripts mark roles: T^s is a static type, T^u an exact runtime type, and T_X^σ, the *cap* of X, the bound of X when it names no parameter of m, and $\top$ otherwise. A subscript on $\vdash$ names what a judgment consults besides its environments: $\vdash_\sigma$ the signature σ, with the directions and bounds of its parameters, and $\vdash_{\sigma,\overline{\tau_j}}$ also the static assignment.

Occurrence paths and reads The occurrences of X that inference reads are those in the formal parameter types, $(X, \rho, v) \in \mathsf{occ}(T_i)$ (Section 3.7.1); eligibility and the interval also consult those in T_r and in the bounds. None of these variances is composed with a root variance (Section 3.7.1), since we ask how X varies with the type in which it occurs: a parameter of type `T` is an out occurrence of `T`.

A *read*, the formal counterpart of an observation (Section 2.9), finds what an argument's type T says about X at one occurrence: it follows the occurrence path ρ into T and gives the type found there, $\mathsf{read}_w(\rho, T)$, where w is the variance composed from the levels already followed, out at the start. Where T says nothing about X, the read gives instead the marker none, which is not a type. Figure 4.4 gives one rule for each case: Read-Here ends a read, Read-Step enters a generic type, Read-And passes an intersection and Read-None gives none.

$\boxed{\mathcal{D} \mid \Delta \vdash \mathsf{read}_w(\rho, T) = T'}$

$$\frac{}{\mathcal{D} \mid \Delta \vdash \mathsf{read}_w(\epsilon, T) = T}\ \textsc{Read-Here}$$

$$\frac{\mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, N) = \overline{p_l} \qquad \mathcal{D} \mid \Delta \vdash \mathsf{read}_{w \circ v'}(\rho, \mathsf{end}_{v'}(p_i)) = T'}{\mathcal{D} \mid \Delta \vdash \mathsf{read}_w((N, i, v')\rho, T) = T'}\ \textsc{Read-Step}$$

$$\frac{\mathcal{D} \mid \Delta \vdash \mathsf{read}_{\mathsf{out}}(\rho, T) = T'}{\mathcal{D} \mid \Delta \vdash \mathsf{read}_{\mathsf{out}}((\&, i, \mathsf{out})\rho, T) = T'}\ \textsc{Read-And}$$

$$\frac{\rho = (N, i, v')\rho' \qquad (w = \mathsf{out} \wedge T = \bot) \vee (w = \mathsf{in} \wedge T = \top)}{\mathcal{D} \mid \Delta \vdash \mathsf{read}_w(\rho, T) = \mathsf{none}}\ \textsc{Read-None}$$

Figure 4.4: Reads; a read gives a type or the marker none. At inv, end_v is defined only on a point interval, which every runtime read of an eligible method meets; a static read that meets another leaves the call ill typed, as does any read of $\bot$ or $\top$ that Read-None does not cover.

The rules follow the path one level at a time. Read-Here ends the read at the empty path and gives the type reached. Read-Step enters a level (N, i, v'), and does two things that comparing the formal and argument types level by level would not. First, the argument need not be an instantiation of N: a `List[Cat]`, or a class `CatList : List[Cat]` with no type arguments, passed for `IEnumerable[T]` has no `IEnumerable` to look into, so Read-Step views it through its principal instantiation at N (Section 3.3.2), for both $[\bot, \texttt{Cat}]$. Second, that instantiation gives the slot an interval p_i rather than a type, since the argument may be a projection such as `List[out Cat]`; Read-Step takes the end $\mathsf{end}_{v'}(p_i)$ that the level's variance selects, the upper end at an `out` level and the lower end at an `in` level, and continues the read from that type. Read-And passes an intersection level without moving, since a type is below $T_1 \,\&\, T_2$ exactly when it is below each conjunct (Sub-And-R, Figure 3.7); it applies only where the levels above compose to out, a condition of eligibility (Section 4.5). Figure 4.5 reads a parameter two levels deep, through both

`List` and `CatList`.

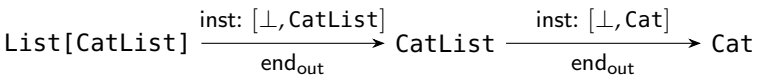


Figure 4.5: Reading `T` at `IEnumerable[IEnumerable[T]]` from `List[CatList]`.

Some arguments say nothing about X at an occurrence. Take a parameter of type `IEnumerable[IEnumerable[T]]` and an argument of type `IEnumerable[Never]`. The first level gives `Never` with a level still to enter; since `Never` is below `IEnumerable[X]` for every X, the argument fits the parameter whatever `T` is. The read must therefore not give a type: Read-Step cannot enter the next level, since $\bot$ has no principal instantiation, and giving $\bot$ as the result would wrongly bound `T`. Read-None gives the marker none instead, which inference ignores. It applies whenever the read meets $\bot$ at an out position, or $\top$ at an in position, before the path ends. Where the path has ended, $\bot$ is an ordinary result: a `List[Never]` passed for `List[T]` gives `T` exactly `Never` by Read-Here.

Inference uses the type that a read gives as a bound on X, according to the variance of the occurrence, made precise under *Directions* below: a lower bound at an out occurrence, an upper bound at an in occurrence, and an equality at an inv one. For example, reading `IEnumerable[Animal]` at the out occurrence of `T` in `IEnumerable[T]` gives `Animal`, so `Animal` $<:$ `T`. Type checking reads the arguments' static types, in which a type parameter X' of the caller is read through its bound, and runtime inference their exact runtime types (Section 4.8).

Directions The *direction* $\mathsf{dir}(X) \in \{\mathsf{out}, \mathsf{in}, \mathsf{fixed}\}$ of a type parameter X says which of two admissible type arguments is better: an *out-directed* parameter prefers a smaller type argument, an *in-directed* one a larger, and a *fixed* one prefers no type argument to another. It comes from the occurrences of X in the formal parameter types.

An occurrence of variance out or in gives that direction, as in `IEnumerable[T]` or `IHandler[T]`. An occurrence of variance inv makes X fixed when every level (N, i, v) of a head on its path has the level variance $v = \mathsf{inv}$, its lv (Section 3.7.1), as in `List[T]`: a `List[Cat]` fits `List[T]` only for `T` $=$ `Cat`, so the argument determines the type argument exactly. If such a level has $v \neq \mathsf{inv}$, as `IEnumerable` has in `IEnumerable[List[T]]`, the occurrence gives no direction (Table 4.1), since that slot may hold `Never`, and an `IEnumerable[Never]` fits `IEnumerable[List[X]]` for every X.

Table 4.1: The direction that a single occurrence gives `T`.

Formal parameter type	Variance	dir(T)
`IEnumerable[T]`	out	out
`IHandler[T]`	in	in
`List[T]`	inv	fixed
`IEnumerable[List[T]]`	inv	none: an out level

When the occurrences of X give different directions, the more decisive one wins: X is fixed if any occurrence makes it fixed, and otherwise out-directed if any occurrence is out, and in-directed if any is in. We write $\vdash_\sigma \mathsf{dir}(X) = d$ when X has the direction d in σ. A parameter whose occurrences give no direction has none, so that no judgment $\vdash_\sigma \mathsf{dir}(X) = d$ holds, except that one occurring in no T_i is fixed when its bound is $\bot$, which admits only $\bot$ itself. Inference reads each occurrence at its own variance v, except an inv occurrence with a level $v' \neq \mathsf{inv}$, whose variance does not describe how the argument constrains X; such an occurrence is read in the direction of X. This *effective variance* $\mathsf{eff}(X, \rho, v)$ is thus $\mathsf{dir}(X)$ for such an occurrence and v for every other: in `Feed` below, the occurrence in `IHandler[T]` has effective variance in, although `T` is out-directed.

When X is out-directed, its out occurrences determine the type argument and its in occurrences only check it. `Feed[T](IEnumerable[T] src, IHandler[T] sink)` infers `T` from the source; given an `IEnumerable[Cat]` and an `IHandler[Animal]`, the optimum is `Cat`, which the handler of animals accepts. No optimum is lost, and the in occurrences need no runtime read: each type read at run time from an out occurrence lies below the static type argument, which type checking has already checked to lie below each type read from an in occurrence, so these occurrences never reject the inferred type argument. Figure 4.6 shows the call.

```
static void Feed[T](IEnumerable[T] src, IHandler[T] sink) { ... }
IEnumerable[Cat] cats = ...;
IHandler[Animal] animals = new Shelter();
Program.Feed(cats, animals);  // T = Cat, which the handler accepts
```

Figure 4.6: A parameter read at both variances: `T` is read from the source and only checked against the handler.

Coupled parameters A bound that names another parameter couples the two, so the parameters can no longer be inferred one at a time. In `Pair[T, U : T]` (Figure 4.2), reading the arguments separately gives `Cat` for `T` and `Dog` for `U`, which violate `U <: T`; joining the type of `U` into that of `T` gives the optimum (`Animal`, `Dog`). Coupling therefore starts from each parameter's own answer, its *current type* $\hat{T}_j$, and adjusts the current types bound by bound. A parameter is *coupled* when its bound names another parameter or another parameter's bound names it.

Each bound $X_q{:}T$ that names another parameter *adjusts* the current types, written $\mathsf{adjust}_{X_q:T}(\overline{\hat{T}_j}) = \overline{\hat{T}'_j}$ (Figure A.14 in Appendix A). A bound $X_q{:}X_l$ requires $X_q <: X_l$. If X_l is out-directed and X_q is not in-directed, the current type of X_q is joined into that of X_l (Couple-Join), as for `Pair` in Figure 4.7; if X_q is in-directed and X_l is not out-directed, the current type of X_l is met into that of X_q (Couple-Meet). If X_l is not out-directed and X_q is not in-directed, the bound leaves the current types unchanged (Couple-Skip): X_q then moves only down and X_l only up, so a bound that the static assignment meets stays met. If X_l is out-directed and X_q in-directed, the two moves conflict and no rule applies. This is the case of `Link[T, U : T]` (Figure 4.1), where `T` would move down and `U` up, which can break `U <: T`, so `Link` is not eligible (Section 4.5). Couple-Through handles a bound that names X_l inside a generic type, such as `U : IEnumerable[T]`: it reads the current type of `U` at the paths where `T` occurs in `IEnumerable[T]` and joins the types found into the current type of the out-directed `T`.

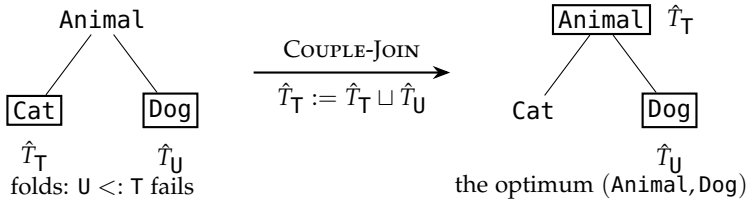


Figure 4.7: Coupling in `Pair[T, U : T]` on a `List[Cat]` and a `List[Dog]`: the bound joins the current type of `U` into that of `T`.

After the adjustments, an F-bound may still fail (Section 2.7). For `Family[T : IFamily[T]]` (Figure 4.8), the static assignment `Animal` meets the bound, but on a `List[Kitten]` the current type `Kitten` does not: `Kitten` inherits `IFamily[Cat]` from `Cat`, not `IFamily[Kitten]`. Joining `Kitten` with that type argument `Cat` gives `Cat`, which meets the bound and is the optimum; fix_X, defined next, performs this join.

```
shape interface IFamily[out T] { }
class Animal : IFamily[Animal] { ... }
class Cat : Animal, IFamily[Cat] { ... }
class Kitten : Cat { ... }
static List[out T] Family[T : IFamily[T]](IEnumerable[T] xs) { ... }
IEnumerable[Animal] view = kittens;          // a List[Kitten]
List[out Animal] f = Program.Family(view);   // T = Cat
```

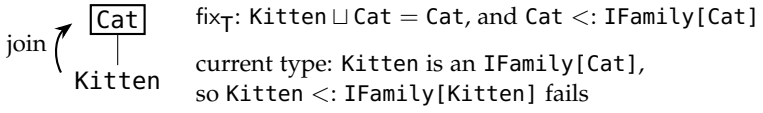


Figure 4.8: An F-bound met by a join: on a `List[Kitten]`, fix_T raises `Kitten` to `Cat`.

In general, for an out-directed X with an F-bound, fix_X moves the type up to the least valid type above it that meets the bound,

$$\mathsf{fix}_X(\hat{T}) = \min\{T' \mid \hat{T} <: T',\ \mathcal{D} \mid \Delta \vdash T',\ T' <: T^b_X[X \mapsto T']\}$$

and $\mathsf{fix}_X(\hat{T}) = \hat{T}$ for every other X; where no such type exists, fix_X is undefined and so is the assignment that needs it. The minimum is taken under $\mathcal{D} \mid \Delta$ at compile time, and at run time among the ground types of the running program, with T^b_X closed by the caller's closing (Section 4.4). When the bound is $N[\overline{a_i}]$ and X occurs in it only as the type arguments a_i, $i \in J$, at out slots, as in `IFamily[T]`, the minimum is computed by joining the type with the upper ends of its principal instantiation at N, as `Kitten` was joined with `Cat`:

$$\mathsf{fix}_X(\hat{T}) = \hat{T} \ \text{ if } \hat{T} <: T^b_X[X \mapsto \hat{T}], \qquad \mathsf{fix}_X(\hat{T}) = \mathsf{fix}_X\big(\hat{T} \sqcup \bigsqcup_{i \in J} hi_{p_i}\big) \ \text{ otherwise,} \quad (4.1)$$

where $\mathcal{D} \mid \Delta \vdash \mathsf{inst}(\hat{T}, N) = \overline{p_i}$. Every type T' in the set above lies above each hi_{p_i}: from $\hat{T} <: T'$, the upper end of the slot of $\hat{T}$ at N lies below that of T', which $T' <: T^b_X[X \mapsto T']$ places below T'. So every step stays below the minimum, and the step that meets the bound reaches it. A step that does not meet the bound raises the type, since some hi_{p_i} is not below it, and the types it reaches lie in the inheritance closure, so the joins end (Property 4.4). Since the adjustments only raise such a member by joins, applying fix_X once after all of them gives the least valid type above every type they produced.

Bounds form no cycle (Section 3.7.1), so they can be ordered so that each adjustment sees the types already moved; the order is not a preference between parameters, since optimality uses the product order (Section 2.9). Together, $\mathsf{couple}_\sigma(\overline{\hat{T}^0_j})$ applies the adjustments of the bounds to the types $\overline{\hat{T}^0_j}$ in that order, and then fix_{X_j} to every parameter X_j. Type checking applies it to the best type argument that each parameter's own reads allow, which gives the static assignment (described next); runtime inference applies it to the parameters' *start types*, their folds or, where the interval is a point, $\eta\tau_X$ (Section 4.8.1).

The static assignment Type checking determines the static assignment $\overline{\tau_j}$ from the arguments' static types $\overline{T^s_i}$ alone, as local type inference does [Pierce and Turner, 2000], with the reads above in place of constraint solving. Each occurrence of X is read from the static type of its argument, and the type read bounds X according to the effective variance of the occurrence. This gives each parameter its lower bounds

$$L_X = \{T \mid (X, \rho, v) \in \mathsf{occ}(T_i),\ \mathsf{eff}(X, \rho, v) = \mathsf{out},\ \mathcal{D} \mid \Delta \vdash \mathsf{read}_{\mathsf{out}}(\rho, T^s_i) = T \neq \mathsf{none}\},$$

and, defined alike, its upper bounds U_X, read where $\mathsf{eff}(X, \rho, v) = \mathsf{in}$, and its equalities Q_X, read where $\mathsf{eff}(X, \rho, v) \in \{\mathsf{inv}, \mathsf{fixed}\}$. The best type argument of X, written $\mathsf{best}(X, \overline{T^s_i})$, is its equality if it has one (Best-Equal), and otherwise the join of its lower bounds if it is out-directed (Best-Out), the meet of its upper bounds

and T^{ς}_{X} if it is in-directed (Best-In), and `Never` if it is fixed with the bound `Never` (Best-Never). A fixed parameter with no equality, as for a `Never` argument of `List[T]`, which fits every type argument, has none. Assign-Call applies couple_{σ} to the best type arguments, which gives the static assignment, written $\mathcal{D} \mid \Delta \vdash_{\sigma} \mathsf{assign}(\overline{T^{s}_{i}}) = \overline{\tau_{j}}$ (Figure A.15 in Appendix A). If a read is undefined, equalities disagree, no case applies or couple_{σ} is undefined, the call has no static assignment and is ill typed, as a call whose arguments do not fit is. For `Program.ToList(view)`, reading `IEnumerable[Animal]` at the `out` occurrence of `T` gives $L_{\mathsf{T}} = \{\texttt{Animal}\}$, and Best-Out gives $\tau_{\mathsf{T}} = \texttt{Animal}$.

4.4 Admissible and optimal assignments

Consider an RI call $c = r.m(\overline{e_{i}})$ (Figure 3.1) whose arguments have static types $\overline{T^{s}_{i}}$, typed under $\mathcal{D} \mid \Delta$ with static assignment $\overline{\tau_{j}}$. This section specifies what runtime inference must compute for c, making precise the notions that Section 2.9 introduced. The rules of the following sections compute this answer (Property 4.2).

Definition 4.1 (Execution)**.** An *execution* $\omega = (\mathcal{D}', \eta, \overline{u_{i}})$ of c consists of a program $\mathcal{D}'$ with $\mathcal{D} \sqsubseteq \mathcal{D}'$ (Section 3.7.2), a closing η of Δ valid in $\mathcal{D}'$ (Section 3.4), and initialised argument values $\overline{u_{i}}$ whose exact runtime types satisfy $\mathcal{D}' \mid \emptyset \vdash T^{u}_{i} <: \eta T^{s}_{i}$.

An execution fixes what type checking cannot know: which declarations the running program has, which types the enclosing type parameters stand for, and which values the arguments hold. The guarantee must hold in every execution. η concerns the caller, not the called method: `Program.ToList(view)` is made in `main`, which has no type parameters and lies in the non-generic class `Program`, so Δ and η are empty. In one execution the program is as written and `view` holds a `List[Cat]`; in another the program also declares `class Kitten : Cat` and `view` holds a `List[Kitten]`. Both are executions, since `List[Cat]` and `List[Kitten]` are subtypes of the static type `IEnumerable[Animal]`. Applied to an assignment, η closes each type argument, as in $\eta\overline{\tau_{j}}$.

Definition 4.2 (Admissible set)**.** The *admissible set* $\mathcal{A}(c, \omega)$ of c in an execution ω is the set of assignments $\overline{\theta_{j}}$ of ground types that satisfy, under $\mathcal{D}' \mid \emptyset$,

$\vdash \theta_{j}$ and $\vdash \theta_{j} <: \eta T^{b}_{j}[\overline{\theta_{l}}]$ for every j	(valid, within bounds)
$\vdash T^{u}_{i} <: \eta T_{i}[\overline{\theta_{j}}]$ for every i	(arguments fit)
$\vdash \eta T_{r}[\overline{\theta_{j}}] <: \eta T_{r}[\overline{\tau_{j}}]$	(result agreement)

where T^{b}_{j}, T_{i} and T_{r} come from the signature $\sigma = [\overline{X_{j} : T^{b}_{j}}](\overline{T_{i}}) \to T_{r}$ of m looked up at c (Section 4.3), and T^{u}_{i}, η and $\mathcal{D}'$ from ω.

The first two conditions are the premises of T-Call-Explicit, checked against the runtime types of the argument values rather than the static types of the arguments: $\overline{\theta_{j}}$ could have been given explicitly in this execution. The third, *result agreement*, keeps

the result within the static type that the caller received, which is all the caller relies on.

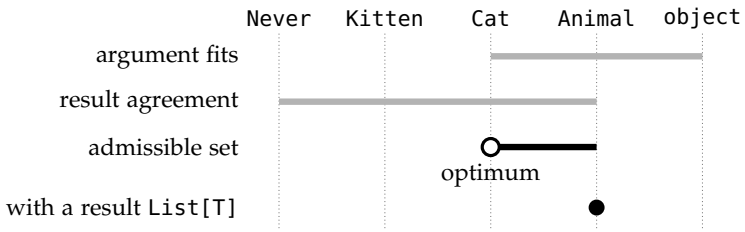


Figure 4.9: The type arguments θ for `T` that each condition allows when `ToList` is called on a `List[Cat]` seen as an `IEnumerable[Animal]`. The admissible set satisfies both; `T` is out-directed, so its least element, circled, is the optimum.

Take `ToList` on a `view` that holds a `List[Cat]` (Figure 4.9). Its parameter type is `IEnumerable[T]`, its result type `List[out T]`, and its static assignment $\tau_{\mathtt{T}} =$ `Animal`. The argument fits, `List[Cat]` <: `IEnumerable[`θ`]`, for `Cat` and every type above it, and result agreement, `List[out` θ`]` <: `List[out Animal]`, holds for `Animal` and every type below it. The admissible type arguments are those in between, from `Cat` to `Animal`. Result agreement is why `ToList` returns `List[out T]` (Section 3.3.1): with an exact result `List[T]`, `List[`θ`]` <: `List[Animal]` holds only for $\theta =$ `Animal`, and runtime inference could not refine the copy.

Definition 4.3 (Optimum)**.** A type T is *at least as good as* T' for X, written $T \preceq_X T'$, when $T <: T'$ if X is out-directed, $T' <: T$ if it is in-directed, and $T \equiv T'$ if it is fixed (Section 4.3). Assignments are compared in the product order of Section 2.9: $\overline{\theta_j} \preceq \overline{\theta'_j}$ when $\theta_j \preceq_{X_j} \theta'_j$ for every j. An *optimum* of $\mathcal{A}(c, \omega)$ is an element $\overline{\theta^*_j}$ of it with $\overline{\theta^*_j} \preceq \overline{\theta_j}$ for every $\overline{\theta_j}$ in it. Subtyping is that of $\mathcal{D}' \mid \emptyset$, and of $\mathcal{D} \mid \Delta$ when the same order compares static types. Two optima give equivalent type arguments to every type parameter, and canonical types (Section 3.5) give them one representation, so a set of assignments has at most one optimum.

