João Afonso Oliveira Pereira

Bachelor of Science

Adding Type List Constraints and Type Inference to Featherweight Generic Go

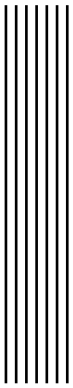
Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021



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 extensions 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. As such, this work proposes to study these extensions.

The new features that will be considered are primitive types and operations, type lists in interfaces and inference of type parameters. The primitive operators present some challenges related to the heterogeneity they exhibit; type lists arise as the solution for allowing primitive operations between values of 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 will study how these features can be integrated in FGG and how they impact its properties of type safety and the monomorphisation process. In particular, we will extend the FG and FGG prototype implementations with primitive types and operations, add type lists to FGG interfaces and implement their translation to FG, and implement type parameter inference using a bidirectional typing technique.

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

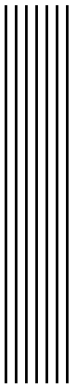
Resumo

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 extensões que não foram investigadas em contexto formal, abrindo espaço para interações imprevistas entre *features*, que podem só vir a ser descobertas demasiado tarde no processo de desenho. Como tal, este trabalho propõe estudar essas extensões.

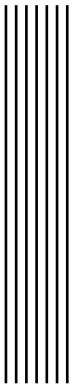
As novas features a considerar são os tipos e operadores primitivos, as listas de tipos em interfaces e a inferência de parâmetros de tipo. Os operadores primitivos levantam alguns desafios relacionados com a heterogeneidade que exibem; as listas de tipos surgem como a solução para permitir operações primitivas entre valores de tipos variáveis, 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.

Estudaremos como é que estas features podem ser integradas no FGG e que impacto causam sobre as propriedades de *type safety* e sobre o processo de monomorfização. Particularmente, iremos estender os protótipos de implementação do FG e do FGG com tipos e operadores primitivos, adicionar listas de tipos às interfaces do FGG e implementar a sua tradução para FG, e implementar inferência de parâmetros de tipo usando uma técnica de *bidirectional typing*.

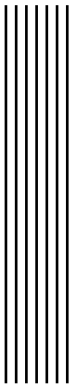
Palavras-chave: Golang, Teoria de linguagens de programação, Sistemas de tipos, Linguagens peso-pluma, Listas de tipos, Tipificação bidirecional



Contents



List of Figures



Listings

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 the developers request the most. 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 [13]. However, it also includes some real-world oriented extensions whose foundations have not yet been thoroughly studied. It is known that these discrepancies between theory and practice may be problematic, as one might overlook unexpected interactions between seemingly unrelated features - such as the relationship between subtyping and arrays in Java [21], which results in a slow down of *every* assignment to an array.

In particular, the 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 in their code. In addition, the design further includes type parameter inference, a feature that was also not investigated in the work on Featherweight Generic Go [13].

The primitive operators present some challenges related to the heterogeneity they exhibit - e.g., it is not possible to add an int64 to an int. Type lists introduce a completely different notion of interface, where one explicitly states the types that can implement that interface, instead of just specifying a set of methods. Moreover, type lists act as a form of intersection of types (rather than an union), since the operators supported by the

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

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

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 can then have distinct methods.

Therefore it becomes important to understand how these extensions interact with the features already considered in the core underlying theories of Featherweight Go and Featherweight Generic Go. Namely if they can compromise the type-safety properties of the model languages, and how they impact *monomorphisation* - the technique employed in the compilation of generics, that generates specialized code for each possible instantiation of a generic type. As for type lists, it's plausible that explicitly specifying all the primitive types that may instantiate a type parameter actually aids in speeding monomorphisation.

1.1 Goals

In this work, we aim to study the problem of integrating type lists in FGG and its impact on monomorphisation and type inference. To this end, we will:

- Extend the implementation of FG and FGG (typechecker and interpreter) with primitive types and operations;
- Add type lists to FGG interfaces (again, in the typechecker and the interpreter);
- Extend the monomorphisation algorithm to account for type lists;
- Add type parameter inference to FGG.

