NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO AFONSO OLIVEIRA PEREIRA
Bachelor in Computer Science

FEATHERWEIGHT GENERIC GO WITH UNTYPED CONSTANTS, STRUCTURAL TYPE DEFINITIONS AND TYPE INFERENCE

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FEATHERWEIGHT GENERIC GO WITH UNTYPED CONSTANTS, STRUCTURAL TYPE DEFINITIONS AND TYPE INFERENCE

JOÃO AFONSO OLIVEIRA PEREIRA

Bachelor in Computer Science

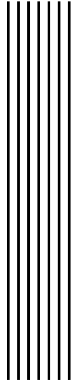
Adviser: Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, NOVA University of Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

Featherweight Generic Go with Untyped Constants, Structural Type Definitions and Type Inference

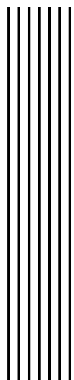
Copyright © João Afonso Oliveira Pereira, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.



Acknowledgements

A very big thank you to my advisor Prof. Bernardo Toninho for the excellent guidance, the impressive availability and the freedom he allowed me in exploring topics not directly related to this work.

Thanks to my family for all the support and love. Thank you Soraya for everything, I do believe this would not be possible without you. Finally, thanks to my fishing comrades for showing me how much fun it can be to look at a rod for hours on end.



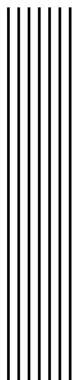
Abstract

Generics are a key ingredient of modular program design in modern programming languages. This feature, which is absent from the Go programming language, has consistently been the most requested among Go developers. The Go Team has been examining how to add generics to the language for years, and recently approved a language change proposal that includes support for generics. The proposal suggests a design based on type parameters, backed by the foundational work on Featherweight (Generic) Go. The proposed design includes some features which were not investigated in a formal setting, opening the door to unforeseen interactions between features that might only be acknowledged too late into the design process.

In particular, the features to be considered are primitive types and operations, type lists in interfaces and inference of type parameters. Primitive types present some challenges related to the heterogeneity they exhibit and the distinction Go makes between values of primitive type and *untyped constants*; type lists in interfaces arise as a solution to allow primitive operations between values of (generic) variable type, but introduce some intricacies since they represent a form of intersection type; type parameter inference doesn't add any functionality to the formal design, but its interference with the type system must still be accounted for.

We have designed and developed an implementation of the above features in the theoretically driven implementation of Featherweight (Generic) Go, which includes a type checker, an interpreter and a monomorphiser. We show how the system can be formally extended to account for this extended feature set. The implementation includes run time validation of type preservation, which supports the correctness of our formal design.

Keywords: Golang, Programming language theory, Type systems, Featherweight languages, Type lists, Bidirectional typing



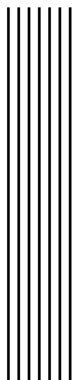
Resumo

Os genéricos são um ingrediente chave para um desenho modular de programas em linguagens de programação modernas. Esta *feature*, ausente da linguagem de programação Go, tem sido consistentemente a mais pedida entre os programadores de Go. A Go Team tem vindo a examinar como adicionar genéricos à linguagem há anos, e recentemente aprovou uma *language change proposal* que inclui suporte para genéricos. A proposta sugere um desenho baseado em parâmetros de tipo, apoiada pelo trabalho fundamental no Featherweight (Generic) Go. O desenho proposto inclui algumas features que não foram investigadas num contexto formal, abrindo espaço para interações imprevistas entre *features*, que podem só vir a ser descobertas demasiado tarde no processo de desenho.

Em particular, as features consideradas são os tipos e operadores primitivos, as listas de tipos em interfaces e a inferência de parâmetros de tipo. Os tipos primitivos levantam desafios relacionados com a heterogeneidade que exibem, além da distinção que o Go faz entre valores de tipo primitivo e *untyped constants*; as listas de tipos em interfaces surgem como uma solução para permitir operações primitivas entre valores de tipos genéricos, mas introduzem algumas dificuldades já que representam uma forma de tipo interseção; a inferência de parâmetros de tipo não adiciona nenhuma funcionalidade ao desenho formal, no entanto deve ter-se em conta a sua interferência com o sistema de tipos.

Desenhou-se e desenvolveu-se uma implementação destas features no protótipo guiado pela teoria do Featherweight (Generic) Go, que inclui um type checker, um interpretador e um monomorfizador. Neste documento mostra-se como se pode estender formalmente o sistema de forma a considerar este conjunto alargado de features. A implementação inclui uma verificação em run time da preservação de tipos, o que suporta a correção deste desenho formal.

Palavras-chave: Golang, Teoria de linguagens de programação, Sistemas de tipos, Linguagens peso-pluma, Listas de tipos, Tipificação bidirecional



Contents

List of Figures ix

List of Listings xi

1 Introduction 1

1.1 Contributions 2

1.2 Document Structure 2

2 Background 3

2.1 Theory of Programming Languages 3

2.1.1 Syntax 4

2.1.2 Semantics 5

2.1.3 Type Systems 6

2.1.4 Type Safety - Formally 8

2.1.5 Modelling Real-World Programming Languages 8

2.2 Polymorphism 9

2.2.1 Ad-hoc polymorphism 10

2.2.2 Subtype polymorphism 11

2.2.3 Parametric polymorphism 13

2.3 Bidirectional Typing 17

2.3.1 Algorithmic typing rules 19

2.3.2 Application to polymorphic type systems 20

3 Related Work 21

3.1 Featherweight Java 21

3.1.1 Featherweight Generic Java 22

3.2 Featherweight Go 22

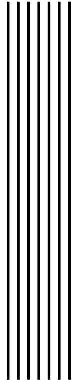
3.2.1 Go vs Java 23
3.2.2 The Featherweight Go Language 23
3.2.3 Featherweight Generic Go 25

4 Extending Featherweight (Generic) Go 28
4.1 Additions to Featherweight Go 28
4.1.1 Primitive types and operations 28
4.1.2 General type definitions 29
4.1.3 Extending Featherweight Go 31
4.2 Additions to Featherweight Generic Go 45
4.2.1 Type lists in interfaces 45
4.2.2 Type definitions revisited 46
4.2.3 Extending Featherweight Generic Go 46
4.3 Adjusting the monomorphisation algorithm 53
4.3.1 Monomorphisation in Featherweight (Generic) Go 53
4.3.2 Monomorphisation and Type Declarations 57
4.3.3 Handling instantiated types in type declarations 59
4.3.4 Monomorphising interface types 61
4.3.5 Note on the remaining features 63

5 Exploring Type Argument Inference 65
5.1 Problem Statement 65
5.2 The base algorithm 67
5.3 Solving a set of subtype constraints 69
5.3.1 Nontrivial constraints 69
5.3.2 Multiple constraints over a single variable 72
5.4 Discussion and Related Work 75
5.4.1 Inferring types for untyped constants 76
5.4.2 Typing the empty list 78
5.4.3 Related Work 79

6 Conclusions and Future Work 81

Bibliography 83



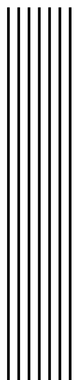
List of Figures

2.1 Syntax of simply typed λ-calculus enriched with booleans and conditionals (λ_B). 5
2.2 Evaluation rules for λ_B. 6
2.3 Typing rules for λ_B. 7
2.4 Syntax of tuples. 12
2.5 Subtyping rules for λ_B enriched with tuples. 12
2.6 Syntax of polymorphic extension of λ_B. 14
2.7 Evaluation rules for type application. 14
2.8 Typing rules for type abstraction and application. 15

3.1 Featherweight Go syntax 24
3.2 FGG syntax 26

4.1 Featherweight Go syntax 31
4.2 FGe auxiliary functions 32
4.3 FGe auxiliary functions for primitive types 33
4.4 FGe predicates 35
4.5 FG typing 37
4.6 The *mostRestrictive* function. 39
4.7 FG expressions typing 40
4.8 FGe reduction 42
4.9 Featherweight Generic Go (extended) syntax 47
4.10 FGGe auxiliary functions 48
4.11 FGGe auxiliary functions for primitives 49
4.12 FGGe (sub)typing 51
4.13 Monomorphisation of type literals (original). 57
4.14 Computing instance sets. 59

4.15 Monomorphisation of types. 61



List of Listings

2.1 check function 18
2.2 synth function 18
3.1 Example program in FGG 27
4.1 Mixing constants and typed variables 29
4.2 FGe assignability between named types and type literals. 36
4.3 Go implicit conversions, example 1 38
4.4 Go implicit conversions, example 2 38
4.5 FGe example: why conversions are needed. 43
4.6 FGe example: why conversions are needed (2). 43
4.7 Example FGG program to be monomorphised. 54
4.8 Selector example: monomorphised (FG) program. 56
4.9 Selector that includes a predicate as its field. 58
5.1 Example program in $FGG_{()}$. 66
5.2 Lists without type arguments 70
5.3 Randomly choosing between two input values. 73
5.4 Randomly choosing between two input Lists. 74
5.5 Lists without type arguments 77
5.6 Simulating rank-2 polymorphism in $FGG_{()}$. 80

1 Introduction

Generics or *parametric polymorphism* are an essential feature present in most current day programming languages. They provide a powerful abstraction mechanism that allows programmers to express algorithms and data structures agnostic to the type of values they manipulate, promoting modularity and encouraging code reutilization.

However, the Go language infamously lacks generics, despite being the feature that developers request the most [28, 29]. The Go Team has been studying this extension for years now, and recently approved a language change proposal that includes generics [1]. The design [2] is based on type parameters and is supported by well-known theory [20]. However, it also includes some real-world oriented extensions whose foundations have not yet been thoroughly studied. It is known that such discrepancies between theory and practice may be problematic, as one might overlook unexpected interactions between seemingly unrelated features - a popular example being the relationship between subtyping and arrays in Java [1, 35], which had to be compensated by a run-time type check, compromising both static type safety and the performance of array stores.

In particular, the proposed design includes primitive types and a form of type lists, a solution that enables bounding type parameters by predetermined sets of primitive types, as a way to support generic functions that use primitive operators within their body. Additionally, the design further includes type parameter inference, a feature that was also not investigated in the context of Featherweight Generic Go [20].

The primitive types present some challenges related both to the heterogeneity they exhibit - e.g., it is not possible to add an int64 to an int - and to the distinction Go makes between values of primitive type and *untyped constants*. Type lists introduce a completely different notion of interface, where one explicitly states the types that can implement

[1] https://github.com/golang/go/issues/43651

[2] https://go.googlesource.com/proposal/+/master/design/go2draft-type-parameters.md

that interface instead of just specifying a set of methods, which may be implemented by arbitrarily many types. Moreover, type lists act both as a form of union of types and an intersection of operators, since the operators supported by the types in the list must match. This feature can be further complicated by the ability to refer to *type aliases* of primitive types in such lists, which may then have distinct methods.

Therefore it is important to understand how these extensions interact with the features already considered in the core underlying theories of Featherweight Go and Featherweight Generic Go. We find, for instance, that expanding the set of possible values requires the addition of an unplanned feature - type conversions - in order to maintain the type safety properties of the model languages.

1.1 Contributions

This work focuses on the problem of integrating type lists in FGG and the extensions necessary to perform interesting experiments with it. Besides, we also study how a type inference algorithm can be implemented in this context. The main contributions of this work encompass the extension of the theoretically driven implementation of Featherweight (Generic) Go with the following features:

- Primitive types, operations and *untyped constants*
- General type declarations, anonymous type literals and type conversions
- Type lists in interfaces
- Monomorphisation of type declarations and type literals
- Type parameter inference

1.2 Document Structure

The remaining of this document is organized as follows. Chapter 2 introduces the background theoretical concepts, reviewing how programming languages and in particular type systems are modelled formally. Chapter 3 discusses the two works which we will be basing on, focusing on the model of Featherweight Generic Go and some of its current limitations. Chapter 4 presents the main contributions of this thesis. It starts by describing and motivating each extension, and then explains how we realize them. Chapter 5 explores the problem of type argument inference in the context of (extended) Featherweight Generic Go and discusses the algorithm we propose to solve this problem. At last, Chapter 6 concludes and suggests some directions for future work.

2 Background

2.1 Theory of Programming Languages

One of the key aspects of a programming language is the set of correctness properties the language ensures about its programs. By imposing these properties, the language is not only reducing the ability of the programmer to introduce errors, but also providing some guarantees about the run-time behavior of the programs it accepts.
For instance, if a language establishes that a function is always applied to the correct number of arguments, not only is the programmer forbidden from calling one with an incorrect number of arguments in their code, but the programmer is also sure that if they pass a function to foreign, unknown code (e.g. a library), the function will be applied correctly.

Consequently, a crucial step in validating a language design is assessing whether the expected properties indeed hold for every program expressible in the language. Moreover, it's an important step not only during the initial design phase, but also whenever a new extension is to be added, ensuring the properties are preserved. As noted by Cardelli [6], this assessment requires a great deal of rigor in order to avoid false conclusions. The remaining of this section introduces some of the tools that enable such rigorous reasoning.

In general, the program correctness of the form mentioned above is enforced by the language's type system, which imposes constraints over the ways the program objects may be manipulated [7]. These constraints might range from simple ones, e.g. not allowing an integer to be used as a pointer, to much stronger ones such as prohibiting the mutation of shared state [25].

The central role the type system plays in ensuring correctness suggests that a good part of the language validation should target its type system. Indeed this is reflected by the fact that one of the most fundamental results we can prove about a language is *type*

safety. Informally, a language L is type safe if, given any well-typed program written in L, it is guaranteed that a certain class of errors is ruled out in any execution of that program. The class of errors depends on each particular language and type system, but it always includes e.g. the use of a function with incorrect arguments and the attempted application of a non-function [47].

Reasoning about such properties while avoiding false conclusions essentially asks for a formal, mathematical proof that the properties really hold. In order to develop such a proof, first it's necessary to state the properties in a concrete and precise way. For example, before proving the informal notion of type safety above for a particular language, one would have to rigorously define the concept of *well-typedness* and a specific class of errors.

Being able to express formal statements about a language implies the language itself has to be modeled under a mathematical framework. The universally adopted model divides a language definition into three dimensions/components: syntax, semantics and type system. In the following subsections we will be showing how to formalize each of these components, referring to the canonical simply typed lambda calculus augmented with booleans and conditionals [35] - henceforth designated as λ_B - in order to ground the abstract notions in concrete examples.
We will then conclude with a formal type safety statement for the example language, followed by a discussion on how to approach the formalization of real-world programming languages.

2.1.1 Syntax

The syntax describes the forms of types and terms; types express static knowledge about programs, whereas terms (statements, expressions, etc.) express the algorithmic behavior [6]. The syntax is generally provided as a BNF-type grammar. It consists of a set of rules that define how symbols are combined to obtain correctly structured terms, and how to combine the terms themselves into more complex ones.

As an example, Figure 2.1 illustrates the syntax definition for λ_B. The rules on the left define the basic form of terms and how to combine them: $\lambda x\!:\!Bool.\ x$ is a well-formed abstraction (i.e. function definition) term, and so is $\lambda x\!:\!Bool.\ \lambda y\!:\!Bool.\ y$ - in λ_B, all the functions are curried. Likewise, $(\lambda x\!:\!Bool.\ x)\ true$ is a well-formed[1] application, as is $(\lambda x\!:\!Bool.\ \lambda y\!:\!Bool.\ y)\ true$. The remaining forms of terms are relatively simple. Of particular importance is the rule for values, which defines a subset of terms that are possible final results of evaluation [35]. Its relevance will be made clear in Section 2.1.4.

Similarly, the production rule for types (on the right) defines that a type is either the base type $Bool$, which has no internal structure, or a combination of two types T_1 and T_2, yielding a function type $T_1 \to T_2$. The contexts and their meaning will be presented in Section 2.1.3.

[1] Although the parenthesis are not part of the syntax, we use them for the sake of readability.

t ::=	**terms:**	T ::=	**types:**
x	variable	$T \to T$	type of functions
$\lambda x{:}T.\ t$	abstraction	$Bool$	type of booleans
$t\ t$	application		
$true$	constant true	Γ ::=	**contexts:**
$false$	constant false	$\emptyset$	empty context
if t then t else t	conditional	$\Gamma, x : T$	term variable binding
v ::=	**values:**		
$\lambda x{:}T.\ t$	abstraction value		
$true$	true value		
$false$	false value		

Figure 2.1: Syntax of simply typed λ-calculus enriched with booleans and conditionals (λ_B).

2.1.2 Semantics

The formalization of semantics specifies how to assign *meaning* to a term. Intuitively, this corresponds to defining how programs are evaluated.

While there are several techniques to formalize the semantics of a programming language, in this presentation we adopt *structural operational semantics* [38, 35]. In this approach, the behavior of a program is described through an hypothetical computer, whose machine code is the language's terms. A state of the machine is essentially a term of the language (it may also include e.g. a memory store). The operation of the machine is then defined by a transition function that, for each state, either:

- yields the next state (i.e. term) by performing a step of simplification or rewriting on the current one
- declares the machine has halted.

The transition function is specified through a set of inductive rules that define the valid transitions of a term based on the possible transitions of its components (i.e. subterms). An example of such a set of rules is shown in Figure 2.2. These model the semantics of λ_B under a call-by-value evaluation strategy, in which only the outermost application is evaluated and it is only evaluated after its argument has been reduced to a value.

The relation $\longrightarrow$ defined by these rules is called the *evaluation relation*, where $t \longrightarrow t'$ reads "term t evaluates to t' in one step". The rules E-Congr1, E-Congr2 and E-If (top half) are *congruence* rules, in the sense that they direct the evaluation to the correct subterm: E-Congr2 can only be applied when v_1 is a *value*, which means that when evaluating an application $t_1\ t_2$, the subterm t_1 must first be reduced to a value before proceeding. The rule E-Congr1 reinforces this order by stating that if t_1 can

E-Congr1
$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

E-Congr2
$$\frac{t_2 \longrightarrow t_2' \quad v_1 \text{ is a value}}{v_1\ t_2 \longrightarrow v_1\ t_2'}$$

E-If
$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

E-AppAbs
$$\frac{v_2 \text{ is a value}}{(\lambda x\!:\!T.\ t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}$$

E-IfTrue
$$\frac{v \text{ is true}}{\text{if } v \text{ then } t_2 \text{ else } t_3 \longrightarrow t_2}$$

E-IfFalse
$$\frac{v \text{ is false}}{\text{if } v \text{ then } t_2 \text{ else } t_3 \longrightarrow t_3}$$

Figure 2.2: Evaluation rules for λ_B.

be simplified, then that simplification is the step to take. Rule E-If is defined similarly: before selecting a branch, the condition t_1 has to be simplified to a value.

The rules E-AppAbs, E-IfTrue and E-IfFalse are often called β-rules or the *computation* rules. Although E-App also imposes an evaluation order - the application can only be reduced after simplifying the argument to a value - its main purpose is to specify the result of applying a function to an argument. The notation $[x \mapsto v_2]t_{12}$ stands for: replace the parameter x by the argument v_2 in the body of the function, t_{12}. Likewise, the rules E-IfTrue and E-IfFalse stipulate the result of evaluating a conditional - i.e. which branch to pick - based on the value of its condition.

Finally, given the definitions of both congruence and computation rules, the evaluation proceeds as follows. Given an initial term t, look up the rule that is applicable to t (if any) and apply it, producing t'. Then, repeat this process until reaching a term t'' for which no rule is applicable. At this point, we say t'' is the *meaning* of the initial term t. There's one important distinction to be made here: t'' will either be a *value* or simply a term to which no rule applies, like the following:

$$\text{if } \lambda x\!:\!Bool.\ x \text{ then } true \text{ else } false$$

Such a term is called a *stuck state*: neither has the evaluation reached a legal final result - i.e. an abstraction or a boolean value - nor can it proceed: the abstraction can't be simplified further and it isn't a boolean value, hence neither of the rules for the if-construction applies. As discussed next, the avoidance of such states is the main purpose of type systems.

2.1.3 Type Systems

The type system is responsible for assigning *types* to terms. A type represents a set of terms that share some common property. For example, the type *int* represents all the terms that evaluate to an integer.

A type can also be thought of as an upper bound over the range of values that a variable might assume during the program execution [35]. Therefore, a type can be considered

a static approximation to the run-time behavior of a term/program. This observation allows the use of types for reasoning about the behavior of programs, in a tractable way, without the need to run them. Having this capability enables the type system to detect a variety of execution errors statically, at compile time.

A type system is formalized by means of a relation *has-type*, written $\Gamma \vdash t : T$ to mean "term t has type T under context Γ". The context Γ is essentially a set of type assignments $x_i : T_i$ for the free variables x_i in t. Resuming the λ_B example, its typing relation is defined by the rules displayed in Fig. 2.3.

$$\textsc{T-Var}\quad \frac{x\!:\!T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{T-Abs}\quad \frac{\Gamma, x\!:\!T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x\!:\!T_1.\ t_2 : T_1 \to T_2} \qquad \textsc{T-App}\quad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

$$\textsc{T-True}\quad \frac{}{\Gamma \vdash true : Bool} \qquad \textsc{T-False}\quad \frac{}{\Gamma \vdash false : Bool} \qquad \textsc{T-If}\quad \frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

Figure 2.3: Typing rules for λ_B.

These rules express the knowledge we have *a priori* about the behavior of the programs. For instance, given a program in λ_B whose outermost construct is an if, i.e., a program of the form:

$$\text{if } complex\text{-}term \text{ then } t_2 \text{ else } t_3$$

we know for sure that if *complex-term* doesn't evaluate to either $true$ or $false$, then the program will get stuck. Although we deduce this from the evaluation rules (or from a lack of rule for if's with non-boolean conditions), there is nothing in them that prevents such a program from being written and evaluated. The error would only manifest itself during the program execution. But that is exactly what rule T-If imposes: in order to be able to assign a type to a term of the form: if t_1 then t_2 else t_3, its conditional t_1 must belong to type $Bool$, which represents all the terms that evaluate to either $true$ or $false$ (moreover, it also enforces that the type of that term will be the type of either t_2 or t_3, as long as they have the same type).

When we can assign a type to a term we say the term is *well-typed*. For a term t to be well-typed means its subterms are of the appropriate types, i.e. they meet the pre-conditions necessary not to lead the evaluation of t into a stuck state. In other words, a term is well-typed only if its subterms are well-typed. Given this recursive definition, the proof of well-typedness for a term naturally assumes the form of a derivation tree, where the premises used to conclude a statement may themselves be conclusions of their own subtree. As an example, follows the demonstration that the term $(\lambda x\!:\!Bool.\ x)\ true$ has

type *Bool*:

$$
\text{(T-APP)}\ \dfrac{\text{(T-ABS)}\ \dfrac{\text{(T-VAR)}\ \dfrac{x\!:\!Bool \in \Gamma, x\!:\!Bool}{\Gamma, x\!:\!Bool \vdash x : Bool}}{\vdash \lambda x\!:\!Bool.\ x : Bool \to Bool} \qquad \text{(T-TRUE)}\ \dfrac{}{\vdash true : Bool}}{\vdash (\lambda x\!:\!Bool.\ x)\ true : Bool}
$$

Finally, in an actual language implementation, the type rules are enforced by a type checking algorithm. Its job is, given a program source, to assign a type to each term - i.e. to prove that every term in it is well-typed. If that isn't provable, then the program *may* lead to an error/stuck state, and thus it is rejected before even being evaluated. This is the role of the type system in avoiding the execution of (possibly) erroneous programs.

2.1.4 Type Safety - Formally

Having formalized the three components, the final step in validating the language is proving that it is *type safe*. A language with this property is one whose semantics and type system are in harmony: it states that the type of a term and the type of its results (after evaluation) should be the same, or related by some sound type relation (e.g. subtyping)[21].

We will be referring to λ_B again in order to target a specific class of errors, namely stuck states as defined at the end of Section 2.1.2. Section 2.1.3 further defined what it means for a term to be well-typed. Hence we are now equipped to formally state type safety for λ_B:

Theorem 1 (Type safety)**.** *Given a term t of λ_B, if t is well-typed, then the evaluation of t never reaches a stuck state. More precisely:*

1. *(Progress) If t is well-typed, then t is either a value or there exists t' such that $t \longrightarrow t'$.*
2. *(Preservation) If $t : T$ for some T and there exists t' such that $t \longrightarrow t'$, then $t' : T$.*

As a side note, the type safety theorem enunciated here is the *syntactic* variant proposed by Wright and Felleisen [47] and later simplified in textbook presentations [21, 35]. Recent research [25] suggests instead a semantic approach to type safety/soundness, arguing that a syntactic technique lacks the power to identify as safe code that uses potentially unsafe features, even if the unsafe fragment is well encapsulated inside an abstraction[12]. However the development of techniques and logics suited to model the semantics of different type systems is still an active topic of research. For this reason, we will only consider the syntactic variant in the remaining of this presentation.