Take `Link[T, U : T]` (Figure 4.1) on an `IEnumerable[Cat]` and an `IHandler[Animal]`, where `T` is out-directed and `U` in-directed. The admissible (`Cat`, `Cat`) is better for `T`, and the admissible (`Animal`, `Animal`) is better for `U`, so neither is at least as good as the other: they are *incomparable*, and the admissible set has no optimum. A lexicographic order, which compares the first type parameter and consults the second only on a tie, would instead prefer (`Cat`, `Cat`) merely because `T` is declared first; the product order gives no parameter such priority. In `Pair[T, U : T]` (Figure 4.2), both type parameters are out-directed, and the optimum (`Animal`, `Dog`) is at least as good as every admissible assignment, such as (`Animal`, `Animal`), while the better (`Cat`, `Dog`) violates `U` <: `T` and is not admissible.

Definition 4.4 (Eligibility contract)**.** A generic method m satisfies the *eligibility contract* when every RI call c of m, typed as above, satisfies in every execution ω:

1. $\mathcal{A}(c, \omega)$ has an optimum $\overline{\theta_j^*}$; and
2. runtime inference infers an assignment $\overline{\theta_j}$ with $\mathcal{D}' \mid \emptyset \vdash \theta_j \equiv \theta_j^*$ for every j.

The rest of the chapter establishes the contract from declarations alone. A method is eligible when signature checking finds every type parameter of it inferable (Section 4.5); type checking accepts an RI call only of an eligible method (Section 4.6); the selection rules infer its assignment (Section 4.8); and Property 4.2 states that every eligible method satisfies the contract. There is no fallback: runtime inference never answers $\eta\overline{\tau_j}$, or any other admissible assignment, instead of the optimum.

4.5 Eligibility

Signatures that cannot guarantee an optimum Some signatures cannot guarantee an optimum, whatever their arguments. Table 4.2 shows three kinds, each with a call that has no optimum or none that its arguments determine. Signature checking must find such signatures from the declaration alone, and it cannot rely on a call's context, which never chooses a type argument (Section 4.6).

Table 4.2: Three kinds of signature that cannot guarantee an optimum, with a call that has none or that nothing in its arguments determines. In the third, `Cat` implements `IComparable[Cat]` and `Dog` implements `IComparable[Dog]`, but `Animal` implements neither.

Signature	Problem	A call
`void Link[T, U : T] (IEnumerable[T] src, IHandler[U] sink)`	`T` prefers smaller and `U` larger, but the bound keeps `U` below `T`	on a source of cats and a handler of animals, (`Cat`, `Cat`) and (`Animal`, `Animal`) are incomparable (Figure 4.1)
`List[T] Empty[T]()`	no argument mentions `T`	in `List[Cat] xs = Program.Empty()`, only the expected type could choose `T`, and typing does not consult it
`void Sort[T : IComparable[T]] (IHandler[T] sink)`	`T` prefers larger, from a contravariant slot, but the types meeting its F-bound need not have a greatest	on a handler of animals, `Cat` and `Dog` are admissible, and no larger type meets the bound

Besides rejecting invalid declarations (Section 3.7), signature checking decides whether each type parameter X of a signature σ is *inferable*, written $\mathcal{D} \vdash_\sigma X$ inferable,

by Infer-Param (Figure 4.10). One of its premises is that the bound of X is *satisfiable*, $\mathcal{D} \vdash_\sigma (X{:}T^b_X)$ satisfiable: whatever the arguments, the runtime can make the type argument meet the bound without giving up the optimum. Table 4.3 below lists the bounds that are, by the rules of Figure A.16 in Appendix A. A signature is *eligible*, $\mathcal{D} \vdash \sigma$ eligible, when all of its type parameters are inferable (Sig-Eligible), and a method found by member lookup through a type T is eligible, $\mathcal{D} \mid \Delta \vdash T.m : \sigma$ eligible, when its declared signature is (Look-Decl, Look-Combine and Look-Static). For example, the `T` of `ToList` is inferable, while the `U` of `Link` is not, since its bound `U <: T` ties an in-directed `U` below an out-directed `T`.

$\boxed{\mathcal{D} \vdash_\sigma X \text{ inferable} \qquad \mathcal{D} \vdash \sigma \text{ eligible}}$

$$\frac{\vdash_\sigma \mathsf{dir}(X) = d \qquad \forall T \in \overline{T_i} \cup \overline{T^b_j},\, \rho,\, v.\ (X, \rho, v) \in \mathsf{occ}(T) \implies \mathsf{precise}(\rho) \qquad \forall \rho, v.\ (X, \rho, v) \in \mathsf{occ}(T_r) \implies \& \notin \rho \qquad \mathcal{D} \vdash_\sigma (X{:}T^b_X) \text{ satisfiable}}{\mathcal{D} \vdash_\sigma X \text{ inferable}}\ \textsc{Infer-Param}$$

$$\frac{\overline{\mathcal{D} \vdash_\sigma X_j \text{ inferable}}}{\mathcal{D} \vdash \sigma \text{ eligible}}\ \textsc{Sig-Eligible}$$

$\boxed{\mathcal{D} \mid \Delta \vdash T.m : \sigma \text{ eligible}}$

$$\frac{\mathcal{D} \mid \Delta \vdash T.m : \sigma \qquad \mathsf{near}(T, m, \mathsf{arity}(\sigma)) = \{N\} \qquad \mathcal{D} \vdash N \ni m : \sigma_0 \qquad \mathcal{D} \vdash \sigma_0 \text{ eligible}}{\mathcal{D} \mid \Delta \vdash T.m : \sigma \text{ eligible}}\ \textsc{Look-Decl}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T.m : \sigma \qquad |\mathsf{near}(T, m, \mathsf{arity}(\sigma))| \geq 2 \qquad \mathcal{D} \vdash \sigma \text{ eligible}}{\mathcal{D} \mid \Delta \vdash T.m : \sigma \text{ eligible}}\ \textsc{Look-Combine}$$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ C[\overline{X_i}]\ \{\ \overline{M_l}\ \} \qquad \texttt{static}\ H\ \{\ \overline{s_q}\ \} \in \overline{M_l} \qquad \mathsf{name}(H) = m \qquad \mathcal{D} \vdash \mathsf{sig}(H) \text{ eligible}}{\mathcal{D} \mid \Delta \vdash (\mathsf{static}\ C[\overline{T_i}]).m : \mathsf{sig}(H)[\overline{X_i \mapsto T_i}] \text{ eligible}}\ \textsc{Look-Static}$$

Figure 4.10: Inferable type parameters and eligible signatures and methods, for $\sigma = [\overline{X_j : T^b_j}](\overline{T_i}) \to T_r$; $\mathsf{precise}(\rho)$ and σ_0 are defined below.

Inferable type parameters A type parameter X is inferable (Infer-Param) when four conditions hold.

- X has a direction, $\vdash_\sigma \mathsf{dir}(X) = d$; the `T` of `Empty` has none.
- Every occurrence of X in a parameter type or a bound is *precise*, $\mathsf{precise}(\rho)$: no level of its path but the last is contravariant, and every intersection on it lies at an out position; that is, when ρ has the levels (N_1, i_1, v_1) to (N_n, i_n, v_n), $v_l \neq \mathsf{in}$ for every $l < n$, and $v_1 \circ \cdots \circ v_{l-1} = \mathsf{out}$ for every l with $N_l = \&$. Below a contravariant level, a read no longer gives the constraint on X exactly: a handler

fits `IHandler[List[out X]]` when the lower end of its slot is a supertype of `List[out X]`, which depends on the supertypes of `List`, not on one slot. An intersection elsewhere than at an `out` position makes the constraint a choice (Section 4.7).

- No occurrence of X in the result type T_r lies under an intersection, $\& \notin \rho$, since an intersection can absorb X (Section 4.7).
- The bound of X is satisfiable, as described next.

Bounds the runtime can satisfy At run time, the type argument of an out-directed parameter starts from the *fold*, the join of the types read for it from the arguments, and that of an in-directed parameter from the meet (Section 4.8.1). The bound must then hold without giving up the optimum. Table 4.3 lists the bounds that the rules of Figure A.16 accept, each with an example; every other bound makes its parameter not inferable.

Table 4.3: The bounds that the runtime can satisfy, by rule. X_q is the parameter whose bound it is, and X_l another parameter that the bound names.

Rule	Bound	Accepted when	Example
Sat-Ground	names no parameter of m	X is not in-directed, or no conjunct of the bound is a shape	`T : object` of `ToList`
Sat-Fixed	a bound naming only X	X is fixed	`Copy[T : IComparable[T]] (List[T] xs)`
Sat-Self	an F-bound $N[\overline{a_i}]$ with X as type arguments at slots of one variance	X is out-directed	`T : IFamily[T]` of `Family`
Sat-Param	X_q:X_l	not both X_l out-directed and X_q in-directed	`U : T` of `Pair`, but not of `Link`
Sat-Through	X_l inside a generic type, only at `out`	X_l out-directed and X_q not in-directed	`U : IEnumerable[T]`

A bound that names no parameter of m only caps the interval, within which the fold lies (Sat-Ground). The exception is an in-directed parameter with a shape among the conjuncts of its bound: the meet moves the type argument down, and the types below a shape need not have a greatest material one. A fixed X has a single type argument, which type checking has already required to meet its F-bound (Sat-Fixed).

Sat-Self accepts the F-bound of an out-directed X when X occurs in it directly as type arguments at slots of one variance; N is a shape, as the cycle restrictions require

of every F-bound (Section 3.7.1). Each variance is safe for its own reason. At `in` slots, every type below one that meets the bound meets it too, so the fold, which lies below the static type argument, meets it. At one invariant slot, the bound pins X to its static type argument. At `out` slots, the fold may fail the bound, and fix_X joins it with its own type arguments at those slots until it holds, by (4.1), as one join raises `Kitten` to `Cat` for `Family` (Figure 4.8). An in-directed X moves down by meets, and the types below an F-bound need not have a greatest one, as `Sort` in Table 4.2 shows, so no F-bound of an in-directed parameter is accepted.

A bound that names another parameter couples the two (Section 4.3). Sat-Param accepts $X_q{:}X_l$ exactly when a coupling rule applies to it, so it rejects the bound of `Link`, where X_l is out-directed and X_q in-directed. Sat-Through accepts a bound that names X_l inside a generic type when Couple-Through can join the reads of X_l into an out-directed X_l.

`ToList` is eligible: its `T` occurs in its parameter only in the covariant slot of `IEnumerable[T]`, a precise occurrence, and its bound `object` names no parameter. `Tag[T](List[out T] xs)` is eligible for the same reason, since `T` is read at an `out` slot: a call on a `List[out Animal]` view is well typed, and when the view holds a `CatList`, the optimum is `Cat`. `Empty` is not eligible, since its `T` occurs in no parameter type. Figure 4.11 shows the eligibility of some signatures used in this chapter.

Signature	Eligible
`List[out T] ToList[T](IEnumerable[T] source)`	yes: `T` out-directed
`void Feed[T](IEnumerable[T] s, IHandler[T] h)`	yes: `T` out-directed
`List[T] Copy[T](List[T] xs)`	yes: `T` fixed
`void Link[T, U : T](IEnumerable[T] s, IHandler[U] h)`	no: bound `U : T`
`List[T] Empty[T]()`	no: `T` has no direction

Figure 4.11: Eligibility of example signatures; a call of `Link` or `Empty` must give its type arguments.

Lookup of an eligible method A method found by member lookup has two signatures. Its *declared* signature σ_0 is the one its declaration writes, in terms of the type parameters of the declaring head N, $\mathcal{D} \vdash N \ni m : \sigma_0$. The signature σ in $\mathcal{D} \mid \Delta \vdash T.m : \sigma$ is σ_0 *seen through* the receiver type T: Mem-View (Figure 3.9) replaces the type parameters of N by the intervals that T guarantees at N, $\sigma = \mathsf{view}(\sigma_0, \mathsf{inst}(T, N))$, and leaves the method's own type parameters as they are. For example, if `List[T]` declared `void AddFrom[U : T](IEnumerable[U] src)`, then σ_0 has the bound `U` $<:$ `T`, and through `List[Cat]`, σ has the bound `U` $<:$ `Cat`. Look-Decl judges σ_0, so a method is eligible or not once, at its declaration, whatever receiver a call uses; its bounds are judged before the view, which may replace a type parameter of N in them by an open intersection. When several heads combine by Mem-Combine, no declaration writes the combined signature, and Look-Combine judges σ itself; for a static method, Look-Static judges $\mathsf{sig}(H)$, the signature of the method `static` H that Mem-Static (Figure A.6) finds, before it substitutes the class's type arguments.

4.6 Typing an RI call

T-Call-RI (Figure 4.12) types an RI call. It types the receiver and looks up m, which must be eligible and have type parameters. It then synthesises the static type T_i^s of each argument and determines the static assignment from these types alone, $\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{assign}(\overline{T_i^s}) = \overline{\tau_j}$ (Section 4.3). Finally it checks the remaining premises of T-Call-Explicit (Figure 3.13) at $\overline{\tau_j}$: each τ_j is valid and below its bound, and each argument checks at its formal parameter type. The call has type $T_r[\overline{X_j \mapsto \tau_j}]$. An RI call is therefore ill typed when its method is not eligible, which signature checking decides from the declaration, or when a premise about its arguments fails, as when they give it no static assignment (Sections 4.3 and 4.7); every well-typed RI call of an eligible method has a unique optimal assignment, at compile time and at run time (Properties 4.1 and 4.2). Type checking also gives each type parameter X the interval $\mathcal{I}_X$ in which runtime inference must choose its type argument (4.2). For `Program.ToList(view)`, `ToList` is eligible, $\tau_\mathsf{T} = \texttt{Animal}$, the call has type `List[out Animal]`, and $\mathcal{I}_\mathsf{T} = [\bot, \texttt{Animal}]$.

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}$

$$\frac{\begin{array}{c}\mathcal{D} \mid \Delta \mid \Gamma \vdash r \Rightarrow T \qquad \mathcal{D} \mid \Delta \vdash T.m : \sigma \text{ eligible} \qquad \sigma = [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r \\ |\overline{X_j}| \geq 1 \qquad \overline{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_i \Rightarrow T_i^s} \qquad \mathcal{D} \mid \Delta \vdash_\sigma \mathsf{assign}(\overline{T_i^s}) = \overline{\tau_j} \\ \overline{\mathcal{D} \mid \Delta \vdash \tau_j} \qquad \overline{\mathcal{D} \mid \Delta \vdash \tau_j <: T_j^b[\overline{X_l \mapsto \tau_l}]} \qquad \overline{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_i \Leftarrow T_i[\overline{X_j \mapsto \tau_j}]}\end{array}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash r.m(\overline{e_i}) \Rightarrow T_r[\overline{X_j \mapsto \tau_j}]}\ \text{T-Call-RI}$$

Figure 4.12: Typing an RI call.

Principal typing The call's context takes no part in typing: an expected type is checked against the call's type by T-Sub but never chooses a type argument, so `List[Cat] xs = Program.Empty()` is ill typed although its context expects a `List[Cat]`. An RI call therefore has a principal type, $T_r[\overline{X_j \mapsto \tau_j}]$, built from its arguments' types alone, so eligible methods have principal typing in Muehlboeck's sense (Section 2.9); its static assignment is moreover the unique optimum for those types (Property 4.1). The static assignment gives the call its static type; the body runs with the type arguments that runtime inference chooses.

Intervals The interval of X is an interval in the sense of Section 3.3.1, the intersection $\cap$ of argument intervals of τ_X, which keeps the type arguments that all of them allow, as the Meet column of Table 3.1 does at an invariant slot; for a fixed X, $\mathsf{den}_\mathsf{fixed} = \mathsf{den}_\mathsf{inv}$:

$$\mathcal{I}_X = [\bot, T_X^g] \cap \mathsf{den}_{\mathsf{dir}(X)}(\tau_X) \cap \bigcap_{(X,\rho,v)\in\mathsf{occ}(T_r)} \mathsf{den}_v(\tau_X) \cap \bigcap_{(X,\rho,\mathsf{inv})\in\mathsf{occ}(T_X^b)} \mathsf{den}_\mathsf{inv}(\tau_X). \tag{4.2}$$

Normally it runs from τ_X towards the preferred type arguments, down to $\bot$ for an out-directed X and up to the cap T_X^g for an in-directed one: for `ToList`, $\mathcal{I}_{\mathsf{T}} = [\bot, \texttt{Animal}]$. It shrinks to the point $[\tau_X, \tau_X]$ in three cases: when X is fixed; when the result type has an occurrence of X that is invariant or opposite to $\mathsf{dir}(X)$, since result agreement then admits only $\eta\tau_X$ (Section 4.4); and when the bound of X mentions X at an invariant slot. Where the interval is a point, runtime inference answers $\eta\tau_X$ in every execution, so the call behaves as if τ_X had been given explicitly.

4.7 Intersections and inference

An intersection may appear wherever a type may, but runtime inference can use it only where it constrains a type parameter in one way. A type is below T_1 & T_2 exactly when it is below both conjuncts (Sub-And-R), so at an `out` position an intersection gives both constraints at once, and Read-And reads through it. In the other direction, T_1 & T_2 is below a type when either conjunct is (Sub-And-L), or when its principal instantiation is (Sub-Inst). Where a type parameter makes that instantiation undefined, only the first remains, so the constraint becomes a choice between the conjuncts, and their answers need not be comparable. With interfaces `R1[X] : R[X]` and `R2[Y] : R[Y]`, an `IHandler[R[int]]` fits a parameter of type `IHandler[R1[X] & R2[Y]]` when `X` = `int` or when `Y` = `int`, so neither parameter has a best type argument. Table 4.4 lists every position and what G# does there.

Table 4.4: Intersections by position, with what G# does and the rule that does it; `IA[T]` and `IB[U]` inherit the contravariant `ICons[T]` and `ICons[U]`.

Position	Example	Outcome
bound or supertype	`U : T & IA` `C[T] : D[T & IA]`	invalid: not ground (Valid-Decl, Valid-Sig)
two paths to one head	`IC[T, U] : IA[T], IB[U]`	invalid: the ends are not stable (Valid-Inh)
parameter type at `out`	`T & IA x` `IEnumerable[T & IA] xs`	read through each conjunct (Read-And)
parameter type at `in` or `inv`	`IHandler[R1[X] & R2[Y]] h` `Box[T & IA] b`	not inferable: a choice between conjuncts (precise, Infer-Param)
result type	`T & Cat F[T](IHandler[T] h)`	not inferable: the intersection absorbs `T` (Infer-Param)
argument's static type	`X & ICons[in Cat]` with `X : ICons[in Y]`	call ill typed: no principal instantiation (Assign-Call)
inferred type argument	`X & Y` from two `ICons` arguments	an ordinary answer (Best-In)

Choices in a signature At an invariant position the choice is between two solutions: a `Box[Dog & IA]` fits `Box[T & IA]` at both `T = Dog` and `T = Dog & IA`, and an invariant slot has no direction to choose between them. In the result type, an intersection breaks the occurrence-by-occurrence reading that gives the interval (Section 4.6). In `T & Cat F[T](IHandler[T] h)`, `T` is in-directed, and for an `h` of static type `IHandler[Animal]`, the `out` occurrence in the result keeps `T` at or below `Animal`; yet when `h` holds an `IHandler[object]`, the better `object` gives the same result `Cat`, which result agreement admits.

Choices in an argument A static read needs the principal instantiation of the argument's static type at each head of its path (Read-Step), and an open intersection can lack one. In `H[Y, X : ICons[in Y]]`, an argument of type `X & ICons[in Cat]` passed to `F[T](ICons[in T] c)` has two entries at `ICons` whose lower ends `Y` and `Cat` are not stable, so their meet is undefined (Section 3.3.1). Each conjunct alone allows `T <: Y` or `T <: Cat`, the same kind of choice as in `IHandler[R1[X] & R2[Y]]`, so no read is defined and the RI call is ill typed, although each explicit form, such as `F[Cat](c)`, is well typed. At run time the value has one `ICons` entry, so the call has an optimum in every execution; accepting such calls is left to future work (Section 7.1).

Intersections as answers An intersection that inference computes is an ordinary answer. For `Meet[T](ICons[in T] a, ICons[in T] b)`, called in `Both[X, Y]` on an `ICons[in X]` and an `ICons[in Y]`, the reads give the upper bounds `X` and `Y`, and Best-In gives $\tau_{\mathsf{T}} = \mathtt{X} \sqcap \mathtt{Y} = \mathtt{X\ \&\ Y}$; at `X = Cat` and `Y = Animal` it is `Cat`, which no written type argument other than `X & Y` would preserve.

4.8 Inferring the assignment at run time

At run time an argument may be more precise than its static type: a source seen as `IEnumerable[Animal]` may hold a `List[Kitten]`. Runtime inference reads `Kitten`, must not go beyond the static assignment $\tau_{\mathsf{T}} = \mathtt{Animal}$, and must still meet the bound, which for `Family` (Figure 4.8) raises `Kitten` to `Cat`. This section gives the rules for selection (Section 2.9). Type checking passes the runtime the *plan* $\pi = (\sigma, \overline{\tau_j})$ of the call, its looked-up signature and static assignment. In an execution with program $\mathcal{D}'$ and caller's closing η, the plan infers the assignment $\overline{\theta_j}$ from the exact runtime types $\overline{T_i^\mu}$ of the argument values, written $\mathcal{D}' \vdash_\eta \mathsf{infer}_\pi(\overline{T_i^\mu}) = \overline{\theta_j}$ and defined by Sel-Plan: it reads one type for each occurrence, folds the reads of each parameter, and combines the folded types of parameters coupled by bounds. The judgments of selection carry $\mathcal{D}'$ as their environment and the closing η as a subscript of $\vdash$, and name the plan by a subscript of $\vdash$ or of infer. They work on ground types under the empty Δ, so the subtyping and joins in their premises are those of $\mathcal{D}' \mid \emptyset$, and their reads use the rules of Figure 4.4, now applied to exact runtime types.

4.8.1 Folding reads

The fold rules (Figure 4.13) use the interval $\mathcal{I}_X$ of the plan (4.2), closed by η and compared up to $\equiv$, and define the start type $\mathsf{start}(X, \overline{T_i^u})$ of a parameter, its type before coupling. Where $\mathcal{I}_X$ is a point, it is $\eta\tau_X$ (Sel-Point). Otherwise the reads of X at its occurrences of effective variance $\mathsf{dir}(X)$, $R_X = \{T \mid (X, \rho, v) \in \mathsf{occ}(T_i),\ \mathsf{eff}(X, \rho, v) = \mathsf{dir}(X),\ \mathcal{D}' \mid \emptyset \vdash \mathsf{read}_{\mathsf{out}}(\rho, T_i^u) = T \neq \mathsf{none}\}$, are combined by its direction, by join for an out-directed X (Sel-Out) and by meet for an in-directed one (Sel-In), starting from the end of $\mathcal{I}_X$ other than τ_X. The fold must refine the closed static type argument $\eta\tau_X$ in the parameter's direction.