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 that we will be basing on, focusing on the model of Featherweight Generic Go and some of its current limitations. Chapter 4 ends the presentation with the proposal for this thesis.

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 [2], 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 [3]. 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 [18].

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 [29].

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[21] - 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 of 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 [2]. 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 [21]. 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	$\Gamma ::=$	**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* [22, 21]. 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 (on the left) 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 $t_1\ t_2$, t_1 must first be reduced to a value before proceeding. E-Congr1 reinforces this order by stating that if t_1 can be simplified, then that simplification

$$\text{(E-Congr1)}\ \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App)}\ \frac{v_2\ \textit{is a value}}{(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}$$

$$\text{(E-Congr2)}\ \frac{t_2 \longrightarrow t_2' \quad v_1\ \textit{is a value}}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-IfTrue)}\ \frac{v\ \textit{is true}}{\text{if } v \text{ then } t_2 \text{ else } t_3 \longrightarrow t_2}$$

$$\text{(E-If)}\ \frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \qquad \text{(E-IfFalse)}\ \frac{v\ \textit{is false}}{\text{if } v \text{ then } t_2 \text{ else } t_3 \longrightarrow t_3}$$

Figure 2.2: Evaluation rules for λ_B.

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

The rules E-App, 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, and apply it, producing t'. This process is repeated 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, e.g. the term:

```
if λx:Bool. x then true 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 [21]. 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.

$$\text{(T-Var)}\ \frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Abs)}\ \frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2} \qquad \text{(T-True)}\ \frac{}{\Gamma \vdash true : Bool}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-False)}\ \frac{}{\Gamma \vdash false : Bool}$$

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

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

```
if <complex-term1> then <complex-term2> else <complex-term3>
```

we know for sure that if `<complex-term1>` 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*:

$$\dfrac{\dfrac{\dfrac{x{:}Bool \in \Gamma, x{:}Bool}{\Gamma, x{:}Bool \vdash x : Bool}\,(\text{T-Var})}{\vdash \lambda x{:}Bool.\ x : Bool \to Bool}\,(\text{T-Abs}) \qquad \dfrac{}{\vdash true : Bool}\,(\text{T-True})}{\vdash (\lambda x{:}Bool.\ x)\ true : Bool}\,(\text{T-App})$$

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)[14].

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 [29] and later simplified in textbook presentations [14, 21]. Recent research [18] 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[5]. 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 [6] had subtle errors only discovered later [27]. Some works [19, 27] 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 [15, 10, 30, 13] the decision is to favor compactness over completeness, so as to get "maximum insight for minimum investment" [15]. 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 [21]. 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):

```
idBool = λx:Bool. x
  ↪ Bool → Bool
idFun = λx:Bool→Bool. x
  ↪ (Bool→Bool) → (Bool→Bool)
id2ndOrd = λx:(Bool→Bool)→(Bool→Bool). x
  ↪ ((Bool→Bool)→(Bool→Bool)) → ((Bool→Bool)→(Bool→Bool))
```

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

[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 [21].

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 [26, 3]: *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* [21], 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 [26]. 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:

```
id = λx:Bool. x
id = λx:Bool→Bool. x
id = λx:(Bool→Bool)→(Bool→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*[3]. 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 [21]. 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:

$$(\text{T-Sub})\ \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

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* [21]. The syntax is summarized in Fig. 2.4, where the "..." 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. An empty tuple is represented as $\langle\rangle$, and 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; the operation 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:

```
proj1 = λx:⟨Bool⟩. x.1
  ↪ ⟨Bool⟩ → Bool
```

[3]This particular convention 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.

$t ::= \ldots$	**terms:**	$T ::= \ldots$	**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
$v ::= \ldots$	**values:**		
$\langle v_i^{i\in 1..n}\rangle$	tuple value		

Figure 2.4: Syntax of tuples.