2.1.5 Modelling Real-World Programming Languages

The previous sections demonstrated how a programming language is formally modelled and how such a model enables rigorous reasoning over the language's properties. The

model developed throughout those sections for λ_B is *complete*, as it addresses every single feature of the language. Complete models are in a sense ideal: a **correct** proof of type safety for that model conveys absolute certainty that no program accepted by the typechecker will get stuck, no matter how convoluted the program is.

It is possible to devise a complete model for λ_B - and a relatively simple proof of its properties - only because the language comprises a minimal set of constructs. However, this is seldom the case for mainstream programming languages. The amount of features they embody render the proofs for a complete model hard to effectively manage - in general, each feature adds a new proof case. For instance, a type soundness proof for a large subset of Java [13] had subtle errors only discovered later [44]. Some works [30, 44] resorted to machine checking to ensure the correctness of their proofs, but even that approach poses difficulties inherent to adapting the reasoning to the proof assistant's language.

For that reason, when formalizing a "real" programming language one must tackle the tension between completeness and compactness: although a complete model becomes unworkable, the less features the model accounts for the more likely the proofs overlook nefarious interactions between features that weren't considered together.
Often [22, 17, 48, 20] the decision is to favor compactness over completeness, so as to get "maximum insight for minimum investment" [22]. Chapter 3 discusses examples of such models in detail.

2.2 Polymorphism

A polymorphic type system is one where a single piece of code can be used with many types [35]. The motivation for such type systems arises from the observation that many programs are in fact agnostic to the type of values they manipulate. For instance, consider the following identity functions in λ_B[2] (where $\hookrightarrow$ indicates the type assigned to the functions):

$$\begin{aligned}
&\texttt{idBool} = \lambda x\!:\!Bool.\ x \\
&\quad \hookrightarrow Bool \to Bool \\
&\texttt{idFun} = \lambda x\!:\!Bool \to Bool.\ x \\
&\quad \hookrightarrow (Bool \to Bool) \to (Bool \to Bool) \\
&\texttt{id2Ord} = \lambda x\!:\!(Bool \to Bool) \to (Bool \to Bool).\ x \\
&\quad \hookrightarrow ((Bool \to Bool) \to (Bool \to Bool)) \ \to \ ((Bool \to Bool) \to (Bool \to Bool))
\end{aligned}$$

[2] λ_B as presented in the previous section doesn't account for named functions. In this section we assume the existence of a naming mechanism that allows us to reference a previously defined function, in order to simplify the presentation. Such mechanism could be a simple let-binder of the form $let\ x = t_1\ in\ t_2$ as presented in section 11.5 of [35].

The three functions above are effectively the same, despite being assigned different types. Albeit the example is simple and not of much interest in practice, it can be easily mapped to the real case of array sorting functions, where one would have to write a separate function for each element type, i.e., one for arrays of integers, another for arrays of strings, etc.

The key insight is that we are trying to capture a common behavior that is independent of particular types. Yet, as the type system isn't expressive enough - it imposes that the argument to an abstraction must have a single type -, it forces us to write one version for each individual instance. In particular, what we would like to express with the identity functions is a single function that given a term - whatever its type - returns that exact same term. Similarly, in the sorting example what we intend is a sole function that, given an array of elements of some type T and a total ordering over T, returns an array sorted according to that ordering, regardless of the actual type of the elements.

There are diverse type system extensions that enable the expression of code applicable to multiple types. In essence, these variants can be broadly classified into one of two general kinds [43, 7]: *universal polymorphism* and *ad-hoc (non-universal) polymorphism*. Universal refers to code that works uniformly for arbitrary types, and can be further refined into *parametric* and *subtype* polymorphism. Conversely, ad-hoc concerns functions that are applicable to a finite, known-in-advance range of types, and that may actually exhibit different behaviors depending on the particular type that instantiates them. The next subsections clarify these distinctions.

2.2.1 Ad-hoc polymorphism

As stated previously, ad-hoc refers to a form of polymorphism in which a portion of code is usable by a pre-determined and finite set of types. Moreover, that code might represent different behaviors depending on the particular type that instantiates it.

The most common form of ad-hoc polymorphism is *function overloading* [35], in which a function symbol/name gets associated to different implementations. Each time the function is applied, the correct implementation is then chosen according to the types of the operands. As an example, many languages overload the operator +, which may represent either integer or real addition, and in some cases (e.g. Java) even string concatenation. We call ambiguous functions of this sort polymorphic as they have several forms depending on their arguments [43]. They are ad-hoc in the sense that an actual implementation must be provided for each argument type we wish to support.

Considering the motivating example presented at the beginning of this section, overloading would reduce the burden on the programmer, but would not alleviate it completely. With overloading we would still need to write the three versions:

$\texttt{id} = \lambda x\!:\!Bool.\ x$
$\texttt{id} = \lambda x\!:\!Bool \to Bool.\ x$
$\texttt{id} = \lambda x\!:\!(Bool \to Bool) \to (Bool \to Bool).\ x$

The advantage is that we can now assign them the same name, and upon invocation we don't have to specify which one we intend, as the compiler/interpreter would take care of choosing the correct one based on the type of the argument passed to it.

Another flavor of ad-hoc polymorphism is *coercion*[7]. Coercion represents implicit type conversions of the arguments to a function application, in order to assign them the type expected by the function and hence avoid type errors. Instances of coercion can be often found in mainstream languages where, for example, one is allowed to provide integer values to functions expecting floats. This variant mainly represents a convenience for the programmer, as the language implementation still has to perform (and verify the correctness of) the type conversions.

It's not clear how coercion would make it easier to define or call the identity functions. One possible idea would be to only write one version for some arbitrary type T, and for each call coerce the argument to T. But there is no way to guarantee that such type conversion would be correct in this setting. Nonetheless, a variant of this idea - where the type T represents more than an arbitrary choice - actually forms the basis for subtype polymorphism, as we discuss next.

2.2.2 Subtype polymorphism

A type S is a *subtype* of another type T, written $S <: T$, if the terms of S can safely be used in any context expecting a term of type T [35]. The concept of subtype polymorphism then arises from the fact that a function expecting a parameter of type T can safely be applied to a term of any type S, as long as $S <: T$.

At the type system level, subtyping is realized by introducing a new rule, which allows a term of a subtype S to assume the supertype T:

$$\textsc{T-Sub}\quad \frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

and by defining what types are related through the subtype relation. The definition of <: is best illustrated through an example. For that purpose, assume the example language λ_B is now enriched with *tuples* [35]. The syntax is summarized in Fig. 2.4, where the ellipses represent the forms already defined for λ_B in Fig. 2.1 and $\top$ stands for the maximum type - whose role will be cleared at the end of this section. A tuple consists of an ordered list of n elements grouped inside angle brackets, where each element t_i is a term of type T_i.

t ::= ...	**terms:**	T ::= ...	**types:**
$\langle t_i^{i\in 1..n}\rangle$	tuple	$\langle T_i^{i\in 1..n}\rangle$	tuple type
$t.i$	projection	$\top$	maximum type
v ::= ...	**values:**		
$\langle v_i^{i\in 1..n}\rangle$	tuple value		

Figure 2.4: Syntax of tuples.

An empty tuple is represented as $\langle\rangle$; a 1-tuple has the form $\langle t_1\rangle$ and type $\langle T_1\rangle$[3]. The only operation available over tuples is projection over one of its n components, which works as expected and thus we omit its evaluation rules. The typing rules are also omitted for similar reasons.

Now consider the following function that projects the first component - of type *Bool* - from a tuple:

$$\texttt{proj1} = \lambda x{:}\langle Bool\rangle.\ x.1$$
$$\hookrightarrow \langle Bool\rangle \to Bool$$

Although the function is defined as expecting terms of type $\langle Bool\rangle$, it would be safe to accept both $\langle Bool, Bool\rangle$ and $\langle Bool, Bool \to Bool, Bool\rangle$ - i.e., the function can safely be applied to a tuple of any arity and type, as long as it has a first component of type *Bool*. Generalizing: it is safe to use a tuple of type $\langle T_i^{i\in 1..n+k}\rangle$ anywhere a type $\langle T_i^{i\in 1..n}\rangle$ is expected. This observation is captured by the rule S-TplWidth displayed in Fig. 2.5.

S-Refl
$$\frac{}{S <: S}$$

S-Trans
$$\frac{S <: U \quad U <: T}{S <: T}$$

S-TplDepth
$$\frac{\text{for each } i \in 1..n\ .\ S_i <: T_i}{\langle S_i^{i\in 1..n}\rangle <: \langle T_i^{i\in 1..n}\rangle}$$

S-TplWidth
$$\frac{}{\langle T_i^{i\in 1..n+k}\rangle <: \langle T_i^{i\in 1..n}\rangle}$$

S-Arrow
$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

Figure 2.5: Subtyping rules for λ_B enriched with tuples.

The rules of Fig. 2.5 together define the subtype relation for the types in λ_B. Rules S-Refl and S-Trans are self-explanatory given the definition of subtyping stated at

[3] This particular convention (the type being $\langle T_1\rangle$ and not just T_1) is important to avoid deriving e.g. $\langle Bool, Bool\rangle <: Bool$, that would lead the typechecker to accept terms of the form if $\langle true, false\rangle$ then t_2 else t_3 as well-typed.

the beginning of this section. The rule S-TplDepth is straightforward as well: if S can be used wherever T is expected, then the same applies to $\langle S \rangle$ and $\langle T \rangle$. S-Arrow in turn requires a more careful look. The intuition is the following: assume two functions $g : S_1 \to S_2$ and $f : T_1 \to T_2$; if g is to be used in place of f, then: (1) g must accept any element of f's domain ($T_1 <: S_1$), and (2) no result produced by g can *surprise* the context where f is expected ($S_2 <: T_2$).

Having the subtype relation settled, subtype polymorphism can be applied to the identity functions example (beginning of 2.2) by defining the type $\top$ as being a supertype of every type:

$$\text{S-Top} \quad \frac{}{S <: \top}$$

and then writing the function as taking an argument of type $\top$:

$$\texttt{idTop} = \lambda x{:}\top.\ x$$
$$\hookrightarrow \top \to \top$$

The subtype relation coupled with the rule T-Sub will now allow the application of `idTop` to terms of any type. This is essentially the same idea that Java used in its legacy “generic” libraries [5, 18], where the top of the subtype hierarchy is represented by the type `Object` instead of $\top$.

Notice however the return type of `idTop`: no matter the type of the term passed to it, the function will always return it as a $\top$. This illustrates the main caveat of resorting to subtyping to write polymorphic code: by “promoting” a type to its supertype, the information about the original type is effectively lost. For instance, if `idTop` is to be applied to the term *true*, the result of the application can no longer be used as the condition in an `if` term - all the typechecker knows is that `idTop` returns a $\top$, whereas the typing rules require a condition of type *Bool*. The usual solution to this problem is to add a *casting* mechanism to the language, which has itself its own intricacies [22].

2.2.3 Parametric polymorphism

In parametric polymorphism, the ability to define polymorphic code is achieved through the use of *type parameters*. The idea is that by referring to type variables instead of particular types, we can encode *generic* functions[7]: functions that capture a common, uniform behavior while making no assumptions about the type of the values they manipulate. The type parameters of a generic function can then be *instantiated* with concrete types, yielding a specialized version of the function that operates over those types.

In formal terms, one can extend λ_B with support for parametric polymorphism by introducing two new constructs: *type abstraction* and *type application*, depicted on the left of Fig. 2.6. Type abstraction, written $\Lambda X.\ t$ (where Λ is a capital λ, and X represents

$t ::= ...$	**terms:**	$T ::= ...$	**types:**
$\Lambda X.\ t$	type abstraction	X	type variable
$t\ [T]$	type application	$\forall X.\ T$	universal type
$v ::= ...$	**values:**	$\Gamma ::= ...$	**contexts:**
$\Lambda X.\ t$	type abstraction value	Γ, X	type variable binding

Figure 2.6: Syntax of polymorphic extension of λ_B.

E-TCongr
$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T] \longrightarrow t_1'\ [T]}$$

E-TApp
$$\frac{}{\Lambda X.\ t\ [T] \longrightarrow [X \mapsto T]\ t}$$

Figure 2.7: Evaluation rules for type application.

a type variable), is what enables the expression of generic functions. Conversely, type application is written $t\ [T]$ (where the argument $[T]$ is a type expression) and represents the instantiation of a polymorphic term with some type T.

The result of applying a type abstraction to a type argument is similar to that of applying a λ-abstraction to a term argument: the parameter is replaced by the argument in the body of the abstraction, as specified by the rule E-TApp in Fig. 2.7. The rule E-TCongr is again a *congruence* rule.

Returning to the identity function example, this extension allows us to express it as:

$$\texttt{id} = \Lambda X.\lambda x{:}X.\ x$$
$$\hookrightarrow \forall X.X \rightarrow X$$

We can now obtain type-specific identity functions by applying `id` to concrete types, e.g. `id` $[Bool]$, which would reduce to the following function:

$$\lambda x{:}Bool.\ x$$
$$\hookrightarrow Bool \rightarrow Bool$$

The last step in modelling the extension concerns the typing of type abstractions. The `id` abstraction above already gives an hint: applying it to $Bool$ produces a function of type $Bool \rightarrow Bool$; likewise, applying it to $Bool \rightarrow Bool$ would yield a function typed $(Bool \rightarrow Bool) \rightarrow (Bool \rightarrow Bool)$. In general, applying `id` to a type T produces a type $T \rightarrow T$, which means the type assigned to the generic function depends on the type it is supplied as an argument. This dependency is captured by specifying `id`'s type as

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t \ : \ \forall X.T} \ \text{(T-TAbs)} \qquad \frac{\Gamma \vdash t_1 : \forall X.T_1}{\Gamma \vdash t_1\,[T_2] \ : \ [X \mapsto T_2]\ T_1} \ \text{(T-TApp)}$$

Figure 2.8: Typing rules for type abstraction and application.

$\forall X.\ X \rightarrow X$ [35]. The rule T-TAbs displayed in Fig. 2.8 defines how types are assigned to polymorphic functions in the general case. T-TApp specifies how an instantiation of a polymorphic abstraction is typed (replace the occurrences of the type variable X in T_1 by the type T_2), while also imposing that only a type abstraction can be applied to a type expression.

The language we obtain by adding these two constructs and their evaluation/typing rules - and by stripping out booleans and conditionals, which add no expressive power to the language and were introduced just to clarify the presentation - is commonly called *System F*[35] or *polymorphic lambda calculus*[40].

This language is very expressive, allowing to type e.g. the untyped self-application term $\lambda x.\ x\ x$, to which λ_B can assign no type. In System F, this term becomes typable if we define x as having a polymorphic type and instantiate it appropriately [35]:

$$\texttt{selfApp} = \lambda x{:}\forall X.\ X \rightarrow X.\ \ (x\ [\forall X.X \rightarrow X])\ x$$
$$\hookrightarrow (\forall X.\ X \rightarrow X)\ \rightarrow\ (\forall X.\ X \rightarrow X)$$

The aspect of System F that enables the expression of functions like self-application is the fact that the type variables can themselves be instantiated with polymorphic types (i.e. types that contain quantifiers) - e.g. the subterm $x\ [\forall X.X \rightarrow X]$ above - a feature known as impredicativity [35, 42]. Section 2.2.3.2 discusses how this expressiveness can be penalizing in the context of *type inference*.

2.2.3.1 Bounded parametric polymorphism

A yet more powerful refinement of System F is often called $F_{<:}$ ("F sub")[35]. $F_{<:}$ extends System F by introducing *bounded quantification*, which combines parametric polymorphism with subtyping. The goal of mixing these two features is to address some shortcomings that each of them suffer from, when used individually. In particular:

- When resorting to subtype polymorphism (Section 2.2.2) to let a function accept many subtypes S of a type T, we lose information about the original type S. For instance, the λ_B term $(\lambda x{:}\langle Bool\rangle.\ x\ \ \langle true, false\rangle).2$ would raise a type error - even though the term passed to the identity originally had a second field, that information was lost after promoting it to a $\langle Bool\rangle$.

- Using parametric polymorphism the original types are kept, but by turning the types into variables we also lose information that might be needed inside the polymorphic function.

To understand how the information lost by parametric polymorphism could be necessary, consider the following example (adapted from [35]). The goal is to write a kind of identity function for tuples of any arity ≥ 1, that takes a tuple t and returns a new 2-tuple t', such that the first component of t' is $t.1$ and the second is t itself. Using solely parametric polymorphism, this function would look like:

$$\Lambda X.\lambda t{:}X.\ \langle t.1, t\rangle$$

```
↪ Error: expected tuple type.
```

As the function is encoded *generically*, X could be instantiated to *any* type - the type checker can't guarantee that type provides a projection operation, and thus rejects the program.

Bounded quantification lets us express restrictions over the types that may be used to instantiate type parameters. These restrictions are encoded through the subtype relation: if X is a subtype of T, then X can be treated as a T, which means it offers at least the same operations as T. Thus our goal function can be expressed in $F_{<:}$ as:

$$\Lambda Y.\Lambda X{<:}\langle Y\rangle.\ \lambda t{:}X.\ \langle t.1, t\rangle$$
$$\hookrightarrow \forall Y.\forall X{<:}\langle Y\rangle.\ X \to \langle Y, X\rangle$$

Notice the extra type parameter Y, that was added in order to accept tuples with first components of any type. Although we don't give $F_{<:}$ a formal treatment in this section, Section 3.2 illustrates an example of a type system with bounded quantification.

2.2.3.2 Type inference for polymorphic systems

In a real programming language it would be convenient not to have to explicitly instantiate polymorphic functions, when the intended instance is made obvious by the argument to which it is applied.

For instance, instead of invoking `id` as $id\ [Bool]\ true$ or $id\ [\langle Bool,\ Bool\rangle]\ \langle true,\ false\rangle$, it would be preferable to just write $id\ true$ and $id\ \langle true, false\rangle$ and let the compiler *infer* the correct instance from the type of the arguments - resembling what occurs in function overloading (Section 2.2.1), but here the particular instances would all behave uniformly and would be auto-generated by the compiler.

Taking this concept further, one could think of eliding all the type annotations and e.g.

express *id* simply as $\lambda x.\ x$ - that is, to make functions *implicitly* polymorphic, a distinguishing feature of the ML family of languages.

The drawback of eliding every type annotation is that complete type inference for a system as expressive as System F is undecidable [46]. One alternative to overcome the undecidability result is to restrict the allowed forms of polymorphism. The most popular [35] restriction is the one found in Hindley-Milner type systems, allowing only *predicative* polymorphism [42], i.e. imposing that type variables cannot be instantiated with polymorphic types. This approach has the downside that it limits the expressiveness of the system. The next section presents an alternative that takes a different compromise, obtaining a technique scalable to powerful type system features at the cost of requiring some explicit type annotations.

2.3 Bidirectional Typing

When implementing a set of typing rules in a real compiler there are two important considerations to be made: first, typing rules do not necessarily describe an algorithm - section 2.3.1 discusses this matter in detail; secondly, as mentioned in the previous section, although the typing rules require the polymorphic types to be explicitly instantiated, it would be convenient if the *surface syntax* of the language hid these details from the programmer, allowing some type annotations to be inferred. But by resorting to pure inference, we are limited by the undecidability result.

Bidirectional typing is a frequent solution to both these problems [14, 16]. By combining *type checking* with *type inference* or *synthesis*, it is able to handle rich sets of typing features while requiring relatively few annotations. Moreover, its algorithmic nature also makes it easy to translate a set of bidirectional rules into an implementation.

In both *checking* and *synthesis*, the primary goal of the typing process is to prove judgments of the form $\Gamma \vdash t : T$, that state the term t is well-typed. This process can be seen as the bottom-up construction of a derivation [33]. In that sense, we can view a set of typing rules as describing an algorithm[4], where each rule defines a different case of the algorithm based on the form of the term t in its conclusion. For example, one can interpret the typing rule for conditionals:

$$\text{(T-IF)}\ \frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

as describing the case for if terms in a function `check` or `synth` (resp. if checking or synthesizing). Listings 2.1 and 2.2 illustrate a simplified implementation of such functions in pseudo-code (adapted from [8]). In Listing 2.1, we check that the if-term has the provided input type *ty* by verifying that: (1) t_1 has type *Bool*, and (2) both branches t_1, t_2 have the

[4] Again, section 2.3.1 discusses why this isn't always the case, i.e. why not every set of rules describes an algorithm.

Listing 2.1: check function

```
check ctx (If t1 t2 t3) (Some ty) =
  case (check ctx t1 BoolTy,
        check ctx t2 ty,
        check ctx t3 ty)
  of
    (Some BoolTy, Some ty2, Some ty3) → Some ty
    _                                 → None
```

Listing 2.2: synth function

```
synth ctx (If t1 t2 t3) =
  case (synth ctx t1,
        synth ctx t2,
        synth ctx t3)
  of
     (Some BoolTy, Some ty2, Some ty3) → if ty2 = ty3
                                          then Some ty2
                                          else None
     _                                 → None
```

expected type ty. Notice that the function returns an optional type[5] just for convenience of presentation - the context where the function was called already knows the returned type ty. The alternative was to do nothing if every term checks against the expected type, and throw some kind of exception otherwise.
In contrast, in Listing 2.2 we first synthesize types for t_1, t_2 and t_3. Only then we compare the inferred type for t_1 with *Bool* and verify that the types inferred t_2 and t_3 match. The function returns a type only if this last test passes, and returns None otherwise to mean the term is *ill-typed*.

The key distinction between checking and synthesizing then lies in what parts of the judgment $\Gamma \vdash t : T$ constitute the input to the algorithm. In checking, the goal is to *verify* that a given program has a known type, hence all of (Γ, t, T) should be available (i.e. input). Conversely, the objective of synthesizing is to *determine* the type of a given program, which means T is the output of the algorithm.

Bidirectional typing combines the two approaches by splitting the typing judgment $\Gamma \vdash t : T$ into two mutually recursive judgments (where the inputs are marked with + and outputs with – [33]):

- $\Gamma^+ \vdash t^+ \Rightarrow T^-$ "under assumptions in Γ, the term t synthesizes type T"

- $\Gamma^+ \vdash t^+ \Leftarrow T^+$ "under assumptions in Γ, the term t checks against type T"

[5] In the style of Haskell's Maybe or Rust's Option.

The point of using two judgments is that the typing algorithm will be alternating between checking and synthesizing duty. On one hand, we want to elide as much type annotations as possible, hence the algorithm will *synthesize* whenever there is enough information in the term and the context. On the other, there are some terms whose body (together with the context) doesn't provide enough information (e.g. un-annotated λ-abstractions [8]) - these are the terms that require annotations. Although an annotation allows to trivially infer the type of the annotated term, the algorithm still needs to *check* the term against that type. Of course, in the process of checking it might change to synthesize mode if a needed type can be inferred.

As an example, consider the pair of rules for function abstraction and application - together with rules for annotations and variables that aid in connecting the two:

$$\textsc{BT-App}\quad \frac{\Gamma \vdash t_1 \Rightarrow T \to T' \quad \Gamma \vdash t_2 \Leftarrow T}{\Gamma \vdash t_1\ t_2 \Rightarrow T'} \qquad \textsc{BT-Ann}\quad \frac{\Gamma \vdash t \Leftarrow T}{\Gamma \vdash (t : T) \Rightarrow T}$$

$$\textsc{BT-Abs}\quad \frac{\Gamma, x{:}T \vdash t \Leftarrow T'}{\Gamma \vdash \lambda x.\ t \Leftarrow (T \to T')} \qquad \textsc{BT-Var}\quad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \Rightarrow T}$$

An application is a *destructor*[33] - it generates a result of a smaller type from a component of a larger type - hence its type can be inferred; and if we are applying a function, then we must already know the type of that function in advance, either through an annotation (BT-Ann) or from the context (BT-Var). Notice t_2 is just checked against T - T is already known after inferring the type of t_1.
An abstraction is a *constructor* as it produces a larger type, so its subterms don't have enough information - it must be checked against some type. This type will then be propagated to the premise check, allowing to *assume* a type for the argument (in the context). This is why bidirectional systems normally require us to annotate (the outermost) function declarations [14, 33].

2.3.1 Algorithmic typing rules

As referred in the previous section, not all sets of typing rules can be outright turned into an algorithm by "reading" the rules bottom-up. Consider for example a type system crafted to be type checked, offering subtyping capabilities through the rule:

$$\textsc{T-Sub}\quad \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

The reason this rule is problematic is that it imposes no constraints over the shape of the term t. This means that when checking any term t' there will always be two options: either applying the rule whose conclusion matches the form of t', or applying T-Sub.

But a deterministic algorithm would have no way of knowing which rule to employ at each step. The key issue is that this rule was not written in a *syntax-directed* way - it's not possible to decide which rule to apply solely based on the syntactic shape of a term.

Another problematic situation arises from the rule for transitivity of subtyping:

$$\textsc{S-Trans}\quad \frac{S <: U \quad U <: T}{S <: T}$$

if we were to read this rule from bottom to top, we would find ourselves in trouble while checking the first premise: it requires us to "invent" a seemingly arbitrary type U, as we only get S and T as input. This rule violates the principle of *mode-correctness* [14], which states that the inputs to each premise should be uniquely determined by the inputs to the conclusion and the outputs of earlier premises.