$\boxed{\mathcal{D}' \vdash_{\pi,\eta} \mathsf{start}(X, \overline{T_i^u}) = T}$

$$\frac{\mathcal{I}_X = [\tau_X, \tau_X]}{\mathcal{D}' \vdash_{\pi,\eta} \mathsf{start}(X, \overline{T_i^u}) = \eta\tau_X}\ \textsc{Sel-Point}$$

$$\frac{\mathcal{I}_X \neq [\tau_X, \tau_X] \qquad \mathsf{dir}(X) = \mathsf{out} \qquad \hat{T} = \eta lo_{\mathcal{I}_X} \sqcup \bigsqcup R_X \qquad \mathcal{D}' \mid \emptyset \vdash \hat{T} <: \eta\tau_X}{\mathcal{D}' \vdash_{\pi,\eta} \mathsf{start}(X, \overline{T_i^u}) = \hat{T}}\ \textsc{Sel-Out}$$

$$\frac{\mathcal{I}_X \neq [\tau_X, \tau_X] \qquad \mathsf{dir}(X) = \mathsf{in} \qquad \hat{T} = \eta hi_{\mathcal{I}_X} \sqcap \bigsqcap R_X \qquad \mathcal{D}' \mid \emptyset \vdash \eta\tau_X <: \hat{T}}{\mathcal{D}' \vdash_{\pi,\eta} \mathsf{start}(X, \overline{T_i^u}) = \hat{T}}\ \textsc{Sel-In}$$

$\boxed{\mathcal{D}' \vdash_{\eta} \mathsf{infer}_{\pi}(\overline{T_i^u}) = \overline{\theta_j}}$

$$\frac{\overline{\mathcal{D}' \vdash_{\pi,\eta} \mathsf{start}(X_j, \overline{T_i^u}) = \hat{T}_j^0} \qquad \overline{\theta_j} = \mathsf{couple}_{\sigma}(\overline{\hat{T}_j^0}) \qquad \forall l.\ X_l\ \mathsf{coupled} \implies \mathcal{D}' \mid \emptyset \vdash \theta_l <: \eta T_l^b[\overline{\theta_j}]}{\mathcal{D}' \vdash_{\eta} \mathsf{infer}_{\pi}(\overline{T_i^u}) = \overline{\theta_j}}\ \textsc{Sel-Plan}$$

Figure 4.13: Selection: the start type of each parameter and the inferred assignment.

This is a check, not a fallback: on a well-typed RI call it always holds (Section 4.10), and the rules define no substitute for it. A bound that mentions the parameter itself is met afterwards: for `Family`, fix_X (Section 4.3), applied in Sel-Plan, moves the fold `Kitten` to `Cat`. Figure 4.14 folds two reads.

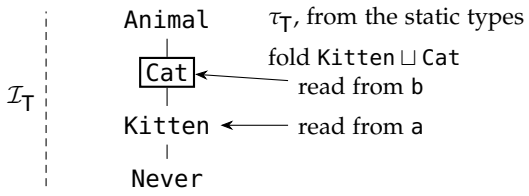


Figure 4.14: Folding two reads in `Concat[T](IEnumerable[T] a, IEnumerable[T] b)` on a `List[Kitten]` and a `List[Cat]`, both seen as `IEnumerable[Animal]`: the fold lies within $\mathcal{I}_T$.

4.8.2 Bounds and coupled parameters

At run time the folded types are combined by the same couple_σ (Section 4.3): for `Pair`, with a `List[Cat]` and a `List[Dog]`, the folds `Cat` and `Dog` become the optimum `(Animal,Dog)`. Sel-Plan (Figure 4.13) starts each parameter at its start type, applies couple_σ, and checks the bound of every coupled parameter at the complete assignment. A bound that names no other parameter needs no check: fix_X meets an F-bound, and the interval meets a bound that names no parameter of m, since an out-directed fold lies below $\eta\tau_X$, which meets the bound, and an in-directed one below the cap. Like the check of the fold rules, this bound check always succeeds on a well-typed RI call (Property 4.2).

4.9 Execution of an RI call

A configuration $\langle h, \Phi, e\rangle$ consists of a heap h, which maps each object to its fields and its exact runtime type, a frame stack Φ and the code e being executed, all under the environment $\mathcal{D}'$ of the running program. A value u is an integer, a Boolean, the Unit value or a reference to an object (Section 3.2). The top frame (η, λ) holds the closing η of the running code and the map λ from its local variables to values; each frame below it, (η, λ, E), also holds the evaluation context E of the call it waits for. Evaluation contexts E are the standard ones, which evaluate subexpressions once, from left to right in source order, and S ranges over the contexts of a statement in a method body.

Once its receiver and arguments are evaluated, to a receiver r, a value or a static class, and values $\overline{u_i}$, a call becomes the code $\mathsf{call}(m\texttt{[}\overline{T_j^a}\texttt{]}, r, \overline{u_i})$ when it gives type arguments $\overline{T_j^a}$, and $\mathsf{call}(m^\pi, r, \overline{u_i})$ when it is an RI call with plan π. Four functions connect the rules to the heap: $\mathsf{types}(h, \overline{u_i})$ gives the exact runtime types $\overline{T_i^u}$ of the argument values; $\mathsf{dispatch}(m, h, r)$ gives the body b that implements the statically resolved method m for r, chosen by the exact type of r or, for a static class, by its declaration; $\mathsf{body}(b)$ is the code of b; and $\mathsf{enter}(b, r, \overline{\theta_j}, \overline{u_i})$ gives the callee's frame, which binds the parameters to $\overline{u_i}$, the method's type parameters to $\overline{\theta_j}$, those of the class C that declares b to the type arguments of the static class or, for an instance method, of the exact type of r at C, its principal instantiation (Section 3.3.2), whose slots are points, and, for an instance method, `this` to r. E-Call-Explicit and E-Call-RI (Figure 4.15) obtain the assignment $\overline{\theta_j}$, explicit or inferred by the plan, and enter the body, while the caller's frame keeps the context E of the call. E-Return resumes that context with the returned value; a body that completes without `return` returns the Unit value. Dispatch chooses the body; it does not choose the type arguments.

The compiler emits each well-typed RI call with the plan of its T-Call-RI derivation, from which the intervals and the order of the bounds that the runtime rules consult are computed. The other expressions evaluate as usual: `e is T` tests the value's runtime type against T with the frame's type arguments substituted, and `new C[`$\overline{T_j}$`](`$\overline{e_i}$`)` runs the constructor and yields the new object, whose runtime type is $C[\overline{T_j}]$ under the same substitution; we omit their rules. There is no rule for a

$\boxed{\mathcal{D}' \vdash \langle h, \Phi, e\rangle \to \langle h', \Phi', e'\rangle}$

$$\frac{\overline{\theta_j = \eta T_j^a} \qquad b = \mathsf{dispatch}(m,h,r) \qquad \Phi' = \mathsf{enter}(b,r,\overline{\theta_j},\overline{u_i}){:}(\eta,\lambda,E){:}\Phi}{\mathcal{D}' \vdash \langle h, (\eta,\lambda){:}\Phi,\ E[\mathsf{call}(m[\overline{T_j^a}],r,\overline{u_i})]\rangle \to \langle h, \Phi', \mathsf{body}(b)\rangle}\ \text{E-Call-Explicit}$$

$$\frac{\mathcal{D}' \vdash_\eta \mathsf{infer}_\pi(\mathsf{types}(h,\overline{u_i})) = \overline{\theta_j} \qquad b = \mathsf{dispatch}(m,h,r) \qquad \Phi' = \mathsf{enter}(b,r,\overline{\theta_j},\overline{u_i}){:}(\eta,\lambda,E){:}\Phi}{\mathcal{D}' \vdash \langle h, (\eta,\lambda){:}\Phi,\ E[\mathsf{call}(m^\pi,r,\overline{u_i})]\rangle \to \langle h, \Phi', \mathsf{body}(b)\rangle}\ \text{E-Call-RI}$$

$$\frac{}{\mathcal{D}' \vdash \langle h, (\eta',\lambda'){:}(\eta,\lambda,E){:}\Phi,\ S[\texttt{return } u\texttt{;}]\rangle \to \langle h, (\eta,\lambda){:}\Phi, E[u]\rangle}\ \text{E-Return}$$

Figure 4.15: Entry to and return from a generic call.

failed inference, since no read, fold or bound check can fail on a well-typed RI call; Section 4.10 shows this (Property 4.2). Figure 4.16 derives the call of Section 4.2. Only the callee's frame carries `Cat`; the caller's `List[out Animal]` is unchanged.

Signature: `T` occurs at $\rho = (\texttt{IEnumerable}, 1, \mathsf{out})$ in T_1 and at `out` in T_r	
1 $\mathsf{dir}(\texttt{T}) = \mathsf{out}$, ρ precise, bound `object`: `T` inferable, `ToList` eligible	Infer-Param
Compile time: $T_1^s = \texttt{IEnumerable[Animal]}$	
2 $\mathsf{read}_{\mathsf{out}}(\rho, T_1^s) = \texttt{Animal}$, so $L_\texttt{T} = \{\texttt{Animal}\}$	Read-Step
3 $Q_\texttt{T} = \emptyset$, so $\tau_\texttt{T} = \bigsqcup L_\texttt{T} = \texttt{Animal}$	Best-Out
4 $\texttt{view} \Leftarrow T_1[\overline{\tau_j}]$, so $\mathcal{D} \mid \Delta \mid \Gamma \vdash c \Rightarrow \texttt{List[out Animal]}$	T-Call-RI
5 $\mathsf{dir}(\texttt{T}) = \mathsf{out}$, T_r reads `T` at `out`, $T_\texttt{T}^g = \top$: $\mathcal{I}_\texttt{T} = [\bot, \texttt{Animal}]$	Equation (4.2)
Run time: `view` holds an object of exact type $T_1^u = \texttt{List[Cat]}$	
6 $\mathsf{inst}(T_1^u, \texttt{IEnumerable}) = [\bot, \texttt{Cat}]$, so $\mathsf{read}_{\mathsf{out}}(\rho, T_1^u) = \texttt{Cat}$	Read-Step
7 $\bot \sqcup \texttt{Cat} = \texttt{Cat} <: \texttt{Animal}$, so $\mathsf{start}(\texttt{T}, \overline{T_i^u}) = \texttt{Cat}$	Sel-Out
8 $\mathcal{D}' \vdash_\eta \mathsf{infer}_\pi(\overline{T_i^u}) = (\texttt{Cat})$; the body runs under it	Sel-Plan, E-Call-RI

Figure 4.16: Derivation of `Program.ToList(view)`. Step 1 is signature checking, steps 2 to 5 type checking and steps 6 to 8 run time.

4.10 Optimality and soundness

We state what the rules guarantee as properties, each with the reason it holds: Property 4.2 is the eligibility contract (Definition 4.4), and Property 4.7 that no compiled program halts because of a type error. The reasons are informal, and the premises they rely on are stated before Property 4.7. Throughout, $\vdash \mathcal{D}$, c is an RI call of an eligible method typed by T-Call-RI with plan π, and $\omega = (\mathcal{D}', \eta, \overline{u_i})$ is an execution of c.

Optimality

Property 4.1 (Static optimality)**.** $\overline{\tau_j}$ is the optimum, in the product order of Section 4.4 under $\mathcal{D} \mid \Delta$, of the assignments that satisfy the premises of T-Call-Explicit for the arguments' static types $\overline{T_i^s}$, unique up to $\equiv$ under Δ.

Every occurrence of a type parameter of an eligible method in a parameter type or bound is precise (Infer-Param), so each read gives exactly the constraint that subtyping places on its parameter at that occurrence: at an `out` level, Sub-Inst (Figure 3.7) compares the upper end of the slot, which Read-Step takes, and at an `in` level the lower end. Best-Out and Best-In therefore take the least or greatest type argument that the constraints of each parameter allow, and couple_σ moves a coupled parameter only as far as its bound requires.

Property 4.2 (Optimality)**.** There is a $\overline{\theta_j}$ with $\mathcal{D}' \vdash_\eta \mathsf{infer}_\pi(\overline{T_i^u}) = \overline{\theta_j}$, and $\overline{\theta_j}$ is the optimum of $\mathcal{A}(c, \omega)$, unique up to $\equiv$.

The runtime reads exact types with the same rules, so the fold of each parameter is the least or greatest type argument its reads allow. The closed static assignment $\eta\overline{\tau_j}$ is admissible, since T-Call-RI checked it and $T_i^u <: \eta T_i^s$, so the fold lies within $\mathcal{I}_X$ and the check of Sel-Out or Sel-In holds; a bound that mentions its own parameter is met by fix_X, whose steps (4.1) stay below the optimum, and coupled parameters move as at compile time. Each condition of Infer-Param is what one of these arguments needs (Section 4.5).

Property 4.3 (Stability)**.** If $\mathcal{D} \sqsubseteq \mathcal{D}'$, every judgment of signature checking and type checking derived in $\mathcal{D}$ is derivable in $\mathcal{D}'$, so the eligibility of m, the static assignment and the plan of c are unchanged.

Adding declarations changes no supertype, variance or bound of an existing type (Section 3.7.2), and joins and canonical meets are unchanged because G# has no union types (Section 3.5).

Termination and soundness of RI calls

Property 4.4 (Termination)**.** Every derivation of $\mathcal{D}' \vdash_\eta \mathsf{infer}_\pi(\overline{T_i^u}) = \overline{\theta_j}$ is finite: a read takes one principal-instantiation lookup per level of its path, a fold one join or meet per read, coupling one join or meet per bound and one join per read of Couple-Through, and fix_X at most $|\mathsf{cl}(\overline{T_i^u})|$ steps of (4.1), where cl is the inheritance closure (Section 3.7.1).

Paths, reads and steps are fixed by σ, each type operation terminates by the cycle restrictions of Section 3.7.1, and the steps of (4.1) stay in the inheritance closure, which the finitary usage graph keeps finite.

Property 4.5 (Progress of RI calls)**.** E-Call-RI applies to $\mathsf{call}(m^\pi, r, \overline{u_i})$ in every execution.

The plan derives the optimum (Property 4.2) in finitely many steps (Property 4.4), and dispatch finds a body as for an explicit call.

Property 4.6 (Preservation for RI calls)**.** The frame that E-Call-RI enters is well typed: the body runs with type arguments $\overline{\theta_j}$ that are valid and below their bounds, its parameters hold values of types below $\eta T_i[\overline{\theta_j}]$, and its result has a type below the call's static type $\eta T_r[\overline{\tau_j}]$.

$\overline{\theta_j} \in \mathcal{A}(c, \omega)$, whose conditions are the premises of T-Call-Explicit on the argument values, so the call enters the body as the explicit call with $\overline{\theta_j}$ would; result agreement places its result below the type T-Call-RI gave the call.

Soundness of programs A state $\langle h, \Phi, e \rangle$ is *well typed* when every value it holds lies below the type that the rules of Chapter 3 give its holder: for each frame (η, λ) or (η, λ, E), whose code is typed under some $\mathcal{D} \mid \Delta \mid \Gamma$, η is a valid closing of Δ in $\mathcal{D}'$ and $\mathcal{D}' \mid \emptyset \vdash \mathsf{types}(h, \lambda(x)) <: \eta\Gamma(x)$ for each local x, and $E[u]$ is well typed there for every value u whose exact type lies below the static type of the call that E waits for, closed by η; for each object of exact type T and field f with $\mathcal{D}' \mid \emptyset \vdash T.f : T_f$ (Mem-Field), the field's value has an exact type below T_f; and the running code e is well typed under the top frame's $\mathcal{D} \mid \Delta \mid \Gamma$. The properties rely on two premises, which we assume rather than prove:

(P1) *safety of the rest of the language*: from a well-typed state, every step other than an RI call leads to a well-typed state, and only a finished program or a `throw` takes no step. The core of G# is that of Featherweight Generic Java, whose safety Igarashi et al. [2001, Section 3.4] prove; its projections, intersections, shapes and definite initialisation lie outside that proof;

(P2) *termination of the type operations*, which the finitary usage graph ensures: Material–Shape Separation [Greenman et al., 2014, Sections 3.2–3.3 and 5.1] and the absence of growing cycles [Kennedy and Pierce, 2007, Theorem 9];

These properties concern the rules; how compiled code follows them is the subject of Section 4.11.

Property 4.7 (Soundness)**.** Assume (P1), (P2), $\vdash \mathcal{D}$ and $\mathcal{D} \sqsubseteq \mathcal{D}'$. No execution of $\mathcal{D}$ under $\mathcal{D}'$ halts because of a type error: every RI call that it reaches executes by E-Call-RI with the optimum of its admissible set and leaves a well-typed state.

By induction on the steps of the execution, whose initial state, that of `main`, is well typed. A step other than an RI call is covered by (P1). At an RI call c, the state is well typed, so its argument values have exact types below their static types closed by the frame's η, and $\omega = (\mathcal{D}', \eta, \overline{u_i})$ is an execution of c; by Property 4.3 the plan derived in $\mathcal{D}$ is the plan of c in $\mathcal{D}'$, and Properties 4.5 and 4.6 give the step and a well-typed result.

4.11 Implementation

Our implementation compiles G# to native code through LLVM, both for just-in-time execution and as linked executables; Chapter 5 measures its cost. This section explains how the compiled code follows the rules: its records of ancestry agree with principal instantiations, inference reads only completed values, and each RI call performs the steps of the rules. None of this is proved.

Ancestry Every object's first word points to the descriptor of its exact type (Section 2.6), so upcasting moves no data. A descriptor holds the dispatch table, the type arguments, each an interval of descriptors because a projection denotes one (Section 3.3.1), and the *ancestry*, which holds the principal instantiation $\mathsf{inst}(T, N)$ for every head N above the type, built the first time the type is tested, so the read of Read-Step at one level is one lookup (Figure 4.17). A type that no source line writes, such as the intersection a join produces, receives its descriptor on first use, looked up by its canonical form so that one type has one descriptor. Descriptors built at run time instantiate heads of the one compiled image; new declarations enter as a recompiled image. As in the CLR's shared generic code [Kennedy and Syme, 2001; Yu et al., 2004], a generic method is compiled once and receives its type arguments, explicit or inferred, as hidden descriptor parameters.

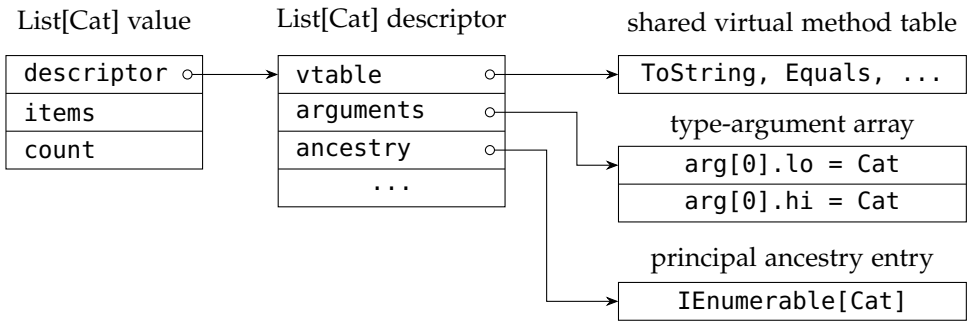


Figure 4.17: Selected logical descriptor links for the ToList source. The exact `Cat` argument has equal endpoints, and the entry `IEnumerable[Cat]` stands for the interval $[\bot, \texttt{Cat}]$ of its covariant slot. Interface dispatch and other fields are omitted.

Completed values Definite initialisation prevents an RI call from reading an unfilled slot, and the emitted code keeps the object and its descriptor reachable by the garbage collector during the read.

Correspondence with the rules An RI call performs the steps of Section 4.9 in order: it evaluates the arguments once, reads each argument's ancestry at the heads of its occurrence paths, folds the reads, checks each fold against $\eta\tau_X$, applies couple_σ, whose fix_X takes the steps of (4.1) through the ancestry, and passes the assignment to the callee; a type argument whose interval is a point is emitted as if explicit. The

type operations and the plan are implemented once, in a library that the checker and the runtime share, and an RI call that caches its answers (Section 5.3.2) stores the assignment that the plan infers. The correspondence is not proved.

Chapter 5

Evaluation

This chapter asks two questions about runtime inference: what it gives programmers, and whether it can be provided at low cost. For the first, an inferred type argument gives programs behaviour that explicitly typed languages obtain only by registering extra type evidence (Section 5.1), and in a port of a LINQ library and other existing benchmarks most generic calls omit their type arguments, with the library's signatures unchanged apart from projected results (Section 5.2). For the second, the cost of inference is small at run time and is paid mainly in the size of the generated code (Section 5.3).

5.1 Programming capabilities

Two programs show what an inferred type argument changes while the caller keeps its static view, and a comparison with three explicitly typed languages shows what they need instead.

5.1.1 Collection materialisation

Copying isolates what the type argument controls: the type of the newly constructed list. Figure 5.1 gives the complete body. It reuses the enumeration interfaces of Figure 2.4; the supporting list keeps its elements in a `T[]` whose unused slots hold a fill value given to its constructor, so the body takes the source's first element as that value and halts with a `throw` on an empty source (Appendix B). The program calls `ToList` on a `List[Cat]` held through the view `IEnumerable[Animal]`, once with the explicit type argument `Animal` and once as an RI call. Both copies retain the contents and order and expose animals through the static `List[out Animal]` view. They differ in the instantiation: the explicit copy is a `List[Animal]`, the inferred copy a `List[Cat]`. A type test `is IEnumerable[Cat]` therefore succeeds on the inferred copy, which can be passed to a consumer requiring cats, and fails on the explicit one even though its current elements happen to be cats. This is a consequence of the container's type, not of a scan.