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.

$$\text{(S-TplWidth)}\ \frac{}{\langle T_i^{i\in 1..n+k}\rangle <: \langle T_i^{i\in 1..n}\rangle} \qquad \text{(S-TplDepth)}\ \frac{\text{for each } i \in 1..n \quad S_i <: T_i}{\langle S_i^{i\in 1..n}\rangle <: \langle T_i^{i\in 1..n}\rangle}$$

$$\text{(S-Refl)}\ \frac{}{S <: S} \qquad \text{(S-Trans)}\ \frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-Arrow)}\ \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

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. S-Refl and S-Trans are self-explanatory given the definition of subtyping stated at 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:

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

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

```
idTop = λx:⊤. x
  ↪ ⊤ → ⊤
```

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 [1, 11], 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 [15].

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[3]: 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 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.

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

Figure 2.6: Syntax of polymorphic extension of λ_B.

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 below. The rule E-TCongr is again a *congruence* rule.

$$\text{(E-TCongr)}\ \frac{t_1 \longrightarrow t_1'}{t_1\ [T] \longrightarrow t_1'\ [T]} \qquad \text{(E-TApp)}\ \frac{}{\Lambda X.\ t\ [T] \longrightarrow [X \mapsto T]\ t}$$

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

```
id = ΛX.λx:X. x
  ↪ ∀X. X → X
```

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

```
λx : Bool. x
  ↪ Bool → 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 $Bool \to Bool$; likewise, applying it to $Bool \to Bool$ would yield a function typed $(Bool \to Bool) \to (Bool \to Bool)$. In general, applying *id* to a type T produces a type $T \to T$, 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 $\forall X.\ X \to X$ [21]. The rule T-TAbs displayed below 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.

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

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*[21] or *polymorphic lambda calculus*[23].
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 [21]:

```
selfApp = λx:∀X.X → X. (x [∀X.X → X]) x
  ↪ (∀X. X → X) → (∀X. X → 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 $x\ [\forall X.X \to X]$ above - a feature known as impredicativity [21, 25]. 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")[21]. $F_{<:}$ 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 [21]). 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:

```
ΛX.λt:X. ⟨t.1,t⟩
  ↪ 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:

```
ΛY.ΛX<:⟨Y⟩. λt:X. ⟨t.1,t⟩
  ↪ ∀Y.∀X<:⟨Y⟩. X → ⟨Y,X⟩
```

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 writing $id\ [Bool]\ true$ or $id\ [\langle Bool,\ Bool\rangle]\ \langle true,\ false\rangle$ it would be desirable 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 [28]. One alternative to overcome the undecidability result is to restrict the allowed forms of polymorphism. The most popular [21] restriction is the one found in Hindley-Milner type systems, allowing only *predicative* polymorphism [25], 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 [7, 9]. 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 main 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 [20]. 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 [4]). In Listing 2.1, we check that the if-term has 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.

provided input type ty by verifying that: (1) t_1 has type *Bool*, and (2) both branches t_1, t_2 have the expected type ty. Notice that the function returns a type 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*.

Listing 2.1: check function

