SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Generating Executable Go Code from the Isabelle Theorem Prover

Matthias Stübinger

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Generating Executable Go Code from the Isabelle Theorem Prover

Generierung von ausführbarem Go Code aus dem Beweisassistenten Isabelle

Author:	Matthias Stübinger
Supervisor:	Prof. Tobias Nipkow
Advisor:	Lars Hupel
Submission Date:	June 15, 2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, June 15, 2023 Matthias Stübinger

Abstract

The Isabelle proof assistant includes a small functional language, which allows users to write programs and prove propositions about these programs. So far, the so-written code could be extracted into a number of functional languages: Standard ML, OCaml, Scala, and Haskell.

This work adds support for Go as a fifth target language for the code generator. Unlike the previous target languages, Go is not a functional language and encourages code in an imperative style, thus many of the features of Isabelle's language (particularly data types, pattern matching, and type classes) have to be emulated using imperative language constructs in Go.

The developed code generation is provided both as an out-of-tree development which can be used with any recent development version of Isabelle by simply importing it, and as an in-tree set of patches for Isabelle itself which allow deeper integration with existing Isabelle tools.

Finally, methods of testing the generated code for bugs, i.e. cases where its behaviour differs from the Isabelle formalisation from which it was extracted, are discussed.

Contents

1. Introduction

The interactive theorem prover *Isabelle* of the LCF tradition (Nipkow and Klein 2014) is based on a small, well-established and trusted kernel written in Standard ML. All higher-level tools and proofs, such as those included in the most commonly-used logic *Isabelle/HOL*, have to work through this kernel.

Some (but by far not all) of the tools available to users in *Isabelle/HOL* feel immediately familiar to anyone with experience in functional programming languages: it is possible to define data types (via the **datatype** command), functions (via **fun** and **function**), as well as type classes and instances akin to Haskell (via **class** and **instance**).

1.1. Motivation

Unlike most other languages, Isabelle's nature as a theorem prover means that it is easy to formalise and prove propositions about the behaviour of the programs written in it, for example to ensure that a specific invariant is never violated.

To allow use of such programs outside of the proof assistant's environment, Isabelle comes equipped with a *code generator*, allowing users to extract source code in Haskell, Standard ML, Scala, or OCaml, which can then be compiled and executed. This translation of code works via an intermediate language called *Thingol* shared by all target languages; finally the code of the intermediate language is transformed into code in the individual target languages by the principle of *shallow embedding*, that is, by representing constructs of the source language using only a well-defined subset of the target language (Haftmann 2009; Haftmann and Nipkow 2010).

On the other hand, Go is a programming language introduced by Google in 2009 (Griesemer, Pike, et al. 2009). It is a general-purpose, garbage-collected, and statically typed language (Go Team 2022). In contrast to the existing targets of Isabelle's code generator, it is not a functional language, and encourages programming in an imperative style. However, it is a very popular language[1], and many large existing code bases have been written in it.

[1]As a rough indication, the Stack Overflow 2023 Survey lists it as the 13th most commonly used language, above the existing targets Scala, Haskell, and OCaml on places 27, 31, and 43 respectively. Standard ML is not included in the ranking.

The aim of this work is to give a translation schema from programs in Thingol to programs in Go, and thereby allow Isabelle users to also generate Go code from their theory files. This extension to the code generator exists (almost, compare Appendix A) entirely as an out-of-tree development separate from the main Isabelle distribution, which can be used with a suitably recent development version of Isabelle. Its source code is made available at `https://git.sr.ht/~stuebinm/isabelle-go-codegen`.

1.2. Structure

The next two chapters 2 and 3 will give an overview of Thingol, the intermediate language used during code generation, and of the fragment of Go which is used as target for the shallow embedding process.

Chapter 4 will then discuss the translation itself. Given the fundamental differences between Go and the previous target languages, some challenges present themselves: in particular, obvious equivalents for data types (discussed in Section 4.4) and pattern matching (discussed in Section 4.5) are features which have existed (with minor differences) in all previous target languages, but are entirely absent in Go, and thus must be emulated using different features. Type classes likewise have no easy equivalent in Go; here the well-known dictionary construction for type classes (Haftmann and Nipkow 2010; Hupel 2019) which is also used for the Standard ML and OCaml targets is adapted for Go in Section 4.7.

Finally, several ways to approach testing *correctness* of the generated code are discussed in Chapter 6.

There are two appendices: while the Go code generator is in principle an out-of-tree development, a small number of minor changes to Thingol itself had to be made to accommodate type information needed to generate Go code, which all of the previous targets could infer on their own; these are listed in Appendix A. A short guide for users of the code generator with Go is given in Appendix B.

2. Thingol

Isabelle's code generation pipeline works in multiple stages. Most importantly, all definitions made in Isabelle are first translated into an abstract intermediate language called *Thingol*, which is the last stage shared between all target languages. The final stage then uses a shallow embedding to translate the Thingol program into source code of the target language.

Thingol is meant to capture all target languages' common essence, as a "rest point" between the formal world of Isabelle, and the "dirtier", less specified and diverse world of target languages.

Of course, if one intends to add a new target language, this implies most (if not all) of the work will already lie beyond this rest point: Thingol has no formal description of its semantics, and neither has Go (or any of the other target languages except Standard ML). Thus for most of the way, there can only be informal reasoning; the satisfaction of achieving verification "all the way," i.e. being able to confidently assert that the generated code will "always" behave as the original Isabelle equations from which it was translated thus seems very far away.

Nevertheless, we can still attempt to make as sure as we reasonable can that, *in practice*, the generated code will most likely never go wrong—or at least, by applying the same testing strategies programmers use for "ordinary" code which has never been formalised, we can hope to be sure that if the generated code ever does go wrong, it will do so less often than it would have had it been hand-written in the target language from the start.