5.1.2 Typed event delivery

An event bus delivers each event to the handlers that can accept it, and the generic delivery operation should work when the caller holds an event through a common supertype. Figure 5.2 gives its declarations. The registry contains an `ActivityLog`

```
static List[out T] ToList[T](IEnumerable[T] source) {
    IEnumerator[T] e = source.GetEnumerator();
    if (!e.MoveNext()) { throw "empty source"; }
    List[T] result = new List[T](e.Current());
    result.Add(e.Current());
    while (e.MoveNext()) {
        result.Add(e.Current());
    }
    return result;
}
```

Figure 5.1: The same copying body executes with the inferred type argument. It traverses the source; inference does not inspect the elements to guess their types.

handler accepting `Event` and an `AutoBackup` handler accepting `FileSaved`, both implementing `IHandler[in E]` (Figure 2.5). The generic operation `Publish[E](E e)` calls each registered handler that passes the test `is IHandler[E]`. The complete program is in Appendix B. The caller holds its events as `Events`. The explicit

```
class EditorEvent : Event { ... }
class FileOpened : EditorEvent { ... }
class FileSaved : EditorEvent { ... }
class ActivityLog : IHandler[Event] { ... }
class AutoBackup : IHandler[FileSaved] { ... }

void Publish[E](E e) {                    // in class Bus
    for (int j = 0; j < handlers.Length(); j = j + 1) {
        if (handlers[j] is IHandler[E] h) { h.Handle(e); }
    }
}

Event[] events = ...;  // a FileOpened, then a FileSaved
bus.Publish[Event](events[i]);            // explicit
bus.Publish(events[i]);                   // RI call
```

Figure 5.2: The declarations and calls of the event bus, with constructors and counters omitted.

call fixes `E` = `Event`, so the test is `is IHandler[Event]`: `ActivityLog` passes, but `AutoBackup` cannot handle an arbitrary `Event` and fails. The RI call infers `E` from each event's runtime type: `E` is itself the parameter type, a direct occurrence, so the call reads the event's exact runtime type (Table 2.1 and Figure 5.3). For the `FileSaved` the test is `is IHandler[FileSaved]`, which `AutoBackup` passes, and so does `ActivityLog`, since `IHandler` is contravariant and a handler of every `Event`

handles a `FileSaved`. The caller's code is the same; only the type argument differs, and it decides which handlers run.

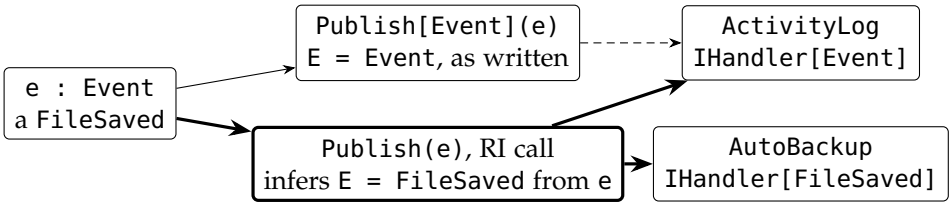


Figure 5.3: Handlers reached by each call form for a `FileSaved` held as an `Event`; the RI call is bold. A `FileOpened` reaches only `ActivityLog` under either form.

5.1.3 Typed alternatives

C#, Java and Kotlin programs written for the same delivery behaviour are excerpted in Appendix C, and Table 5.1 compares the extra construct each needs [Microsoft, 2026b, 2021; Oracle, 2025c,d; JetBrains, 2025].

Table 5.1: What each language adds to reproduce the delivery behaviour of the inferred G# call. The C# program prints the same output as the G# program's RI call; the Java and Kotlin programs are written to it but were not compiled.

Language	Extra construct at registration	Where the event type is learned
G#	none	Inferred at the `Publish` call from the event's descriptor
C#	a generic adapter class retaining `T`, with a type test in `Deliver`	Registration, by the adapter's instantiation
Java	a `Class<T>` token stored with an `IHandler<? super T>`	Registration, by the token's `isInstance` test
Kotlin	a class token, or an inline function with a reified type parameter	Registration, by the token or the reified parameter

5.2 Applicability to existing generic code

The capabilities above come from small programs written to show them. This section asks how often inference applies in generic code written for another language, and what that code must change to obtain it.

Workloads The workloads are translations of existing benchmarks plus small programs that exercise inference. The two largest are .NET performance suites for LINQ, the generic query library of .NET: `LinqBenchmarks` (*LINQ operators*), of which we

Table 5.2: The three classes of LINQ operator in the port, by whether signature checking finds them eligible.

Class	Examples	Eligible
Occurs at out only	`Concat`, `Skip`, `Take`, `Count`, `ToList`	yes, out-directed
Occurs at out and in	`Where`, `Select`, `Zip`, `OrderBy`, `GroupBy`, `Contains`; predicate forms of `Count`, `Any`, `First`	yes, out-directed; the other arguments checked
Not determined by an argument	`Cast`, `Empty`	no

Table 5.3: Method type-parameter sites of the inferred variants: inferred at run time (RI), or explicit, either written in both variants or filled in before type checking where the call site determines the answer.

			Explicit	
Workload	Sites	RI	Written	Filled in
LINQ operators	61	48	13	0
LINQ enumerable	113	93	19	1
Sieve	11	8	0	3
Float	4	1	0	3
Intersort	5	0	0	5
Six smaller programs	15	11	3	1
Total	209	161	35	13

translate 17 of its 20 benchmark methods over 77 product records (the three query-syntax cases are excluded), and `Perf_Enumerable` (*LINQ enumerable*), of which we translate 52 methods and 98 benchmark instances [.NET Foundation, 2026b,a]. Both use one G# library of sequence operators, collection materialisation, maps and sorting. The others are the *sieve*, *float* and *intersort* programs of the MonNom benchmarks for nominal gradual typing [Muehlboeck and Tate, 2021] and six smaller programs: a sequence pipeline (*LINQ pipeline*), *collection copy*, map and set construction with sorting (*maps and sets*), *event delivery*, an expression-tree *visitor* and a *numeric kernel*. The cost figures and tables list them in this order and in these three groups, without intersort for the reason below. Every program is written twice: an *explicit* variant (E) that gives the type arguments of each generic method call explicitly and an *inferred* variant (R) that omits them wherever the method is eligible. The two differ only in how they write generic calls and must print identical output. All of intersort's generic calls have their type arguments filled in before type checking (Table 5.3), so its variants compile alike and it has no pair whose cost differs.

A LINQ library LINQ operators such as `Where` and `Select` take a sequence and return a new one, and operators such as `Count` and `ToList` consume one. They are generic in the element type of a sequence, are called through views of sequences whose element type has been widened, and take functions over that element type, which is the setting runtime inference is designed for. Table 5.2 sorts the library's generic operators by what signature checking decides. Every operator keeps the parameter types System.Linq gives it, with each delegate parameter written as a function-object interface, since G# has no function values; the only change is that an operator that builds a collection, such as `ToList`, returns it through a projected view.

Inference on the workloads Table 5.3 classifies the method type-parameter sites of each workload's inferred variant. A *site* is one type parameter of one generic call. The filled-in sites omit their type arguments in the source, but the call site determines them before execution, as for an argument constructed in place; the compiler writes them in before type checking, by the call-site rules of Section 7.2, so the type system sees explicit calls, which need no inference at run time. They include all of intersort's sites and three of float's four. In the two LINQ suites, every operator call the benchmark methods make is inferred except the `Cast` and `Empty` calls of the enumerable suite, whose type parameter no argument determines (Table 5.2); the other explicit sites are calls inside the library's own generic helpers, which pass their own type parameter on explicitly, as the ported source does.

5.3 Cost of runtime inference

We first measure what omitting type arguments costs the workloads of Section 5.2, and then the cost of one RI call, which explains where that cost arises and where it grows.

5.3.1 Whole programs

Programs run as single-threaded linked ARM64 executables on an Apple M4. Each has eleven repetitions with E and R alternating, counting instructions with the collector disabled and wall time with it enabled. Two zero-inference controls compare the explicit LINQ operator program with itself after reordering its declarations or adding source padding, which shows how far layout alone moves a reading. Figure 5.4 shows the results. With collection disabled, eight of the ten programs execute the same number of instructions in both variants to within 0.05%. The two LINQ suites, which have the most RI sites (48 and 93, Table 5.3), execute 0.1% and 0.2% more; the declaration-order control, which involves no inference, moves by 0.14%, so layout alone can change a count by about that much, and the 0.2% of LINQ enumerable is only slightly above it. With the collector enabled, two wall-time intervals exclude zero. LINQ enumerable runs 0.5% slower in its inferred variant; maps and sets runs 1.8% faster, although its two variants execute the same number of instructions, so the instruction count does not explain that difference. The other intervals, including

both controls', contain zero and extend up to about 4% above it, so smaller wall-time differences are not resolved.

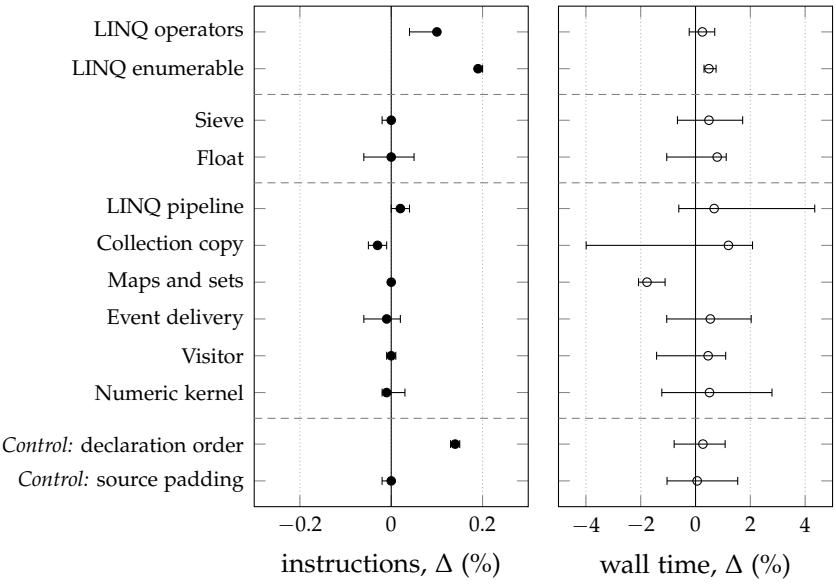


Figure 5.4: Runtime cost of omitting type arguments in whole programs: the median of the paired change $100(R_i/E_i - 1)$ of the inferred over the explicit variant, with its 95% bootstrap confidence interval. Dashed lines separate the LINQ suites, the MonNom ports, the smaller programs and the two layout controls, which involve no inference. The medians are in Table D.1.

Inference adds no measurable memory: where the heap holds the program's data (about 170 to 920 MiB), peak resident size differs by less than 0.1% between the variants, and in the few-MiB LINQ programs it follows the collector's schedule rather than the variant. The cost of omitting arguments falls instead on compilation, because each RI site emits its own lookups, combine and cache: the machine code of the inferred LINQ suites is 4% and 5% larger and that of the LINQ pipeline 9% larger. Building takes up to 7% longer.

These results come from one machine and from benchmarks we translated, with reference objects and function objects in place of .NET's value types and delegates; they compare G# with and without inference, not G# with .NET.

5.3.2 Cost of one RI call

At run time an RI call reads its arguments' descriptors, combines the reads and checks the result (Section 4.11). A read at one level of an occurrence path is one lookup in the argument's ancestry, which is built the first time its type is tested. Where a parameter has a single read and nothing to combine it with, as in `ToList`, that read is the answer. Otherwise the call caches its answers: each step of the fold looks up a table keyed by the type read, the result so far and $\eta\tau_X$, and on a miss combines and checks once and stores the answer. The guarantee is what makes

this cache sound: the optimum is unique and depends only on the arguments' exact types and the closed static assignment, so a stored answer is the one the computation would give again. A table holds at most 4,096 rows; when it is full, an older answer is evicted, and a call that needs it again computes the same answer again.

We measured these costs with a program that executes one RI call in a loop and counts the retired instructions per call (Figure 5.5). Three quantities vary: the depth of the inheritance chain above the argument's class, from 1 to 16; the number of conjuncts of the inferred type, an intersection of 2 to 200 conjuncts; and the number of distinct argument types that the loop cycles through, from 1,024 to 8,192. A *first* call is the first on its argument type, before the ancestry and the answer exist; a *reused* call repeats an argument type whose answer is cached.

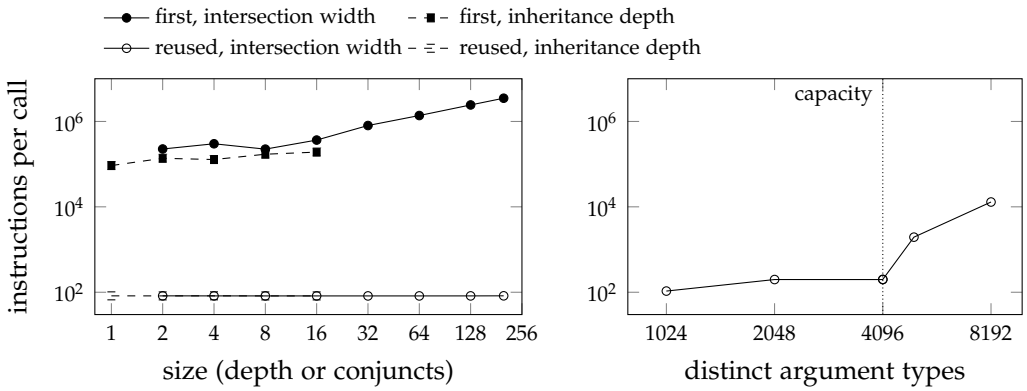


Figure 5.5: Runtime cost of one RI call, in retired instructions (medians of eleven runs, log scales). Left: first and reused calls by inheritance depth and intersection width. Right: reused cost as the number of distinct argument types crosses the 4,096-row capacity of a table.

A reused call costs 82 instructions at every depth and width, the cost of one lookup. A first call costs between 0.09 and 0.19 million instructions for depths 1 to 16, with wide variation between runs, and from 16 conjuncts grows with their number, to 3.5 million for 200. Cycling through 8,192 argument types, twice a table's capacity, evicts answers before they are reused, and a call then costs about 13,000 instructions, with every answer unchanged. Below the capacity the reused cost already rises, to about 200 instructions from 2,048 argument types. A program therefore pays the first-call cost once for each new argument type and the cached cost on every later call.

Chapter 6

Related Work

This chapter relates G# to four lines of work: inference of type arguments at run time (Section 6.1), inference at compile time (Section 6.2), runtimes that retain type arguments (Section 6.3), and the variance and subtyping results that G#'s type system builds on (Section 6.4). For each we describe what is inferred or chosen, from what information, and what is guaranteed about the answer, and compare it with the guarantee of G# (Definition 4.4): from a method's signature alone, every well-typed RI call has a unique optimal assignment, for every argument value it may receive and every addition of type declarations, and the runtime computes that assignment.

6.1 Inference at run time

Principally inferable generic methods The closest prior design is the last part of Muehlboeck's dissertation, which starts from C#. With `B1` and `B2` extending `A`, `Singleton(x)` building a one-element list and `Snoc(l, x)` appending `x` to `l`, C# infers `B1` for both calls in `Snoc(Singleton(b1), b1)` when `b1` is declared at `B1`, so the invariant result `List<B1>` cannot take `b2`, although giving `b1` the static type `A` would have allowed it [Muehlboeck, 2019, Section 8.2.2]. `Singleton`, unlike `Snoc`, can always be given a principal type, a property of its declaration; even so, reified type arguments make C#'s semantics incoherent, since inferring its type argument statically as `A` or dynamically as `B1` allocates lists that accept different elements [Muehlboeck, 2019, Sections 8.2.3 and 8.2.4].

The dissertation's calculi therefore check at each method signature that the inferable type parameters are principally inferable (Section 2.9). Each is written with a variance that fixes its strategy: an invariant parameter must be uniquely determined by the argument types, a covariant one is inferred as the minimal type satisfying its constraints and a contravariant one as the maximal, and the return type must be covariant in these parameters, so that the strategies give the most precise return type [Muehlboeck, 2019, Section 8.5.1]. Inferability is checked one component of the signature at a time, each later parameter imposing no new constraint on the type parameters already inferred [Muehlboeck, 2019, Section 8.5.3]. To keep reified generics coherent, the calculi give method invocation a semantics that infers the principal type arguments for the runtime types of the arguments; the type arguments inferred at compile time only ensure that the invocation is safe and bound its result [Muehlboeck, 2019, Sections 8.5 and 8.7.1]. The semantics is declarative: signature inferability guarantees that principal type arguments exist for every ty-

pable invocation, and a proof of that judgement yields an algorithm [Muehlboeck, 2019, Section 8.7.2]. The dissertation evaluates the design by a corpus study of how many generic methods would be inferable and leaves measuring runtime inference as future work [Muehlboeck, 2019, Sections 8.5.5 and 8.7.1].

G# adopts this division of labour between compile time and run time and differs in what it states and supplies. It derives each direction from the signature, which the dissertation notes is possible in most cases but writes explicitly for simplicity [Muehlboeck, 2019, Section 8.5.1]. It does not require the result to be covariant: an occurrence of a parameter in the result type that is invariant or opposite to its direction shrinks its interval to a point, and the call then behaves as if the type argument were explicit (Section 4.6); `ToList` returns the projection `List[out T]` so that its result does not hold `T`. Its guarantee asks for one optimum of the whole assignment in the product order, including parameters coupled by bounds, in every execution of Section 4.4, including those with added declarations; and it gives the computation as runtime rules (Section 4.8), implements them and measures their cost (Chapter 5).

Multiple dispatch With *overloading*, several declarations share one method name and differ in their parameter types; under *multiple dispatch*, the dynamic types of the arguments choose among them, and dispatch is *symmetric* when no argument is privileged [Park et al., 2019]. Castagna, Ghelli and Longo place two conditions on a set of overloaded branches: any two whose domains have a common subtype need a third at their greatest lower bound, the *meet rule*, so that the most specific branch is defined and unambiguous; and a branch with a smaller domain must have a smaller result type, so that resolution preserves types [Castagna et al., 1995]. Allen et al. extend these conditions to Fortress, with symmetric multiple dispatch, multiple inheritance, parametric polymorphism and modular checking, as the No Duplicates, Meet and Return Type Rules [Allen et al., 2011, Section 3.2]. To make useful overloadings valid, Fortress also restricts declarations: types related by *exclusion* may have no nontrivial common subtype, and a type may not extend several instantiations of one generic type [Allen et al., 2011, Section 1]. When dispatch selects a declaration more specific than the one used at compile time, its type parameters must be inferred at run time, and Allen et al. leave how to do so beyond their scope [Allen et al., 2011, Section 3.1].

Park et al. address that step. In their calculus a call selects at run time the most specific applicable declaration and solves its method type parameters from the dynamic types of the arguments and the call's static return type [Park et al., 2019]. Their uniqueness result concerns the declaration: a unique most specific applicable one exists [Park et al., 2019, Theorem 4.1]. For its type arguments they prove that a valid substitution is found whenever one exists [Park et al., 2019, Lemma 5.4], and the solver takes the intersection of the upper bounds; the authors considered the union of the lower bounds but could not prove it correct [Park et al., 2019, Section 5.4]. The answer is therefore valid, but not optimal by any stated preference, as their own example illustrates [Park et al., 2019, Section 5.2.4]. With a covariant `List`, classes `C1`

and `C2` below `C3`, and `append[P, Q]` over `List[P]` and `List[Q]` with `P` below `Q`, a call on two lists of static type `List[C3]` that hold a `SortedCons[C1]` and a `Cons[C2]` is solved right to left, `Q` and then `P` each taking the intersection of its upper bounds, giving `C3` for both: the static type arguments. Under the rules of Chapter 4, the same signature with the result written `List[out Q]` gives `P` the type `C1` and `Q` the join of `C1` and `C2`, the most precise types the bound allows. Method type parameters in their calculus are scoped left to right and occur only in lower bounds, which is what lets one right-to-left pass solve them [Park et al., 2019, Sections 3.3 and 5.2.3], and their rules are checked over a fixed program. For a call such as that of Figure 4.1, where `(Cat, Cat)` and `(Animal, Animal)` are both admissible and neither is better, no best answer exists at all.

G# makes the opposite trade. A method is identified by its name and arity, with no overloading on parameter types (Section 3.7), so dispatch chooses only a body and there is one admissible set of assignments per call. The Return Type Rule [Allen et al., 2011, Section 3.2] reappears as result agreement (Section 4.4); a type may extend several instantiations of one generic type, merged by principal instantiation (Section 3.3.2), as the Ancestor Meet Rule allows, which later relaxed this Fortress restriction [Park et al., 2019, Figure 4 and Section 5.4]; and where the dispatch line checks the meet rule once per set of declarations, G# decides eligibility once per signature, so that every call has an optimum, or its type arguments must be written.

Recovering hidden type arguments TameFJ [Cameron et al., 2008] recovers a type argument that a wildcard hid, a type argument written `?` that stands for an unknown type, in a formal model of Java's source semantics whose values are written `new C<T>(v)`, while Java itself erases the arguments the model carries [Oracle, 2025b, Section 4.7]. The hidden argument is read off the argument's class type by subclassing, only from occurrences within a generic type; the paper states type soundness, by progress and subject reduction, and no uniqueness or optimality [Cameron et al., 2008, Sections 3.4–3.6]. Cimadamore's gCVM, a virtual machine with reified generics, performs the closest runtime step we know of: when a generic method is called with an argument whose static type hides a type argument behind a wildcard, it finds the method's type variable in the formal parameter type and follows that path in the descriptor of the argument's runtime type; for `copyFst(Pair<U,V> p)` called on a `Pair<?,String>` that holds a `Pair<Integer,String>`, `U` becomes `Integer` [**?**, Chapter 4, Section 4.3.2]. This is the read of Read-Step, taken from the first occurrence found, for a variable that Java's capture, which names a wildcard's hidden argument by a fresh type variable, makes exact, so there is nothing to choose. G# reads every occurrence, combines the reads by the direction of the parameter, and states at the signature when the combination is the unique optimum.

Swift opens an existential value passed to a generic function, so that its hidden type becomes the generic argument, but only where one existential argument binds the parameter directly, since a single underlying type must be guaranteed; the result type is erased back to an upper bound [Gregor, 2022]. Like eligibility, this refuses rather than chooses, but it decides per call from the shape of the arguments. Both

lines read a hidden argument as an *existential type*: `List<?>` is a list of some type X, $\exists X.\,\texttt{List}[X]$ [Tate et al., 2011, Section 2.2]. G# treats a projection as a type, not a binder: it names a set of possible arguments (Section 3.3.1) but introduces no variable, so an inferred type argument is always a type the program can write, and positions that only an existential variable can relate are what G# gives up.

Julia and two patents Julia chooses a method at run time by subtyping over parametric types; Belyakova et al. show a core of this subtyping undecidable and recover decidability by restricting value types to use-site variance [Belyakova et al., 2024]. Julia is dynamically typed, so no static type bounds the answer, and a call with no most specific method is a runtime error [Zappa Nardelli et al., 2018]; G# decides before execution which calls may leave their type arguments to run time. Two patents infer type arguments at run time without a stated guarantee. Bracha's runtime keeps uncertain bounds for the type arguments of generic objects created, and generic methods invoked, without type arguments, which Java calls raw, and tightens them as they are used [Bracha, 2010], so its answer depends on the execution history. Naden et al. infer over a type interval for overloaded functions, choosing its upper or lower end by a heuristic over the variance of the parameter's occurrences, starting from the return type, to favour the most specific types [Naden et al., 2014]; the interval and the use of variance are shared with G#, but no uniqueness or optimality is claimed.