```
check ctx (If t1 t2 t3) 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
```

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 – [20]):

- $\Gamma^+ \vdash t^+ \Rightarrow T^-$ "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"

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 [4]) - 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:

$$(\text{BT-App})\ \frac{\Gamma \vdash t_1 \Rightarrow T \to T' \quad \Gamma \vdash t_2 \Leftarrow T}{\Gamma \vdash t_1\ t_2 \Rightarrow T'} \qquad (\text{BT-Abs})\ \frac{\Gamma, x{:}T \vdash t \Leftarrow T'}{\Gamma \vdash \lambda x.t \Leftarrow (T \to T')}$$

$$(\text{BT-Ann})\ \frac{\Gamma \vdash t \Leftarrow T}{\Gamma \vdash (t : T) \Rightarrow T} \qquad (\text{BT-Var})\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x \Rightarrow T}$$

An application is a *destructor*[20] - 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 [7, 20].

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:

$$(\text{T-Sub})\ \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

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:

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

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* [7], 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 $\Gamma \vdash t \Leftarrow T$ [20, 16]. In particular, one typically defines a rule like

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

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 [21].

2.3.2 Application to polymorphic type systems

Recent work [17, 9, 7, 25] 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 [8, 17]; existential polymorphism and GADTs [9, 24] and even impredicative polymorphism [25]. 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 [13] is discussed in greater detail as this work derives directly from it.

3.1 Featherweight Java

Featherweight Java [15] 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. One 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 [15], 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. [15] extend FJ with *generics* as a way to investigate how parametric polymorphism can be integrated into Java. The main technique they use, following GJ [1], 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 runtime.

In (1), the usual way to make representations uniform is to consider all the types as pointers to heap-allocated objects [3]. 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 [13] follows the footsteps laid by Featherweight Java [15], 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 receives 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 states that a struct *implicitly* implements (is a subtype of) an interface if the struct defines all the methods specified by the interface; and 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 ($\Leftrightarrow$ object creation), field selection, method calls and type assertions ($\Leftrightarrow$ casts).

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

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

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

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. For instance, the code shown below defines the method Apply for the struct type *incr*.

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

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 [13]. 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:

$$\text{(T-Func)}\ \frac{t_S\ ok \qquad \overline{t\ ok} \qquad u\ ok \qquad distinct(x, \overline{x}) \qquad x : t_S, \overline{x : t} \vdash e : t \qquad t <: u}{\texttt{func}\ (x\ t_S)\ m(\overline{x\ t})\ u\ \{\texttt{return}\ e\}\ ok}$$

[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 [13].

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:

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

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):

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

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 $\texttt{type}\ \overline{\alpha\ \tau_I}$, where α is a type parameter and the bound τ_I is an interface type.
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 $\texttt{struct}\ \{\overline{f\ \tau}\}$ 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

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	$\Phi, \Psi ::= \texttt{type}\ \overline{\alpha\ \tau_I}$
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	$\texttt{struct}\ \{\overline{f\ \tau}\}$	Structure literal	$\tau_S\{\overline{e}\}$
Interface	$\texttt{interface}\ \{\overline{S}\}$	Select	$e.f$
Declaration	$D ::=$	Method call	$e.m(\overline{\tau})(\overline{e})$
Type declaration	$\texttt{type}\ t(\Phi)\ T$	Type assertion	$e.(\tau)$
Method declaration	$\texttt{func}\ (x\ t_S(\Phi))\ mM\ \{\texttt{return}\ e\}$		
Program	$P ::= \texttt{package main};\ \overline{D}\ \texttt{func main}()\ \{\_ = e\}$		

Figure 3.2: FGG syntax

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){}}}
}
```

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 [13]) to the specifications in the interface. This is the reason `Nil` also needs a type parameter.
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 [13].

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 [12], 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 Proposed Work

4.1 Type list constraints

FGG only allows us to constrain type parameters with interface types, (e.g. `type a Any`). This means that the only thing we can do with a variable of polymorphic type is to call the methods specified in the interfaces that bound the variable. We are not allowed to apply primitive operations such as comparisons <, > or arithmetic operators as +, − to values of a variable type, as the primitive operations are not defined with methods and hence cannot be specified in an interface. However these operations play a major role in day-to-day programming, forming the building blocks for routines as common as sorting data collections.

The solution proposed in the official design proposal for generics [1] is to add type lists to interfaces. Essentially, the idea is to explicitly declare in an interface that will be used as a constraint, what primitive types can be used to instantiate a type variable constrained by that interface - Listing 4.1 illustrates an example declaration of a constraint for primitive types that can be ordered. Then, in any use of a primitive operation over variables of polymorphic type, the compiler can verify that the interface used to constrain that variable contains only primitive types for which that operation is defined - and hence the operation can safely be applied. As an example, Listing 4.2 shows, in plain Go (adapted from the official proposal), how the previously defined constraint can be used in a generic function that finds the smallest element of a slice. As `v` and `r` have both type `T`, constrained by `Ordered`, `v` and `r` can be compared by the operator < in a safe way.