In general, bidirectional systems solve the first problem by handling subsumption implicitly in the definition of the judgment $\Gamma \vdash t \Leftarrow T$ [33, 23]. In particular, one typically defines a rule like

$$\textsc{BT-Sub}\quad \frac{\Gamma \vdash t \Rightarrow S \quad S <: T}{\Gamma \vdash t \Leftarrow T}$$

which replaces the use of subsumption, but is only applied when no other *check* rule applies. For instance, in the rule BT-App presented in the previous section, the premise $\Gamma \vdash t_2 \Leftarrow T$ would actually represent two premises $\Gamma \vdash t_2 \Rightarrow S$ and $S <: T$. Notice the rule BT-Sub is also mode-correct: as S is synthesized in the first premise, the inputs to the second premise $S <: T$ are both available.

The second problem is handled similarly in both bidirectional and non-bidirectional systems, making transitivity implicit by bundling together rules that could only be "pasted together" in a derivation by the transitivity rule [35].

2.3.2 Application to polymorphic type systems

Recent work [24, 16, 14, 42] has applied the ideas of bidirectional typing to develop bidirectional systems for languages with advanced forms of polymorphism such as first- and higher-rank polymorphism [15, 24]; existential polymorphism and GADTs [16, 41] and even impredicative polymorphism [42]. The key observation is that bidirectional systems can account with relative ease for various forms of polymorphic type instantiation through a combination of algorithmic subtyping (as seen in the previous section) and local constraint solving. The approach is flexible enough to be able to integrate Hindley-Milner style inference (when it can be applied) with type annotations that are needed for various instances of higher-rank polymorphism or impredicativity.

3 Related Work

In this chapter we discuss the two key works that support this thesis. Both used small languages to model the central features of industrial languages, as a way to attain manageable type-safety proofs and investigate the addition of *generics*. Featherweight Go [20] is discussed in greater detail as this work derives directly from it.

3.1 Featherweight Java

Featherweight Java [22] is a functional, object-oriented core of the Java language, obtained by distilling the original language into a minimal set of features. The goal of the authors in considering just this core language was to develop a compact model that allowed a type safety proof as succinct as possible, while still capturing the key ideas that would shape such a proof for full Java.

The model encompasses only five forms of expressions: variables, object creation, field access, method invocation and casting; computation happens at the latter three. The most interesting of these features are casts. On one hand, their operation depends on the definition of the subtype relation, which is relevant because it is defined in a fundamentally different way when compared to Go. In Java (and in FJ) subtyping is *nominal*: a class `C` is a subtype of a class `D` if this is declared on `C`'s source code through the `extends` keyword (or if `C` extends some `C'` such that `C'` is a subtype of `D`, i.e. it is transitive) - and also through `implements`, in the case of Java. On the other hand, the formal modelling of casts under a small-step semantics also unveiled an error in a type soundness proof for Classic Java [22], which was solved in FJ by introducing the concept of *stupid casts*.

3.1.1 Featherweight Generic Java

One of the reasons for considering a small model is that it can be extended with interesting features while retaining its compactness. This allows a rigorous study of these features in a somewhat isolated setting, focusing on the particular aspects that characterize them.

Igarashi et al. [22] extend FJ with *generics* as a way to investigate how parametric polymorphism can be integrated into Java. The main technique they use, following GJ [5], is based on the idea of translating a language *with* generics (FGJ) to a language *without* generics (FJ). By doing so, the formal definition of the translation specifies a semantics for FGJ, which can be used as a specification for an implementation.

The translation essentially consists of *erasing* all the type parameters, replacing them by their bounds. For example, a class declared as *class C<T extends Object>{...}* is erased to *class C<Object>{...}*. This approach has the advantages that it is simple to implement and that it generates only one version of the code for each parameterized type. However it has two major drawbacks:

(1) the underlying code is shared by all the instantiations, which requires that every type must be represented uniformly;

(2) there is no information about the types used to instantiate type parameters in run-time.

In (1), the usual way to make representations uniform is to consider all the types as pointers to heap-allocated objects [7]. But that implies that when instantiating a type parameter with a primitive type, first it's needed to wrap it in some reference type - e.g. wrap an int in an $Integer$. This is both inconvenient for the programmer and a source of runtime overhead, as it requires the introduction of boxing and unboxing operations.

The problem with (2) is mostly that it imposes some unnatural restrictions on the source language. For example, inside a class parameterized on T it's not possible to create arrays with elements of type T, nor is it allowed to test if a term is an instance of the type T.

3.2 Featherweight Go

The work on formalizing generics in Go [20] follows the footsteps laid by Featherweight Java [22], in the sense that they also (1) restrict their model to a small functional subset of the language (FG); and (2) formalize generics by enriching the model with parametric polymorphism capabilities (FGG) and then translating it back to the original version (FG). Despite that, the languages being modelled have some key differences. In particular, while in Java there are classes, in Go there are structs; and while in Java subtyping is *nominal* (Section 3.1), in Go it is *structural*. Besides, the two works also diverge in their strategy for translating generic code: in FJ they resort to *erasure*, while in FG they use *monomorphisation*.

3.2.1 Go vs Java

The main distinction between Go's structs and Java's classes is how they define methods. In Java, a class *has* a method if that method is either present in its source code or in the source of one of its superclasses (in which case the method is inherited). This means that after a class is compiled, there is no way to add methods to it without having to recompile the class's code. While in Go a struct S defines a method if there is a declaration for a function with a *receiver* of type S. For instance, the following declaration defines a method `Apply`, that takes an **`int`** and returns an **`int`**, for structs of type `incr` (the syntax is detailed in the next section):

```
type incr struct { n int }

func (this incr) Apply(x int) int {
    return x + this.n
}
```

The other key aspect that differentiates Go from Java is the definition of the subtype relation. In the nominal subtyping of Java, a class implements an interface only if this is explicitly declared, either in the class itself or in one of its superclasses. Similarly, an interface can only *explicitly* become a subtype of another. This means that a class has a *closed* set of supertypes - only those that were declared. In contrast, Go's structural subtyping stipulates that a struct *implicitly* implements (i.e. is a subtype of) an interface if the struct defines all the methods specified by the interface. Likewise, an interface I implements another interface I' if I specifies all the methods that I' does - with exactly the same signature - and possibly some more. The implicitness of structural subtyping implies that a struct has an *open* set of supertypes - it's always possible to declare a new interface that specifies a subset of the methods defined by a struct, without having to recompile the struct's code.

3.2.2 The Featherweight Go Language

Figure 3.1 illustrates the syntax of Featherweight Go. The notation is standard, where $\overline{f}$ stands for f_1, ..., f_n and similarly $\overline{f\ t}$ means $f_1\ t_1$, ..., $f_n\ t_n$.
The language is a functional fragment of Go that omits burdensome-to-model imperative features such as assignment (and side-effects in general), and concurrency-related constructs such as channels and green threads, for which there isn't a standard formalization strategy. Resembling Featherweight Java, it only has five forms of expressions: variables, structure literals (⇔ object creation), field selection, method calls and type assertions (⇔ casts).

Field name	f	Expression	$d, e ::=$
Method name	m	Variable	x
Variable name	x	Structure literal	$t_S\{\overline{e}\}$
Structure type name	t_S, u_S	Select	$e.f$
Interface type name	t_I, u_I	Method call	$e.m(\overline{e})$
Type name	$t, u ::= t_S \mid t_I$	Type assertion	$e.(t)$
Method signature	$M ::= (\overline{x\ t})\ t$		
Method specification	$S ::= mM$		
Type Literal	$T ::=$		
Structure	**struct** $\{\overline{f\ t}\}$		
Interface	**interface** $\{\overline{S}\}$		
Declaration	$D ::=$		
Type declaration	**type** $t\ T$		
Method declaration	**func** $(x\ t_S)\ mM$ $\{$**return** $e\}$		
Program	$P ::=$ **package main**; $\overline{D}$ **func main**() $\{_ = e\}$		

Figure 3.1: Featherweight Go syntax

There's only two forms of types: structs and interfaces. The declaration:

```
type incr struct { n int }
```

shown before introduces the type `incr`, a struct with a single field `n` of type `int` [1] (in FG, as in Go, the type comes after the variable name). An interface type is introduced in an identical way, except it declares a set of *method specifications* instead of fields. Notice that in an interface a method is specified as name (m) followed by signature (M), while the declaration of a method for a struct (i.e. declaring the struct type t_S implements method m) prefixes the method specification with the keyword **func** and the *receiver* (a variable of type t_S) of the method.

As an example, the code shown below illustrates a representation for functions of integer domain and co-domain (`ItoIFunction`) and an instance of such a function (`incr`). The type `incr` *implements* the interface `ItoIFunction`, since the second declaration defines the method `Apply` - whose signature matches the one specified in the interface - for receivers of type `incr`.

```
type ItoIFunction interface {
    Apply(x int) int
}

func (this incr) Apply(x int) int {
    return x + this.n
}
```

[1] FG doesn't have integers (indeed including them is part of the purpose of this work), but we use them to simplify the examples, which are in general adapted from the original FG paper [20].

Finally, a program in FG is composed by a set of declarations (of types and methods) and a top-level expression. There are no declarations inside method bodies, which means all the declarations are globally scoped.

The language is type-safe in the sense described in Section 2.1.4 [20]. For instance, consider method calls, one of the (only) three forms of reducible expressions. First, a method declaration is judged as being *syntactically* well-formed according to the following rule:

$$\textsc{T-Func}\quad \frac{\begin{array}{c} distinct(x, \overline{x}) \\ t_S\ ok \qquad \overline{t\ ok} \qquad u\ ok \qquad x : t_S, \overline{x : t} \vdash e : t \qquad t <: u \end{array}}{\textbf{func}\ (x\ t_S)\ m(\overline{x\ t})\ u\ \{\textbf{return}\ e\}\ ok}$$

that is, the declaration is well-formed if: the variable names used in the receiver and the arguments all have distinct names; the receiver, the parameters and the return typed are all well-formed (i.e. the types are defined in the program); and if under the assumption that the arguments are well-typed, it's possible to conclude that the value returned by the method has a type that *implements* (Section 3.2.1) the declared return type. Notice that subtyping is used as a premise instead of including a general (non-algorithmic) subsumption rule (Section 2.3.1). Then a method call is typed according to the rule:

$$\textsc{T-Call}\quad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash \overline{e : t} \qquad (m(\overline{x\ u})\ u) \in methods(t) \qquad \overline{t <: u}}{\Gamma \vdash e.m(\overline{e}) : u}$$

which verifies that the receiver and the arguments are well-typed, checks that there is such a method defined for the type of the receiver, and that the arguments' types implement the types of the parameters. It's fairly straightforward to conclude from the previous two typing rules, together with the following evaluation rule (where *body* represents an auxiliary function that looks up the declaration of a method, given the type of the receiver and the method name):

$$\textsc{E-Call}\quad \frac{(x : t_S, \overline{x : t}).e = body(type(v).m)}{v.m(\overline{v}) \longrightarrow e[x \mapsto v, \overline{x \mapsto v}]}$$

that if a method call expression is assigned a type u, then the result of evaluating this expression will also have either type u or a type t such that $t <: u$. A similar reasoning can be applied to the other two reducible forms, field selection and type assertion.

3.2.3 Featherweight Generic Go

Featherweight Generic Go extends Featherweight Go with parametric polymorphism. Its syntax is presented in Fig. 3.2, where the differences from FG are highlighted. The main change is that type and method declarations in FGG can include *bounded* type parameters, represented by *type formals* Φ, Ψ of the form **type** $\overline{\alpha\ \tau_I}$, where α is a type parameter and the bound τ_I is an interface type.

Field name	f	Type	$\tau, \sigma ::=$
Method name	m	Type parameter	α
Variable name	x	Named type	$t(\overline{\tau})$
Structure type name	t_S, u_S	Structure type	$\tau_S, \sigma_S ::= t_S(\overline{\tau})$
Interface type name	t_I, u_I	Interface type	$\tau_I, \sigma_I ::= t_I(\overline{\tau})$
Type name	$t, u ::= t_S \mid t_I$	Interface-like type	$\tau_J, \sigma_J ::= \alpha \mid \tau_I$
Type parameter	α	Type formal	$\Phi, \Psi ::= \textbf{type}\ \overline{\alpha\ \tau_I}$
Method signature	$M ::= (\Psi)(\overline{x\ \tau})\ \tau$	Type actual	$\phi, \psi ::= \overline{\tau}$
Method specification	$S ::= mM$	Expression	$e ::=$
Type Literal	$T ::=$	Variable	x
Structure	$\textbf{struct}\ \{\overline{f\ \tau}\}$	Structure literal	$\tau_S\{\overline{e}\}$
Interface	$\textbf{interface}\ \{\overline{S}\}$	Select	$e.f$
Declaration	$D ::=$	Method call	$e.m(\overline{\tau})(\overline{e})$
Type declaration	$\textbf{type}\ t(\Phi)\ T$	Type assertion	$e.(\tau)$
Method declaration	$\textbf{func}\ (x\ t_S(\Phi))\ mM\ \{\textbf{return}\ e\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{_ = e\}$		

Figure 3.2: FGG syntax

Type names are replaced by types τ, σ which can either be type parameters α or named types $t(\overline{\tau})$. This is reflected in structure declarations **struct** $\{\overline{f\ \tau}\}$ and literals $\tau_S\{\overline{e}\}$, and type assertions $e.(\tau)$. A method call now also has to provide type arguments before the "regular" arguments: $e.m(\overline{\tau})(\overline{e})$.

As an example, Listing 3.1 illustrates most of these constructs. It starts by declaring the empty-interface type `Any`, which can be declared in plain FG, and is trivially implemented by every other type (a type implements an interface if it implements all the methods defined by the interface). It then defines the polymorphic interface `Function`, parameterized on two "unbounded" type variables `a`,`b` (the bound `Any` imposes no constraints), that defines a single method `Apply` which receives an `a` and returns a `b`.
Next it declares an interface for polymorphic lists. This interface represents all the types that implement the polymorphic method `Map`, which produces a `List` of `b`'s by applying the function `f` to every element of the receiver `List` (of `a`'s). Notice how the type `Function(a,b)` of the argument `f` refers to both the type parameter of the interface, `a`, and the parameter of the method, `b`. Following is the declaration of the structs that will implement that interface - this encoding of lists gives a taste of the functional style of FG/FGG. Observe that the empty list still needs to be parameterized.
`Nil` and `Cons` are defined as implementing the interface `List` by declaring a `Map` method for them. The receiver type parameters are also specified in the method's header, and can be referred in the types of the method's arguments. For a struct to implement an interface, the specifications of the struct's methods must correspond exactly (modulo variable names [20]) to the specifications in the interface. This is the reason `Nil` also needs a type parameter.

Listing 3.1: Example program in FGG

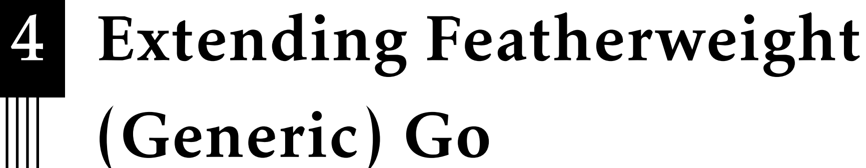
```
type Any interface {}
type Function(type a Any, b Any) interface {
    Apply(x a) b
}

type List(type a Any) interface {
    Map(type b Any)(f Function(a, b)) List(b)
}
type Nil(type a Any) struct {}
type Cons(type a Any) struct {
    head a
    tail List(a)
}

func (xs Nil(type a Any))
    Map(type b Any)(f Function(a,b)) List(b) {
  return Nil(b){}
}
func (xs Cons(type a Any))
    Map(type b Any)(f Function(a,b)) List(b) {
  return Cons(b)
    {f.Apply(xs.head), xs.tail.Map(b)(f)}
}

func main() {
    _ = Cons(int){3, Cons(int){6, Nil(int){}}}
}
```

Finally, the main function simply creates a list to illustrate how every instantiation of the polymorphic structs has to explicitly provide the type arguments. Albeit not depicted here, the explicit type arguments are also required when instantiating polymorphic methods.

The properties of type safety for FG are also enjoyed by FGG, requiring some adaptations detailed in the original paper [20].

3.2.3.1 Considerations

Being a small model in the spirit of Featherweight Java, FGG doesn't account for primitive types. Although they may be *representable* in the language [19], it is not evident how generic types and methods will interact with the primitive types of Go and the primitive operations over them, which are not defined using methods.
Besides, as the main function in Listing 3.1 illustrates, instantiating a polymorphic type can be very verbose, requiring explicit type arguments in every instantiation, even when they are obvious from the context.



4 Extending Featherweight (Generic) Go

This chapter describes the main contributions of this thesis: extending Featherweight (Generic) Go with type lists. Ir order to investigate type lists in a context where they might exhibit non-trivial behaviors, we add some extra features to F(G)G. In particular, we add primitive types and operations, as well as a generalized form of type declarations. Both extensions have significant implications either on the type system, the operational semantics or the monomorphisation process. Some of these implications are independent from the presence of generics, hence we start by exploring them in the simpler context of Featherweight Go.

4.1 Additions to Featherweight Go

We begin by describing the features we add to Featherweight Go, explaining the key challenges they pose, and then show how we realize them formally in the typing system and the operational semantics. We use the inductive rules more as a form of pseudo-code than with the intent of proving type safety over them, as the proofs are out of the scope of this work.

4.1.1 Primitive types and operations

Primitive types in Go bear some intricacies. On one hand, it is not permitted to mix variables of different types in a primitive operation, regardless of how safe the operation might appear. For instance, the following function's body is not typable:

```
func add(x int32, y int64) int64 { return x + y }
```

and likewise, it is not allowed to provide a `float32`-typed argument to a function expecting a parameter of type `float64`.

Listing 4.1: Mixing constants and typed variables

```
func addF(x float32) float32 {
    return x + 3
}

func addI(x int64) int64 {
    return x + 3.0
}

func main() {
    _ = addF(10 - 0.5) // ok
    _ = addI(10)       // ok
    _ = addI(9.5)      // error
}
```

On the other hand, these rules are not as restrictive as they might appear when considering primitive literal (constant) values in addition to primitive-typed variables. The program of Listing 4.1 demonstrates some examples of this. Notice that it's possible to sum a `float32` with an "integer" constant, and so is adding an `int64` with a "float" constant (as long as its numeric value corresponds to an integer). Moreover, mixing "integer" and "float" constants is also allowed (e.g. `10 - 0.5`). Yet note that passing a "float" constant (whose value is not an integer) to a context expecting a `int64` would still raise a type error.

Although in a language with implicitly-inserted type conversions (e.g. Java, C, C++) this behavior wouldn't be surprising, the rigid typing rules of Go imply that constants must be given a special treatment to achieve this kind of flexibility. In Go terms, constants of this sort [1] are considered *untyped*: "An untyped constant is just a value, one not yet given a defined type that would force it to obey the strict rules that prevent combining differently typed values." [37].

Therefore, a model of Go that includes primitive types should take this aspect into account, as one needs the ability to write primitive literals in order to insert values for the programs to manipulate. In section 4.1.3 we define typing rules that model most of the untyped constants' flexibility.

4.1.2 General type definitions

Another aspect of Go that is only partially modelled in FG is the general mechanism to declare types by binding a new name to an already existing type. In Go, type declarations have two forms: alias declarations and type definitions [45]. We won't consider alias declarations as they only introduce a "synonym" for another type – i.e. the alias denotes the same type as the *aliased*. Instead we focus on type definitions and hence from here on we refer to type declarations and type definitions interchangeably.

[1] Go allows declaring explicitly-typed constants, but this work doesn't encompass constant declarations.

A type definition has the form:

$$\texttt{type}\ identifier\ srcType$$

and it binds an identifier to a new, distinct type with the same *underlying* type and operations as the source type. Type definitions may be used as a way to enforce stronger static checking, since they introduce new types which can't be implicitly converted to the types they were created from – nor to other "similar" types. For instance, in a program that makes computations using different units of temperature, it might be helpful to introduce two new types:

```
type Celsius float32
type Kelvin  float32
```

so as to make sure the different units are not mixed, while still exploiting all the primitive operations that `float32` supports. Additionally, declaring a new type adds the possibility of defining methods for that type – which otherwise would not be possible for base types. Referring to the previous example, it could be useful to add methods such as `ToKelvin` and `ToCelsius` to the respective types.

While FG includes a form of type declarations, they are a restricted version of Go's. They don't allow, for example, to declare the two different types:

```
type Point struct { x, y float64 }
type Polar Point
```

The reason is that FG's type declarations only allow creating types from type literals (cf. Fig. 3.1, section 3.2.2). Besides, type literals can only appear in type declarations, meaning they aren't recognized as first class types. Thus, adapting FG's type declarations to match Go's requires generalizing the source type so that it can be any type. This generalization has two main implications: first, type literals must also be considered types. This introduces the possibility of having parameters whose type is an "anonymous" struct/interface type. For example, in Go we can define a function

```
func Fun (p struct{x,y float64}) { ... }
```

which may then be called with either of the following arguments:

```
Fun( struct{x,y float64}{0.0, 0.0} )
Fun( Point{1.1, 1.1} )
Fun( Polar{2.2, 2.2} )
```

Secondly, since a type declaration may refer to any type as its source type, including other named types, the type system needs some general mechanism to determine, for a given declared type, (1) what operations (primitives or e.g. field selection) it supports, and (2) which values it accepts. This is necessary because when a type is created with a declaration, that type is considered distinct from the type used to create it. Go solves this

Field name	f
Method name	m
Variable name	x
Primitive type	$B ::= bool \mid string \mid int32 \mid \ldots$
Undefined primitive type	B_U
Type name	$t, u \quad (t, u \neq B)$
Type Literal	$L ::=$
Structure	$\textbf{struct}\ \{\overline{f\ T}\}$
Interface	$\textbf{interface}\ \{\overline{S}\}$
Type	$T, U ::= B \mid t \mid L$
Extended Types	$T^+ ::= T \mid B_U$
Method signature	$M ::= (\overline{x\ T})\ T$
Method specification	$S ::= mM$
Declaration	$D ::=$
Type declaration	$\textbf{type}\ t\ T$
Method declaration	$\textbf{func}\ (x\ t_S)\ mM\ \{\textbf{return}\ e\}$
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{_ = e\}$

Expression	$d, e ::=$
Primitive Literal	ℓ
Variable	x
Structure literal	$T\{\overline{e}\}$
Select	$e.f$
Method call	$e.m(\overline{e})$
Type assertion	$e.(T)$
Type conversion	$T^+(e)$
Binary operation	$e \oplus e$
Comparison	$e \asymp e$
Literal	$\ell ::=$
Bool literal	ℓ_B
Int literal	ℓ_I
Float literal	ℓ_F
String literal	ℓ_S

Figure 4.1: Featherweight Go syntax

with the notion of *underlying types*: a type admits the same operations and values as its underlying type. Section 4.1.3.2 formalizes how to adapt this notion to FG.

4.1.3 Extending Featherweight Go

4.1.3.1 Syntax

Figure 4.1 describes the syntax of FG enriched with the proposed extensions. We add the primitive (or base) types $bool$, $string$, $int32$, $int64$, $float32$ and $float64$. The goal was to have a set of types diverse enough for most of the heterogeneity issues to show up: we have operators specific to one type (||, &&), overloaded operators (+) and, as we show in Section 4.1.3.3, the possibility of mixing different numeric types.

As hinted in the previous section, type literals were "promoted" and thus a type T is now either a primitive type B, a named type t or a type literal L. We also introduce a notion of extended types T^+, which augment the set of types with undefined primitive types B_U. These types, as detailed in Section 4.1.3.3, serve to model the untyped constants and are not part of the surface syntax of the language; they are only used in typing rules. Regarding expressions, we add the primitive literals corresponding to the supported base types, the binary operations +, −, && and || – these are all written as $\oplus$ for compactness – and relational operations < and > (represented as $\asymp$). This explicit distinction is due to the operations' result types: in a binary operation, it's the same as (or related to) the inputs'; in a comparison, it is always a boolean. Albeit not explicit in the grammar, the operators follow the expected precedence and associativity rules.

$$\frac{}{underlying(L) = L} \qquad \frac{}{underlying(B) = B} \qquad \frac{}{underlying(B_U) = B_U}$$

$$\frac{\textbf{type}\ t\ T \in \overline{D} \quad underlying(T) = U}{underlying(t) = U} \qquad \frac{underlying(T^+) = B_U}{primType(T^+)} \qquad \frac{underlying(T^+) = B}{primType(T^+)}$$

$$\frac{underlying(T) = \textbf{struct}\{\overline{f\ T}\}}{struct(T)} \qquad \frac{underlying(T) = \textbf{interface}\ \{\overline{S}\}}{interface(T)}$$

$$\frac{underlying(T) = \textbf{struct}\{\overline{f\ T}\}}{fields(T) = \overline{f\ T}} \qquad \frac{(\textbf{func}\ (x\ t)\ m(\overline{x\ T})\ U\ \{\textbf{return}\ e\}) \in \overline{D}}{body(t.m) = (x : t, \overline{x : T}).e : U}$$

$$\frac{}{methods(B_U) = \emptyset} \qquad \frac{}{methods(B) = \emptyset} \qquad \frac{}{methods(\textbf{struct}\{\overline{f\ T}\}) = \emptyset}$$

$$\frac{underlying(T) = \textbf{interface}\ \{\overline{S}\}}{methods(T) = \overline{S}} \qquad \frac{\neg interface(t)}{methods(t) = \{mM \mid (\textbf{func}\ (x\ t)\ mM\ \{\textbf{return}\ e\}) \in \overline{D}\}}$$

$$tdecls(\overline{D}) = [t \mid (\textbf{type}\ t\ T) \in \overline{D}] \qquad mdecls(\overline{D}) = [t.m \mid (\textbf{func}\ (x\ t)\ mM\ \{\textbf{return}\ e\}) \in \overline{D}]$$

$$\frac{mM_1, mM_2 \in \overline{S} \text{ implies } M_1 = M_2}{unique(\overline{S})}$$

Figure 4.2: FGe auxiliary functions

We further observe that expressions include a form of type conversions $T^+(e)$ - although this was not a planned extension, it emerged as a consequence of introducing more forms of values (cf. Fig. 4.8, sec. 4.1.3.5), and the need to distinguish between them at the type level. This is detailed in Section 4.1.3.5.