6.2 Inference at compile time

Local inference Local type inference [Pierce and Turner, 2000] and C# method type inference, introduced in Section 2.9, fix a call's type arguments from static types alone. Pierce and Turner choose the type arguments that give the application its least result type, and reject the application when no choice does: for `id(x)` with `x` an `Int`, both `Int` and `Real` are valid and `Int` is chosen, while `f()` with `f` of type $\forall X.\,() \to (X \to X)$ has no best result type and is rejected [Pierce and Turner, 2000, Section 3]. Since each term of their explicitly typed internal language has a unique minimal type [Pierce and Turner, 2000, Theorem 2.3.1], every application they accept has a principal type, but whether one is accepted depends on the application; G# makes the analogous decision at the declaration, so that `Empty`, like `f`, is not eligible and every call writes its type argument. C# fixes each type parameter to the one candidate among its bounds to which all the others convert, so its method type inference computes no join, and when there is none the method takes no part in overload resolution [ECMA, 2023, Sections 12.6.3.1 and 12.6.3.12]. Java's `lub` only approximates a least upper bound [Oracle, 2025a, Section 4.10.4], and Smith and Cartwright show that Java 5 inference is by design neither sound nor locally complete: an unsound answer is caught when the call is checked, and a missing one is supplied by writing the type arguments [Smith and Cartwright, 2008]. For the running `ToList` call all three answer `Animal`, the element type of the static view. G#'s type checking of an RI call is of this kind and supplies the static assignment (Section 4.3); the runtime then

reads the retained type within the checked interval. Since no later check can repair a wrong runtime answer, G# decides at the signature whether calls may omit their type arguments.

Global inference Global inference infers the types of declarations as well as calls. For Featherweight Generic Java without polymorphic recursion it is NP-complete [Stadelmeier et al., 2022, Theorem 16], and Java-TX extends it to Java with wildcards, with a type-soundness proof for its core calculus; its underlying unification is NP-hard [Stadelmeier et al., 2026]. G# infers no declaration types and solves each call independently.

Principality G# ensures principal typing in Muehlboeck's sense (Section 2.9), checked at the declaration [Muehlboeck, 2019, Sections 8.2.3 and 8.5.3]. It is weaker than Jim's principal typings, which also infer the type environment [Jim, 1996, Section 1], whereas Γ is given in G#, and it is not the principal type scheme that algebraic subtyping provides.

Algebraic subtyping MLsub introduced algebraic subtyping, in which types form a lattice with union and intersection, constraints are solved by biunification and every typable expression has a principal type scheme [Dolan and Mycroft, 2017; Dolan, 2016]. Simple-sub obtains the same system with a simpler algorithm that agrees with MLsub on generated expressions [Parreaux, 2020], and MLstruct adds first-class unions, intersections and negation over mostly structural types, and is by its authors' account the first language with principal polymorphic inference for union and intersection types [Parreaux and Chau, 2022]. G# shares the polarity discipline that separates in from out positions and the use of joins and meets to combine constraints, but differs in four respects.

First, the object inferred differs. MLsub gives `select p v d = if (p v) then v else d` the scheme $\forall \alpha, \beta. (\alpha \to \mathtt{bool}) \to \alpha \to \beta \to (\alpha \sqcup \beta)$, whose result is the union of what flows in [Dolan and Mycroft, 2017, Section 1]; the scheme stands for every use, and no type argument is chosen. A G# method `Select[T](IPred[in T] p, T v, T d)` has one type parameter, out-directed, and for a `Cat` and a `Dog` the runtime chooses their nominal join, `Animal`, against which the predicate is checked. G# must pick one concrete assignment for one execution, and principality of a scheme does not imply that a unique optimal assignment exists. Second, mutable state: MLsub's calculus has none [Dolan and Mycroft, 2017, Section 1.1], and Dolan's thesis types mutable collections by replacing each invariant parameter with an input and an output type [Dolan, 2016, Section 9.1.1]; in G# an invariant occurrence fixes its type argument and a projected result hides writes, so result agreement is part of the guarantee. Third, the family is at its core structural with first-class unions, whereas G# is nominal, with declared ancestry and declared variance, which lets the runtime read one ancestry entry per head; a union, as in `X f[X](Int | X x)` applied to an `Int`, would leave `X` unconstrained and offers no single head for that read (Section 3.5).

Fourth, the family solves once, at compile time. Dolan's system is extensible: adding types never makes a typable program untypable [Dolan, 2016, Section 1.3.1], where the added types leave the subtyping between existing types unchanged [Dolan and Mycroft, 2017]. G# needs more, since the assignment it computes must stay optimal when subtypes of existing types are added, so the supertypes of existing declarations are fixed (Section 2.8). The loss of unions is a genuine limitation: Castagna argues that set-theoretic types are needed to type some common patterns precisely [Castagna, 2021].

6.3 Runtimes that retain type arguments

Runtime inference needs retained type information; retention is a premise of G#, not a contribution. The CLR keeps a type record for each instantiation and makes it available to type tests and reflection (Kennedy and Syme, 2001, Sections 1 and 4.2; Yu et al., 2004). A type argument that a C# call leaves out [Kennedy and Syme, 2001, Section 2.3] is fixed at compile time [ECMA, 2023, Section 12.6.3], or, under `dynamic`, by the same inference applied at run time to the run-time types [ECMA, 2023, Sections 12.3.3 and 12.8.9.1]. Muehlboeck and Tate show what that costs [Muehlboeck and Tate, 2017, Section 10.3.2]: in their copying-and-appending example, the inner `Enumerable.ToList` call is given the type parameter `T` of the statically typed `SnocS`, which is `object` in their call, but `string` in the dynamically typed `SnocD`, where appending the integer `5` then fails. They call type-argument inference for gradual generics an unsolved problem; MonNom's implementation carries type arguments in its gradual-typing checks [Muehlboeck and Tate, 2021] but describes no inference of a call's type arguments. G#'s eligibility rules out such failures before permitting inference, and also establishes optimality.

Other designs pass type arguments without choosing them. Monomorphisation specialises code per instantiation [Griesemer et al., 2020; Ellis et al., 2022], dictionary passing hands the argument to shared code [Ellis et al., 2022], NextGen keeps a wrapper class per instantiation [Cartwright and Steele, 1998; Allen et al., 2003], and GJ erases it after inferring it locally [Bracha et al., 1998]. Java's `Class` tokens and Kotlin's reified parameters carry type evidence that the caller or compiler supplies from the call site's static types [Oracle, 2025d; JetBrains, 2025], which is what the typed alternatives of Section 5.1.3 must add. Harper and Morrisett pass types to polymorphic code that branches on them with `typecase` [Harper and Morrisett, 1995], and Crary et al. do so with term-level representations of types [Crary et al., 1998]; there the program inspects a type and chooses, whereas in G# the compiler fixes which types a call reads and the runtime combines them into the single optimal argument. In each of these designs, the type arguments that a call passes come from static information; none is chosen from the values of its arguments.

6.4 Variance and subtyping

Variance Igarashi and Viroli read a projection with a simple argument as the set of objects whose exact instantiation lies within its bound [Igarashi and Viroli, 2006, Section 3.1], Craciun et al. infer use-site annotations as intervals of a lower bound for writing and an upper bound for reading [Craciun et al., 2009], and Altidor et al. combine definition-site and use-site variance [Altidor et al., 2011]; C# and Java supply legality rules that G# follows or avoids [Emir et al., 2006; Torgersen et al., 2004]. All decide at compile time which conversions and member accesses are permitted. G# also uses the interval at run time, as the range in which an observation must fall.

Joins and principal instantiation Greenman, Muehlboeck and Tate separate nominal types into materials, which can be used as value types, and shapes, which occur only in inheritance and as bounds, and prove joins computable under that separation for arguments with explicit in and out bounds [Greenman et al., 2014, Sections 3.2–3.3 and 5.3]. G# stores each argument as such a pair of bounds but declares variance at the declaration, so it states its own restrictions (Section 3.7.1) and termination argument. Muehlboeck and Tate formalise the principal instantiation of an intersection, joining contravariant arguments and meeting covariant ones, and a disjointness relation that is decidable and sound but not necessarily complete [Muehlboeck and Tate, 2018, Sections 6.2–6.3]. Their principal instantiation joins contravariant arguments by a union, so it exists for an intersection of type variables too, and $A \cap B$ of two type variables is the only principal type in their motivating example [Muehlboeck and Tate, 2018, Section 2]; their extension to bounded type variables reads a variable through its upper bound [Muehlboeck and Tate, 2018, Section 7.4], as G# reads a type parameter through its bound. G#'s join takes the place of their union, and gives the same principal instantiation for ground types; for types that mention type parameters, the join under Δ can exceed the join at a closing, so G# combines two arguments only where they are stable (Section 3.3.1); an argument without such a principal instantiation leaves its RI call ill typed rather than letting typing choose between its conjuncts (Section 4.7). Its disjointness is, like theirs, sound but incomplete. Their formalisation assumes single-instantiation inheritance; the principal-instantiation inheritance that G# uses comes from Ceylon [King, 2017], which, unlike G#, marks a use-site type argument as covariant or contravariant but not both, and Scala 3 states the corresponding identity for a covariant constructor [Scala Center and EPFL, n.d.a]. Muehlboeck and Tate also criticise a nominal semantic model that assumes a closed world and so treats interfaces with no current common implementing class as disjoint [Muehlboeck and Tate, 2018, Section 7.7]; this is the hazard that G#'s guarantee quantifies over added declarations to exclude.

Decidability Kennedy and Pierce show nominal subtyping with variance undecidable, conjecture that undecidability needs contravariant constructors, *expansive* inheritance, in which a cycle of declarations makes type arguments grow without bound, and several supertypes with one head together, and show subtyping between closed

types decidable without expansive inheritance [Kennedy and Pierce, 2007]. Grigore proves Java's subtyping undecidable and refutes the third ingredient: contravariance and expansive inheritance suffice [Grigore, 2017, Section 8]. Misonizhnik and Mordvinov show that satisfiability of subtyping constraints over open types stays undecidable even without expansive inheritance [Misonizhnik and Mordvinov, 2019]. G# permits several supertypes with one head but rejects expansive inheritance, even through shapes (Section 3.7.1), and its runtime works on ground types, so it never decides satisfiability over open types.

Values Runtime inference reads the descriptor of every argument value, so a value without one, such as null, would leave it nothing to read. Amin and Tate turn any type into any other in Java and Scala through null, with wildcard capture in Java and path-dependent types in Scala; the Java programs need no `null` expression, since a field read before its constructor assigns it holds null [Amin and Tate, 2016]. Bierhoff avoids the unsoundness by ensuring that witness types, values showing that an existential type is inhabited, exist when an existential is introduced [Bierhoff, 2022], and Fähndrich and Leino track partially constructed objects with a separate raw type [Fähndrich and Leino, 2003]. G# has neither source: its types are non-null, definite initialisation covers every slot (Section 3.2), and its projections bind no type variable.

6.5 Summary

Table 6.1 compares what each line of work states against four properties of the G# guarantee: the answer is optimal, it is unique, it depends on the runtime types of the arguments, and it stays valid when declarations are added. Most of these systems prove theorems, about different objects; G# contributes a formal statement of its guarantee, rules designed to secure it and an implementation, under premises (P1) and (P2) of Section 4.10.

Table 6.1: The properties of the G# guarantee that each line of work states; – means not stated or not applicable.

Approach	Optimal	Unique	Runtime types	Stable under new types
Pierce and Turner	partly	yes	no	–
Java 5 (Smith and Cartwright)	no	–	no	–
MLsub, Simple-sub	partly	partly	no	yes
Castagna et al.	yes	yes	yes	–
Fortress	yes	yes	yes	yes
Park et al.	partly	partly	yes	no
Muehlboeck	yes	yes	yes	–
TameFJ, gCVM, Swift	–	partly	yes	–
Julia, patents	–	–	yes	–
G#	yes	yes	yes	yes

Concluding Remarks

We first discuss the limitations of this work and the directions we think most fruitful for future work, and then close with an overview of the thesis.

7.1 Limitations

The language cost of the guarantee The guarantee is bought with restrictions that a programmer meets, each justified in Chapter 3. The most visible is that a result refined at run time is returned through a projected view, and a projection restricts the members that use its type argument according to its direction (Section 3.3.1). Through `List[out Animal]`, the result of `ToList`, elements are read as `Animal`, but `Add` would need a `Never` and cannot be called; through `List[in Cat]`, `Add` accepts a `Cat`, but elements are read only as `object`. The C# `List<Animal>` permits both reading and adding animals, and is often what the programmer wants. An exact result type cannot give this for a copy whose type argument is chosen at run time: the copy may be a `List[Cat]`, so adding a `Dog` would need a check at run time that can fail (Section 4.4). The other declaration restrictions of Chapter 3 are smaller; the one with most effect on existing code is the absence of union types (Section 3.5).

Completeness of eligibility Eligibility is a sufficient test, not a complete one. Because it is decided from the declaration alone, a method is not eligible when its signature cannot establish an optimum for every call, although a particular call may have one, such as `Link` (Figure 4.1) on a source and a sink both constructed at `Animal`; the programmer must then give the type arguments explicitly. A more complete signature test, for instance for a type parameter inside a generic type below an `in` level, as in `IHandler[List[out T]]`, which eligibility rejects although a particular call may have an optimum, would accept more methods. Type checking is likewise incomplete for an argument whose static type is an open intersection without a principal instantiation (Section 4.7): its call must give its type arguments, although the value read at run time determines an optimum.

Proof We have not proved the properties of Section 4.10. Their arguments are informal and rest on premises (P1) and (P2), which we assume rather than re-establish for the projections, intersections, shapes and definite initialisation of G#. Nor have we proved that the compiled code follows the rules (Section 4.11).

7.2 Future work

Call-site rules Some calls have type arguments that the call site, though not the signature, determines. An argument whose static type no declaration could extend, as a language with a modifier forbidding subclasses would allow, or one constructed in place has exactly its static type at run time, and a context may expect a type, as `List[Cat]` does for `Program.Empty()`. A tool outside the language could collect such special cases as rules and write the type arguments they determine into the source before compilation, as the compiler does for the filled-in sites of the workloads (Section 5.2). The rewritten call is an explicit one, checked by T-Call-Explicit: it needs no inference at run time, and it may call a method that is not eligible, such as `Link` above. Neither eligibility nor the guarantee would depend on such rules.

Mechanised proof A mechanised proof of the properties of Section 4.10 is a natural next step, as is verifying that the compiled code follows the rules (Section 4.11).

Language features G# has no `null` and no optional type, so a container takes a fill value for its unused slots when it is constructed; an optional type designed for runtime inference, whose absent value carries a type that inference can read, is future work. G# also has no function values or lambda expressions; adding them needs rules for captured state and for inference through function types. Value types with unboxed storage could reduce allocation, but must still carry the type information that runtime inference observes.

Concurrency and dynamic loading The language has no threads; concurrency would need a memory model for publishing descriptors. The implementation runs one compiled image; loading modules would need identifier relocation, declaration checks and safe publication, although the guarantee already covers every permitted addition of type declarations.

7.3 Conclusion

In this thesis we have presented the design and implementation of G#, a statically typed nominal object-oriented language with runtime type-argument inference. A generic call may omit its type arguments and have them chosen from the runtime types of its argument values, so it can use type information that the caller's static types have discarded, without weakening the guarantee that no execution of a compiled program halts because of a type error. The running `ToList` call ends where it began: it is well typed, the runtime reads `Cat` from the source's descriptor, and the copy is a `List[Cat]` behind the caller's `List[out Animal]` view.

The central idea is to settle at the declaration what each call needs. Signature checking decides from a method's declaration alone whether it is eligible. Eligibility

ensures principal typing for the method's RI calls and, beyond it, that every well-typed one, for every value its static argument types allow, every valid closing of the enclosing type parameters and every addition of type declarations, has a unique optimal assignment both statically and at run time, which runtime inference computes (Section 4.10). This follows Muehlboeck's principally inferable generic methods, whose invocations also infer their type arguments from runtime types [Muehlboeck, 2019, Chapter 8]; G# derives the inference direction of each type parameter from its occurrences, asks for one optimum of the whole assignment, including parameters coupled by bounds and declarations added later, admits results that are not covariant by fixing their parameters, and gives, implements and measures the runtime computation. Where multiple dispatch with variance solves a selected declaration's type arguments to some valid substitution [Park et al., 2019], and algebraic subtyping gives a principal type scheme at compile time [Dolan and Mycroft, 2017], G# chooses one concrete assignment per execution and guarantees that it is the best one. Chapter 3 gave the language whose declaration rules make joins and meets exist and keep them valid as declarations are added, and Chapter 4 formalised signature checking, type checking of RI calls and runtime selection, stated the properties the rules guarantee and described how the compiler and runtime follow the rules. In the evaluation, a port of LINQ keeps System.Linq's parameter types, with function objects in place of delegates, and its client calls omit their type arguments except where an operator's signature cannot determine them; an RI call whose answer is cached costs 82 instructions, whole programs execute less than one per cent more instructions, and the main price is larger generated code (Chapter 5).

The insight behind this design is that a safe choice of type arguments is not enough. Where several admissible assignments are incomparable, any deterministic choice makes execution depend on an arbitrary rule, so a language must either guarantee a unique optimum or leave the choice to the programmer. G# makes that decision once, at each signature, and pays for it in language restrictions (Section 7.1) and larger generated code rather than in execution time. We hope that this makes the information carried by reified generics usable by ordinary generic calls, not only by reflection and type tests.

Supporting Rules

These figures complete Chapters 3 and 4. The body gives the rules on which runtime inference builds and the rules of runtime inference itself; this appendix gives the remaining rules, which the body describes in prose, in the order of its sections. Each section names the section that it completes.

A.1 Declarations and parents

Declaration lookup gives the rules access to the program (Section 3.1), and the parents of a type are the steps that principal instantiation walks (Section 3.3.2).

$\boxed{\mathcal{D} \vdash k\ N[\overline{P_i}] : \overline{T_j}\ \{\ \overline{B_l}\ \}}$

$$\frac{\texttt{class}\ C[\overline{P_i}]\ :\ \overline{T_j}\ \{\ \overline{M_l}\ \} \in \mathcal{D}}{\mathcal{D} \vdash \mathsf{mat}\ C[\overline{P_i}] : \overline{T_j}\ \{\ \overline{M_l}\ \}}\ \textsc{Decl-Class}$$

$$\frac{\texttt{interface}\ I[\overline{P_i}]\ :\ \overline{T_j}\ \{\ \overline{S_l}\ \} \in \mathcal{D}}{\mathcal{D} \vdash \mathsf{mat}\ I[\overline{P_i}] : \overline{T_j}\ \{\ \overline{S_l}\ \}}\ \textsc{Decl-Iface}$$

$$\frac{\texttt{shape interface}\ I[\overline{P_i}]\ :\ \overline{T_j}\ \{\ \overline{S_l}\ \} \in \mathcal{D}}{\mathcal{D} \vdash \mathsf{shape}\ I[\overline{P_i}] : \overline{T_j}\ \{\ \overline{S_l}\ \}}\ \textsc{Decl-Shape}$$

$\boxed{\mathcal{D} \vdash N \ni m : \sigma}$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ C\ \{\ \overline{M_l}\ \} \qquad H\ \{\ \overline{s_q}\ \} \in \overline{M_l}}{\mathcal{D} \vdash C \ni \mathsf{name}(H) : \mathsf{sig}(H)}\ \textsc{Decl-Method}$$

$$\frac{\mathcal{D} \vdash k\ I\ \{\ \overline{S_l}\ \} \qquad H\texttt{;} \in \overline{S_l}}{\mathcal{D} \vdash I \ni \mathsf{name}(H) : \mathsf{sig}(H)}\ \textsc{Decl-Sig}$$

Figure A.1: Declaration lookup. A declaration by its kind, and the methods it declares.

$\boxed{\mathcal{D} \mid \Delta \vdash \mathsf{parents}(T) = \mathcal{T}}$

$$\frac{\mathcal{D} \vdash k\ N'[\overline{X_i}] : \overline{T'_j}}{\mathcal{D} \mid \Delta \vdash \mathsf{parents}(N'[\overline{T_i}]) = \{\overline{T'_j[\overline{X_i \mapsto T_i}]}\}}\ \textsc{Par-Inst}$$

$$\frac{}{\mathcal{D} \mid \Delta \vdash \mathsf{parents}(T_1 \,\&\, T_2) = \{T_1, T_2\}}\ \textsc{Par-And} \qquad \frac{X \in \mathsf{dom}(\Delta)}{\mathcal{D} \mid \Delta \vdash \mathsf{parents}(X) = \{\Delta(X)\}}\ \textsc{Par-Param}$$

$$\frac{T \in \{\mathtt{int}, \mathtt{bool}, \mathtt{void}, \top, \bot\}}{\mathcal{D} \mid \Delta \vdash \mathsf{parents}(T) = \emptyset}\ \textsc{Par-Base}$$

Figure A.2: Parents of a type T. N' is the own head of an instantiation.

A.2 Valid types and bounds

This section gives the rules for valid types and type arguments, and for the types that may appear as bounds and supertypes (Section 3.4).

$\boxed{\mathcal{D} \mid \Delta \vdash T}$

$$\frac{X \in \mathsf{dom}(\Delta)}{\mathcal{D} \mid \Delta \vdash X}\ \textsc{Type-Param} \qquad \frac{T \in \{\mathtt{int}, \mathtt{bool}, \mathtt{void}, \top, \bot\}}{\mathcal{D} \mid \Delta \vdash T}\ \textsc{Type-Base}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T_1 \quad \mathcal{D} \mid \Delta \vdash T_2}{\mathcal{D} \mid \Delta \vdash T_1 \,\&\, T_2}\ \textsc{Type-And}$$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ N[\overline{v_i\ X_i : T_i^b}] \qquad \overline{\mathcal{D} \mid \Delta \vdash a_i\ \mathsf{ok}_{v_i}} \qquad (\forall l.\ \mathsf{exact}(a_l)) \implies \overline{\mathcal{D} \mid \Delta \vdash a_i <: T_i^b[\overline{X_l \mapsto a_l}]}}{\mathcal{D} \mid \Delta \vdash N[\overline{a_i}]}\ \textsc{Type-Inst}$$

$\boxed{\mathcal{D} \mid \Delta \vdash a\ \mathsf{ok}_v}$

$$\frac{\mathcal{D} \mid \Delta \vdash T}{\mathcal{D} \mid \Delta \vdash T\ \mathsf{ok}_v}\ \textsc{Arg-Exact} \qquad \frac{\mathcal{D} \mid \Delta \vdash T \quad v \neq \mathsf{in}}{\mathcal{D} \mid \Delta \vdash \mathtt{out}\ T\ \mathsf{ok}_v}\ \textsc{Arg-Out}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T \quad v \neq \mathsf{out}}{\mathcal{D} \mid \Delta \vdash \mathtt{in}\ T\ \mathsf{ok}_v}\ \textsc{Arg-In} \qquad \frac{}{\mathcal{D} \mid \Delta \vdash ?\ \mathsf{ok}_v}\ \textsc{Arg-Any}$$

Figure A.3: Valid types and type arguments.