Primitive types and type lists are not included in FG/FGG, the main theoretical work

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

Listing 4.1: Example constraint interface declaration in Go/FGG

```
// Ordered is a type constraint that matches any ordered type.
// An ordered type is one that supports the <, <=, >, and >= operators.
type Ordered interface {
  type int, int8, int16, int32, int64,
      uint, uint8, uint16, uint32, uint64, uintptr,
      float32, float64,
      string
}
```

Listing 4.2: Example constraint utilization in Go

```
// Smallest returns the smallest element in a slice.
// It panics if the slice is empty.
func Smallest[T Ordered](s []T) T {
  r := s[0] // panics if slice is empty
  for _, v := range s[1:] {
    if v < r {
      r = v
    }
  }
  return r
}
```

that supports the proposal. As such, we propose to implement and study the consequences of these extensions.

The first step will then be to add some primitive types and operations to both FG and FGG, as they will be necessary to model type lists. This step will encompass extending the type checker with base types and the possibility to create type aliases to them, as well as typing rules for the operators. After that, we will add the corresponding evaluation rules to the interpreter.

The next step is to add type lists to FGG - they won't "exist" in FG, as it doesn't have type parameters and consequently no type parameter constraints. We will need to define how to check the *implements* relation between a type used to instantiate a parameter and a constraint interface, which in principle could be a relatively simple membership test, but might be complicated by the fact that one can also define type aliases to primitive types, and that type lists can refer to many type aliases.

The final task in implementing type lists is to add the translation of FGG with type lists in interfaces to bare FG. This will be done by extending the monomorphisation algorithm, possibly enhancing it with the explicit information conveyed by type lists.

4.1.1 Validation

In order to validate each of these implementation steps, we will use the tools developed in the context of the prototype implementation of FG and FGG [2].

[2] https://github.com/rhu1/fgg/

In particular, we will take advantage of the fact that the interpreters support dynamic checking of preservation and progress: if after the implementation of each new feature we perform extensive testing, and every test passes the dynamic checks, we can attain a fair degree of certainty that the extension doesn't break the type-safety properties.

We will also test our extension to the monomorphisation algorithm following the approach employed in the FG work. Namely, we will use the test of bisimulation: given a well-typed program P in FGG (with type lists) - say with top level expression d-, we generate its monomorphised version P' in FG - say with top level expression e. We apply one (small-) step of evaluation to both d and e and verify that the obtained FGG term d' still monomorphises to the FG term e'; this process is then repeated until the programs terminate.

4.2 Inference of type instantiations

As section 3.2.3 illustrated, the instantiation of polymorphic types and functions can be very verbose. As such, and following the official language change proposal, we propose implementing a form of type parameter inference based on bidirectional typing (Section 2.3), as long as time permits.

Although FGG doesn't have polymorphism as sophisticated as the systems mentioned in Section 2.3, we can still follow the bidirectional "recipe" [7] and integrate techniques of constraint solving commonly employed in more general forms of inference. The goal of this inference is to allow the programmer to omit type annotations in most of the instantiations of type parameters, although the annotations will always be required in the declaration of polymorphic types or methods. We propose to further investigate the power of the implemented inference technique by adding a form of un-annotated local variable declaration (such as Go's `var`) that requires more inference. Of course, this will be a non-mutable variable so as to avoid introducing mutable state.

The validation of the implementation will be performed in a manner similar to that described in the previous section.

4.3 Work plan

Figure 4.1 displays the proposed work plan. We will start by studying the existing prototype implementations. Then, we will add primitive types and operations to both the interpreter and the typechecker. Next, we will add support for type aliases. All these steps are necessary before we can implement type lists. After that we will implement the translation of interfaces with type lists in FGG to FG, by extending the monomorphisation algorithm. Then we will implement the type parameter inference, and finally write the dissertation document. Although the chart doesn't make it explicit, every implementation step will be validated according to the methodology described in Section 4.1.1.