4.1.3.2 Auxiliary functions

Figure 4.2 presents some auxiliary functions which will be used throughout the next sections [2]. We start by formalizing the notion of *underlying type*. It is a recursive definition, whose base cases are the types who are not named types – for these, the underlying types are the types themselves. For named types, the definition recurses on the type used as source in the named type's declaration. The definition must be recursive since we can have, as seen before, the two declarations:

[2] Although this doesn't quite follow the order in which the features were introduced in sections 4.1.1 and 4.1.2, these auxiliary functions are presented right away because they don't depend on any additional definition.

$$\frac{}{type(T\{\overline{v}\}) = T} \qquad \frac{}{type(T(\ell)) = T} \qquad \frac{}{type(\ell_B) = bool_U} \qquad \frac{}{type(\ell_S) = string_U}$$

$$\frac{\ell \in range(int32)}{type(\ell) = int32_U} \qquad \frac{\ell \in range(int64)}{type(\ell) = int64_U} \qquad \frac{\ell \in range(float32)}{type(\ell) = float32_U} \qquad \frac{\ell \in range(float64)}{type(\ell) = float64_U}$$

$$\frac{underlying(T^+) = B_U}{tag(T^+) = B} \qquad \frac{underlying(T^+) = B}{tag(T^+) = B} \qquad \frac{primType(T^+) \quad B = tag(T^+)}{fitsIn(B_U, T^+)}$$

$$\frac{primType(T^+) \quad isNumeric(T^+) \quad isNumeric(B_U) \qquad B \leq tag(T^+)}{fitsIn(B_U, T^+)}$$

Figure 4.3: FGe auxiliary functions for primitive types

```
type Point struct { x, y float64 }
type Polar Point
```

where, although `Point` and `Polar` designate different types, their underlying type is the same.

We then define predicates that check whether a type is a primitive, struct or interface type. They essentially abstract over the *underlying* function – for instance, the test $struct(T)$ would succeed when applied to either `Polar`, `Point` or even `struct{ x, y float64 }`. These predicates will be used mostly to check if a type has some desired characteristics or, in other words, what operations it supports. Therefore *primType* groups B_U and B types under the same umbrella, since the (primitive) operations available for them are the same.

The lower half of the figure consists of straightforward adaptations of the auxiliary functions already defined in the original FG paper [20]. Here we can see, for example, the first appearance of anonymous struct types: they always have empty method sets. Notice, though, that anonymous interface types are also featured, but the function *methods* makes no distinction between named and anonymous interfaces.

4.1.3.3 Auxiliary functions for primitives

Figure 4.3 presents definitions that mainly target undefined primitive types. Figure 4.4 introduces some predicates that will be useful in formalizing primitive operations. We start by expanding on the definitions depicted in the former.

The upper half of Fig. 4.3 defines the function *type*, generalizing a function that was also defined in the FG paper [20]. The purpose is fundamentally the same: to return the type of a *value*. Morally it is a function that is only used during evaluation, but we reuse it in the typing rules (Fig 4.7, sec. 4.1.3.4), as it factors the typing of primitive literals (or untyped constants).

The first clause adapts the one defined in FG to also take anonymous structs into account. The second derives from the need to separate untyped constants from typed primitive values; this aspect is thoroughly explained on Section 4.1.3.5.

The remaining clauses specify how to type a primitive literal. Although in Go they are referred to as *untyped* constants, formalizing a type system requires us to assign a type to every form of expression.
The main idea is then to assign to a literal its *least restrictive* type. For instance, an integer literal 1 would be assigned the type $int32_U$, while the constant $2,147,483,648$ would already be typed as $int64_U$ (since it doesn't belong to the range of values that $int32$ represents). The same happens for float literals, save some exceptions: in particular, a literal such as 1.0 is initially assigned the type $int32_U$. This fact is implied by the notation employed – in that, for numeric literals, we write *type*(ℓ) instead of *type*(ℓ_I) and *type*(ℓ_F) (ℓ encompasses both ℓ_I and ℓ_F) – and we further assume that the *range* function generates such literals (e.g. 1.0, 1.00, ...) when applied to integer types. [3]

The undefined types are similar to the primitive types in that e.g. a $int32_U$ supports the same operations as an $int32$ (e.g. addition of 32 bit integers). This is reflected by the notation: an undefined primitive type B_U is represented as a primitive type B plus an "undefined mark" $_U$. We introduce the function *tag* as a mean to allow comparing defined and undefined types.

What distinguishes undefined primitive types is that, in a primitive operation, they are compatible with arbitrarily many other types, while their defined counterparts are only compatible with themselves. The relation *fitsIn* captures the conditions necessary for an undefined type to be compatible with another type. It is based on the notion of *restrictiveness*: $int32$ is less restrictive than $int64$ because the former can always safely be converted to the latter. Similarly between $int64$ and $float32$ – although in some cases we may lose precision, but we don't take precision losses into consideration. We thus assume, in $fitsIn$, the following order between numeric tags:

$$int32 < int64 < float32 < float64$$

As an example, a value `1` would be initially assigned the type $int32_U$. This value may then be added to values such as `2147483648`, `float64(1.5)` or even `Kelvin(0)`, since the pairs $(int32_U, int64_U)$, $(int32_U, float64)$ and $(int32_U, Kelvin)$ all belong to the relation *fitsIn*. Notice that *fitsIn* also includes a clause asking for *tag* equality. It targets non-numeric undefined types, since e.g. a $bool_U$ is only compatible with bool-like types.

All in all, the undefined primitive types behave very much like the primitive types in languages with automatically-inserted conversions (e.g. the "usual arithmetic conversions" of C++). They allow mixing operands of different types and, when it happens, the operand with the least restrictive type is implicitly converted to the other type, as we will

[3] In the implementation this is actually realized by a "preprocessing" phase, during parsing, that truncates the float literals whose fractional part consists only of zeros – effectively turning them into integer literals.

$$\frac{tag(T^+) = bool}{isBool(T^+)} \qquad \frac{tag(T^+) = string}{isString(T^+)} \qquad \frac{tag(T^+) \in \{int32, int64\}}{isInt(T^+)}$$

$$\frac{tag(T^+) \in \{float32, float64\}}{isFloat(T^+)} \qquad \frac{primType(T^+) \quad tag(T^+) \neq bool}{isOrdered(T^+)}$$

$$isNumeric = Or(isInt, isFloat) \qquad \frac{}{Or(P,Q) = \lambda T^+.\ P(T^+)\ \vee\ Q(T^+)}$$

$$\frac{primType(T^+) \quad P(T^+)}{T^+ \models P}\ (\textsc{Satisfies})$$

$$OpPred = \begin{cases} + & \mapsto Or(isNumeric, isString) \\ - & \mapsto isNumeric \\ \&\& & \mapsto isBool \\ || & \mapsto isBool \\ < & \mapsto isOrdered \\ > & \mapsto isOrdered \end{cases}$$

Figure 4.4: FGe predicates

show in Section 4.1.3.5. On a side remark, note how the definitions *tag* and *fitsIn* use the notation T^+ to indicate where an undefined type might appear. This detail will be recurrent throughout the rest of this chapter.

Operations and predicates Figure 4.4 defines predicates akin to the predicates defined in the previous section, in that they essentially abstract over the *underlying* function and group under them the types that support the same operations. The function *OpPred* takes advantage of this idea, but in reverse: it specifies, for each operator, what are the predicates its operands must satisfy. As an example usage:

$$OpPred(\&\&) = isBool$$

Essentially this is an alternative way of listing which types support a given primary operation, but a mapping in this direction is useful to factor the check "operation *op* is defined for type T" – cf. rule T-BINOP in Fig. 4.7, Sec. 4.1.3.4. This check is formalized by the use of *OpPred* together with the relation $T^+ \models P$, which reads "type T^+ satisfies predicate P". For instance, the operation $<$ is defined for the type $int32$ because $OpPred(<) = isOrdered$ and $isOrdered(int32)$.

Listing 4.2: FGe assignability between named types and type literals.

```
type R   struct {} // just a dummy receiver
type Box struct {x int32}

func (this R) RetLit() struct{x int32} {
    return Box{42}
}
func (this R) RetBox() Box {
    return struct{x int32}{43}
}
```

Surely in the FG context one could opt for the simpler version – merely writing $P(T^+)$ to mean the same as $T^+ \models P$ – but this extra indirection is convenient when we consider type list constraints (Section 4.2). As we explain there, some types (type parameters) satisfy a predicate iff a list of types (from the type parameter's constraint) satisfy said predicate. Therefore we also define $\models$ here for consistency.

A consequence of using $\models$ is that we must provide a single predicate as its right hand side. But some operations naturally map to a disjunction of two predicates, as is the case for $+$: it accepts either numeric or string types. For this purpose we define the "higher order" function *Or*, which combines two predicates into a single one. We write it in lambda notation to indicate that the result is a single predicate that may be applied to a type T^+. When applied, it will in turn apply both *captured* predicates to the same type T^+ and return the disjunction of the results.
Despite both $\models$ and *Or* seem overcomplicated, Section 4.2.3.3 clarifies, through examples, the value of introducing such definitions.

4.1.3.4 Typing

Figures 4.5 and 4.7 together formalize the type system of Featherweight Go extended with the proposed features. We start by presenting Fig 4.5.

The judgments T *ok* and S *ok* simply adapt their FG counterparts to account for the new forms of types. We adjust the implements ($<:$) relation such that it only considers interface types in its range. As a consequence, now a non-interface type T^+ doesn't implement itself anymore.

Then we introduce the relation $\prec:$ as a means to formalize the Go concept of *assignability*, where $T^+ \prec: U^+$ reads "type T^+ is assignable to type U^+". This relation can be seen as a generalization of FG subtyping, in that it includes $<:$ and it is the relation that we now use to check e.g. if a method body's type *conforms* to the type declared in the method's signature (rule T-FUNC). The rule concluding $B_U \prec: T^+$ simply delegates to the *fitsIn* relation (cf. Sec. 4.1.3.3). The final two clauses are the ones who relate type literals and named types created from such literals. With these rules, the methods depicted in Listing 4.2 can now be typed.

Well-formed types $\boxed{T\ ok}$

T-PRIM
$$\frac{}{B\ ok}$$

T-UND
$$\frac{}{B_U\ ok}$$

T-NAMED
$$\frac{\textbf{type}\ t\ T \in \overline{D}}{t\ ok}$$

T-STRUCT
$$\frac{distinct(\overline{f}) \quad \overline{T\ ok}}{\textbf{struct}\ \{\overline{f\ T}\}\ ok}$$

T-INTERFACE
$$\frac{unique(\overline{S}) \quad \overline{S\ ok}}{\textbf{interface}\ \{\overline{S}\}\ ok}$$

Well-formed method specifications $\boxed{S\ ok}$

T-SPECIFICATION
$$\frac{distinct(\overline{x}) \qquad \overline{T\ ok} \qquad T\ ok}{m(\overline{x\ T})\ T\ ok}$$

Implements $\boxed{T^+ <:\ U}$

$<:_I$
$$\frac{interface(U) \quad methods(T^+) \supseteq methods(U)}{T^+ <:\ U}$$

Assignable to $\boxed{T^+ \prec: U^+}$

$$\frac{}{T^+ \prec: T^+} \qquad \frac{T^+ <:\ U}{T^+ \prec: U} \qquad \frac{fitsIn(B_U, T^+)}{B_U \prec: T^+}$$

$$\frac{underlying(t) = L}{t \prec: L} \qquad \frac{underlying(t) = L}{L \prec: t}$$

Well-formed declarations $\boxed{D\ ok}$

T-TYPE
$$\frac{T\ ok}{\textbf{type}\ t\ T\ ok}$$

T-FUNC
$$\frac{\begin{array}{c} distinct(x, \overline{x}) \\ t\ ok \qquad \overline{T\ ok} \qquad U\ ok \qquad x : t, \overline{x : t} \vdash e : T \qquad T \prec: U \end{array}}{\textbf{func}\ (x\ t)\ m(\overline{x\ T})\ U\ \{\textbf{return}\ e\}\ ok}$$

Programs $\boxed{P\ ok}$

T-PROG
$$\frac{distinct(tdecls(\overline{D})) \qquad distinct(mdecls(\overline{D})) \qquad \overline{D\ ok} \qquad \emptyset \vdash e : T^+}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{_ = e\}\ ok}$$

Figure 4.5: FG typing

Listing 4.3: Go implicit conversions, example 1

```
func RetI(b Box) I {
    return b
}
func main() {
    _ = RetI( struct{x int32}{42} ).m()
}
```

Listing 4.4: Go implicit conversions, example 2

```
type MyInt int32
func (this MyInt) m() {}

func RetI(m MyInt) I {
    return m
}
func main() {
    _ = RetI(1).m()
}
```

An important remark is that, although $<:$ can be seen as a generalization of FG subtyping, it is missing the property of *transitivity*. To see why, assume we add to the program of Listing 4.2 the following declarations:

```
type I interface { m() }
func (this Box) m() {}
```

i.e. we introduce an interface `I` that specifies a method `m()`, and we make `Box` implement that method. Now we have that both these statements hold:

$$\texttt{struct \{x int32\}} <: \texttt{Box}$$
$$\texttt{Box} <: \texttt{I}$$

yet, the following is not true:

$$\texttt{struct \{x int32\}} <: \texttt{I}$$

since a type literal can't implement any methods – and thus transitivity is broken. Note that this also happens in Go. Furthermore, notice that in Go we can still write the program of Listing 4.3, where the `RetI` function types, as expected, but its use in main might seem odd. The insight here is that upon passing a `struct{x int32}` to `RetI`, the literal is implicitly converted to a `Box`. Moreover, this is the case not only for struct literals, but also for primitive literals, as Listing 4.4 further illustrates.

The general case is, in fact, that a type conversion may be inserted – automatically, by the Go compiler – wherever $<:$ needs to be tested. In our FG implementation we modelled

$$\frac{T^+ <: U^+}{mostRestrictive(T^+, U^+) = U^+} \qquad \frac{U^+ <: T^+}{mostRestrictive(T^+, U^+) = T^+}$$

Figure 4.6: The *mostRestrictive* function.

this fact by making `AssignableTo` (the method that represents <:) return two values: a boolean, the result of the test, and a *coercion*, in the style of *coercion semantics* as defined in [35]. This coercion, which might be a no-op, is then applied to the respective term. This means that our typechecker effectively modifies the program's AST. For instance, the body of the `main` function from Listing 4.4 would be transformed into:

```
_ = RetI( MyInt(1) ).m()
```

In the formalism, we represent this fact by (conservatively) inserting type conversions, during reduction, wherever they might be needed. The next section further details this aspect.

Finally, the judgments $D\ ok$ and $P\ ok$ from Fig. 4.5 are also straightforward adjustments of the FG ones. A subtle detail that we are not formalizing – but is handled in the implementation – is that verifying that a type declaration is well formed requires more work now. In particular, to check that the declaration is not recursive. This was in part approached in FG – named structs can't refer to themselves in their fields – but it's necessary to also eliminate cases such as:

```
type A B
type B A
```

Expressions Figure 4.7 defines how types are assigned to the different forms of expressions. All the rules until T-STUPID adapt the ones from FG by integrating the new sets of types T/T^+ and the notion of assignability.

The general form of the judgment is written $\Gamma \vdash e : T^+$ (i.e. the result is an extended type) since the literals are assigned undefined types, as dictated by the rule T-PRIMLIT (cf. 4.3 and Sec. 4.1.3.3). The rules T-CONV$_{<:}$ and T-CONV$_{=}$ handle type conversions. The former covers cases where an expression is converted to a more restrictive type, such as in `MyInt64(1)` or $float32_U(1)$. The latter is applicable when the conversion simply changes the type but not the underlying value/representation of the expression, e.g. in `MyInt(int32(1))`. Note that the rules overlap for expressions involving type literals, as in `Box(struct{x int32}{42})`. In this case, an implementation might choose to apply either.

The remaining two rules show how to type binary and relational operations. They start by typing both operands and then check that both types support the given operation – this mechanism, that uses the function *OpPred* and the relation $\models$, is detailed at the end

Expressions $\boxed{\Gamma \vdash e : T^+}$

T-VAR
$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

T-CALL
$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash \overline{e : T^+} \qquad (m(\overline{x\ U})\ U) \in methods(T) \qquad \overline{T^+ <: U}}{\Gamma \vdash e.m(\overline{e}) : U}$$

T-STRUCTLIT
$$\frac{T\ ok \quad struct(T) \qquad \Gamma \vdash \overline{e : T^+} \qquad (\overline{f\ U}) = fields(T) \qquad \overline{T^+ <: U}}{\Gamma \vdash T\{\overline{e}\} : T}$$

T-FIELD
$$\frac{\Gamma \vdash e : T \qquad struct(T) \qquad (\overline{f\ U}) = fields(T)}{\Gamma \vdash e.f_i : U_i}$$

T-ASSERT$_I$
$$\frac{T\ ok \quad interface(T) \qquad \Gamma \vdash e : U \quad interface(U)}{\Gamma \vdash e.(T) : T}$$

T-ASSERT$_S$
$$\frac{T\ ok \quad \neg interface(T) \qquad \Gamma \vdash e : U \quad interface(U) \qquad T <: U}{\Gamma \vdash e.(T) : T}$$

T-STUPID
$$\boxed{\frac{T\ ok \quad \neg interface(T) \qquad \Gamma \vdash e : U \quad \neg interface(U)}{\Gamma \vdash e.(T) : T}}$$

T-PRIMLIT
$$\frac{B_U = type(\ell)}{\Gamma \vdash \ell : B_U}$$

T-CONV$_{<:}$
$$\frac{T^+\ ok \quad \Gamma \vdash e : U^+ \quad U^+ <: T^+}{\Gamma \vdash T^+(e) : T^+}$$

T-CONV$_=$
$$\frac{T\ ok \quad \Gamma \vdash e : U \quad underlying(U) = underlying(T)}{\Gamma \vdash T(e) : T}$$

T-BINOP
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1^+ \quad \Gamma \vdash e_2 : T_2^+ \\ P = OpPred(\oplus) \qquad T_1^+ \models P \qquad T_2^+ \models P \qquad U^+ = mostRestrictive(T_1^+, T_2^+)\end{array}}{\Gamma \vdash e_1 \oplus e_2 : U^+}$$

T-COMP
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1^+ \quad \Gamma \vdash e_2 : T_2^+ \\ P = OpPred(\asymp) \qquad T_1^+ \models P \qquad T_2^+ \models P \qquad _ = mostRestrictive(T_1^+, T_2^+)\end{array}}{\Gamma \vdash e_1 \asymp e_2 : bool_U}$$

Figure 4.7: FG expressions typing

of Section 4.1.3.3. Finally, the types T_1^+ and T_2^+ might be different if at least one of them is an undefined type, as in either of these cases:

```
10 - float32(4.2)
42 < MyInt(1)
```

Thus we use the partial function *mostRestrictive*, defined in Fig 4.6, that returns a result iff the types are *compatible*, i.e., if either is assignable to the other. Notice that the application

$$mostRestrictive(MyInt, int32)$$

would return no result and hence an expression such as `MyInt(2) - int32(1)` is ill typed. Finally, in the rule for binary operations, the type of the expression corresponds to the most restrictive type involved. For instance, the expression `10 - float32(4.2)` would be assigned the type $float32$. On the other hand, the Go Spec [45] determines that a comparison always returns an untyped boolean, thus the rule T-COMP ignores the result of *mostRestrictive*, exploiting it just to check the compatibility of the operands' types.

4.1.3.5 Reduction

Figure 4.8 presents the small-step reduction (or evaluation) rules for our extended version of FG. It starts by describing the syntactic forms of values and the evaluation contexts E in the usual way.

We define a judgment $e\ \ value$ that holds when an expression that has the appearance of a value has indeed finished reducing. The motivation is that reduction might encounter an expression such as `float32(1)`. Although it has the form of a value, $T(\ell)$, the literal must be converted to float32 representation before being used in float32-specific operations. We consider this representation conversion to be a step of evaluation, which is in turn captured by the rule R-CONVERT-RAW-T. The remaining two rules of the judgment $e\ \ value$ aren't really necessary but we display them for completion.

The first line of rules for the judgment $d \longrightarrow e$ are adaptations of the FG ones. They feature an important addition: the introduction of type conversions. To motivate their need, consider the FG program depicted in Listing 4.5, which essentially gathers the examples shown in the previous section in a single figure. Note that this program is well typed. Now consider what happens when we take one (small) step of evaluation over the main's body. The first expression to evaluate is the method call:

```
Dummy{}.RetI( struct{x int32}{42} )
```

If we were to reuse the following rule for method calls, as defined in FG [20]:

R-CALL

$$\frac{(x : t, \overline{x : T}).e = body(type(v).m)}{v.m(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]}$$

we would simply substitute the method's body for `struct{x int32}{42}` and return this value. This means the main body would be reduced, in one step, to the expression:

$$\text{Value} \qquad v ::= T\{\overline{v}\} \mid T(\ell) \mid \ell$$

Evaluation context	$E ::=$	Type assertion	$E.(T)$
Hole	$\square$	Type convertion	$T^{+}(E)$
Method call receiver	$E.m(\overline{e})$	Binary operation lhs	$E \oplus e$
Method call arguments	$v.m(\overline{v}, E, \overline{e})$	Binary operation rhs	$v \oplus E$
Structure	$T\{\overline{v}, E, \overline{e}\}$	Relational operation lhs	$E \asymp e$
Select	$E.f$	Relational operation rhs	$v \asymp E$

Value $\boxed{e\ \ value}$

$$\textsc{v-struct}\ \frac{struct(T) \quad \overline{e\ \ value}}{T\{\overline{e}\}\ \ value} \qquad \textsc{v-typed}\ \frac{B_U = type(\ell) \quad tag(B_U) = tag(T)}{T(\ell)\ \ value} \qquad \textsc{v-lit}\ \frac{}{\ell\ \ value}$$

Reduction $\boxed{d \longrightarrow e}$

$$\textsc{r-field}\ \frac{\overline{(f\ \ T)} = fields(T)}{T\{\overline{v}\}.f_i \longrightarrow T_i(v_i)} \qquad \textsc{r-call}\ \frac{(x : t, \overline{x : T}).e : U = body(type(v).m)}{v.m(\overline{v}) \longrightarrow U(e[x := v, \overline{x := T(v)}])}$$

$$\textsc{r-assert}\ \frac{type(v) <: T}{v.(T) \longrightarrow v} \qquad \textsc{r-context}\ \frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

$$\textsc{r-convert-raw-u}\ \frac{\ell' = rawConv(\ell, B)}{B_U(\ell) \longrightarrow \ell'} \qquad \textsc{r-convert-raw-t}\ \frac{B_U = type(\ell) \quad B' = tag(T) \quad B < B' \quad \ell' = rawConv(\ell, B')}{T(\ell) \longrightarrow T(\ell')}$$

$$\textsc{r-convert-typed}\ \frac{}{T(U(\ell)) \longrightarrow T(\ell)} \qquad \textsc{r-convert-s}\ \frac{struct(T)}{T(U\{\overline{v}\}) \longrightarrow T\{\overline{v}\}} \qquad \textsc{r-convert-i}\ \frac{interface(T)}{T(U\{\overline{v}\}) \longrightarrow U\{\overline{v}\}}$$

$$\textsc{r-binop}\ \frac{T_1^{+} = type(v_1) \quad T_2^{+} = type(v_2) \quad U^{+} = mostRestrictive(T_1^{+}, T_2^{+}) \quad \boxplus = match(\oplus, U^{+})}{v_1 \oplus v_2 \longrightarrow U^{+}(v_1) \boxplus U^{+}(v_2)}$$

$$\textsc{r-relop}\ \frac{T_1^{+} = type(v_1) \quad T_2^{+} = type(v_2) \quad U^{+} = mostRestrictive(T_1^{+}, T_2^{+}) \quad \lesseqgtr = match(\asymp, U^{+})}{v_1 \asymp v_2 \longrightarrow U^{+}(v_1) \lesseqgtr U^{+}(v_2)}$$

Figure 4.8: FGe reduction

Listing 4.5: FGe example: why conversions are needed.

```
type Dummy struct {} // dummy receiver
type I interface { m() }

type Box struct {x int32}
func (this Box) m() {}

func (this Dummy) RetI(b Box) I {
    return b
}

func main() {
    _ = Dummy{}.RetI( struct{x int32}{42} ).m()
}
```

Listing 4.6: FGe example: why conversions are needed (2).

```
type I interface { m() }

type MyInt int32
func (this MyInt32) m() {}

type BoxM struct {x MyInt}

func main() {
    _ = BoxM{42}.x.Foo()
}
```

```
struct{x int32}{42}.m()
```

Yet this expression is clearly ill-typed, as the type literal `struct{x int32}` doesn't define any methods. For this reason, we need to convert `struct{x int32}` into a `Box` before substituting it in the body of the `RetI` method. Listing 4.6 illustrates a variation of this issue for the case of field selection. The pattern is the same: if we were to use the original reduction rule

$$\text{R-FIELD}\quad \frac{(\overline{f\ T}) = \mathit{fields}(T)}{T\{\overline{v}\}.f_i \longrightarrow v_i}$$

the main body would be reduced into

```
42.m()
```

resulting again in a type error. It is evident that maintaining these rules would break type safety, as programs that are initially well typed are made ill typed by evaluating them.