$\boxed{\mathcal{D} \mid \Delta \vdash T \ \mathsf{bound}}$

$$\frac{\mathcal{D} \mid \Delta \vdash T}{\mathcal{D} \mid \Delta \vdash T \ \mathsf{bound}} \textsc{Bound-Type} \qquad \frac{\mathcal{D} \mid \Delta \vdash T_1 \ \mathsf{bound} \quad \mathcal{D} \mid \Delta \vdash T_2 \ \mathsf{bound}}{\mathcal{D} \mid \Delta \vdash T_1 \,\&\, T_2 \ \mathsf{bound}} \textsc{Bound-And}$$

$$\frac{\mathcal{D} \vdash \mathsf{shape}\ N\,[\overline{v_i\ X_i : T_i^b}] \qquad \overline{\mathcal{D} \mid \Delta \vdash a_i\ \mathsf{ok}_{v_i}} \qquad (\forall l.\ \mathsf{exact}(a_l)) \implies \overline{\mathcal{D} \mid \Delta \vdash a_i <: T_i^b[\overline{X_l \mapsto a_l}]}}{\mathcal{D} \mid \Delta \vdash N[\overline{a_i}] \ \mathsf{bound}} \textsc{Bound-Shape}$$

Figure A.4: Valid bounds and supertypes.

A.3 Expressions and statements

This section gives the rules for field reads and static methods, checking, construction, type tests and their bindings, and statements (Section 3.6), with the types of members read through a projection (Section 3.3.1).

$$\mathsf{at}_v^{\overline{p_i}}(X_i') = \mathsf{end}_v(p_i)$$

$$\mathsf{at}_v^{\overline{p_i}}(N''[\overline{a_l}]) = N''[\overline{\mathsf{at}_{v \circ \mathsf{lv}(a_l, v_l')}^{\overline{p_i}}(a_l)}] \ \text{ where } \mathsf{vars}(N'') = \overline{v_l'}$$

$$\mathsf{at}_v^{\overline{p_i}}(\mathtt{out}\ T) = \mathtt{out}\ \mathsf{at}_v^{\overline{p_i}}(T) \qquad \mathsf{at}_v^{\overline{p_i}}(\mathtt{in}\ T) = \mathtt{in}\ \mathsf{at}_v^{\overline{p_i}}(T)$$

$$\mathsf{at}_v^{\overline{p_i}}(T_1 \,\&\, T_2) = \mathsf{at}_v^{\overline{p_i}}(T_1) \,\&\, \mathsf{at}_v^{\overline{p_i}}(T_2)$$

$$\mathsf{view}(\sigma, \overline{p_i}) = [\overline{X_j : \mathsf{at}_{\mathsf{in}}^{\overline{p_i}}(T_j^b)}](\overline{\mathsf{at}_{\mathsf{in}}^{\overline{p_i}}(T_l)}) \to \mathsf{at}_{\mathsf{out}}^{\overline{p_i}}(T_r).$$

Figure A.5: Definitions of types read through the intervals $\overline{p_i}$ of the slots of N, whose type parameters $\overline{X_i}$ are written $\overline{X_i'}$ here, and a signature $\sigma = [\overline{X_j : T_j^b}](\overline{T_l}) \to T_r$ read through them; every other type and type argument is unchanged.

$\boxed{\mathcal{D} \mid \Delta \vdash T.f : T' \qquad \mathcal{D} \mid \Delta \vdash T.m : \sigma}$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ C[\overline{X_i}]\ \{\,\overline{M_l}\,\} \qquad T_f\ f \in \overline{M_l} \qquad \mathsf{near}(T, f) = \{C\} \qquad \mathcal{D} \mid \Delta \vdash \mathsf{inst}(T, C) = \overline{p_i}}{\mathcal{D} \mid \Delta \vdash T.f : \mathsf{at}_{\mathsf{inv}}^{\overline{p_i}}(T_f)} \textsc{Mem-Field}$$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ C[\overline{X_i}]\ \{\,\overline{M_l}\,\} \qquad \mathtt{static}\ H\ \{\,\overline{s_q}\,\} \in \overline{M_l} \qquad \mathsf{name}(H) = m}{\mathcal{D} \mid \Delta \vdash (\mathsf{static}\ C[\overline{T_i}]).m : \mathsf{sig}(H)[\overline{X_i \mapsto T_i}]} \textsc{Mem-Static}$$

Figure A.6: Fields and static methods. $T_f\ f \in \overline{M_l}$ means that $\overline{M_l}$ declares the field f of type T_f, with or without an initialiser, and $\mathsf{near}(T, f)$ is defined as for methods from the heads that declare f; a field that two heads declare, neither inheriting the other, is not available.

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow T}$ $\qquad$ $\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash r \Rightarrow T}$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T' \quad \mathcal{D} \mid \Delta \vdash T' <: T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow T}\ \text{T-Sub} \qquad \frac{\mathcal{D} \mid \Delta \vdash C[\overline{T_j}]}{\mathcal{D} \mid \Delta \mid \Gamma \vdash C\,[\,\overline{T_j}\,] \Rightarrow \mathsf{static}\, C[\overline{T_j}]}\ \text{Recv-Class}$$

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}$

$$\frac{\Gamma(x) = T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash x \Rightarrow T}\ \text{T-Var} \qquad \frac{\Gamma(\texttt{this}) = T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \texttt{this} \Rightarrow T}\ \text{T-This}$$

$$\frac{}{\mathcal{D} \mid \Delta \mid \Gamma \vdash z \Rightarrow \texttt{int}}\ \text{T-Int} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash b \Rightarrow \texttt{bool}}\ \text{T-Bool}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T \quad \mathcal{D} \mid \Delta \vdash T.f : T'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e\,.\,f \Rightarrow T'}\ \text{T-Field}$$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\, C\,[\,\overline{X_j}\,]\ \{\,\overline{M_q}\,\} \qquad C\,(\,\overline{T_i\ x_i}\,) \ldots \in \overline{M_q} \qquad \mathcal{D} \mid \Delta \vdash C[\overline{T_j^a}] \qquad \overline{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_i \Leftarrow T_i[\overline{X_j \mapsto T_j^a}]}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \mathsf{new}\, C\,[\,\overline{T_j^a}\,]\,(\overline{e_i}) \Rightarrow C[\overline{T_j^a}]}\ \text{T-New}$$

Figure A.7: Checking, receivers, variables and literals, construction and field reads. In T-New, $C\,(\,\overline{T_i\ x_i}\,) \ldots$ is a constructor of C with as many parameters as the call has arguments.

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi'}$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T' \quad \mathcal{D} \mid \Delta \vdash T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e\ \texttt{is}\ T \Rightarrow \texttt{bool} \triangleright \emptyset; \emptyset}\ \text{T-Is} \qquad \frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T' \quad \mathcal{D} \mid \Delta \vdash T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e\ \texttt{is}\ T\ x \Rightarrow \texttt{bool} \triangleright x{:}T; \emptyset}\ \text{T-Bind}$$

$$\frac{e \text{ is not a test} \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow \texttt{bool}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \emptyset; \emptyset}\ \text{T-Cond} \qquad \frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash\ !e \Rightarrow \texttt{bool} \triangleright \Psi'; \Psi}\ \text{T-Not}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1 \Rightarrow \texttt{bool} \triangleright \Psi_1; \Psi_1' \quad \mathcal{D} \mid \Delta \mid \Gamma, \Psi_1 \vdash e_2 \Rightarrow \texttt{bool} \triangleright \Psi_2; \Psi_2'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1\ \texttt{\&\&}\ e_2 \Rightarrow \texttt{bool} \triangleright \Psi_1, \Psi_2;\ \Psi_1' \cap (\Psi_1, \Psi_2')}\ \text{T-And}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1 \Rightarrow \texttt{bool} \triangleright \Psi_1; \Psi_1' \quad \mathcal{D} \mid \Delta \mid \Gamma, \Psi_1' \vdash e_2 \Rightarrow \texttt{bool} \triangleright \Psi_2; \Psi_2'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1\ \texttt{||}\ e_2 \Rightarrow \texttt{bool} \triangleright \Psi_1 \cap (\Psi_1', \Psi_2);\ \Psi_1', \Psi_2'}\ \text{T-Or}$$

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}$

$$\frac{e \text{ is a test} \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool}}\ \text{T-Test}$$

Figure A.8: Type tests and their bindings. A *test* is an expression `e is T`, `e is T x`, `!e`, `e && e` or `e || e`; $\Psi \cap \Psi'$ keeps the bindings that both sets introduce at the same test.

$\boxed{\mathcal{D} \mid \Delta \mid \Gamma \vdash s}$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi' \quad \mathcal{D} \mid \Delta \mid \Gamma, \Psi \vdash s_1 \quad \mathcal{D} \mid \Delta \mid \Gamma, \Psi' \vdash s_2}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2} \textsc{Stmt-If}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow \texttt{bool} \triangleright \Psi; \Psi' \quad \mathcal{D} \mid \Delta \mid \Gamma, \Psi \vdash s}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \texttt{while (}e\texttt{)}\ s} \textsc{Stmt-While}$$

$$\frac{}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \{\}} \textsc{Stmt-Empty}$$

$$\frac{\mathcal{D} \mid \Delta \vdash T \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow T \quad \mathcal{D} \mid \Delta \mid \Gamma, x{:}T \vdash \{\,\overline{s_i}\,\}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \{\,T\ x\ \texttt{=}\ e\texttt{;}\ \overline{s_i}\,\}} \textsc{Stmt-Local}$$

$$\frac{s \text{ is not a declaration} \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash s \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash \{\,\overline{s_i}\,\}}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \{\,s\ \overline{s_i}\,\}} \textsc{Stmt-Seq}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow \Gamma(x)}{\mathcal{D} \mid \Delta \mid \Gamma \vdash x\ \texttt{=}\ e\texttt{;}} \textsc{Stmt-Assign}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1 \Rightarrow T \quad \mathcal{D} \mid \Delta \vdash T.f : T' \quad \mathcal{D} \mid \Delta \mid \Gamma \vdash e_2 \Leftarrow T'}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e_1\texttt{.}f\ \texttt{=}\ e_2\texttt{;}} \textsc{Stmt-Field}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash e\texttt{;}} \textsc{Stmt-Expr}$$

$$\frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Leftarrow \Gamma(\mathsf{ret})}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \texttt{return}\ e\texttt{;}} \textsc{Stmt-Return} \qquad \frac{\mathcal{D} \mid \Delta \mid \Gamma \vdash e \Rightarrow T}{\mathcal{D} \mid \Delta \mid \Gamma \vdash \texttt{throw}\ e\texttt{;}} \textsc{Stmt-Throw}$$

Figure A.9: Statements. `if (e) s` abbreviates `if (e) s else {};` a declaration's scope is the rest of its block, and a declaration that is not in a block is checked as the block that holds only it; ret is the result type (Figure A.12).

A.4 Valid declarations and programs

This section gives the rules for refinement of inherited signatures, consistent inheritance, class bodies, and valid declarations and programs (Section 3.7).

$\boxed{\mathcal{D} \mid \Delta \vdash \sigma \leq_N \sigma'}$

$$\frac{}{\mathcal{D} \mid \Delta \vdash \sigma \leq_C \sigma} \textsc{Refine-Class}$$

$$\frac{\Delta' = \Delta, \overline{X_j : T_j^b} \quad \overline{\mathcal{D} \mid \Delta' \vdash T_i' <: T_i} \quad \mathcal{D} \mid \Delta' \vdash T_r <: T_r'}{\mathcal{D} \mid \Delta \vdash [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r \leq_I [\overline{X_j : T_j^b}](\overline{T_i'}) \to T_r'} \textsc{Refine-Iface}$$

Figure A.10: Refinement of a signature declared in N, with the method type parameters of the two signatures renamed to one list $\overline{X_j}$.

$\boxed{\mathcal{D} \mid \Delta \vdash N \text{ inherits}}$

$$\frac{\begin{array}{c}\mathcal{D} \vdash k\ N[\overline{X_i}] : \overline{T_j} \qquad (\forall j.\ \exists N' \neq \texttt{Array}, \overline{T'_l}.\ T_j = N'[\overline{T'_l}]) \\ |\{j \mid \exists C', \overline{a_l}.\ T_j = C'[\overline{a_l}]\}| \leq 1 \qquad N = I \implies \nexists j, C', \overline{a_l}.\ T_j = C'[\overline{a_l}] \\ \forall N'' \in \mathsf{heads}(N[\overline{X_i}]).\ \exists \overline{p_l}.\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(N[\overline{X_i}], N'') = \overline{p_l} \wedge \forall l.\ \mathcal{D} \mid \Delta \vdash lo_{p_l} <: hi_{p_l} \\ N = C \implies \forall N'', \overline{p_l}, m, \sigma.\ \mathcal{D} \mid \Delta \vdash \mathsf{inst}(C[\overline{X_i}], N'') = \overline{p_l} \wedge \mathcal{D} \vdash N'' \ni m : \sigma \implies \\ \mathcal{D} \mid \Delta \vdash \mathsf{impl}(C, m, \mathsf{arity}(\sigma)) \leq_{N''} \mathsf{view}(\sigma, \overline{p_l})\end{array}}{\mathcal{D} \mid \Delta \vdash N \text{ inherits}}\ \textsc{Valid-Inh}$$

Figure A.11: Consistent inheritance. N' ranges over class and interface names, so $T_j = N'[\overline{T'_l}]$ excludes primitive types, $\top$, $\bot$, type parameters and intersections.

$\boxed{\mathcal{D} \mid \Delta \vdash_C M}$

$$\frac{\mathcal{D} \mid \Delta \mid \texttt{this}{:}C[\overline{X_i}] \vdash e \Leftarrow T}{\mathcal{D} \mid \Delta \vdash_C T\ f\ \texttt{=}\ e\texttt{;}}\ \textsc{Body-Field} \qquad \frac{}{\mathcal{D} \mid \Delta \vdash_C T\ f\texttt{;}}\ \textsc{Body-Decl}$$

$$\frac{\begin{array}{c}\mathcal{D} \mid \Delta, \overline{X_j : T_j^b} \mid \texttt{this}{:}C[\overline{X_i}], \overline{x_n : T_n}, \mathsf{ret}{:}T_r \vdash \texttt{\{}\ \overline{s_q}\ \texttt{\}} \\ T_r \neq \texttt{void} \implies \mathsf{returns}(\overline{s_q})\end{array}}{\mathcal{D} \mid \Delta \vdash_C T_r\ m\texttt{[}\overline{X_j\ \texttt{:}\ T_j^b}\texttt{]}\texttt{(}\overline{T_n\ x_n}\texttt{)}\ \texttt{\{}\ \overline{s_q}\ \texttt{\}}}\ \textsc{Body-Method}$$

$$\frac{\begin{array}{c}\mathcal{D} \mid \Delta, \overline{X_j : T_j^b} \mid \overline{x_n : T_n}, \mathsf{ret}{:}T_r \vdash \texttt{\{}\ \overline{s_q}\ \texttt{\}} \\ T_r \neq \texttt{void} \implies \mathsf{returns}(\overline{s_q})\end{array}}{\mathcal{D} \mid \Delta \vdash_C \texttt{static}\ T_r\ m\texttt{[}\overline{X_j\ \texttt{:}\ T_j^b}\texttt{]}\texttt{(}\overline{T_n\ x_n}\texttt{)}\ \texttt{\{}\ \overline{s_q}\ \texttt{\}}}\ \textsc{Body-Static}$$

$$\frac{\begin{array}{c}\mathsf{cbase}(C) = C'[\overline{T'_k}] \implies \mathcal{D} \mid \Delta \mid \overline{x_n : T_n} \vdash \mathsf{new}\ C'\texttt{[}\overline{T'_k}\texttt{]}\texttt{(}\overline{e_p}\texttt{)} \Rightarrow C'[\overline{T'_k}] \\ \mathsf{cbase}(C)\ \text{undefined} \implies \overline{e_p}\ \text{is empty} \\ \mathcal{D} \mid \Delta \mid \texttt{this}{:}C[\overline{X_i}], \overline{x_n : T_n}, \mathsf{ret}{:}\texttt{void} \vdash \texttt{\{}\ \overline{s_q}\ \texttt{\}}\end{array}}{\mathcal{D} \mid \Delta \vdash_C C\texttt{(}\overline{T_n\ x_n}\texttt{)}\ \texttt{:}\ \texttt{base(}\overline{e_p}\texttt{)}\ \texttt{\{}\ \overline{s_q}\ \texttt{\}}}\ \textsc{Body-Ctor}$$

$\boxed{\mathcal{D} \mid \Delta \vdash C \text{ bodies}}$

$$\frac{\mathcal{D} \vdash \mathsf{mat}\ C\texttt{[}\overline{X_i}\texttt{]}\ \texttt{\{}\ \overline{M_l}\ \texttt{\}} \qquad \overline{\mathcal{D} \mid \Delta \vdash_C M_l} \qquad \mathsf{init}(C)}{\mathcal{D} \mid \Delta \vdash C \text{ bodies}}\ \textsc{Valid-Body}$$

Figure A.12: Class bodies. $\overline{X_i}$ are the type parameters of C, $\overline{X_j}$ those of a method and $\mathsf{cbase}(C)$ its class base, if any; `this` is bound to the receiver and ret to the result type, which `return e;` checks. returns holds of `return` and `throw`, of an `if` whose two branches it holds of, and of a block with a statement it holds of. $\mathsf{init}(C)$ is definite initialisation (Section 3.2), whose flow rules we omit.

$\boxed{\vdash \mathcal{D} \qquad \mathcal{D} \vdash N \text{ valid}}$

$$\frac{\begin{array}{c}\forall k, N.\ \mathcal{D} \vdash k\ N \implies \mathcal{D} \vdash N \text{ valid} \\ \forall D_1, D_2 \in \mathcal{D}.\ \mathsf{name}(D_1) = \mathsf{name}(D_2) \implies D_1 = D_2 \qquad \mathsf{finitary}(\mathsf{usage}(\mathcal{D}))\end{array}}{\vdash \mathcal{D}}\ \textsc{Valid-Prog}$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash k\ N[\overline{v_i\ X_i : T_i^b}] : \overline{T_j}\ \{\ \overline{B_l}\ \} \qquad \Delta = \overline{X_i : T_i^b} \qquad \mathsf{acyclic}(\Delta) \\ \overline{\mathcal{D} \mid \Delta \vdash T_i^b \text{ bound}} \qquad \overline{\mathcal{D} \mid \Delta \vdash T_j \text{ bound}} \qquad \mathcal{D} \mid \Delta \vdash N \text{ inherits} \\ \overline{\mathsf{gi}(T_i^b)} \qquad \overline{\mathsf{gi}(T_j)} \qquad \mathsf{respects}(N) \qquad N = C \implies \mathcal{D} \mid \Delta \vdash C \text{ bodies} \\ \forall H \in \overline{B_l}.\ \mathcal{D} \mid \Delta \vdash \mathsf{sig}(H) \qquad \mathsf{distinct}(\overline{B_l}) \\ \forall T\ f \in \overline{B_l}.\ \mathcal{D} \mid \Delta \vdash T \qquad \forall N(\overline{T_n\ x_n}) \ldots \in \overline{B_l}.\ \overline{\mathcal{D} \mid \Delta \vdash T_n}\end{array}}{\mathcal{D} \vdash N \text{ valid}}\ \textsc{Valid-Decl}$$

$\boxed{\mathcal{D} \mid \Delta \vdash \sigma}$

$$\frac{\begin{array}{c}\Delta' = \Delta, \overline{X_j : T_j^b} \quad \mathsf{acyclic}(\Delta') \quad \overline{\mathcal{D} \mid \Delta' \vdash T_j^b \text{ bound}} \quad \overline{\mathsf{gi}(T_j^b)} \\ \overline{\mathcal{D} \mid \Delta' \vdash T_i} \quad \mathcal{D} \mid \Delta' \vdash T_r\end{array}}{\mathcal{D} \mid \Delta \vdash [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r}\ \textsc{Valid-Sig}$$

Figure A.13: Valid programs, declarations and signatures. H, $\overline{T\ f}$ and $N(\ldots)\ldots$ range over the method headers, fields and constructors in $\overline{B_l}$; gi is defined in Section 3.4; the rules do not check that `main` exists.

A.5 Coupled parameters

This section gives the adjustments that bounds make to the current types of coupled parameters (Section 4.3).

$\boxed{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{adjust}_{X_q:T}(\overline{\hat{T}_j}) = \overline{\hat{T}'_j}}$

$$\frac{\mathsf{dir}(X_l) = \mathsf{out} \quad \mathsf{dir}(X_q) \neq \mathsf{in}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{adjust}_{X_q:X_l}(\overline{\hat{T}_j}) = \overline{\hat{T}_j}[l \mapsto \hat{T}_l \sqcup \hat{T}_q]}\ \textsc{Couple-Join}$$

$$\frac{\mathsf{dir}(X_l) \neq \mathsf{out} \quad \mathsf{dir}(X_q) = \mathsf{in}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{adjust}_{X_q:X_l}(\overline{\hat{T}_j}) = \overline{\hat{T}_j}[q \mapsto \hat{T}_q \sqcap \hat{T}_l]}\ \textsc{Couple-Meet}$$

$$\frac{\mathsf{dir}(X_l) \neq \mathsf{out} \quad \mathsf{dir}(X_q) \neq \mathsf{in}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{adjust}_{X_q:X_l}(\overline{\hat{T}_j}) = \overline{\hat{T}_j}}\ \textsc{Couple-Skip}$$

$$\frac{\mathsf{ftv}(T) \cap \overline{X_j} = \{X_l\} \quad T \neq X_l \quad X_l \neq X_q \quad \mathsf{dir}(X_l) = \mathsf{out} \quad \mathsf{dir}(X_q) \neq \mathsf{in} \quad \mathcal{R} = \{T' \mid (X_l, \rho, v) \in \mathsf{occ}(T),\ \mathcal{D} \mid \Delta \vdash \mathsf{read}_{\mathsf{out}}(\rho, \hat{T}_q) = T' \neq \mathsf{none}\}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{adjust}_{X_q:T}(\overline{\hat{T}_j}) = \overline{\hat{T}_j}[l \mapsto \hat{T}_l \sqcup \bigsqcup \mathcal{R}]}\ \textsc{Couple-Through}$$

Figure A.14: Adjustments of the current types of coupled parameters of the signature σ of m (Section 4.3). $\overline{\hat{T}_j}$ are the current types, and $\overline{\hat{T}_j}[l \mapsto T]$ replaces the current type of X_l by T. In Couple-Join, Couple-Meet and Couple-Skip, X_l is a type parameter of m other than X_q.