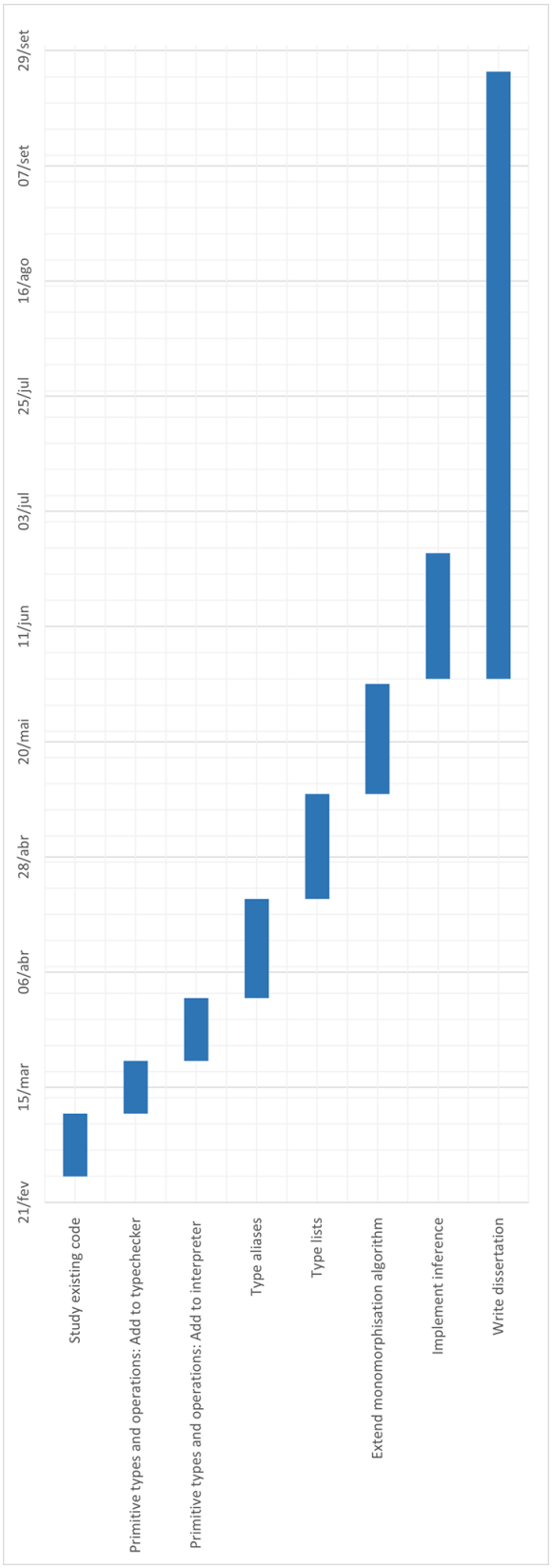
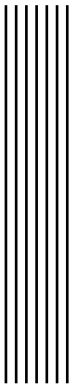


Figure 4.1: Work plan



Bibliography

[1] 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.

[2] L. Cardelli. "Type Systems". In: *The Computer Science and Engineering Handbook*. Ed. by A. B. Tucker. CRC Press, 1997, pp. 2208–2236.

[3] 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.

[4] D. R. Christiansen. *Bidirectional typing rules: A tutorial*. Accessed: 2021-02-21. 2013. URL: http://davidchristiansen.dk/tutorials/bidirectional.pdf.

[5] 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/.

[6] 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.

[7] J. Dunfield and N. Krishnaswami. "Bidirectional Typing". In: *CoRR* abs/1908.05839 (2019). arXiv: 1908.05839. URL: http://arxiv.org/abs/1908.05839.

[8] 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.

[9] 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.

[10] 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.

[11] J. Gosling, W. N. Joy, and G. L. S. Jr. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-201-63451-1.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.