The insight is that type conversions might be needed wherever assignability is tested for, since sometimes we need to distinguish the values at the type level. For instance,

we can create the struct literal `BoxM{42}` because `BoxM` takes a field of type `MyInt`, and `42` – which is typed as $int32_U$ – is assignable to `MyInt`. Yet when projecting the field we want it to behave as a `MyInt` and not as an $int32_U$.
In this formalism we solve this problem by injecting type conversions, during reduction, in every expression that might have asked for assignability. In particular: when projecting a field; when calling a method – both the arguments and the result are converted to the declared types; in binary and relational operations. Note that these directly correspond to the typing rules that test for assignability.

The next five reduction rules (prefixed by R-CONVERT) define the operational semantics of type conversions. The first two rules employ the function *rawConv*, meaning that they indeed alter the representation of the value. The rule R-CONVERT-RAW-U covers cases such as $float32_U(42)$. We view the function *rawConv* as a black box, assuming only the following: when it is applied to a literal ℓ and a *tag* B, it produces a literal ℓ' such that the application *type*(ℓ') returns B_U. For instance, *rawConv*$(1, float32)$ would produce a literal $1'$ such that *type*$(1') = tfloat_U$. Observe that, in an implementation, *rawConv* would actually change the binary representation of its argument.
Rule R-CONVERT-RAW-T is similar to the previous but it targets expressions involving defined types, as in `float64(23)` – where the literal doesn't yet have the desired representation (otherwise it would fall into V-TYPED). The difference is that this rule yields a *typed* literal, contrarily to R-CONVERT-RAW-U which produces a literal of undefined primitive type.
The three remaining conversion rules are mostly straightforward. R-CONVERT-TYPED reduces expressions like `MyInt(int32(1))`; R-CONVERT-S handles struct type conversions such as `Point(struct{x,y float64}{1.1, 2.2})` or `Polar(Point{1.1, 2.2})`. The idea behind R-CONVERT-I is mostly the same as R-ASSERT.

The final two rules specify the semantics of binary and relational operations, at a quite high level. They start by converting both operands to the same, most restrictive type. Then they refer to the function *match*, for which we assume the following semantics: given a type and an operation symbol, it returns the corresponding operation specific to that type. As an example, *match*$(+, int64)$ would return "addition of 64 bit integers". We don't formally model the low level details of these operations and instead just write either $e \boxplus e$ or $e \lesseqgtr e$ to represent them.

4.2 Additions to Featherweight Generic Go

In this section we present our additions to Featherweight Generic Go. The section follows the same overall structure as Section 4.1, where we begin by introducing the proposed features informally and then show how to realize them in the typing system. However, we don't present the reduction rules since they are essentially the same as in the previous section.

4.2.1 Type lists in interfaces

The work on Featherweight Go [20] formalized the addition of parametric polymorphism to Go, focusing on how to integrate generics in a language where subtyping is *structural*. They found that the interfaces of Go turn out to be a great fit for specifying bounds for type parameters, as interfaces simply represent a set of methods. This is a clean way of expressing which capabilities (methods) are assumed from a type parameter inside a generic function. Also, it provides great flexibility for the callers of such functions, only needing to ensure the type arguments implement the required methods. Moreover, as the type system of FGG permits mutual recursion on the bounds of type parameters, this language enjoys an expressive power comparable to that of other academic languages such as Emerald [4] or Theta [10].

However, Go disallows primitive operators to be defined in terms of methods, as they are built into the language. This means that using FGG interfaces as type parameter constraints is not enough to express e.g. a generic min function :

```
func min(type T Ordered)(x T, y T) {
    if x < y {
        return x
    } else {
        return y
    }
}
```

since we can't define `Ordered` in a way that captures the operator $<$.

The solution proposed by the Go Team is to specify such operators indirectly. More concretely, the idea is to explicitly list the types that may satisfy a constraint and consider that the operators specified by that constraint are the intersection of the operators that each of the listed types supports. For instance, if we consider just the operations listed in Section 4.1.3.1, then the list of types: $int32$, $float32$, $string$ indirectly specifies the operators $+$, $<$ and $>$, since both $int32$ and $float32$ support the operators $+, -, <, >$, while $string$ only supports $+, <, >$. An immediate observation we can make is that these type lists represent both a union of types and an intersection of operators, since a type list with e.g. $int32$ and $string$ dictates that the type parameter it bounds might be instantiated by

either $int32$ or $string$, but the operations the type parameter supports are the intersection of the operations supported by $int32$ and $string$.

Section 4.2.3 details how we added type lists to FGG interfaces and explores some of its consequences.

4.2.2 Type definitions revisited

An interesting case that arises when we extend interfaces with type lists is that it becomes possible to have constraints that specify both a type list and a set of methods. I.e., it's allowed to write interfaces like the following (the syntax is detailed in Sec. 4.2.3.1):

```
type StringerNumber interface {
    type int32, int64, float32, float64
    String() string
}
```

This interface would constrain a type parameter such that it can only be instantiated by a type that (1) has underlying type $int32, int64, float32$ or $float64$ and (2) implements a method `String() string`. In general, such constraints can only be satisfied by types such as `MyInt`, since primitive types can't implement methods. As we've seen in Section 4.1.2, this requires generalizing the FG type declarations so that they may refer to an arbitrary type as the source type. This generalization has two further consequences in the context of FGG, in particular on the process of monomorphisation, which we detail in Section 4.3.

4.2.3 Extending Featherweight Generic Go

4.2.3.1 Syntax

Figure 4.2.3.1 describes the syntax of FGG enriched with type lists in interfaces. It follows the idea employed in Fig. 3.2 in that it highlights what has changed in comparison with the syntax described in Fig 4.1. In particular, type and method declarations may now take type parameters, which are bound by interface types. As such, utilizing a parameterized type or method requires passing it type arguments ϕ to instantiate the parameters. The most relevant modification is that interface types used as type parameter constraints may optionally specify a *type list*.

We write type lists following the previous proposal's syntax, i.e. a type list has the form:

$$\textbf{type } T_1, T_2, \ldots$$

The new proposal [4] that is based on the concept of type sets uses a rather different syntax that doesn't include the **type** keyword, but for our purposes the semantics are fundamentally the same. This proposal also defines that, in a type list, writing a type T represents

[4] https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md

Field name	f	Expression	$d, e ::=$
Method name	m	Primitive Literal	ℓ
Variable name	x	Variable	x
Primitive type	$B ::= bool \mid string \mid int32 \mid \ldots$	Structure literal	$T\{\overline{e}\}$
		Select	$e.f$
Undefined primitive type	B_U	Method call	$e.m(\phi)(\overline{e})$
Type name	$t, u \quad (t, u \neq B)$	Type assertion	$e.(T)$
Type Literal	$L ::=$	Type conversion	$T^+(e)$
Structure	$\textbf{struct}\ \{\overline{f\ T}\}$	Binary operation	$e \oplus e$
Interface	$\textbf{interface}\ \{\textbf{type}\ \overline{T}; \overline{S}\}$	Comparison	$e \asymp e$
Type parameter	α	Literal	$\ell ::=$
Type	$T, U ::= B \mid t(\overline{T}) \mid L \mid \alpha$	Bool literal	ℓ_B
Extended Types	$T^+ ::= T \mid B_U$	Int literal	ℓ_I
Type formal	$\Phi, \Psi ::= \textbf{type}\ \overline{\alpha\ T}$	Float literal	ℓ_F
Type actual	$\phi, \psi ::= \overline{T}$	String literal	ℓ_S
Method signature	$M ::= (\Psi)(\overline{x\ T})\ T$		
Method specification	$S ::= mM$		
Declaration	$D ::=$		
Type declaration	$\textbf{type}\ t(\Phi)\ T$		
Method declaration	$\textbf{func}\ (x\ t(\Phi))\ mM\ \{\textbf{return}\ e\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{_ = e\}$		

Figure 4.9: Featherweight Generic Go (extended) syntax

"just the type T" (exact matching), and it adds a new syntactic construct called an *approximation element*, written $\sim T$, to mean "all the types whose underlying type is T". The exact matching targets a context where type lists may also represent a form of sum types, but we don't consider such possibility here. Hence we don't distinguish T from $\sim T$ and assume T stands for all the types whose underlying type is T.

4.2.3.2 Auxiliary functions

Figure 4.10 presents auxiliary definitions. The top of the figure repeats the functions defined in FG [20] that return a mapping η from type formals Φ to actuals ϕ. Then we adjust the function *underlying* to take type parameters into consideration. The main difference is that for (parameterized) named types this function also performs a substitution. As an example, given the declaration

```
type Pair(type X, Y any) struct {x X, y Y}
```

we have that the underlying type of the instantiation `Pair(int32, int32)` is the type `struct{x int32, y int32}`.

Next we define the function $tlist_\Delta$ [5] which takes a type and returns the type list it represents. We designate that the type list of a non-interface type is the singleton set

[5] The beginning of the next section explains the meaning of subscript Δ.

$$\frac{(\textbf{type}\ \overline{\alpha\ T}) = \Phi \qquad \eta = (\overline{\alpha := T})}{(\Phi := \overline{T}) = \eta} \qquad \frac{(\textbf{type}\ \overline{\alpha\ T}) = \Phi \qquad \eta = (\Phi := \phi) \qquad \Delta \vdash (\overline{\alpha <: T})[\eta]}{(\Phi :=_\Delta \phi) = \eta}$$

$$\frac{}{underlying(L) = L} \qquad \frac{}{underlying(B) = B} \qquad \frac{}{underlying(B_U) = B_U} \qquad \frac{}{underlying(\alpha) = \alpha}$$

$$\frac{\textbf{type}\ t(\Phi)\ T \in \overline{D} \quad U = underlying(T) \quad \eta = (\Phi := \phi)}{underlying(t(\phi)) = U[\eta]}$$

$$\frac{\neg interface(T) \quad T \neq \alpha}{tlist_\Delta(T) = \{T\}} \qquad \frac{}{tlist_\Delta(\alpha) = tlist_\Delta(bounds_\Delta(\alpha))}$$

$$\frac{underlying(T) = \textbf{interface}\ \{\textbf{type}\ \overline{T};\ \overline{S}\} \quad \overline{U} = \bigcup tlist_\Delta(T_i)}{tlist_\Delta(T) = \overline{U}}$$

$$\frac{}{T \lhd \emptyset} \qquad \frac{(T \in \overline{U})\ \vee\ (underlying(T) \in \overline{U})}{T \lhd \overline{U}} \qquad \frac{\bigwedge T_i \lhd \overline{U}}{\overline{T} \lhd \overline{U}}$$

Figure 4.10: FGGe auxiliary functions

constituted by that type. This, together with the judgment $T \lhd \overline{U}$ (explained below) allows us to define the implements relation with a single rule, as we show in Section 4.2.3.4. For an interface type that specifies a type list $\overline{T}$, its $tlist_\Delta$ is the union of each T_i's $tlist_\Delta$. It must be a *flattened* union instead of just $\overline{T}$, since we can have, for example:

```
type Int    interface {type int32, int64}
type Float  interface {type float32, float64}
type Number interface {type Int, Float}
```

i.e., an interface's type list might contain other interfaces that list types themselves. Notice that if an interface I specifies no type list, then $tlist_\Delta(I) = \emptyset$.

At last we define the judgment $T \lhd \overline{U}$, which reads "type T is represented in the set of types $\overline{U}$". Essentially, it holds if either the type T or its underlying type are present in $\overline{U}$. The axiom $T \lhd \emptyset$ is an ad-hoc addition necessary to uniformize the treatment of interfaces whether they specify a type list or not. This aspect is explained in Section 4.2.3.4.
The final rule sort of overloads this judgment, as its left hand side is not a type T but a set of types $\overline{T}$. It dictates that the whole set $\overline{T}$ is represented in $\overline{U}$ iff each type T_i is represented in $\overline{U}$. Thus we have, for instance:

$$\{int32, float32\} \lhd \{int32, float32, string\}$$
$$\{MyInt32, MyFloat32\} \lhd \{int32, float32\}$$

$$\frac{primType(T^+) \quad B = tag(T^+)}{fitsIn_\Delta(B_U, T^+)}$$

$$\frac{primType(T^+) \quad isNumeric(T^+) \quad isNumeric(B_U) \qquad B \leq tag(T^+)}{fitsIn_\Delta(B_U, T^+)}$$

$$\frac{fitsIn_\Delta(B_U, bounds_\Delta(\alpha)\,)}{fitsIn_\Delta(B_U, \alpha)} \qquad \frac{interface(T) \quad \overline{U} = tlist_\Delta(T) \quad |\overline{U}| > 0 \quad \bigwedge fitsIn_\Delta(B_U, U_i)}{fitsIn_\Delta(B_U, T)}$$

$$\frac{primType(T^+) \quad P(T^+)}{T^+ \models_\Delta P} \qquad \frac{bounds_\Delta(\alpha) \models_\Delta P}{\alpha \models_\Delta P}$$

$$\frac{interface(T) \quad \overline{U} = tlist_\Delta(T) \quad |\overline{U}| > 0 \quad \bigwedge U_i \models_\Delta P}{T \models_\Delta P}$$

Figure 4.11: FGGe auxiliary functions for primitives

The relation $\lhd$ can be thought of as an enhanced subset check $\subseteq$ that also checks for the presence of underlying types on its right hand side. This is useful for establishing the implements relation between two interfaces that both specify type lists, as Section 4.2.3.4 demonstrates.

4.2.3.3 Auxiliary functions for primitives

Figure 4.11 shows the adjusted versions of the relations *fitsIn* and $\models$ (Section 4.1.3.3). Both are added a subscript Δ to indicate that the type environment is also an input to the rule. They both refer to the function $bounds_\Delta$ which was defined in FG [20]. This function essentially retrieves the bound of a type parameter α from the type environment Δ.

The new rules for $fitsIn_\Delta$ are fairly straightforward: an undefined type B_U is compatible with a type parameter α iff it is compatible with every type listed in α's bound. As an example, given the two constraints:

```
type Number      interface {type int32, int64, float64}
type NumberAndS interface {type int32, int64, float64, string}
```

we have that the type B_U is compatible with `Number` but not with `NumberAndS`. Also, observe that an undefined type is never compatible with a constraint that specifies no type list. As we saw in Section 4.1.3.3, the *fitsIn* relation is responsible for determining which types can be mixed with an undefined type in a primitive operation. Therefore, the new definition $fitsIn_\Delta$ now allows one to write the following method:

```
func (this Dummy) Inc(type T Number)(x T) T {
    return x + 1
}
```

Yet note that to type the method Inc we must also verify that + is defined for the type T. For this purpose, the relation $\models_\Delta$ is adjusted in a very similar sense: it defines that a type parameter α satisfies a predicate P iff every type listed in α's bound satisfies P. Recalling Section 4.1.3.3, we have that:

$$OpPred(+) = Or(isNumeric, isString)$$

One can intuitively conclude that all the types listed by Number satisfy this predicate – as they are all numbers – from which it follows that + is defined for the type parameter T of Inc. Now consider the following method:

```
func (this Dummy) Double(type T NumberAndS)(x T) T {
    return x + x
}
```

it's also intuitive to conclude that NumberAndS supports +. But notice the usefulness of defining Or as an "higher order" function: checking that NumberAndS supports + amounts to verifying that each type it lists is either numeric or a string. Without a way to compose two predicates, we could be inclined to check something like (where U_i represents each of the types listed by NumberAndS):

$$\left(\bigwedge U_i \models_\Delta isNumeric\right) \quad \bigvee \quad \left(\bigwedge U_i \models_\Delta isString\right)$$

but both sides of the disjunction would be false, as the types are neither all numeric nor all strings. This would lead us to conclude NumberAndS doesn't support +, which is clearly wrong.

As a final remark, note that since both final clauses of $fitsIn_\Delta$ and $\models_\Delta$ recurse on the respective relation in their last premises, both definitions work even for type list constraints embedded in other constraints.

4.2.3.4 Typing

Figure 4.12 presents the adjusted judgment $\Delta \vdash T\ \ ok$ and the implements and assignableTo relations. We don't display the judgments $S/D/P\ ok$ as they mostly repeat the rules described in Section 4.1.3.4. The typing rules for expressions are also not depicted since they essentially merge the ones defined in Fig 4.7 with the rules for FGG [20]. Instead we focus on the main consequences of adding type lists to interfaces.

We redefine the judgment $T\ \ ok$ so that it also takes the type environment as input. The first line of rules are simple adjustments to take this into account. Rules T-PARAM and T-NAMED are the same as in FGG and are displayed just for completion. Rule T-INTERFACE now checks the listed types are all well formed and it prohibits the use of

Well-formed types $\boxed{\Delta \vdash T \; ok}$

T-PRIM
$$\frac{}{\emptyset \vdash B \; ok}$$

T-UND
$$\frac{}{\emptyset \vdash B_U \; ok}$$

T-STRUCT
$$\frac{distinct(\overline{f}) \quad \Delta \vdash \overline{T \;\; ok}}{\Delta \vdash \textbf{struct} \; \{\overline{f \;\; T}\} \; ok}$$

T-PARAM
$$\frac{(\alpha : T) \in \Delta}{\Delta \vdash \alpha \; ok}$$

T-NAMED
$$\frac{\Delta \vdash \overline{T} \; ok \quad (\textbf{type} \; t(\Phi) \; T) \in \overline{D} \quad \eta = (\Phi :=_\Delta \overline{T})}{\Delta \vdash t(\overline{T}) \; ok}$$

T-INTERFACE
$$\frac{\overline{T \neq \alpha} \quad \Delta \vdash \overline{T \; ok} \quad unique(\overline{S}) \quad \Delta \vdash \overline{S \;\; ok}}{\Delta \vdash \textbf{interface} \; \{\textbf{type} \; \overline{T}; \; \overline{S}\} \; ok}$$

Implements $\boxed{\Delta \vdash T^+ <: U}$

$<:_I$
$$\frac{interface(U) \quad methods_\Delta(T^+) \supseteq methods_\Delta(U) \quad tlist_\Delta(T^+) \lhd tlist_\Delta(U)}{\Delta \vdash T^+ <: U}$$

Assignable to $\boxed{\Delta \vdash T^+ \prec: U^+}$

$$\frac{}{\Delta \vdash T^+ \prec: T^+} \qquad \frac{\Delta \vdash T^+ <: U}{\Delta \vdash T^+ \prec: U} \qquad \frac{fitsIn_\Delta(B_U, T^+)}{\Delta \vdash B_U \prec: T^+}$$

$$\frac{underlying(t) = L}{\Delta \vdash t \prec: L} \qquad \frac{underlying(t) = L}{\Delta \vdash L \prec: t}$$

Figure 4.12: FGGe (sub)typing

stand-alone type parameters in a type list. The reason is that there is no way for an outside type to be *represented* (Section 4.2.3.2) by such a type, since we assume that each type parameter α is unique.

The adjusted implements relation reflects the major consequence of adding type lists to interfaces. Now there are two conditions necessary for a type T^+ to implement an interface type U: (1) T^+ implements the methods specified by U, and (2) the $tlist_\Delta$ that T^+ stands for is represented in the $tlist_\Delta$ of U. By referring to the auxiliary definitions $tlist_\Delta$ and $\lhd$, we are able to neatly condense all the cases of implements in a single rule. For example, if `I` is an interface that only specifies methods:

```
type I interface { m1(); m2(); ... }
```

then a type T implements that interface simply by implementing those methods, as expected, since $tlist_\Delta(I) = \emptyset$ and the definition of $\lhd$ states that a type is always *represented* in an empty set. Conversely, if an interface `C` just specifies a type list $\overline{T}$:

```
type C interface { type T̄ }
```

then a non-interface type T implements `C` (or *satisfies* the constraint `C`) if the list $\overline{T}$ contains either T or its underlying type. Recall that if an interface type has no type list, then its $tlist_\Delta$ is the empty set, which means it wouldn't be represented in a set $\overline{T}$ – and thus an interface with no type list can never satisfy a type list constraint.

Finally, the most interesting case arises when both types are interfaces that specify type lists. As an example, if we have the following two types:

```
type C1 interface { type int32, float32 }
type C2 interface { type int32, float32, string }
```

Then the definition allows us to conclude $C1 <:_\Delta C2$ (recall the end of Section 4.2.3.2). This can be justified intuitively from two different views.
First, `C2` supports a superset of the types supported by `C1`, hence there is no way for an instantiation of `C1` to "surprise" `C2`.
Secondly, code that uses a type parameter α bound by `C2` can only use operations defined for all three of $int32$, $float32$ and $string$. Thus, instantiating α with either $int32$ or $float32$ (or a type defined over them) will always be safe, since α might only use a subset of the operations supported by $int32$ and $float32$. To make our point clearer, consider that we (informally) define a function Ops that, when applied to a type list constraint, returns the set of primitive operations it supports. Hence we would have that:

$$Ops(C1) = \{+, -, <, >\}$$
$$Ops(C2) = \{+, <, >\}$$

This allows us to conclude that $C1 <:_\Delta C2$ because $Ops(C1) \supseteq Ops(C2)$, which is exactly the same idea already used to decide the implements relation between sets of methods.

Finally, the assignable relation is defined by exactly the same set of rules. A type parameter is only assignable to itself and thus falls under the rule $T^+ \prec: T^+$. A type parameter is, at the same time, considered a type different from all the others, and thus we never have that e.g. $int32 \prec: \alpha$. This means that a method such as the following is not allowed (recall the typing of binary operations from Section 4.1.3.4):

```
func (this Dummy) AddInt32(type T Number)(x T, y int32) T {
    return x + y
}
```

The only types flexible enough to be mixed with (assignable to) type parameters are the undefined types. This case is handled just by modifying the function *fitsIn*, as we showed in the previous section.

Notice the distinction between (1) mixing type parameters and primitive types and (2) instantiating type parameters with primitive types. While the former asks for assignability of either $\alpha <:_\Delta int32$ or $int32 <:_\Delta \alpha$, the latter requires $int32 <:_\Delta bounds_\Delta(\alpha)$ - which would redirect to $int32 <:_\Delta bounds_\Delta(\alpha)$. This is why we can't add values of types $int32$ and T in the previous example, yet we would be allowed to instantiate `AddInt32`'s type parameter with $int32$ (if `AddInt32` was well typed).

4.3 Adjusting the monomorphisation algorithm

In this section we explore how generalizing type declarations impacts the monomorphisation process. We begin with an high level review of the formal algorithm as defined originally [20]. Then, Section 4.3.2 delineates two (loose) cases that the original algorithm isn't prepared to handle. At last, in Sections 4.3.3 and 4.3.4, we show how to adapt monomorphisation so as to account for both of these issues.

4.3.1 Monomorphisation in Featherweight (Generic) Go

The work on Featherweight Go [20] defines precise semantics for generic FGG code by specifying a formal translation of FGG programs into their non-generic FG counterparts. The translation is based on the technique of monomorphisation, which means that each instantiation of a polymorphic method/type gives rise to a different, type-specific (monomorphic) version of the generic code. For example, given the FGG type declaration:

```
type List(type a Any) interface { ... }
```

the two instantiations `List(bool)` and `List(string)` translate to two FG types `List<bool>` and `List<string>` [6], along with their corresponding type declarations:

```
type List<bool>   interface {/* a/bool */}
type List<string> interface {/* a/string */}
```

This monomorphisation algorithm encompasses two phases. First, it collects a set of type and method instantiations from a FGG program. Then, it generates the equivalent (specialized) FG program, following the instance set computed in the first phase.