A.6 The static assignment

This section gives the best type argument of each parameter and the static assignment of an RI call (Section 4.3).

$\boxed{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X, \overline{T_i^s}) = \tau}$

$$\frac{T \in Q_X \quad \forall T' \in Q_X.\ \mathcal{D} \mid \Delta \vdash T' \equiv T}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X, \overline{T_i^s}) = T}\ \textsc{Best-Equal}$$

$$\frac{Q_X = \emptyset \quad \mathsf{dir}(X) = \mathsf{out}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X, \overline{T_i^s}) = \bigsqcup L_X}\ \textsc{Best-Out} \qquad \frac{Q_X = \emptyset \quad \mathsf{dir}(X) = \mathsf{fixed} \quad T_X^b = \bot}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X, \overline{T_i^s}) = \bot}\ \textsc{Best-Never}$$

$$\frac{Q_X = \emptyset \quad \mathsf{dir}(X) = \mathsf{in}}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X, \overline{T_i^s}) = \bigsqcap U_X \sqcap T_X^g}\ \textsc{Best-In}$$

$\boxed{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{assign}(\overline{T_i^s}) = \overline{\tau_j}}$

$$\frac{\forall i, j, \rho, v.\ (X_j, \rho, v) \in \mathsf{occ}(T_i) \implies \exists t.\ \mathcal{D} \mid \Delta \vdash \mathsf{read}_{\mathsf{out}}(\rho, T_i^s) = t \quad \overline{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{best}(X_j, \overline{T_i^s}) = \tau_j^0} \quad \overline{\tau_j} = \mathsf{couple}_\sigma(\overline{\tau_j^0})}{\mathcal{D} \mid \Delta \vdash_\sigma \mathsf{assign}(\overline{T_i^s}) = \overline{\tau_j}}\ \textsc{Assign-Call}$$

Figure A.15: The static assignment for a call of m, with signature σ, whose arguments have static types $\overline{T_i^s}$. L_X, U_X and Q_X are the lower bounds, upper bounds and equalities read for X (Section 4.3); $\bigsqcup \emptyset = \bot$ and $\bigsqcap \emptyset = \top$; in Assign-Call, t is a type or none; disagreeing equalities, or a parameter whose best is undefined, give no static assignment. couple_σ is defined in Section 4.3.

A.7 Bounds the runtime can satisfy

This section gives the bounds that eligibility accepts (Section 4.5). Sat-Ground uses the conjuncts of a type, $\mathsf{conj}(T_1 \,\&\, T_2) = \mathsf{conj}(T_1) \cup \mathsf{conj}(T_2)$ and $\mathsf{conj}(T) = \{T\}$ for any other T.

$\boxed{\mathcal{D} \vdash_\sigma (X{:}T)\ \mathsf{satisfiable}}$

$$\frac{\mathsf{ftv}(T) \cap \overline{X_j} = \emptyset \qquad \mathsf{dir}(X) = \mathsf{in} \implies \forall N[\overline{a_i}] \in \mathsf{conj}(T).\ \mathcal{D} \vdash \mathsf{mat}\ N}{\mathcal{D} \vdash_\sigma (X{:}T)\ \mathsf{satisfiable}}\ \textsc{Sat-Ground}$$

$$\frac{\begin{array}{c}\mathsf{dir}(X) = \mathsf{out} \qquad T = N[\overline{a_i}] \qquad J = \{i \mid a_i = X\} \neq \emptyset \\ \forall i \notin J.\ X \notin \mathsf{ftv}(a_i) \qquad \mathsf{ftv}(T) \cap \overline{X_j} = \{X\} \\ \mathsf{vars}(N) = \overline{v_i} \qquad (\forall i \in J.\ v_i = \mathsf{in}) \ \vee\ (J = \{i\} \wedge v_i = \mathsf{inv}) \ \vee\ (\forall i \in J.\ v_i = \mathsf{out})\end{array}}{\mathcal{D} \vdash_\sigma (X{:}T)\ \mathsf{satisfiable}}\ \textsc{Sat-Self}$$

$$\frac{\mathsf{dir}(X) = \mathsf{fixed} \quad \mathsf{ftv}(T) \cap \overline{X_j} = \{X\}}{\mathcal{D} \vdash_\sigma (X{:}T)\ \mathsf{satisfiable}}\ \textsc{Sat-Fixed} \qquad \frac{\mathsf{dir}(X_l) \neq \mathsf{out} \vee \mathsf{dir}(X_q) \neq \mathsf{in}}{\mathcal{D} \vdash_\sigma (X_q{:}X_l)\ \mathsf{satisfiable}}\ \textsc{Sat-Param}$$

$$\frac{\begin{array}{c}\mathsf{ftv}(T) \cap \overline{X_j} = \{X_l\} \quad T \neq X_l \quad X_l \neq X_q \\ \forall \rho, v.\ (X_l, \rho, v) \in \mathsf{occ}(T) \implies v = \mathsf{out} \qquad \mathsf{dir}(X_l) = \mathsf{out} \qquad \mathsf{dir}(X_q) \neq \mathsf{in}\end{array}}{\mathcal{D} \vdash_\sigma (X_q{:}T)\ \mathsf{satisfiable}}\ \textsc{Sat-Through}$$

Figure A.16: Bounds the runtime can satisfy, for a signature $\sigma = [\overline{X_j : T_j^b}](\overline{T_i}) \to T_r$. T is the bound of X, or of X_q; in Sat-Param, X_l is a type parameter of m.

Appendix B

Complete G# Examples

These listings give the complete programs underlying the short examples in the body, including supporting classes and traversal code omitted from the figures. The type arguments inferred for them, and their behaviour, are explained in Chapters 1, 3 and 5.

B.1 ToList

```
class Animal {
    private string name;
    public Animal(string n) { name = n; }
    public string Name() { return name; }
}
class Cat : Animal {
    public Cat(string n) : base(n) { }
}
interface IEnumerator[out T] {
    bool MoveNext();
    T Current();
}
interface IEnumerable[out T] {
    IEnumerator[T] GetEnumerator();
}
class ListEnumerator[T] : IEnumerator[T] {
    private T[] items;
    private int count;
    private int index;
    public ListEnumerator(T[] buffer, int length) {
        items = buffer; count = length; index = -1;
    }
    public bool MoveNext() {
        if (index + 1 >= count) { return false; }
        index = index + 1;
        return true;
    }
    public T Current() {
        return items[index];
    }
}
class List[T] : IEnumerable[T] {
    private T[] items;
    private T fill;
```

```
    private int count;
    public List(T f) { fill = f; items = new T[](4, f); count = 0; }
    public void Add(T x) {
        if (count == items.Length()) {
            T[] grown = new T[](items.Length() * 2, fill);
            for (int i = 0; i < count; i = i + 1) { grown[i] = items[i]; }
            items = grown;
        }
        items[count] = x; count = count + 1;
    }
    public IEnumerator[T] GetEnumerator() {
        return new ListEnumerator[T](items, count);
    }
}
class Program {
    static List[out T] ToList[T](IEnumerable[T] source) {
        IEnumerator[T] e = source.GetEnumerator();
        if (!e.MoveNext()) { throw "empty source"; }
        List[T] result = new List[T](e.Current());
        result.Add(e.Current());
        while (e.MoveNext()) { result.Add(e.Current()); }
        return result;
    }
    public static int main() {
        List[Cat] cats = new List[Cat](new Cat("Tom"));
        cats.Add(new Cat("Tom")); cats.Add(new Cat("Felix"));
        IEnumerable[Animal] view = cats;
        List[out Animal] explicitCopy = Program.ToList[Animal](view);
        List[out Animal] runtimeCopy = Program.ToList(view);
        Console.WriteLine(explicitCopy);
        Console.WriteLine(runtimeCopy);
        IEnumerator[Animal] a = explicitCopy.GetEnumerator();
        IEnumerator[Animal] b = runtimeCopy.GetEnumerator();
        while (a.MoveNext()) { Console.WriteLine(a.Current().Name()); }
        while (b.MoveNext()) { Console.WriteLine(b.Current().Name()); }
        return 0;
    }
}
```

B.2 Running example

The program also uses the collection classes of the previous section, `IEnumerator`, `IEnumerable`, `ListEnumerator` and `List`, unchanged; they are not repeated here.

```
class Animal { }
interface IPet { }
interface INamed { }
class Cat : Animal { }
class Dog : Animal { }
class Kitten : Cat, IPet, INamed { }
class Puppy : Dog, IPet { }
interface IHandler[in E] { void Handle(E e); }
class Shelter : IHandler[Cat], IHandler[Dog] {
    public void Handle(Animal e) { Console.WriteLine("handled"); }
```

```
}
class Program {
    static List[out T] ToList[T](IEnumerable[T] source) {
        IEnumerator[T] e = source.GetEnumerator();
        if (!e.MoveNext()) { throw "empty source"; }
        List[T] result = new List[T](e.Current());
        result.Add(e.Current());
        while (e.MoveNext()) { result.Add(e.Current()); }
        return result;
    }
    public static int main() {
        List[Cat] cats = new List[Cat](new Cat());
        cats.Add(new Cat());
        IEnumerable[Animal] view = cats;
        List[out Animal] copy = Program.ToList(view);
        bool exact = copy is List[Cat];
        Console.WriteLine(exact);
        IHandler[Animal] h = new Shelter();
        h.Handle(new Dog());
        IPet p = new Kitten();
        Console.WriteLine(p is INamed);
        return 0;
    }
}
```

B.3 Typed event delivery

```
class Event { public Event() {} }
class EditorEvent : Event { public EditorEvent() {} }
class FileOpened : EditorEvent { public FileOpened() {} }
class FileSaved : EditorEvent {
    private string contents = "";
    public FileSaved(string contents) { this.contents = contents; }
    public string Contents() { return contents; }
}
interface IHandler[in E] { void Handle(E e); }
class ActivityLog : IHandler[Event] {
    private int count;
    public ActivityLog() { count = 0; }
    public void Handle(Event e) { count = count + 1; }
    public int Count() { return count; }
}
class AutoBackup : IHandler[FileSaved] {
    private int count;
    private string snapshot = "";
    public AutoBackup() { count = 0; snapshot = ""; }
    public void Handle(FileSaved e) {
        count = count + 1;
        snapshot = e.Contents();
    }
    public int Count() { return count; }
    public string Snapshot() { return snapshot; }
```

```
}
class Bus {
    private object[] handlers;
    public Bus(object[] handlers) { this.handlers = handlers; }
    public void Publish[E](E e) {
        for (int j = 0; j < handlers.Length(); j = j + 1) {
            if (handlers[j] is IHandler[E] h) { h.Handle(e); }
        }
    }
}
class Program {
    static void Run(bool runtime) {
        ActivityLog log = new ActivityLog();
        AutoBackup backup = new AutoBackup();
        object[] handlers = new object[](2, log);
        handlers[0] = log;
        handlers[1] = backup;
        Bus bus = new Bus(handlers);
        Event[] events = new Event[](2, new FileOpened());
        events[0] = new FileOpened();
        events[1] = new FileSaved("edited file contents");
        string mode = "static";
        if (runtime) { mode = "runtime"; }
        for (int i = 0; i < events.Length(); i = i + 1) {
            if (runtime) { bus.Publish(events[i]); }
            else { bus.Publish[Event](events[i]); }
            Console.WriteLine(mode + " " + i + ": "
                + log.Count() + ", " + backup.Count());
        }
        Console.WriteLine("snapshot: " + backup.Snapshot());
    }
    public static int main() {
        Program.Run(false);
        Program.Run(true);
        return 0;
    }
}
```

Appendix C

Typed Event-Delivery Alternatives

These excerpts show how C#, Java and Kotlin reproduce the delivery behaviour compared in Section 5.1.3. Each program declares the events and handlers of Figure 5.2, registers `ActivityLog` at `Event` and `AutoBackup` at `FileSaved`, and publishes a `FileOpened` and a `FileSaved` through an `Event`-typed array. The listings show only what differs from the G# program of Appendix B: an adapter that retains the event type and a bus that stores adapters behind a common delivery interface, declared as in the C# listing.

The C# program, compiled with the .NET 10 SDK with nullable checking and warnings as errors, prints the same counts and snapshot as the G# program's RI calls: the log receives both events and the backup only `FileSaved`. The Java and Kotlin programs are written to the same output but were not compiled.

C.1 C#: an adapter retaining T

The adapter's generic instantiation retains `T`, and `Deliver` tests the `Event` value against `T` before invoking the typed handler [Microsoft, 2026b].

```
interface IDelivery {
    void Deliver(Event e);
}
class TypedDelivery<T> : IDelivery where T : Event {
    private readonly IHandler<T> handler;
    public TypedDelivery(IHandler<T> handler) { this.handler = handler; }
    public void Deliver(Event e) {
        if (e is T value) { handler.Handle(value); }
    }
}
class Bus {
    private readonly List<IDelivery> deliveries = new List<IDelivery>();
    public void Register<T>(IHandler<T> handler) where T : Event {
        deliveries.Add(new TypedDelivery<T>(handler));
    }
    public void Publish(Event e) {
        foreach (IDelivery delivery in deliveries) { delivery.Deliver(e); }
    }
}
```

C.2 Java: an adapter with a class token

The adapter retains a `Class<T>` token alongside an `IHandler<? super T>`. The token's `isInstance` test guards the cast, which suffices for these nongeneric event classes but would not recover the erased arguments of a generic event type; the wildcard permits a handler accepting `T` or a supertype [Oracle, 2025d,c].

```
    static class TypedDelivery<T extends Event> implements Delivery {
        private final Class<T> eventType;
        private final IHandler<? super T> handler;
        TypedDelivery(Class<T> eventType, IHandler<? super T> handler) {
            this.eventType = eventType;
            this.handler = handler;
        }
        public void deliver(Event e) {
            if (eventType.isInstance(e)) { handler.handle(eventType.cast(e)); }
        }
    }
    static class Bus {
        private final List<Delivery> deliveries = new ArrayList<>();
        <T extends Event> void register(Class<T> eventType, IHandler<? super T>
  handler) {
            deliveries.add(new TypedDelivery<>(eventType, handler));
        }
        void publish(Event e) {
            for (Delivery delivery : deliveries) { delivery.deliver(e); }
        }
    }
```

C.3 Kotlin: reified evidence at registration

The inline registration function obtains the class token of its reified `T` and stores it in an adapter, which later tests and casts the event; the bus, omitted, stores adapters through an `add` method and delivers as in the Java listing. `publish` itself still receives only an `Event` [JetBrains, 2025; Oracle, 2025d].

```
class TypedDelivery<T : Event>(
    private val eventType: Class<T>,
    private val handler: IHandler<T>
) : Delivery {
    override fun deliver(e: Event) {
        if (eventType.isInstance(e)) { handler.handle(eventType.cast(e)) }
    }
}
inline fun <reified T : Event> register(bus: Bus, handler: IHandler<T>) {
    bus.add(TypedDelivery(T::class.java, handler))
}
```

Appendix D

Evaluation Programs

These listings show the parts of the evaluation programs of Chapter 5 that the chapter discusses, in the order of Section 5.2: the LINQ library and representative calls of the two LINQ suites, of the MonNom ports and of the six smaller programs. Members the listings do not use are omitted. The explicit (E) and inferred (R) variants of a program differ only in how they write generic calls: R omits the type arguments and, in some programs, the class name of a static call. Table D.1 gives the medians plotted in Figure 5.4.

D.1 The LINQ suites

Figure D.1 lists the interfaces and operators of the library, whose lazy sequence classes are omitted, Figures D.2 and D.3 the storage and the list class, and Figure D.4 the call sites of the two suites in both variants. The source writes a projection that repeats a declared variance, as in `IEnumerable[out T]`; the listings omit it, since `IEnumerable[out T]` and `IEnumerable[T]` denote the same type (Section 3.3.1).

```
interface IEnumerator[out T] { bool MoveNext(); T Current(); }
interface IEnumerable[out T] { IEnumerator[T] GetEnumerator(); }
interface ICollection[out T] : IEnumerable[T] { int Count(); }
interface IList[out T] : ICollection[T] { T At(int i); }
interface IPred[in T] { bool Test(T x); }
interface IFunc[in A, out B] { B Apply(A a); }

class Enumerable {
  public static IEnumerable[T] Where[T](
      IEnumerable[T] source, IPred[T] pred) {
    return new WhereSeq[T](source, pred);
  }
  public static IEnumerable[B] Select[A, B](
      IEnumerable[A] source, IFunc[A, B] f) {
    return new SelectSeq[A, B](source, f);
  }
  public static IEnumerable[T] Cast[T](IEnumerable[object] source) {
    return new CastSeq[T](source);
  }
  public static int Consume[T](IEnumerable[T] source) {
    int n = 0;
    IEnumerator[T] e = source.GetEnumerator();
```

```
    while (e.MoveNext()) { T x = e.Current(); n = n + 1; }
    return n;
  }
  public static Vec[out T] ToList[T](IEnumerable[T] source) {
    Vec[T] result = new Vec[T](Enumerable.Capacity[T](source));
    IEnumerator[T] e = source.GetEnumerator();
    while (e.MoveNext()) { result.Add(e.Current()); }
    return result;
  }
  private static int Capacity[T](IEnumerable[T] source) {
    if (source is ICollection[T] c) { return c.Count(); }
    return 8;
  }
}
```

Figure D.1: Interfaces and operators of the LINQ library used in Section 5.2. `Where` and `Select` keep their System.Linq signatures, with function objects in place of delegates, and have the element type at out and at in; `Consume` counts the elements of a sequence; `ToList` copies one into the `Vec` of Figure D.3.

```
interface ISlots[T] {
  int Length();  T Get(int i);  ISlots[T] Put(int i, T item);
}
class EmptySlots[T] : ISlots[T] {
  private int length = 0;
  public EmptySlots(int n) { length = n; }
  public int Length() { return length; }
  public T Get(int i) { throw "unfilled slot"; }
  public ISlots[T] Put(int i, T item) {
    T[] items = new T[](length, item);
    items[i] = item;
    return new FilledSlots[T](items);
  }
}
class FilledSlots[T] : ISlots[T] {
  private T[] items;
  public FilledSlots(T[] xs) { items = xs; }
  public int Length() { return items.Length(); }
  public T Get(int i) { return items[i]; }
  public ISlots[T] Put(int i, T item) { items[i] = item; return this; }
}
```

Figure D.2: Backing storage for a fixed number of slots. An array needs a fill value of its element type, which a collection has only once its first element arrives, so the array is allocated by the first `Put`, with that element as the fill; reading a slot before then is an error.

```
class Vec[T] : IList[T] {
```

```
  private ISlots[T] data;  private int count = 0;
  public Vec(int capacity) { data = new EmptySlots[T](capacity); }
  public void Add(T item) {
    if (count == data.Length()) {
      ISlots[T] bigger = new EmptySlots[T](count * 2 + 1);
      for (int i = 0; i < count; i = i + 1) {
        bigger = bigger.Put(i, data.Get(i));
      }
      data = bigger;
    }
    data = data.Put(count, item);
    count = count + 1;
  }
  public int Count() { return count; }
  public T At(int i) { return data.Get(i); }
  // GetEnumerator walks the slots by index
}
```

Figure D.3: The list class of the library, over the storage of Figure D.2.

```
class SoldOut : IPred[Product] {
  public bool Test(Product p) { return p.UnitsInStock() == 0; }
}

// LinqBenchmarks: Where00LinqMethodX(Vec[Product] products, ...)
SoldOut pred = new SoldOut();
IEnumerable[Product] soldOutProducts =
    Enumerable.Where[Product](products, pred);    // explicit variant
IEnumerable[Product] soldOutProducts =
    Enumerable.Where(products, pred);             // inferred variant

// Perf_Enumerable: CastToSameType(IEnumerable[Num] c)
return Enumerable.Consume[Num](Enumerable.Cast[Num](c));  // explicit
return Enumerable.Consume(Enumerable.Cast[Num](c));       // inferred
```

Figure D.4: Two call sites of the LINQ suites in both variants. `Product` is the benchmark's product record and `Num` an `int` held by reference. The inferred `Where` omits its type argument; `Cast` gives its type argument explicitly in both variants because no argument mentions it.

Table D.1: Execution cost of the explicit (E) and inferred (R) variant of each program. Instructions are counted with the collector disabled; wall time with it enabled. E and R are the medians of eleven repetitions of the explicit and inferred variants. Δ is the median of the per-repetition change $100(R_i/E_i - 1)$ with its 95% bootstrap confidence interval; it is not the ratio of the two medians, and the two can differ, even in sign.