4.3.1.1 Instance collection

The instance collection phase is formalized by a judgment $P \blacktriangleright \Omega$. Here, Ω (and ω) range over instance sets, which contain elements of *closed* type τ or pairs of a type with a method and its type arguments, $\tau.m(\psi)$ [7]. The judgment holds if Ω is the instance set for the program P, and is defined by the following rule:

I-PROG

$$\frac{\emptyset; \emptyset \vdash e \blacktriangleright \omega \qquad \Omega = \lim_{n \to \infty} G^n_{\emptyset}(\omega)}{\textbf{package main}; \ \overline{D} \ \textbf{func main}() \ \{_ = e\} \blacktriangleright \Omega}$$

This rule is, in turn, specified in terms of two auxiliary definitions. First, the judgment $\Delta; \Gamma \vdash e \blacktriangleright \omega$, which holds under conditions similar to the previous – if ω is the instance set for expression e, given environments Δ and Γ. In essence, it recursively traverses an expression's AST and collects the instantiations that may be found at each node. For

[6] Angle brackets and commas "<,>" are assumed to be part of FG identifiers, for convenience [20].

[7] FG makes a clear distinction between first class types (τ) and type literals (T) – c.f. Fig. 3.2, Section 3.2.3. We adopt the same convention in this review section.

Listing 4.7: Example FGG program to be monomorphised.

```
type Pred interface {
    Test(type a Number)(x a) bool
}

type Pos struct {}
func (p Pos) Test(type a Number)(x a) bool {
    return x > 0
}

type Selector struct {}
func (s Selector) SelectI(p Pred, v, d int32) int32 {
    if p.Test(int32)(v) {
        return v
    } else {
        return d
    }
}

func main() {
    _ = Selector{}.SelectI(Pos{}, -5, 0)
}
```

instance, upon reaching a method-call node, i.e., an expression of the form $e.m(\psi)(\overline{e})$, it applies the rule:

$$\textsc{I-call}\quad \frac{\Delta;\Gamma \vdash e : \tau \quad \Delta;\Gamma \vdash e \blacktriangleright \omega \quad \overline{\Delta;\Gamma \vdash e \blacktriangleright \omega}}{\Delta;\Gamma \vdash e.m(\psi)(\overline{e}) \blacktriangleright \{\tau, \tau.m(\psi)\} \cup \omega \cup \overline{\omega}}$$

That is, it records the type instance τ of the receiver together with the method instantiation $\tau.m(\psi)$, and proceeds inductively on the expressions of the receiver and the arguments.

The second auxiliary definition is the function G, whose fixed point corresponds to the final instance set Ω of the program. Initially, this function takes the instance set ω collected from the top-level expression ($_ = e$), and then it is repeatedly applied until its result contains all the type and method instantiations entailed by ω.
The original definition of G is depicted in Fig. 4.14 (grayed out), although the reader is advised to consult [20] for a detailed explanation of each auxiliary function. In short, the purpose of G is to find all the type and method instantiations necessary to monomorphise declarations (*F-closure*, *M-closure*) as well as to preserve the <: relation (*I-closure*, *S-closure*).

As an example, consider the program of Listing 4.7. It defines abstract predicates over numbers, followed by a particular one that tests if a number is positive. The predicates are then used in the method `SelectI` (specialized in integers, for the sake of simplicity), which

returns the first argument `v` if it satisfies the predicate, or a default value `d` otherwise. Starting from `main`'s body, the first premise of I-PROG yields:

$$\emptyset; \emptyset \vdash \texttt{Selector\{\}.SelectI(Pos\{\}, -5, 0)} \blacktriangleright \{\texttt{Selector}, \texttt{Selector.SelectI()}, \texttt{Pos}\}$$

Then, initializing $\omega_0 = \{\texttt{Selector}, \texttt{Selector.SelectI()}, \texttt{Pos}\}$, G is applied until it returns an ω_i such that $G_\Delta(\omega_i) = \omega_i$. From the first iteration we have:

$$\omega_1 = G_\emptyset(\omega_0) = \omega_0 \cup \{ \texttt{Pred}, \textbf{int32}, \texttt{Selector.SelectI()}, \texttt{Pred}, \texttt{Pred.Test(}\textbf{int32}\texttt{)()} \}$$

Where `Pred` and **`int`** result from *M-closure*, while the remaining instances are obtained from *S-closure*. The repeator `Selector.SelectI()` stems from the fact that `Selector` implements itself; `Pred` and `Pred.Test(int)()` constitute the instance set for the body of `Selector.SelectI()` – note how the function *S-closure* is the responsible for navigating method bodies.

The second iteration reaches a fixed point of G with $\omega_2 = \omega_1 \cup \{ \texttt{Pos.Test(}\textbf{int}\texttt{)()} \}$, where `Pos.Test(int)()` is collected via *S-closure* since `Pos <: Pred`, and thus we have that $\Omega = \omega_2 = G_\emptyset(\omega_2)$ is the instance set for this program.

4.3.1.2 Generation of monomorphic code

After computing the instance set Ω, monomorphising a program essentially consists of generating specialized declarations for each instantiation $t(\phi)$ or $\tau.m(\psi)$ in Ω. Besides, it is also necessary to replace the occurrences of such instantiations by suitable FG identifiers (e.g. `List(bool)` $\mapsto$ `List<bool>` or `Pred.Test(int32)` $\mapsto$ `Pred.Test<int32>`) [8]. The formal algorithm further includes some nuances, such as fabricating dummy methods to ensure the preservation of subtyping, but those are not relevant for this presentation and were already sorted out [20].

A parameterized FGG declaration D translates to zero or more specialized declarations $\mathscr{D}$, one for each instantiation of its type parameters. This is captured by the judgment $\Omega \vdash D \mapsto \mathscr{D}$, where the set of monomorphised declarations $\mathscr{D}$ is determined by the instances contained in Ω.

Declarations may introduce either types or methods; for illustrative purposes, we focus on the case of type declarations. These are translated according to the following rule:

$$\textsc{m-type}\quad \frac{\mathscr{D} = \left\{ \textbf{type } t^\dagger\ T^\dagger \;\middle|\; \begin{array}{l} t(\phi) \in \Omega,\ \eta = (\Phi := \phi),\ \mu = \{m(\psi) \mid t(\phi).m(\psi) \in \Omega\}, \\ \eta \vdash t(\phi) \mapsto t^\dagger, \quad \eta; \mu \vdash T \mapsto T^\dagger \end{array} \right\}}{\Omega \vdash \textbf{type } t(\Phi)\ T \mapsto \mathscr{D}}$$

That is, for each instantiation $t(\phi)$, we record the substitution η that produces this particular type and select the set of corresponding method instances μ from Ω. The pair

[8] Again, we assume the parenthesis are punctuation, contrarily to the angle brackets which are part of identifiers.

Listing 4.8: Selector example: monomorphised (FG) program.

```
type Top struct {}

type Pred interface {
    Test<int32>(x int32) bool
    Test<0>() Top
}

type Pos struct {}
func (p Pos) Test<int32>(x int32) bool {
    return x > 0
}
func (p Pos) Test<0>() Top {
    return Top{}
}

type Selector struct {}
func (s Selector) SelectI(p Pred, v, d int32) int32 {
    if p.Test<int32>(v) {
        return v
    } else {
        return d
    }
}
func (s Selector) SelectI<1>() Top {
    return Top{}
}

func main() {
    _ = Selector{}.SelectI(Pos{}, -5, 0)
}
```

$\eta; \mu$ derived from this $t(\phi)$ (and Ω) then guides the generation of the monomorphised identifier $t^\dagger$ and source type $T^\dagger$, forming a monomorphic declaration **type** $t^\dagger$ $T^\dagger$.

Consider, for example, the translation of the declaration for the interface type `Pred` in Listing 4.7, which specifies a generic method. The whole monomorphised program can be consulted in Listing 4.8, where dummy methods are grayed out. Ignoring dummy method signatures, the declaration essentially translates to:

```
type Pred interface {
    Test<int32>(x int32) bool
}
```

since the type `Pred` is not parameterized, and the only pertinent method instance in Ω is `Pred.Test(int32)`. This example illustrates a particular case of the type-literal-monomorphisation judgment $\eta; \mu \vdash T \mapsto T^\dagger$, which is characterized in Fig. 4.13. Namely,

Type literal $\boxed{\eta; \mu \vdash T \mapsto T^{\dagger}}$

M-STRUCT
$$\frac{\eta \vdash \overline{\tau \mapsto \tau^{\dagger}}}{\eta; \mu \vdash \textbf{struct}\{\overline{f\ \tau}\} \mapsto \textbf{struct}\{\overline{f\ \tau^{\dagger}}\}}$$

M-INTERFACE
$$\frac{\eta; \mu \vdash \overline{S \mapsto \mathcal{S}}}{\eta; \mu \vdash \textbf{interface}\{\overline{S}\} \mapsto \textbf{interface}\{\bigcup \overline{\mathcal{S}}\}}$$

Figure 4.13: Monomorphisation of type literals (original).

it illustrates interface monomorphisation, and how it essentially consists of generating specialized versions for each signature S the interface specifies. As shown, this process is guided by a method instance set μ, which is aggregated while translating a type declaration and passed as input to this judgment.

4.3.2 Monomorphisation and Type Declarations

The algorithm as presented in the previous section suffers from two shortcomings when we enrich F(G)G with general type declarations.

The first emerges from the ability to refer to arbitrary types in type declarations. For instance, consider the following declarations:

```
type Pair(type T1, T2 any) struct {
    x T1; y T2
}

type PairInt Pair(int32, int32)
```

As the instance collection phase assumes that the source of a type declaration must be a type literal, it only navigates type declarations with the intent of finding new instantiations within struct fields (function *F-closure*, Fig. 4.14) or method signatures (*M-closure*). Yet, as the example demonstrates, now the source type can itself be a (possibly unique) type instantiation. It is necessary to take this case into account since otherwise we would risk generating the FG declaration:

```
type PairInt Pair<int32,int32>
```

without ever generating a specialized declaration for its source type `Pair<int32,int32>`.

The second issue stems from the possibility of having variables of anonymous interface type. As an example, consider we change the program of Listing 4.7 such that a `Selector` struct now includes a predicate as its field. Moreover, assume we declare the type of this field as the interface literal from which `Pred` is created. The necessary modifications are illustrated in Listing 4.9 (unaltered lines are grayed out).

Among other things, monomorphising this program requires translating the type declaration for `Selector` into the following:

Listing 4.9: Selector that includes a predicate as its field.

```
type Pred interface {
    Test(type a Number)(x a) bool
}

type Pos struct {}
func (p Pos) Test(type a Number)(x a) bool {
    return x > 0
}

type Selector struct {
    p interface{ Test(type a Number)(x a) bool }
}
func (this Selector) SelectI(v, d int32) int32 {
    if this.p.Test(int32)(v) {
        return v
    } else {
        return d
    }
}

func main() {
    _ = Selector{Pos{}}.SelectI(-5, 0)
}
```

```
type Selector struct {
    p interface{
        Test<int32>(x int32) bool
        Test<0>() Top
    }
}
```

Furthermore, although `Pred` is not used in this program, its declaration should also be monomorphised into:

```
type Pred interface {
    Test<int32>(x int32) bool
    Test<0>() Top
}
```

so as to ensure the subtype relation is preserved during the translation – note that, in the FGG program of Listing 4.9, `Pred` implements the anonymous interface and vice-versa.

However, there are two reasons why the original algorithm doesn't produce the desired result. First, the instance collection phase only considers first class (τ) types and thus will never record an instantiation of the form:

$$\textbf{interface}\{\ldots\}\,.\,m(\psi)$$

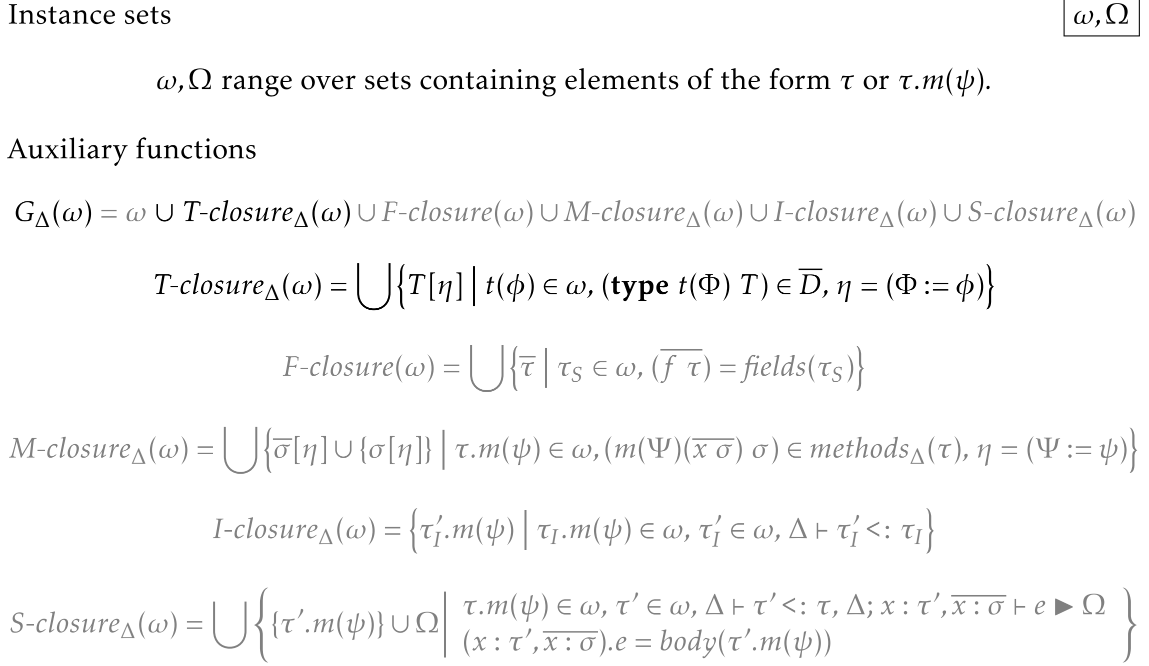


Figure 4.14: Computing instance sets.

but that is mostly a detail solvable by promoting type literals to first class, as discussed in the next section. Secondly, as shown at the end of the previous section, the type-literal-monomorphisation judgment $\eta; \mu \vdash T \mapsto T^{\dagger}$ expects a method instance set μ as input, assuming it is assembled during the translation of a type declaration. But, by definition, an anonymous interface has no declaration and thus no μ is collected before translating it – i.e., the input μ will always be empty in these cases, regardless of the instances in Ω.

4.3.3 Handling instantiated types in type declarations

We solve the first issue by extending the definition of G with an extra auxiliary function *T-closure*, highlighted in Figure 4.14 [9]. This function just collects the source type that corresponds to a closed type instantiation $t(\phi)$, applying to it the same substitution that originated $t(\phi)$. Note that, despite its simple definition, *T-closure* can even handle (arbitrarily long) chains of declarations of the form:

```
type Pair(type T1, T2 any) struct {
    x T1; y T2
}

type PairI1(type T2 any) Pair(int32, T2)
type PairInt PairI1(int32)
```

[9] N.B.: using τ_I and τ_S (as original FG paper) to avoid polluting the rules with checks of the form *struct*(τ)/*interface*(τ) (i.e. $\tau_S \implies$ *struct*(τ_S))

since the function G is applied iteratively: from `PairInt`, *T-closure* can find the instantiation `PairI1(int32)`; in the next iteration, `PairI1(int32)` will be part of ω and, from that instance, *T-closure* can then derive `Pair(int32,int32)`.
As a side remark, observe that by iteratively applying the substitution to the source type, this function ensures that if a type instantiation $t(\phi)$ occurs in Ω, so will its underlying type. For example, if Ω contains the instance `List(int32)`, then *T-closure* will expand it with the instance:

```
interface { Map(type b Any)(f Function(int32, b)) List(b) }
```

where the first type argument of `Function` is also instantiated to `int32`. In formal terms, we represent this guarantee by the following lemma:

Lemma 4.3.1. *If $t(\phi) \in \Omega$ then underlying$(t(\phi)) \in \Omega$.*

Proof outline. We haven't proved that the addition of *T-closure* to G's definition maintains the properties of the *nomono* predicate [20]. As such, we assume Ω may be either finite or infinite and thus this is a proof by coinduction [26]. The proof considers only the part of Ω built by *T-closure*, but is rigorous enough for our purposes.

The set Ω is defined to be the largest set $\mathbb{S}$ satisfying the following property:

Property 1. *If $T \in \mathbb{S}$, then:*

i. T is a base type B or a type literal L, or

*ii. If T is a named type $t(\phi)$, then there exists a declaration (**type** $t(\Phi)\ T'$) and a substitution $\eta = (\Phi := \phi)$ such that $T'[\eta] \in \mathbb{S}$.*

That is, we can think of Ω as the greatest fixpoint of the operator P:

$$P(\mathbb{S}) = \left\{ T \;\middle|\; (T = B \vee T = L) \vee \left(T = t(\phi) \implies \exists (\textbf{type}\ t(\Phi)\ T' \in \overline{D}, \eta = \Phi := \phi).\ T'[\eta] \in \mathbb{S} \right) \right\}$$

Now, we want to show that if $t(\phi) \in \Omega$ then *underlying*$(t(\phi)) \in \Omega$. Suppose

$$t(\phi) \in \Omega$$

Being a named type, $t(\phi)$ must be defined by some type declaration **type** $t(\Phi)\ T'$, where T' is either a base type, a type literal or another named type. If $T' = B$ or $T' = L$, then by Property 1(i) we have that the underlying type of $t(\phi)$ is in Ω. Otherwise, if T' is another named type $s(\phi')$, then *underlying*$(t(\phi))$ = *underlying*$(s(\phi'))$.
But, by the coinduction hypothesis we have that *underlying*$(s(\phi')) \in \Omega$ and thus we can also conclude *underlying*$(t(\phi)) \in \Omega$.

□

The importance of this detail will be made clear below, in the context of interface monomorphisation.

Type monomorphisation $\boxed{\eta \vdash T \mapsto T^{\dagger}}$

M-TNAMED
$$\frac{t^{\dagger} = \langle t(\overline{T[\eta]})\rangle}{\eta \vdash t(\overline{T}) \mapsto t^{\dagger}}$$

M-TPARAM
$$\frac{T = \alpha[\eta] \quad \eta \vdash T \mapsto T^{\dagger}}{\eta \vdash \alpha \mapsto T^{\dagger}}$$

M-STRUCT
$$\frac{\overline{\eta \vdash T \mapsto T^{\dagger}}}{\eta \vdash \mathbf{struct}\{\overline{f\ T}\} \mapsto \mathbf{struct}\{\overline{f\ T^{\dagger}}\}}$$

M-PRIM
$$\frac{}{\vdash B \mapsto B}$$

M-UND
$$\frac{}{\vdash B_U \mapsto B_U}$$

M-INTERFACE
$$\frac{\overline{\eta; \mu \vdash S \mapsto \mathcal{S}}}{\eta; \mu \vdash \mathbf{interface}\{\overline{S}\} \mapsto \mathbf{interface}\{\bigcup \overline{\mathcal{S}}\}}$$

M-INTERFACE
$$\frac{L_I = \mathbf{interface}\{\overline{S}[\eta]\} \quad \mu = \{m(\psi) \mid L_I.m(\psi) \in \Omega\} \quad \overline{\eta; \mu \vdash S \mapsto \mathcal{S}}}{\eta \vdash \mathbf{interface}\{\overline{S}\} \mapsto \mathbf{interface}\{\bigcup \overline{\mathcal{S}}\}}$$

Figure 4.15: Monomorphisation of types.

4.3.4 Monomorphising interface types

The second issue is solved by adapting (instead of extending) some definitions, related both to instance collection and to the generation of monomorphic code. We begin by promoting type literals to first class types, which requires two modifications.
First, we merge the type-monomorphisation and type-literal-monomorphisation judgments into a single one, which we define with the rules presented in Figure 4.15. The first three rules are mostly identical to the original ones [20]; the following two rules are straightforward. Rule M-INTERFACE is detailed below.

Secondly, we rewrite rules and auxiliary functions by replacing uses of τ by T, where T represents the set of all possible types as defined in Figure 4.9 (Section 4.2.3.1). For instance, the function *I-closure* is rewritten into: [10]

$$\textit{I-closure}_{\Delta}(\omega) = \left\{T_I'.m(\psi) \;\middle|\; T_I.m(\psi) \in \omega,\, T_I' \in \omega,\, \Delta \vdash T_I' <: T_I\right\}$$

These two adjustments suffice to fix the instance collection phase, as it will now record instantiations of the form **interface**$\{\dots\}.m(\psi)$. Moreover, together with Lemma 4.3.1 we now have the following guarantee:

Lemma 4.3.2. *Let L_I = underlying$(t(\phi))$ for some $t(\phi) \in \Omega$ such that interface$(t(\phi))$. If $t(\phi).m(\psi) \in \Omega$ then $L_I.m(\psi) \in \Omega$.*

Proof outline. Follows from Lemma 4.3.1 and the definition of *I-closure*. □

After promoting type literals, we still need to revise how interfaces are monomorphised. In particular, we can't assume that the method instance set μ is collected while

[10] Again, T_I represents a type such that *interface*(T_I) holds.

monomorphising type declarations, since anonymous interfaces have no associated declaration. As such, we propose an update to the rule M-INTERFACE, illustrated in Figure 4.15. The key idea is that instead of expecting μ as input, we now construct it within the premises, using just the interface literal to select the appropriate method instances.

Intuitively, from Lemma 4.3.2 it follows that, for named types $t(\phi)$, assembling μ "inside" this rule is equivalent to the previous process, which constructed μ while generating a specialized type declaration for $t(\phi)$. As such, the translation of type declarations can now be expressed by the simpler rule:

$$\frac{\mathcal{D} = \left\{ \textbf{type}\ t^\dagger\ T^\dagger \ \middle|\ t(\phi) \in \Omega, \eta = (\Phi := \phi),\ \eta \vdash t(\phi) \mapsto t^\dagger, \eta \vdash T \mapsto T^\dagger \right\}}{\Omega \vdash \textbf{type}\ t(\Phi)\ T \mapsto \mathcal{D}}\ \text{M-TYPE}$$

4.3.4.1 Example

In order to illustrate the ramifications of these adjustments, we explore an example in depth. Recall the program of Listing 4.9, where the `Selector` struct has a predicate field represented by an anonymous interface. From the top-level expression we get that:

$$\omega_0 = \{\, \texttt{Selector}, \texttt{Selector.SelectI}, \texttt{Pos} \,\}$$

Then, applying G to this set yields:

$$\begin{aligned} G(\omega_0) \quad &= \omega_0 \\ &\cup \{\, \textbf{struct}\texttt{\{ p }\textbf{interface}\texttt{\{Test(}\textbf{type}\texttt{ a Number)(x a) }\textbf{bool}\texttt{\} \}}, \textbf{struct}\texttt{\{\}} \,\} \\ &\cup \{\, \textbf{interface}\texttt{\{Test(}\textbf{type}\texttt{ a Number)(x a) }\textbf{bool}\texttt{\}} \,\} \\ &\cup \{\, \textbf{int32} \,\} \\ &\cup \{\, \textbf{interface}\texttt{\{...\}}, \textbf{interface}\texttt{\{...\}.Test(}\textbf{int32}\texttt{)()} \,\} \end{aligned}$$

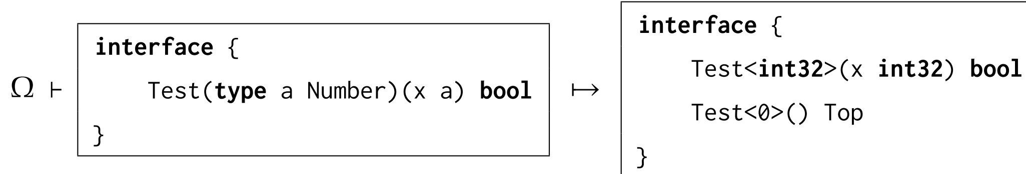
Where the first set is obtained from $\textit{T-closure}_\emptyset(\omega_0)$ (underlying types of `Selector` and `Pos`), the second is obtained from $\textit{F-closure}_\emptyset(\omega_0)$ (type of `Selector`'s field), the third from $\textit{M-closure}_\emptyset(\omega_0)$ and the last one from $\textit{S-closure}_\emptyset(\omega_0)$. This last set already includes an instance of the form **interface**{...}.$m(\psi)$, as desired. Note that in this case we write **interface**{...} to abbreviate the only interface literal in question.
Next we apply G again, reaching a fixed point:

$$G(G(\omega_0)) = G(\omega_0) \cup \{\, \texttt{Pos.Test(}\textbf{int32}\texttt{)()} \,\}$$

Obtaining `Pos.Test(`**int32**`)()` via *S-closure*. The resulting set resembles the Ω we got at the end of Section 4.3.1.1 – replacing `Pred` by **interface**{...} – except for the occurrence of (underlying) type literals. Note that even though `Pred <:` **interface**{...} holds, no method instantiations are collected for `Pred` via *I-closure*, since `Pred` is never added to any ω. Nevertheless, as we show next, this is not problematic as we still collect the instantiations pertaining to `Pred`'s underlying type.

Given the correct instance set, the goal we set at the end of Section 4.3.2 was to generate appropriate, specialized declarations both for `Selector`'s field and for `Pred`. We start

with `Selector`'s field, whose type is declared as `interface{Test(type a Number)(x a) bool}`. We monomorphise this type by applying the rule M-INTERFACE of Figure 4.15. There are no free type variables occurring in this type, hence the substitution has no effect (indeed η is empty) and the L_I obtained is equal to the initial literal. As shown above, Ω contains a method instantiation for this L_I – `Test(int32)()` – and thus from this Ω we can derive:



allowing us to produce the appropriate FG declaration:

```
type Selector struct {
    p interface{
        Test<int32>(x int32) bool
        Test<0>() Top
    }
}
```

The case for `Pred` is analogous. In order to generate a monomorphised declaration, we monomorphise the identifier and the source type (rule M-TYPE, p. 62). Since the source type is an interface literal exactly equal to the one seen above, it is monomorphised into the same FG type. This type is then plugged into the specialized declaration, yielding the desired:

```
type Pred interface{
    Test<int32>(x int32) bool
    Test<0>() Top
}
```

The insight is that by assembling μ inside M-INTERFACE, using only the literal to select the appropriate method instantiations, this rule now becomes "self-contained" or, in other words, modular. This means that for equal (under η) type literals we can just reuse the same rule instance, independently of the context where we are monomorphising said literals, since the corresponding method instance set μ will be the same.

4.3.5 Note on the remaining features

Although we extended F(G)G with other features besides generalized type definitions, this is the only one with non-trivial implications on the monomorphisation process.

Primitive types/values are kept unaltered in FG; primitive operations only require extending the judgment $\Delta; \Gamma \vdash e \blacktriangleright \omega$ so that it recurses in each of the operand expressions.

A type list is only used for constraining type parameters, but the types targeted by monomorphisation are the ones resulting from instantiating such parameters. Indeed, type parameter bounds in general are not relevant for the process of monomorphisation and could be discarded after the type checking phase.

One possible optimization would be to generate, a priori, an instantiation of a type parameter for each type listed in its constraint's type list – with the intent of avoiding part of the instance collection phase. However, the gains from such optimization are not clear and it would result in certain code bloat, particularly for programs where such type parameters are not instantiated "exhaustively". On top of this, the point of the FG formalism is not to model this kind of optimizations, thus we have not investigated further.

5 Exploring Type Argument Inference

This chapter describes our work on type argument inference in the context of Featherweight Generic Go extended with the features presented in the previous chapter. We begin by framing the specific version of the type inference problem that we propose to investigate, first stating it in general terms and then narrowing down the desired characteristics of the solution.

After that, Section 5.2 gives some intuitions on the inner workings of the algorithm we devised to solve this problem, ending with an high-level outline of our solution. At the core of the algorithm stands the constraint resolution mechanism; as such, we devote Section 5.3 to the in-depth exploration of this mechanism.

Finally, the algorithm we devised is a relatively simple adaptation of techniques borrowed from Hindley-Milner as well as bidirectional systems. Yet, most of these techniques were developed targeting idealized languages and thus didn't have to deal with the peculiarities of an industrial language like Go. This indicates there is some novelty to our work, but it also means our solution has some limitations, particularly in the presence of untyped constants. Section 5.4 discusses these limitations as well as possible solutions for them, and finishes by comparing our solution to works that solve similar problems through the use of bidirectional techniques.

The whole chapter follows a rather informal style of presentation, contrasting with the previous one, as the formalization of this system is out of the scope of this thesis and is instead left for future work.

5.1 Problem Statement

The problem we propose to solve consists of inferring correct type arguments for *unannotated* instantiations of generic types and methods. For example, given the program

Listing 5.1: Example program in $\text{FGG}_{()}$.

```
type Dummy struct {}

type Box(type a Any) struct {
    value a
}

type Unboxer struct{}

func (u Unboxer) Unbox(type a Any)(box Box(a)) a {
    return box.value
}

func main() {
    _ = Unboxer{}.Unbox()(Box(){ Dummy{} })
}
```

in Listing 5.1 – where the type arguments were omitted in the top-level expression –, the goal is to figure out types `T1` and `T2` to fill the instantiations:

```
Unboxer{}.Unbox(T1)(Box(T2){ Dummy{} })
```

such that the resulting expression is well typed. Intuitively, in this case we would infer that `T1` = `T2` = `Dummy`. Note that we could also have inferred the type arguments to be `interface{}`, the supertype of all types. However that would make us lose static information, assigning the type `interface{}` to the result of the call `Unboxer.Unbox()`, instead of the more precise `Dummy`. As such, we don't just want types that make the expressions well-typed – we want the *smallest* types that do so [5, 36].

More precisely, we now consider a new external language $\text{FGG}_{()}$ similar to FGG, but where expressions contain no explicit type instantiations – i.e., method calls have the form $e.m()(\overline{e})$ instead of $e.m(\psi)(\overline{e})$ and named types occur in expressions (e.g. struct literals) in the form $t()$ instead of $t(\phi)$. Note that declarations and method signatures are still fully type-annotated in $\text{FGG}_{()}$.
Our goal is then to design a type inference algorithm that maps $\text{FGG}_{()}$ expressions to FGG ones by reconstructing the elided type information [32]. Ideally [1], the algorithm should enjoy the following property: for any FGG expression e and its $\text{FGG}_{()}$ counterpart $e_{()}$, if e type-checks, then so must the mapping for $e_{()}$, where annotations are replaced by inferred types. That is, no program that was previously typeable will now cease to be so [27].

The above constitutes the general problem statement. In addition to that, we impose a restriction on the solution: the algorithm should be able to infer types *locally* [32, 36], i.e., inferring the type of an expression requires only inferring types for its subexpressions. We believe this is a reasonable requirement since top-level type and method declarations are fully type-annotated (type parameters and types of bound variables); furthermore,

[1] Section 5.4 discusses some examples of why this is not the case.

FGG doesn't permit local declarations within method bodies. This effectively means that we know, a priori, the "type schemes" [9] for every generic type and method.

In summary, our aim is to study – and devise a prototype solution for – the problem of local type inference in the context of FGG, which exhibits the following characteristics:

- declarations and method signatures are fully annotated;
- method bodies cannot contain local declarations;
- expressions never contain explicit type instantiations;
- subtyping is structural (and not nominal);
- all the "type constructors" are nonvariant.

5.2 The base algorithm

This section presents the base algorithm we devised to solve the problem described in the previous section. It is a relatively simple algorithm, derived from the application of well established inference techniques [9, 36, 35] to this particular context. We go over an easy example that showcases most of the fundamental ideas, and end with the algorithm outline by generalizing from the example.

Consider again the top-level expression of the program in Listing 5.1, but now focus just on the subexpression:

```
Box(){ Dummy{} }
```

As we've seen, the goal is to find a type `T` such that the resulting expression

```
Box(T){ Dummy{} }
```

is well-typed. The algorithm proceeds as follows. Upon reaching an expression of this form (a struct literal), it starts by *instantiating*, with fresh type variables, all the type parameters found in the struct type's declaration. The declaration for `Box` refers a single type parameter `a`, so we generate a new type variable, say `X1`, and record its bound `Any`. This type variable is then used to instantiate the struct type, yielding `Box(X1)` – whose underlying type is `struct{ value X1 }`. Note that, in general, we may generate zero (for non-parameterized types) or more fresh variables here. This step mimics the technique used in implementations of Hindley-Milner systems [35], assuming the type declaration provides the "generalized" type scheme.

So now we have an expression of the form $e_{(\overline{X})}$, namely

```
Box(X1){ Dummy{} }
```

as well as a type environment $\Delta = \{$ `X1 <: Any` $\}$, and want to resolve a suitable value for the type variable `X1`. The key idea is that we will take advantage of the type information provided by the term arguments to *constrain* the possible values that `X1` might take.
As such, we begin by *inferring* the smallest type for the argument value `Dummy{}`, obtaining the type `Dummy`. Following the base type-checking rules, the expression above is well-typed iff the argument's type, `Dummy`, is a subtype of (assignable to) the corresponding struct field's type, `X1`. Here, we represent this by a *subtype constraint* of the form

$$\texttt{Dummy} \leq_\Delta \texttt{X1}$$

which must be solved by finding a substitution η, mapping type variables to types, such that the *assignable to* judgment

$$\Delta \vdash \texttt{Dummy} \prec: \texttt{X1}[\eta]$$

holds. It's worth noting that some constraints can't be solved – in this case, type inference fails. In theory, this happens when there is no type annotation that would make the expression well-typed.

Resuming the example, the constraint is straightforwardly satisfied by the substitution $\eta = \{ \texttt{X1} \mapsto \texttt{Dummy} \}$. The inference process then ends by applying this substitution to the instantiated expression `Box(X1){ Dummy{} }`, yielding the well-typed FGG expression

```
Box(Dummy){ Dummy{} }
```

This example already illustrates the essence of the algorithm. We focused on struct literals, but the procedure is identical for method calls. The remaining forms of expressions don't directly involve type/method instantiations, so we just recurse on their subexpressions – while also verifying that the basic conditions for well-typedness are met. For instance, the expression:

```
Box(){ Dummy{} }.value
```

is at the top-level a field selection, thus if there are type arguments to infer then it must be in its struct-literal subexpression. However, after inferring the correct type arguments for it, we still need to verify that this struct indeed contains a `value` field to ensure the whole expression is well-typed.

As a recap, we now give an high-level description of the algorithm. Focusing on struct literals and method calls, i.e. expressions of the form $t()\{\overline{e}\}$ and $e.m()(\overline{e})$, and ignoring basic verifications as the one above, the algorithm proceeds as follows:

1. Instantiate the corresponding declaration's type parameters with fresh type variables, $X_0 \ldots X_n$, obtaining an instantiated type $t(X_0,\ldots,X_n)$ or method $e.m(X_0,\ldots,X_n)$ and the corresponding expression (of the form $e_{(\overline{X})}$).

2. From the instantiated type/method, extract a set of formal parameters $\overline{P}$ involving the fresh variables, corresponding to the struct's fields' types or to the method's parameters' types.

3. Infer types $\overline{A}$ for the argument expressions $\overline{e}$.

4. Generate a set of subtype constraints $\overline{C}$, each of the form $A_i \leq_\Delta P_i$, relating the arguments' types to the formal parameters' types.

5. Solve the set of constraints $\overline{C}$, one by one, obtaining a final substitution η. If any constraint is not satisfiable, then fail.

6. Apply η to the instantiated expression $e_{(\overline{X})}$, obtaining the final, well-typed FGG expression.

Yet this description doesn't tell the whole story. In fact, the backbone of a type inference system is the constraint resolution mechanism [39], which we have left unspecified so far. Therefore, this is the subject of the next section.

5.3 Solving a set of subtype constraints

Although the constraint shown in the previous section was trivially solvable by making the type variable equal to the type that was constraining it (i.e., by the substitution `X1` $\mapsto$ `Dummy`), resolving constraints is not so straightforward in the general case. This is mainly due to two reasons:

- a constraint might relate more elaborate types
- there may exist more than one constraint referencing the same type variable

The following two sections expand on each of these issues, detailing our solutions and suggesting possible improvements which were not included in the prototype implementation.

5.3.1 Nontrivial constraints

In general, subtype constraints relate not only types to type variables, but may also relate types to named types instantiated with type variables. Listing 5.2 illustrates a program that leads to the generation of such constraints, where the relevant inference targets are underlined. This program is an adaptation of the `Map` examples already seen in previous chapters, slightly tweaked in order to showcase the two possible forms of nontrivial constraints. The difference is that `MapDup` maps each list element to two duplicated, consecutive elements in the result.

The goal is to infer that the type argument to both underlined instantiations is the type parameter `b`, "inherited" from the context (i.e. the method signature). Note that the type

Listing 5.2: Lists without type arguments

```
type Func(type a Any, b Any) interface {
    Apply(x a) b
}

type List(type a Any) interface {
    MapDup(type b Any)(f Func(a, b)) List(b)
}
type Nil(type a Any) struct {}
type Cons(type a Any) struct {
    head a
    tail List(a)
}

func (xs Nil(type a Any)) MapDup(type b Any)(f Func(a,b)) List(b) {
    return Nil(){}
}

func (xs Cons(type a Any)) MapDup(type b Any)(f Func(a,b)) List(b){
    return Cons(){
        f.Apply(xs.head),
        Cons(){f.Apply(xs.head), xs.tail.MapDup()(f)}
    }
}

func main() {...}
```

parameter b must not be instantiated while typing the body of the method MapDup, since there it actually represents a type and not a variable that must be substituted. Therefore we keep a strict separation between type parameters and type variables while typing methods' bodies; this idea is formalized in e.g. Section 6 of [31], by including the type parameters in a V set that is input to the inference judgments.

As we show in Sections 5.3.1.1 and 5.3.1.2, we conclude that the omitted type argument is b in two different ways, one for each of the underlined instantiations. We first explore the case for the inner expression, `xs.tail.MapDup()(f)`, and then look at the outer one.

5.3.1.1 Introducing equality constraints

According to the algorithm described previously, we infer the type argument for `MapDup()` as follows. We start by generating a fresh variable `X3` (the two enclosing `Cons` instantiations get `X1` and `X2`) and instantiate the expression with it, yielding:

```
xs.tail.MapDup(X3)(f)
```

where the method `MapDup(X3)` gets the signature:

```
MapDup(X3)(f Func(a, X3)) List(X3)
```

Next, we infer the type `Func(a,b)` for the argument `f` by looking in the context. After this, we generate the constraint relating the argument's type to the formal parameter of the instance `MapDup(X3)`:

$$\texttt{Func(a,b)} \leq_\Delta \texttt{Func(a,X3)}$$

and we reach a nontrivial subtype constraint, whose right hand side is a named type instantiated with the variable `X3`.

The solution for this sort of constraints derives from the following observation: subtyping in FGG is *invariant*. This means that for any named type t and two other types S and T, we have that $t(S) <: t(T)$ iff $S = T$. As such, we now introduce *equality constraints* as a way to further constrain type variables that occur in such positions. Thus solving the subtype constraint above entails the creation and resolution of the two equations:

$$\{\, \texttt{a} =_\Delta \texttt{a ; b} =_\Delta \texttt{X3} \,\}$$

which we solve by standard first-order unification [35], producing the final substitution $\{\, \texttt{X3} \mapsto \texttt{b} \,\}$, as intended.

5.3.1.2 Solving constraints by unifying method signatures

We now look at the outer expression. Again, according to the algorithm we start by generating a fresh variable `X1` and instantiate the expression with it, obtaining:

```
Cons(X1){ f.Apply(xs.head), Cons(){f.Apply(xs.head), xs.tail.MapDup()(f)} }
```

where the type `Cons(X1)` has underlying type **`struct`**`{head X1; tail List(X1)}`. Then we go onto inferring types for the arguments. The first argument `f.Apply(xs.head)` is straightforwardly assigned the type `b`, since there are no type arguments to reconstruct.
Regarding the second argument, we've seen in the previous section that the missing type argument in `MapDup()` is inferred to be `b`, thus we may consider that the expression is actually the reconstructed:

```
Cons(){f.Apply(xs.head), xs.tail.MapDup(b)(f)}
```

which is eventually inferred to be of type `Cons(b)` after solving the inequalities

$$\{\, \texttt{b} \leq_\Delta \texttt{X2 ; List(b)} \leq_\Delta \texttt{List(X2)} \,\}$$

using the same mechanism seen above [2]. Now, onto the interesting part. After inferring types for the inner expressions, we generate the following constraints:

$$\{\, \texttt{b} \leq_\Delta \texttt{X1 ; Cons(b)} \leq_\Delta \texttt{List(X1)} \,\}$$

[2] Indeed this is the reason we had to use `MapDup` instead of the simpler `Map` for this exposition.

where the second illustrates a new case: the right hand side is a named type instantiated with a type variable, yet it is not the same name as in the left hand side.

Following the typing rules, for any two named types – say S and T – we have that $S <: T$ if and only if T is an interface type and S implements that interface. Moreover, S implements the interface iff S defines all the methods specified in it, with exactly-matching signatures (modulo variable names). Therefore, we solve constraints such as the one above by *unifying the method signatures*. More concretely, we have (where we renamed the method's type parameter to avoid confusion):

$$methods_{\Delta}(\texttt{Cons(b)}) = \{\, \texttt{MapDup(}\mathbf{type}\texttt{ b' Any)(f Func(b, b')) List(b')} \,\}$$
$$methods_{\Delta}(\texttt{List(X1)}) = \{\, \texttt{MapDup(}\mathbf{type}\texttt{ b' Any)(f Func(X1, b')) List(b')} \,\}$$

from which we extract the equality constraints:

$$\{\, \texttt{Any} =_{\Delta} \texttt{Any} \,;\, \texttt{Func(b, b')} =_{\Delta} \texttt{Func(X1, b')} \,;\, \texttt{List(b')} =_{\Delta} \texttt{List(b')} \,\}$$

that is, we unify type parameter bounds, formal parameter's types and return types. Note that due to the *invariance* of subtyping, solving an equality constraint of the form

$$\texttt{Func(b, b')} =_{\Delta} \texttt{Func(X1, b')}$$

i.e., one that relates two instances of a named type, is the same as if it was a subtype constraint instead. As such, we solve this equation by reducing it to the simpler (as we've seen in the previous section):

$$\{\, \texttt{b} =_{\Delta} \texttt{X1} \,;\, \texttt{b'} =_{\Delta} \texttt{b'} \,\}$$

which we resolve with the substitution $\{\, \texttt{X1} \mapsto \texttt{b} \,\}$, allowing us to conclude the missing type argument is `b` once again, but this time we got there by unifying method signatures.

5.3.2 Multiple constraints over a single variable

The second issue mentioned at the beginning of Section 5.3 is that there may exist multiple constraints referring to the same type variable. When this is the case, solving these constraints usually requires finding a *best* type that satisfies all the constraints simultaneously. As an example, consider the program of Listing 5.3, which defines a method `Choose` that randomly chooses between its two input values [3]. Following our algorithm, inferring the type argument for the call to `Choose()` would lead us to generate the two inequalities:

$$\{\, \texttt{Dummy} \leq_{\Delta} \texttt{X1} \,;\, \texttt{Any} \leq_{\Delta} \texttt{X1} \,\}$$

both constraining the variable `X1`. The typical solution found in the literature [11, 35, 36, 39] is to merge both these constraints into a single one:

[3]We assume the existence of a random number generator as well as local variable declarations for ease of presentation.

Listing 5.3: Randomly choosing between two input values.

```
type Chooser struct {}

func (c Chooser) Choose(type a Any)(v1, v2 a) a {
    if rand.Float64() < 0.5 {
        return v1
    } else {
        return v2
    }
}

func main() {
    var x Any = 5
    _ = Chooser{}.Choose()(Dummy{}, x)
}
```

$$\texttt{Dummy} \sqcup \texttt{Any} \leq_\Delta \texttt{X1}$$

where the operator $\sqcup$, called a *join*, takes the least upper bound of its two arguments – i.e., the smallest type that is a supertype of both `Dummy` and `Any`. In this case, we have that $\texttt{Dummy} \sqcup \texttt{Any} = \texttt{Any}$. Equivalently, we could also consider that when two constraints give rise to two substitutions, e.g.

$$\{\ \texttt{X1} \mapsto \texttt{Dummy}\ ;\ \texttt{X1} \mapsto \texttt{Any}\ \}$$

we can merge both substitutions into the single

$$\{\ \texttt{X1} \mapsto \texttt{Dummy} \sqcup \texttt{Any}\ \}$$

which can also be simplified into $\{\ \texttt{X1} \mapsto \texttt{Any}\ \}$.

As a side remark, note that in a language like FGG (and Go), where the subtyping relation is structural or "implicit", calculating a join $T_1 \sqcup T_2$ essentially amounts to determining which type is a supertype of the other. We can't assume the existence of a type J different from T_1 and T_2 that is a supertype of both without defaulting to $J =$ **interface**{}. This contrasts with e.g. what happens in GJ [5], where one can traverse the subtype hierarchy looking for a common ancestor, concluding for instance that

$$\texttt{Integer} \sqcup \texttt{Float} = \texttt{Number}$$

In our case, this means that when inferring the type argument for an expression like [4]:

```
Chooser{}.Choose()(int32(1), float32(1.5))
```

we would conclude that the omitted type argument is the less satisfactory **interface**{}. It's worth nothing, however, that if the formal type parameter `a` in `Choose()` had a more

[4] We explicitly provide types for the numeric literals to avoid addressing untyped constants, which are somewhat problematic in the context of type inference. Section 5.4.1 discusses the matter in detail.

Listing 5.4: Randomly choosing between two input Lists.

```
func (c Chooser) ChooseL(type a Any)(l1, l2 List(a)) List(a) {
    if rand.Float64() < 0.5 {
        return l1
    } else {
        return l2
    }
}

func main() {
    var x Any = 5
    var l1 List(Dummy) = Cons(){Dummy{}, Nil(){}}
    var l2 List(Any)   = Cons(){x, Nil(){}}

    _ = Recv{}.ChooseL()(l1, l2)
}
```

restrictive bound such as `Number` (cf. p. 48) instead of `Any`, then inference would fail since the only common supertype of **`int32`** and **`float32`** – the empty interface – doesn't implement the bound `Number`.

Returning to the main subject: even though we can merge two substitutions by taking the join of the two types – following the conventional solution – this generally happens in the context of purely functional languages where type constructors are *covariant* by default. Yet, this is not the case in FGG and we've recently found a counter-example due to [3] that shows this solution is still unsound. Listing 5.4 illustrates this case, where `ChooseL` now chooses between two lists, and is called in `main` with two arguments of type `List(Dummy)` and `List(Any)`. The point is that inferring the type argument for `ChooseL()` would lead to generating the constraints:

$$\{\, \texttt{List(Dummy)} \leq_\Delta \texttt{List(X1)} \;;\; \texttt{List(Any)} \leq_\Delta \texttt{List(X1)} \,\}$$

which would eventually simplify into:

$$\{\, \texttt{Dummy} =_\Delta \texttt{X1} \;;\; \texttt{Any} =_\Delta \texttt{X1} \,\}$$

originating again the substitutions

$$\{\, \texttt{X1} \mapsto \texttt{Dummy} \;;\; \texttt{X1} \mapsto \texttt{Any} \,\}$$

However, if we were to merge these substitutions into $\{\, \texttt{X1} \mapsto \texttt{Dummy} \sqcup \texttt{Any} \,\}$ and consequently into $\{\, \texttt{X1} \mapsto \texttt{Any} \,\}$, we would conclude that the missing type argument for `ChooseL()` is the type `Any`. But the instantiation `ChooseL(Any)` is effectively a method that receives arguments of type `List(Any)`, therefore passing it an argument of type `List(Dummy)` is violating the typing rules!

The insight is that we need to distinguish whether a substitution originates from a subtype or from an equality constraint, so as to identify which substitutions may be

merged. We follow the solution devised in [3] for an equivalent problem, by marking the target types of substitutions accordingly. For instance, from the constraints

$$\{\ \texttt{Dummy} =_{\Delta} \texttt{X1}\ ;\ \texttt{Any} =_{\Delta} \texttt{X1}\ \}$$

we extract the marked substitutions

$$\{\ \texttt{X1} \mapsto \texttt{Dummy}^{=}\ ;\ \texttt{X1} \mapsto \texttt{Any}^{=}\ \}$$

both indicating that they can only be merged with substitutions mapping to the same type. As they mention different types, it's not possible to reconciliate them and thus inference fails. On the other hand, the constraints seen at the beginning of the section

$$\{\ \texttt{Dummy} \leq_{\Delta} \texttt{X1}\ ;\ \texttt{Any} \leq_{\Delta} \texttt{X1}\ \}$$

produce the substitutions

$$\{\ \texttt{X1} \mapsto \texttt{Dummy}^{<:}\ ;\ \texttt{X1} \mapsto \texttt{Any}^{<:}\ \}$$

signaling they may be merged with substitutions mapping to types to which `Dummy` (resp. `Any`) can be converted. By definition both types can be converted to $\texttt{Dummy} \sqcup \texttt{Any}$ and hence these substitutions may safely be merged. We may even merge substitutions with different marks, for instance two substitutions

$$\{\ \texttt{X1} \mapsto \texttt{Dummy}^{<:}\ ;\ \texttt{X1} \mapsto \texttt{Any}^{=}\ \}$$

combine into the single, more restrictive $\{\ \texttt{X1} \mapsto \texttt{Any}^{=}\ \}$. However, note that the similar pair

$$\{\ \texttt{X1} \mapsto \texttt{Any}^{<:}\ ;\ \texttt{X1} \mapsto \texttt{Dummy}^{=}\ \}$$

contains two incompatible substitutions since `Any` cannot be converted to `Dummy` and thus leads to failure. For more details and examples, see Section 4.1 of [3].