Program	Metric	E	R	Δ (%)
LINQ operators	Instr. (10^9)	22.97	22.99	+0.10 [0.04, 0.10]
	Wall (ms)	7,506	7,511	+0.25 [−0.23, 0.70]
LINQ enumerable	Instr. (10^9)	25.95	26.00	+0.19 [0.19, 0.20]
	Wall (ms)	8,127	8,182	+0.49 [0.31, 0.75]
Sieve	Instr. (10^9)	11.47	11.47	0.00 [−0.02, 0.01]
	Wall (ms)	969	971	+0.49 [−0.66, 1.72]
Float	Instr. (10^9)	9.44	9.43	0.00 [−0.06, 0.05]
	Wall (ms)	704	703	+0.79 [−1.05, 1.13]
LINQ pipeline	Instr. (10^9)	15.93	15.93	+0.02 [0.00, 0.04]
	Wall (ms)	669	671	+0.68 [−0.61, 4.35]
Collection copy	Instr. (10^9)	13.82	13.82	−0.03 [−0.05, 0.00]
	Wall (ms)	623	616	+1.20 [−3.99, 2.08]
Maps and sets	Instr. (10^9)	31.60	31.60	0.00 [0.00, 0.00]
	Wall (ms)	4,830	4,740	−1.77 [−2.08, −1.11]
Event delivery	Instr. (10^9)	12.10	12.10	−0.01 [−0.06, 0.02]
	Wall (ms)	881	891	+0.54 [−1.04, 2.03]
Visitor	Instr. (10^9)	12.47	12.47	0.00 [−0.02, 0.01]
	Wall (ms)	654	660	+0.46 [−1.42, 1.11]
Numeric kernel	Instr. (10^9)	17.10	17.10	−0.01 [−0.02, 0.03]
	Wall (ms)	973	981	+0.51 [−1.24, 2.79]
Control: declaration order	Instr. (10^9)	22.95	22.99	+0.14 [0.13, 0.15]
	Wall (ms)	7,475	7,470	+0.27 [−0.78, 1.08]
Control: source padding	Instr. (10^9)	22.95	22.95	0.00 [−0.02, 0.00]
	Wall (ms)	7,484	7,496	+0.06 [−1.04, 1.54]

D.2 The MonNom ports

Figure D.5 shows representative calls of the MonNom ports of the sieve and float programs; the intersort port has no call computed at run time, since its element type is invariant and every type argument is fixed by an invariant occurrence, so each call behaves as an explicit one.

```
// sieve:  IStream[T] Make[T](T hd, IFun[IStream[T]] thunk),
//         CountFromFun : IFun[IStream[int]]
return Streams.Make[int](n, new CountFromFun(n));   // explicit
return Streams.Make(n, new CountFromFun(n));        // inferred

// float:  ArrayList[T] ToList[T](IEnumerable[T] e),
//         IEnumerable[S] Map[T, S](IEnumerable[T] e, Fun[T, S] f)
ArrayList[Point] points = Enumerable.ToList[Point](
```

```
    Enumerable.Map[int, Point](new Range(0, n), new PointMapFun()));
ArrayList[Point] points = Enumerable.ToList(
    Enumerable.Map(new Range(0, n), new PointMapFun()));
```

Figure D.5: Representative calls of the MonNom ports of the sieve and float programs, under the signatures of the called methods; the first float call is the explicit variant. The other calls of the two programs differ in the same way.

D.3 The smaller programs

Figure D.6 shows a representative call of each of the six smaller programs.

```
// LINQ pipeline:  ISeq[T] Concat[T](ISeq[T] a, ISeq[T] b),
//   ISeq[T] SkipTake[T](ISeq[T] source, int skip, int take),
//   Vec[out T] SelectToList[T](ISeq[T] source)
chain = Program.Concat[Item](chain, source);          // explicit
ISeq[Item] window = Program.SkipTake[Item](chain, n / 2, n);
last = Program.SelectToList[Item](window);
chain = Concat(chain, source);                        // inferred
ISeq[Item] window = SkipTake(chain, n / 2, n);
last = SelectToList(window);

// collection copying:  Vec[out T] ToList[T](ISeq[T] src)
lastList = Program.ToList[Item](src);        // explicit
lastList = ToList(src);                      // inferred

// maps and sets:  IKeys[out T] BuildMap[T](ISeq[T] source,
//                                          object rawOrder)
map = Program.BuildMap[Item](source, order);  // explicit
map = BuildMap(source, order);                // inferred

// event delivery:  Envelope[out E] Publish[E](E e), with e : Event
last = bus.Publish[Event](e);                // explicit
last = bus.Publish(e);                       // inferred

// visitor:  VisitSummary[out T] Walk[T](IExpr[T] root,
//                                       object rawVisitor)
last = Program.Walk[Item](root, visitor);    // explicit
last = Walk(root, visitor);                  // inferred

// numeric kernel:  KernelResult[out T] Kernel[T](ISeq[T] source,
//                                                object rawProjector)
last = Program.Kernel[Sample](source, projector);   // explicit
last = Kernel(source, projector);                   // inferred
```

Figure D.6: A representative call of each smaller program, under the signature of the called method. The other calls differ in the same way: `ToArray` in the collection-copying program, and `BuildSet` and `Sort` in the maps-and-sets program.

Bibliography

Allen, E.; Bannet, J.; and Cartwright, R., 2003. A first-class approach to genericity. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 96–114. doi:10.1145/949305.949316. https://doi.org/10.1145/949305.949316. (cited on page 74)

Allen, E.; Hilburn, J.; Kilpatrick, S.; Luchangco, V.; Ryu, S.; Chase, D.; and Steele, G. L., 2011. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 973–992. doi:10.1145/2048066.2048140. https://doi.org/10.1145/2048066.2048140. (cited on pages 70 and 71)

Altidor, J.; Huang, S. S.; and Smaragdakis, Y., 2011. Taming the wildcards: Combining definition- and use-site variance. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 602–613. doi:10.1145/1993498.1993569. https://doi.org/10.1145/1993498.1993569. (cited on pages 9, 10, 11, and 75)

Amin, N. and Tate, R., 2016. Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 838–848. doi:10.1145/2983990.2984004. https://doi.org/10.1145/2983990.2984004. (cited on page 76)

Belyakova, J.; Chung, B.; Tate, R.; and Vitek, J., 2024. Decidable subtyping of existential types for Julia. *Proceedings of the ACM on Programming Languages*, 8, PLDI (2024), 1091–1114. doi:10.1145/3656421. https://doi.org/10.1145/3656421. (cited on page 72)

Bierhoff, K., 2022. Wildcards need witness protection. *Proceedings of the ACM on Programming Languages*, 6, OOPSLA2 (2022), 373–394. doi:10.1145/3563301. https://doi.org/10.1145/3563301. (cited on page 76)

Bracha, G., 2010. Reifying generic types while maintaining migration compatibility. US Patent 7,810,077, filed 2005, granted 2010. (cited on page 72)

Bracha, G.; Odersky, M.; Stoutamire, D.; and Wadler, P., 1998. Making the future safe for the past: Adding genericity to the Java programming language.

In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 183–200. doi:10.1145/286936.286957. https://doi.org/10.1145/286936.286957. (cited on page 74)

Cameron, N. R.; Drossopoulou, S.; and Ernst, E., 2008. A model for Java with wildcards. In *ECOOP 2008 – Object-Oriented Programming*, vol. 5142 of *Lecture Notes in Computer Science*, 2–26. Springer. doi:10.1007/978-3-540-70592-5_2. https://doi.org/10.1007/978-3-540-70592-5_2. (cited on pages 3 and 71)

Canning, P.; Cook, W.; Hill, W.; Olthoff, W.; and Mitchell, J. C., 1989. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, 273–280. doi:10.1145/99370.99392. https://doi.org/10.1145/99370.99392. (cited on page 13)

Cardelli, L. and Wegner, P., 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17, 4 (1985), 471–523. https://lucacardelli.name/Papers/OnUnderstanding.A4.pdf. (cited on pages 1, 6, and 34)

Cartwright, R. and Steele, G. L., 1998. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 201–215. doi:10.1145/286936.286958. https://doi.org/10.1145/286936.286958. (cited on page 74)

Castagna, G., 2021. Programming with union, intersection, and negation types. arXiv:2111.03354. Published in *The French School of Programming*, Springer, 2023. (cited on page 74)

Castagna, G.; Ghelli, G.; and Longo, G., 1995. A calculus for overloaded functions with subtyping. *Information and Computation*, 117, 1 (Feb. 1995), 115–135. doi:10.1006/inco.1995.1033. https://doi.org/10.1006/inco.1995.1033. (cited on page 70)

Castagna, G. and Xu, Z., 2011. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 94–106. doi:10.1145/2034773.2034788.

Cimadamore, M. and Viroli, M., 2008. On the reification of Java wildcards. *Science of Computer Programming*, 73, 2–3 (2008), 59–75. doi:10.1016/j.scico.2008.06.005. https://doi.org/10.1016/j.scico.2008.06.005.

Craciun, F.; Chin, W.-N.; He, G.; and Qin, S., 2009. An interval-based inference of variant parametric types. In *Programming Languages and Systems (ESOP 2009)*, vol. 5502 of *Lecture Notes in Computer Science*, 112–127. doi:10.1007/978-3-642-00590-9_9. https://doi.org/10.1007/978-3-642-00590-9_9. (cited on pages 10 and 75)

Crary, K.; Weirich, S.; and Morrisett, G., 1998. Intensional polymorphism in type-erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 301–312. doi:10.1145/289423.289459. https://doi.org/10.1145/289423.289459. (cited on page 74)

Damas, L. and Milner, R., 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 207–212. doi:10.1145/582153.582176. (cited on page 15)

Dolan, S., 2016. *Algebraic Subtyping*. Ph.D. thesis, University of Cambridge. (cited on pages 73 and 74)

Dolan, S. and Mycroft, A., 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 60–72. doi:10.1145/3009837.3009882. https://doi.org/10.1145/3009837.3009882. (cited on pages 3, 73, 74, and 79)

Ducournau, R., 2008. Perfect hashing as an almost perfect subtype test. *ACM Transactions on Programming Languages and Systems*, 30, 6 (2008), 1–56. doi:10.1145/1391956.1391960. https://doi.org/10.1145/1391956.1391960.

Dunfield, J. and Krishnaswami, N., 2021. Bidirectional typing. *ACM Computing Surveys*, 54, 5 (2021), 1–38. doi:10.1145/3450952. https://doi.org/10.1145/3450952. (cited on pages 7 and 15)

ECMA, 2023. ECMA-334: C# language specification. https://ecma-international.org/wp-content/uploads/ECMA-334_7th_edition_december_2023.pdf. 7th edition, December 2023. (cited on pages 15, 21, 32, 72, and 74)

Ellis, S.; Zhu, S.; Yoshida, N.; and Song, L., 2022. Generic go to go: Dictionary-passing, monomorphisation, and hybrid. *Proceedings of the ACM on Programming Languages*, 6, OOPSLA2 (2022), 1207–1235. doi:10.1145/3563331. https://doi.org/10.1145/3563331. (cited on page 74)

Emir, B.; Kennedy, A.; Russo, C.; and Yu, D., 2006. Variance and generalized constraints for C# generics. In *ECOOP 2006 – Object-Oriented Programming*, vol. 4067 of *Lecture Notes in Computer Science*, 279–303. doi:10.1007/11785477_18. https://doi.org/10.1007/11785477_18. (cited on pages 10, 35, and 75)

Fähndrich, M. and Leino, K. R. M., 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 302–312. doi:10.1145/949305.949332. https://www.microsoft.com/en-us/research/wp-content/uploads/2003/11/non-null.pdf. (cited on page 76)

Frisch, A.; Castagna, G.; and Benzaken, V., 2002. Semantic subtyping. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 137–146. https://www.irif.fr/~gc/papers/lics02.pdf. (cited on page 7)

Ghelli, G. and Pierce, B. C., 1998. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193, 1–2 (1998), 75–96. doi:10.1016/S0304-3975(96)00300-3. (cited on page 15)

Goetz, B., 2020. Background: How we got the generics we have (or, how i learned to stop worrying and love erasure). https://openjdk.org/projects/valhalla/design-notes/in-defense-of-erasure. OpenJDK design note, June 2020. Accessed 29 September 2026. (cited on page 11)

Greenman, B.; Muehlboeck, F.; and Tate, R., 2014. Getting F-bounded polymorphism into shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 89–99. doi:10.1145/2594291.2594308. https://doi.org/10.1145/2594291.2594308. (cited on pages 9, 13, 14, 29, 34, 58, and 75)

Gregor, D., 2022. SE-0352: Implicitly opened existentials. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0352-implicit-open-existentials.md. Swift Evolution proposal, implemented in Swift 5.7. (cited on page 71)

Griesemer, R.; Hu, R.; Kokke, W.; Lange, J.; Taylor, I. L.; Toninho, B.; Wadler, P.; and Yoshida, N., 2020. Featherweight Go. *Proceedings of the ACM on Programming Languages*, 4, OOPSLA (Nov. 2020), 1–29. doi:10.1145/3428217. https://doi.org/10.1145/3428217. (cited on page 74)

Grigore, R., 2017. Java generics are Turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 73–85. doi:10.1145/3009837.3009871. https://doi.org/10.1145/3009837.3009871. (cited on page 76)

Harper, R. and Morrisett, G., 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 130–141. doi:10.1145/199448.199475. https://doi.org/10.1145/199448.199475. (cited on page 74)

Hindley, J. R., 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146 (1969), 29–60. doi:10.2307/1995158. (cited on page 15)

Igarashi, A.; Pierce, B. C.; and Wadler, P., 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23, 3 (2001), 396–450. doi:10.1145/503502.503505. https://doi.org/10.1145/503502.503505. (cited on pages 5, 26, and 58)

Igarashi, A. and Viroli, M., 2006. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28, 5 (2006), 795–847. doi:10.1145/1152649.1152650. https://doi.org/10.1145/1152649.1152650. (cited on pages 9, 10, and 75)

JetBrains, 2025. Inline functions: Reified type parameters. https://kotlinlang.org/docs/inline-functions.html. Reified type parameters. Accessed 29 September 2026. (cited on pages 63, 74, and 96)

JetBrains, 2026. Generics: in, out, where. https://kotlinlang.org/docs/generics.html. Accessed 29 September 2026.

Jim, T., 1996. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 42–53. doi:10.1145/237721.237728. (cited on pages 16 and 73)

Kennedy, A. and Syme, D., 2001. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 1–12. doi:10.1145/378795.378797. https://doi.org/10.1145/378795.378797. (cited on pages 1, 12, 59, and 74)

Kennedy, A. J. and Pierce, B. C., 2007. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*. https://www.cis.upenn.edu/~bcpierce/papers/variance.pdf. (cited on pages 6, 34, 58, and 76)

King, G., 2017. The ceylon language. https://web.mit.edu/ceylon_v1.3.3/ceylon-1.3.3/doc/en/spec/html_single. Language specification, version 1.3.3. (cited on pages 12, 25, 26, and 75)

Microsoft, 2021. Variance in generic interfaces. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/variance-in-generic-interfaces. Accessed 29 September 2026. (cited on page 63)

Microsoft, 2023. Using type dynamic. https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/interop/using-type-dynamic. C# programming guide: affected operations are dynamically bound and may fail at run time. Accessed 10 October 2026.

Microsoft, 2024. Generic methods (C#). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-methods. Accessed 29 September 2026. (cited on pages 1 and 2)

Microsoft, 2026a. Enumerable.tolist method. https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolist?view=net-10.0. Official API documentation. Accessed 9 October 2026. (cited on page 2)

Microsoft, 2026b. Type-testing operators and cast expressions. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast. The is operator and declaration patterns. Accessed 4 October 2026. (cited on pages 63 and 95)

Microsoft, n.d. IEnumerable<T> and IEnumerator<T> interfaces. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1 and https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerator-1. Official .NET API documentation. Accessed 29 September 2026. (cited on page 9)

Misonizhnik, A. and Mordvinov, D., 2019. On satisfiability of nominal subtyping with variance. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, vol. 134 of *LIPIcs*, 7:1–7:20. doi:10.4230/LIPIcs.ECOOP.2019.7. https://doi.org/10.4230/LIPIcs.ECOOP.2019.7. (cited on page 76)

Muehlboeck, F., 2019. *Efficient Runtimes for Gradual Typing*. Ph.D. thesis, Cornell University. doi:10.7298/k9ra-2795. (cited on pages 3, 15, 16, 69, 70, 73, and 79)

Muehlboeck, F. and Tate, R., 2017. Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages*, 1, OOPSLA (2017), 1–30. doi:10.1145/3133880. https://doi.org/10.1145/3133880. (cited on page 74)

Muehlboeck, F. and Tate, R., 2018. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2, OOPSLA (2018), 1–29. doi:10.1145/3276482. https://doi.org/10.1145/3276482. (cited on pages 7, 9, 12, 24, 27, and 75)

Muehlboeck, F. and Tate, R., 2021. Transitioning from structural to nominal code with efficient gradual typing. *Proceedings of the ACM on Programming Languages*, 5, OOPSLA (2021), 1–29. doi:10.1145/3485504. https://doi.org/10.1145/3485504. (cited on pages 64 and 74)

Naden, K. B.; Chase, D. R.; and Hilburn, J. R., 2014. Type inference of generic type parameters in overloaded functions using type intervals and inference directions. US Patent 8,893,082 B2, assigned to Oracle; filed 31 August 2012, granted 18 November 2014. (cited on page 72)

.NET Foundation, 2026a. Enumerable microbenchmarks. https://github.com/dotnet/performance/blob/70a3326eb92ab74cf86c1bf613d986b2a1668f0a/src/benchmarks/micro/libraries/System.Linq/Perf.Enumerable.cs. Source version used in this study; accessed 10 October 2026. (cited on page 64)

.NET Foundation, 2026b. LINQ operator microbenchmarks. https://github.com/dotnet/performance/blob/70a3326eb92ab74cf86c1bf613d986b2a1668f0a/src/benchmarks/micro/runtime/Linq/Linq.cs. Source version used in this study; accessed 10 October 2026. (cited on page 64)

Odersky, M.; Zenger, C.; and Zenger, M., 2001. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 41–53. doi:10.1145/360204.360207. https://doi.org/10.1145/360204.360207. (cited on page 15)

Oliveira, B. C. d. S.; Shi, Z.; and Alpuim, J., 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 364–377. doi:10.1145/2951913.2951945.

Oracle, 2025a. The Java language specification, Java SE 25: Least upper bound. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html. Section 4.10.4. Accessed 29 September 2026. (cited on pages 13 and 72)

Oracle, 2025b. The Java language specification, Java SE 25: Reifiable types. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html. Section 4.7. Accessed 11 October 2026. (cited on pages 11 and 71)

Oracle, 2025c. The Java language specification, Java SE 25: Type arguments of parameterized types. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html. Section 4.5.1. Accessed 29 September 2026. (cited on pages 63 and 96)

Oracle, 2025d. Java SE 25 api: Class. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Class.html. IsInstance and cast. Accessed 29 September 2026. (cited on pages 63, 74, and 96)

Park, G.; Hong, J.; Steele, G. L., Jr.; and Ryu, S., 2019. Polymorphic symmetric multiple dispatch with variance. *Proceedings of the ACM on Programming Languages*, 3, POPL (2019), 1–28. doi:10.1145/3290324. https://doi.org/10.1145/3290324. (cited on pages 3, 70, 71, and 79)

Parreaux, L., 2020. The simple essence of algebraic subtyping: Principal type inference with subtyping made easy (functional pearl). *Proceedings of the ACM on Programming Languages*, 4, ICFP (2020), 1–28. doi:10.1145/3409006. https://doi.org/10.1145/3409006. (cited on page 73)

Parreaux, L. and Chau, C. Y., 2022. MLstruct: Principal type inference in a boolean algebra of structural types. *Proceedings of the ACM on Programming Languages*, 6, OOPSLA2 (Oct. 2022), 449–478. doi:10.1145/3563304. https://doi.org/10.1145/3563304. (cited on page 73)

Pierce, B. C. and Turner, D. N., 2000. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22, 1 (2000), 1–44. doi:10.1145/345099.345100. https://doi.org/10.1145/345099.345100. (cited on pages 1, 3, 15, 44, and 72)

Rose, J. et al., 2021. The saga of the parametric VM. https://openjdk.org/projects/valhalla/design-notes/parametric-vm/parametric-vm. OpenJDK design note, 2020–2021.

Roth, O., 2021. Study of the subtyping machine of nominal subtyping with variance. *Proceedings of the ACM on Programming Languages*, 5, OOPSLA (2021), 1–27. doi:10.1145/3485514. https://doi.org/10.1145/3485514.

Scala Center and EPFL, n.d.a. Scala 3 reference: Intersection types. https://docs.scala-lang.org/scala3/reference/new-types/intersection-types.html. Official documentation. (cited on page 75)

Scala Center and EPFL, n.d.b. TypeTags and manifests. https://docs.scala-lang.org/overviews/reflection/typetags-manifests.html. Official Scala documentation.

Siek, J. G.; Vitousek, M. M.; Cimini, M.; and Boyland, J. T., 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32 of *Leibniz International Proceedings in Informatics*, 274–293. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SNAPL.2015.274. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2015.274.

Smith, D. and Cartwright, R., 2008. Java type inference is broken: Can we fix it? In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, 505–524. doi:10.1145/1449764.1449804. https://doi.org/10.1145/1449764.1449804. (cited on page 72)

Stadelmeier, A.; Plümicke, M.; and Thiemann, P., 2022. Global type inference for featherweight generic Java. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, vol. 222 of *LIPIcs*, 28:1–28:27. doi:10.4230/LIPIcs.ECOOP.2022.28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.28. (cited on page 73)

Stadelmeier, A.; Plümicke, M.; and Thiemann, P., 2026. A variation on Java wildcards: Trading expressiveness for global type inference. In *40th European Conference on Object-Oriented Programming (ECOOP 2026)*, vol. 372 of *LIPIcs*, 26:1–26:27. doi:10.4230/LIPIcs.ECOOP.2026.26. https://doi.org/10.4230/LIPIcs.ECOOP.2026.26. (cited on page 73)

Tate, R., 2013. Mixed-site variance. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*. https://fool2013.cs.brown.edu/tate.pdf. (cited on page 10)

Tate, R.; Leung, A.; and Lerner, S., 2011. Taming wildcards in Java's type system. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 614–627. doi:10.1145/1993498.1993570. https://doi.org/10.1145/1993498.1993570. (cited on pages 9 and 72)

Tobin-Hochstadt, S. and Felleisen, M., 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 117–128. doi:10.1145/1863543.1863561. https://doi.org/10.1145/1863543.1863561. (cited on page 32)

Torgersen, M.; Ernst, E.; Hansen, C. P.; von der Ahé, P.; Bracha, G.; and Gafter, N., 2004. Adding wildcards to the Java programming language. *The Journal of Object Technology*, 3, 11 (2004), 97–116. doi:10.5381/jot.2004.3.11.a5. https://doi.org/10.5381/jot.2004.3.11.a5. (cited on pages 10 and 75)

Yu, D.; Kennedy, A.; and Syme, D., 2004. Formalization of generics for the .NET common language runtime. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 39–51. doi:10.1145/964001.964005. https://doi.org/10.1145/964001.964005. (cited on pages 59 and 74)

Zappa Nardelli, F.; Belyakova, J.; Pelenitsyn, A.; Chung, B.; Bezanson, J.; and Vitek, J., 2018. Julia subtyping: A rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2, OOPSLA (2018), 1–27. doi:10.1145/3276483. https://doi.org/10.1145/3276483. (cited on page 72)