5.4 Discussion and Related Work

The present chapter described our algorithm for inferring type arguments for instantiations of generic methods and types. We adopt a rather informal style of presentation, starting with an high-level outline of the algorithm and then expand on the constraint solving mechanism that stands at its core. We show how to settle trivial constraints and how to leverage the distinctive subtyping of FGG to guide the resolution of nontrivial ones. The latter case invariably reduces to solving sets of equality constraints, either to unify type arguments or method signatures, taking advantage of the ubiquitous *nonvariance* of FGG (and Go). As subtype constraints may originate equality constraints, but equality constraints never go back to subtype ones, we have strong reasons to believe our constraint solver always terminates.

Despite the simplicity and apparent good behavior of our solution, it still has some limitations. In particular, the algorithm rests on two fundamental assumptions that do not hold for every $\text{FGG}_{()}$program that "should be typable", which means the algorithm does not enjoy the property enunciated in Section 5.1.
First, we assume that every expression has an *unique manifest type*[36], meaning that we can infer or *synthesize* an unique, smallest type for any form of expression regardless of where it occurs. However, neither FGG nor Go enjoy such property due to the particularities of untyped constants; Section 5.4.1 elaborates on this aspect and its consequences.

The second assumption, which enables such a simple constraint solving mechanism, is that constraints may only contain type variables on their right hand side. This is equivalent to saying that the inference procedure always returns a *solved* expression – i.e. one where all the type variables were already substituted away – since we recursively infer types for the argument expression at step 3. of the algorithm (p. 69) before generating the constraints in the next step. However, this is not the case for every program, as some (valid) expressions contain no arguments that allow us to constrain their type variables. In practical implementations the solution is to disable inference for such cases and instead require explicit type annotations. Nevertheless, our intent is to consider a context without such annotations, thus we explore some possible solutions in Section 5.4.2.

At last, we end this chapter by framing our solution within the body of research aiming to solve problems similar to ours by resorting to bidirectional techniques.

5.4.1 Inferring types for untyped constants

As we hinted above, the treatment of untyped constants is somewhat problematic in the context of type inference. The reason is that a literal such as `1` may be assigned any of the following types: $int32$, $int64$, $float32$, `MyInt`, etc., depending on the context. In the previous chapter, we handled this by assigning such literals an *undefined* type $int32_U$, which can posteriorly be converted to any of the types listed above. However, when inferring types for expressions we need a more definitive answer, as we show below. For an example, consider the $\text{FGG}_{()}$ program in Listing 5.5, which maps the identity function for 64-bit integers over a singleton list containing the literal `1`. Intuitively, this should be typable – indeed the FGG version is, by annotating every instantiation with **`int64`**.

Now consider what happens when we try to type this expression using the inference algorithm. Starting by the `Cons` expression, we would instantiate it:

```
Cons(X1){1, Nil(){}}
```

and infer the type $int32_U$ for `1`. For now, ignore how `Nil` is typed – we explore that in the next section. By solving the constraints we would reconstruct this expression as

$$\texttt{Cons(}int32_U\texttt{)\{1, Nil(}int32_U\texttt{)\{\}\}}$$

But now, inferring an argument for the `Map` call, i.e., for the expression

Listing 5.5: Lists without type arguments

```
type IdInt64 struct {}

func (this IdInt64) Apply(x int64) int64 {
    return x
}

func main() {
    _ = Cons(){1, Nil(){}}.Map()(IdInt64{})
}
```

$$\texttt{Cons(}int32_U\texttt{)\{...\}.Map()(IdInt64\{\})}$$

entails the resolution of the constraint

$$\texttt{IdInt64} \leq_\Delta \texttt{Func(}int32_U\texttt{, X2)}$$

and consequently of:

$$\{\ \textbf{int64} =_\Delta int32_U\ ;\ \textbf{int64} =_\Delta \texttt{X2}\ \}$$

where the first constraint is unsatisfiable – it relates two different types and neither mentions a type variable – and thus we deem the whole expression ill-typed. The problem is that inferring the type $int32_U$ as the argument to a type instantiation is pointless: as subtyping is invariant, a `List(`$int32_U$`)` cannot be used wherever either e.g. a `List(int32)` or a `List(int64)` is expected. Moreover, there is no FGG equivalent for such instantiation, as the programmer cannot express undefined types in source code.

As shown, the idea of undefined types does not blend well with inference, therefore we now examine some solutions. The candidates are:

- Relax invariance by internally allowing "some covariance", such that $t(S) <: t(T)$ as long as S is an undefined type and $S <: T$.
- Associate a default type to primitive literals.
- Devise a non local inference algorithm that assigns to a literal a type variable instead of an undefined type, that will later be solved depending on its context.

The first solution is promptly rejected as the primary goal of the FGG formalism is to model the Go language, an imperative language where mutability is pervasive – which does not combine well with covariance. The second solution is the simplest, although it is also rather unsatisfactory. For example, if we associate the type **`int32`** with the literal `1`, the program of Listing 5.5 would still not be typable, as it would lead to the unsatisfiable constraint **`int32`** $=_\Delta$ **`int64`**. Moreover, it would mean that innocuous expressions such as `1 + 1.5` would also be ill-typed, since we can't mix different types.

The third solution seems to be the most complete, in the sense it would accept the most programs, yet it is also the most difficult to implement. As a side remark, the solution described in the "Type Parameter Proposal" [5] is a mixture of the second and third solutions: the method is non-local since they perform two passes over an expression, where they ignore untyped constants in the first pass and hope "in some cases later arguments can determine the type of an untyped constant". If not, then they default to the predefined type in the second pass.

We finish by noting that this problem doesn't generally occur in other industrial languages with a form of local inference (Java, C++, C$^\sharp$) because they allow decorating literals with suffixes like `l` and `f`, making the type of such literals unequivocal.

5.4.2 Typing the empty list

The other limitation of our algorithm concerns the typing of expressions that don't convey enough information to infer their missing type arguments. The canonical example is the empty list, `Nil(){}`: we know it is missing a type argument because its declaration is parameterized, yet the struct has no field types that allow us to constrain the instantiated type variable. In general, the necessary type information can only be retrieved from the surrounding context. For instance, in a non-empty list, the other elements of the list provide enough information.

This is not a new problem and the literature already suggests solutions, consisting of either: (1) eagerly assigning the empty list a type that is made compatible with the other list types, or (2) keep the empty list's type argument variable, and solve it later.

Regarding the first class of solutions, there are two variations of this idea. The first [34, 36] is to introduce the type `Bot`, make it a subtype of every other type, and consider that type constructors are covariant. The consequence is that the type `Nil(Bot)` is a subtype of `List(T)` for any type `T`. However, this solution was devised in the context of purely functional languages, where covariance is mostly harmless.
The second variation arises in a context closer to ours, GJ [5], where subtyping is invariant by default. They also introduce a 'bottom' type $*$ that is only used internally, and consider a restricted form of covariance, where types instantiated with $*$ subsume instances of the same type where $*$ is replaced by an arbitrary type. That is, $t(*) <: t(T)$ for any t and T. This solution is potentially applicable to our setting, although it is not clear what a bottom type means in the context of Go. For instance, its dual `Top` (or `Object`) is generally presented as an axiom or an ad-hoc addition in other languages, while in Go the top of the subtype hierarchy – the empty interface – is a consequence of the subtyping rules.

The second class of solutions is closer to traditional Hindley-Milner systems, where constraints are not solved locally but instead are first recorded and then solved only after traversing the entire program. We have implemented a prototype of this solution, where

[5] https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#function-argument-type-inference

type variables that escape their scope, such as the one we would generate for `Nil()`, are given some priority. For instance, from the instantiated expression:

```
Cons(X1){Dummy{}, Nil(X2){}}
```

we would extract the substitution `X2` $\mapsto$ `X1` and not the other way around. In a sense, we can draw a parallel between this idea and the ordered typing contexts used in [15], where if a variable $\hat{\alpha}$ occurs to the left of another variable $\hat{\beta}$, then they solve $\hat{\beta}$ to $\hat{\alpha}$ and not the contrary [14].
The drawback of this solution is that we lose the locality of our algorithm and break the assumption that type variables only occur on the right hand side of constraints, which may threaten the decidability of the constraint solving algorithm.

5.4.3 Related Work

We now conclude the exposition by situating our work among the wider context of research in local or bidirectional type inference. We take inspiration from some of the papers we will refer, although we consider a rather unique setting imposed by the restrictions inherited from the Go language itself.

Closest to our work is the foundational *Local Type Inference* [36], as well as some applications of its ideas to more practical contexts such as GJ [5] and (Featherweight) C♯ [2, 3]. In particular, they all solve a problem similar to ours: inferring type arguments for instantiations of generic types and methods/functions, in the presence of bounded quantification, while assuming that generic declarations are fully annotated. In this setting, a bidirectional technique arises naturally.

A further similarity between our system and the one developed in *Local Type Inference* is that they also define both subtyping and equality constraints. However, in their system the equality constraints arise from the fact that they consider the "Kernel" version of $F_{\leq}$ [35], where two quantified types $\forall\overline{X <: B}.\, S$ and $\forall\overline{X <: B}.\, T$ stand in the subtype relation iff the bounds $\overline{B}$ on the quantified variables are identical.
The subtype constraints also differ from ours, as they record both lower and upper bounds for the type variables. The reason being that they also infer type arguments mentioned in the output type of functions, while we only infer arguments that are mentioned in the input types. For example, to make the function type $X \to X$ a subtype of $A \to B$, its necessary to have both $A <: X$ and $X <: B$, which they merge into the single three-place constraint $A <: X <: B$.

Even though their constraints slightly differ from ours, either in meaning or in shape, having these two kinds of constraints implies there must be some way of combining them. For that purpose they define how to combine, via *meets*, two constraints that mention the same type variables – although in this presentation we adopted a more practical style closer to the one on C♯ [3], by showing instead how to combine substitutions.

Listing 5.6: Simulating rank-2 polymorphism in $FGG_{()}$.

```
type IdFun interface {
    Apply(type a any)(x a) a
}

func (this Dummy) ApplyHR(f IdFun) Pair(bool, string) {
    return Pair(){
        f.Apply()(true),
        f.Apply()("hello")
    }
}
```

The distinguishing aspect of our work is the subtype relation we considered and how we use it to guide the constraint solving mechanism. Subtyping in [36] only relates function types, `Top` and `Bot`. The works on GJ and C♯ consider a richer relation, modelling the typical object-oriented subtyping, but in these languages subtyping is nominal, as opposed to FGG.

Higher rank polymorphism On a separate note, our work can also be compared to bidirectional systems aimed at higher ranks of polymorphism [15, 16, 24], as FGG interfaces specifying generic methods can simulate a form of rank-2 polymorphism. As an example, consider the program in Listing 5.6 (adapted from [24]). If we were to assign a "Haskell type" to method `ApplyHR`, it would be something like

```
ApplyHR :: Dummy -> (forall a. a -> a) -> Pair Bool String
```

which is a rank-2 type [24].
One of the central themes in these works is that subtyping actually represents a relation of "more polymorphic than". As such, much of their efforts are directed at finding instantiations that make a polymorphic type a subtype of another possibly polymorphic type. This is done either by instantiating quantifies eagerly using skolemization [24], or lazily, by keeping unsolved unification/existential variables in an ordered context [15, 16]. However, our situation is much simplified by the FGG subtype relation, which never relates a polymorphic type to a monomorphic one. For example, we have that `IdFun` never implements its specializations, e.g. the identity for bools:

```
type IdBool interface {
    Apply(x bool) bool
}
```

Therefore we need not to apply such sophisticated techniques in order to solve our problem.

6 Conclusions and Future Work

The two main goals of this work were to extend the FG/FGG implementations with (1) type list constraints, and (2) type parameter inference. Chapter 4 encompasses the bulk of our work, showing how we realize the former. There, we present the extensions in the form of inductive rules, which act as a form of pseudo code while also trying to prepare ground for future work.

We add primitive types in order to investigate the usage of type lists, which in turn also required us to model the notion of untyped constants in Go. We find that they behave pretty much like primitive types do in languages with implicitly inserted type conversions - an interesting duality since one of the reasons for the strictness of Go type system is to avoid surprising, automatically inserted conversions.

We generalize F(G)G's type declarations in order to experiment with types such as `MyInt` - the only class of types that may satisfy a constraint that specifies both a type list and a set of methods. We explore the consequences of this generalization, finding an aspect that was overlooked in FG – the need to model implicitly inserted type conversions – which is due to them only considering one form of value.
We also find that this generalization has further consequences in the monomorphisation algorithm, requiring the definition of an extra auxiliary function (*T-closure*) for type instance collection, as well as a reformulation of the rule responsible for interface type monomorphisation. The fact that we were able to reformulate this rule instead of adding a separate one targeting only anonymous interfaces, together with the interplay between this rule and *T-closure*, indicate this was indeed a generalization of the original algorithm. For completeness, we also sketch a proof for the correctness of our modifications over the monomorphisation process.

After presenting and discussing the extensions mentioned above, we turned our attention to the problem of type argument inference in the context of FGG extended with

the described features, in Chapter 5. There, we devise an algorithm based on traditional bidirectional inference techniques and discuss its limitations. Among these limitations, we highlight the fact that there seems to be no satisfactory solution to locally infer types for untyped constants, as the definite type of such expressions generally depends on the context where they occur.

Future work will include proving that the developed type system together with the operational semantics enjoy the properties of type safety. On a related note, it will also include the formalization of the type inference algorithm and a rigorous study of its properties.

Other possible direction is to study how to generalize interfaces containing type lists so that they can also represent a form of untagged sum types, which is an hot topic of discussion in Go's issues tracker. A last aspect we didn't investigate is what the Go Team calls "structural constraint" - a type list constraint that specifies a single, composite type - and how it influences type argument inference.

Bibliography

[1] J. A. Bank, A. C. Myers, and B. Liskov. "Parameterized Types for Java". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. Ed. by P. Lee, F. Henglein, and N. D. Jones. ACM Press, 1997, pp. 132–145. DOI: 10.1145/263699.263714. URL: https://doi.org/10.1145/263699.263714 (cit. on p. 1).

[2] G. M. Bierman, E. Meijer, and M. Torgersen. "Lost in translation: formalizing proposed extensions to C#". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by R. P. Gabriel et al. ACM, 2007, pp. 479–498. DOI: 10.1145/1297027.1297063. URL: https://doi.org/10.1145/1297027.1297063 (cit. on p. 79).

[3] G. Bierman. "Formalizing and extending C# type inference". In: *Proceedings of FOOL*. Citeseer. 2007 (cit. on pp. 74, 75, 79).

[4] A. P. Black et al. "Distribution and Abstract Types in Emerald". In: *IEEE Trans. Software Eng.* 13.1 (1987), pp. 65–76. DOI: 10.1109/TSE.1987.232836. URL: https://doi.org/10.1109/TSE.1987.232836 (cit. on p. 45).

[5] G. Bracha et al. "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language". In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*. Ed. by B. N. Freeman-Benson and C. Chambers. ACM, 1998, pp. 183–200. DOI: 10.1145/286936.286957. URL: https://doi.org/10.1145/286936.286957 (cit. on pp. 13, 22, 66, 73, 78, 79).

[6] L. Cardelli. "Type Systems". In: *The Computer Science and Engineering Handbook*. Ed. by A. B. Tucker. CRC Press, 1997, pp. 2208–2236 (cit. on pp. 3, 4).

[7] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Comput. Surv.* 17.4 (1985), pp. 471–522. DOI: 10.1145/6041.6042. URL: https://doi.org/10.1145/6041.6042 (cit. on pp. 3, 10, 11, 13, 22).

[8] D. R. Christiansen. *Bidirectional typing rules: A tutorial*. Accessed: 2021-02-21. 2013. URL: http://davidchristiansen.dk/tutorials/bidirectional.pdf (cit. on pp. 17, 19).

[9] L. Damas and R. Milner. "Principal Type-Schemes for Functional Programs". In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. Ed. by R. A. DeMillo. ACM Press, 1982, pp. 207–212. DOI: 10.1145/582153.582176. URL: https://doi.org/10.1145/582153.582176 (cit. on p. 67).

[10] M. Day et al. "Subtypes vs. Where Clauses: Constraining Parametric Polymorphism". In: *OOPSLA'95, Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Austin, Texas, USA, October 15-19, 1995*. Ed. by R. Wirfs-Brock. ACM, 1995, pp. 156–168. DOI: 10.1145/217838.217852. URL: https://doi.org/10.1145/217838.217852 (cit. on p. 45).

[11] S. Dolan and A. Mycroft. "Polymorphism, subtyping, and type inference in MLsub". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 60–72. DOI: 10.1145/3009837.3009882. URL: https://doi.org/10.1145/3009837.3009882 (cit. on p. 72).

[12] D. Dreyer et al. *What Type Soundness Theorem Do You Really Want to Prove?* Accessed: 2021-02-10. Oct. 2019. URL: https://blog.sigplan.org/2019/10/17/what-type-soundness-theorem-do-you-really-want-to-prove/ (cit. on p. 8).

[13] S. Drossopoulou and S. Eisenbach. "Java is Type Safe - Probably". In: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*. Ed. by M. Aksit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 389–418. DOI: 10.1007/BFb0053388. URL: https://doi.org/10.1007/BFb0053388 (cit. on p. 9).

[14] J. Dunfield and N. Krishnaswami. "Bidirectional Typing". In: *CoRR* abs/1908.05839 (2019). arXiv: 1908.05839. URL: http://arxiv.org/abs/1908.05839 (cit. on pp. 17, 19, 20, 79).

[15] J. Dunfield and N. R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by G. Morrisett and T. Uustalu. ACM, 2013, pp. 429–442. DOI: 10.1145/2500365.2500582. URL: https://doi.org/10.1145/2500365.2500582 (cit. on pp. 20, 79, 80).

[16] J. Dunfield and N. R. Krishnaswami. "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 9:1–9:28. DOI: 10.1145/3290322. URL: https://doi.org/10.1145/3290322 (cit. on pp. 17, 20, 80).

[17] A. D. Gordon and D. Syme. "Typing a multi-language intermediate code". In: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. Ed. by C. Hankin and D. Schmidt. ACM, 2001, pp. 248–260. DOI: 10.1145/360204.360228. URL: https://doi.org/10.1145/360204.360228 (cit. on p. 9).

[18] J. Gosling, W. N. Joy, and G. L. S. Jr. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-201-63451-1 (cit. on p. 13).

[19] R. Griesemer et al. "Featherweight Go". In: *CoRR* abs/2005.11710 (2020). arXiv: 2005.11710. URL: https://arxiv.org/abs/2005.11710 (cit. on p. 27).

[20] R. Griesemer et al. "Featherweight go". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 149:1–149:29. DOI: 10.1145/3428217. URL: https://doi.org/10.1145/3428217 (cit. on pp. 1, 9, 21, 22, 24–27, 33, 41, 45, 47, 49, 50, 53–55, 60, 61).

[21] R. Harper. *Practical Foundations for Programming Languages (2nd. Ed.)* Cambridge University Press, 2016. ISBN: 9781107150300. URL: https://www.cs.cmu.edu/%5C%7Erwh/pfpl/index.html (cit. on p. 8).

[22] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (2001), pp. 396–450. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505 (cit. on pp. 9, 13, 21, 22).

[23] C. Jenkins. "Bidirectional Type Inference in Programming Languages". In: (2018). Accessed: 2021-02-23. URL: https://homepage.cs.uiowa.edu/~cwjnkins/assets/Jen18_Qualifying-Exam.pdf (cit. on p. 20).

[24] S. L. P. Jones et al. "Practical type inference for arbitrary-rank types". In: *J. Funct. Program.* 17.1 (2007), pp. 1–82. DOI: 10.1017/S0956796806006034. URL: https://doi.org/10.1017/S0956796806006034 (cit. on pp. 20, 80).

[25] R. Jung et al. "RustBelt: securing the foundations of the rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154 (cit. on pp. 3, 8).

[26] D. Kozen and A. Silva. "Practical coinduction". In: *Math. Struct. Comput. Sci.* 27.7 (2017), pp. 1132–1152. DOI: 10.1017/S0960129515000493. URL: https://doi.org/10.1017/S0960129515000493 (cit. on p. 60).

[27] S. Krishnamurthi. *Programming languages - application and interpretation*. e-book, 2003. URL: http://www.cs.brown.edu/%5C%7Esk/Publications/Books/ProgLangs/ (cit. on p. 66).

[28] T. Kulesza. *Go Developer Survey 2019 Results*. The Go Blog. Apr. 2020 [Online]. URL: https://go.dev/blog/survey2019-results (cit. on p. 1).

[29] A. Merrick. *Go Developer Survey 2020 Results*. The Go Blog. Mar. 2021 [Online]. URL: https://go.dev/blog/survey2020-results (cit. on p. 1).

[30] T. Nipkow and D. von Oheimb. "Java$_{light}$ is Type-Safe - Definitely". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by D. B. MacQueen and L. Cardelli. ACM, 1998, pp. 161–170. DOI: 10.1145/268946.268960. URL: https://doi.org/10.1145/268946.268960 (cit. on p. 9).

[31] M. Odersky and K. Läufer. "Putting Type Annotations to Work". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by H. Boehm and G. L. S. Jr. ACM Press, 1996, pp. 54–67. DOI: 10.1145/237721.237729. URL: https://doi.org/10.1145/237721.237729 (cit. on p. 70).

[32] M. Odersky, C. Zenger, and M. Zenger. "Colored local type inference". In: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. Ed. by C. Hankin and D. Schmidt. ACM, 2001, pp. 41–53. DOI: 10.1145/360204.360207. URL: https://doi.org/10.1145/360204.360207 (cit. on p. 66).

[33] F. Pfenning. *Lecture notes for 15-312: Foundations of Programming Languages*. Accessed: 2021-02-21. 2004. URL: https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf (cit. on pp. 17–20).

[34] B. C. Pierce. *Bounded quantification with bottom*. Tech. rep. Citeseer, 1997 (cit. on p. 78).

[35] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8 (cit. on pp. 1, 4–6, 8–11, 15–17, 20, 39, 67, 71, 72, 79).

[36] B. C. Pierce and D. N. Turner. "Local type inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (2000), pp. 1–44. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100 (cit. on pp. 66, 67, 72, 76, 78–80).

[37] R. Pike. *Constants*. The Go Blog. Aug. 2014 [Online]. URL: https://go.dev/blog/constants (cit. on p. 29).

[38] G. D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139 (cit. on p. 5).

[39] F. Pottier. "Simplifying Subtyping Constraints: A Theory". In: *Inf. Comput.* 170.2 (2001), pp. 153–183. DOI: 10.1006/inco.2001.2963. URL: https://doi.org/10.1006/inco.2001.2963 (cit. on pp. 69, 72).

[40] J. C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Ed. by B. Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, 1974, pp. 408–423. DOI: 10.1007/3-540-06859-7_148. URL: https://doi.org/10.1007/3-540-06859-7%5C_148 (cit. on p. 15).

[41] T. Schrijvers et al. "Complete and decidable type inference for GADTs". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by G. Hutton and A. P. Tolmach. ACM, 2009, pp. 341–352. DOI: 10.1145/1596550.1596599. URL: https://doi.org/10.1145/1596550.1596599 (cit. on p. 20).

[42] A. Serrano et al. "A quick look at impredicativity". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 89:1–89:29. DOI: 10.1145/3408971. URL: https://doi.org/10.1145/3408971 (cit. on pp. 15, 17, 20).

[43] C. Strachey. "Fundamental Concepts in Programming Languages". In: *High. Order Symb. Comput.* 13.1/2 (2000), pp. 11–49. DOI: 10.1023/A:1010000313106. URL: https://doi.org/10.1023/A:1010000313106 (cit. on p. 10).

[44] D. Syme. "Proving Java Type Soundness". In: *Formal Syntax and Semantics of Java*. Ed. by J. Alves-Foss. Vol. 1523. Lecture Notes in Computer Science. Springer, 1999, pp. 83–118. DOI: 10.1007/3-540-48737-9_3. URL: https://doi.org/10.1007/3-540-48737-9%5C_3 (cit. on p. 9).

[45] *The Go Programming Language Specification*. July 2021 [Online]. URL: https://go.dev/ref/spec (cit. on pp. 29, 41).

[46] J. B. Wells. "Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable". In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 176–185. DOI: 10.1109/LICS.1994.316068. URL: https://doi.org/10.1109/LICS.1994.316068 (cit. on p. 17).

[47] A. K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115.1 (1994), pp. 38–94. DOI: 10.1006/inco.1994.1093. URL: https://doi.org/10.1006/inco.1994.1093 (cit. on pp. 4, 8).

[48] D. Yu, A. Kennedy, and D. Syme. "Formalization of generics for the .NET common language runtime". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by N. D. Jones and X. Leroy. ACM, 2004, pp. 39–51. DOI: 10.1145/964001.964005. URL: https://doi.org/10.1145/964001.964005 (cit. on p. 9).