As such, Thingol is meant purely as an intermediate language: it exists only as a series of data type definitions within Isabelle/ML, and even if it had syntax (and a parser to read the syntax), most programmers would not find it comfortable to write code in it.

However, since all Thingol code the generator receives will always itself be generated from Isabelle, it comes with a few unusual guarantees which make using it as an input language much easier.

2.1. Names

While names of bindings or constants are preserved from Isabelle, Thingol has two basic provisions to deal with the usual issues that may arise:

- There is extensive infrastructure to "deresolve" names into a unique form. In particular, this means that at any point in the target language, all names in scope will be unique, and names will never be shadowed.
- In the same manner, it is always possible to create a guaranteed-unused and therefore "fresh" name, should a target language construct require one, without having to fear it will shadow existing ones and unintentionally change the semantics of the generated program.

Taken together, these can be summed up by saying: the generator (almost) never has to care about names[1] when emitting code, and intricacies of how to avoid shadowing existing names will not be mentioned in the rest of this work.

2.2. Types & Terms

Previous target languages of Isabelle's code generator were all functional, with most of them being part of the "ML tradition" of languages: Standard ML, OCaml, Haskell, and Scala.

As such, Thingol's design reflects the features common to all of them, its terms being essentially a typed λ-calculus with polymorphism, sorts used to implement type classes, and a case-expression used to do pattern matching:

- sorts: $s ::= c_1 \cap \cdots \cap c_n$
- types: $\tau ::= \kappa \;\; \overline{\tau} \;\mid\; \alpha :: s$
- terms: $t ::= f[\overline{\tau}] \;\mid\; x :: \tau \;\mid\; \lambda x :: \tau.\;\; (t :: \gamma) \;\mid\; t_1 \;\; t_2 \;\mid\; \text{case } t :: \tau \text{ of } [\overline{p \to b}]$

Types: consist of either a type constructor κ with a fixed arity, or are a type variable constrained by a sort, which can be read as the type variable being "constrained" by a set of type classes, instances of each must be available for types it is instantiated with.

Sorts: work as they do in Isabelle/Pure[2] and encode a simple order-sorted algebra based on intersections of classes: one sort subsumes another sort if it contains at least all the same classes.

[1]as Goethe would have said, "Gebundene Namen sind Schall und Rauch."

[2]In fact, they are implemented using the same Standard ML type.

Terms: are exactly the abstraction, application, and bound variable known from λ-calculus. Notably, variable names in terms are optional, and may be set to the empty name "_" in patterns.

In addition, there are constants which represent top-level names which exist outside the term (for example, functions defined at a program's top-level), and a case-expressions used for pattern matching on data types.

Constants take a list of type parameters and are thus polymorphic. While they can be used as multiple different types, each usage is annotated with the types $\overline{\tau}$ used to instantiate its parameters.

The case expression's pattern matching works in the usual way: a pattern's term is restricted to variables and fully satisfied applications of data type constructors. Patterns are compared one-by-one in the given order to the term t's value. The entire expression then evaluates to the body b_i of the first matching clause $p_i \to b_i$. If *no* pattern at all matches, the program as a whole aborts.

2.3. Statements

As with terms, Thingol's top-level statements are likewise familiar from other functional programming languages:

```
data $\kappa$ $\overline{\alpha}_k$ = $f_1$ of $\overline{\tau_1}$ | $\cdots$ | $f_n$ of $\overline{\tau_n}$

fun $f$ :: $\forall\overline{\alpha :: s_k}.\tau$ where
  $f$ [$\overline{\alpha :: s_k}$] $\overline{t_1}$ = $t_1$
| ...
| $f$ [$\overline{\alpha :: s_k}$] $\overline{t_n}$ = $t_n$

class $c \subseteq c_1 \cap \cdots \cap c_m$ where
  $g_1$ :: $\forall\alpha :: c.$ $\tau_1$, $\ldots, g_n :: \forall\alpha :: c.$ $\tau_n$

inst $\kappa$ $\overline{\alpha :: s_k} :: c$ where
  $g_1$ [$\kappa$ $\overline{\alpha :: s_k}$] = $t_1$, $\ldots$, $g_n$[$\kappa$ $\overline{\alpha :: s_k}$] $= t_n$
```

Figure 2.1.: Thingol's statements in Haskell-like syntax, taken from Haftmann and Nipkow 2010.

Data types: data types are the only way to introduce new type constructors within a program. As in Standard ML or Haskell, a data type consists of a list of constructors, each with a fixed arity.

Functions: top-level functions may have multiple equations, but are restricted to left-linear constructor patterns in their arguments. In effect, multiple equations of the same function thus implement another kind of case-expressions, and behave the same as a function with just one equation containing a case-expression as their top-level term.

Further unlike the λ-calculus style abstractions allowed in terms, functions can introduce polymorphic types restricted by sorts in their argument's types.

Type Classes: a type class c with superclasses c_1 to c_m defines a set of *methods* or *class parameters* g_1 to g_n.

An instance $\kappa\ \ \overline{s}_k :: c$ instantiates c for a concrete type constructor κ defines terms for each of the *instance parameters* g_1 to g_n, assuming that the arguments $\overline{\alpha}_k$ of κ have sorts $\overline{s}_k$. An instance declaration is only valid if instances $\kappa\ \ \overline{z}_k :: c_i$ for each superclass c_i of c such that each s_j is a subsort for z_k likewise exist.

The intended semantics of type classes is as in Haskell. In particular, it is guaranteed that instances are unique, i.e. there will only be one instance of c for any constructor κ. Further detail is given in (Haftmann 2009).

3. A Fragment of Go

Go is a high-level, statically typed language. However, it is not a functional language, and differs in many aspects from the already-existing target languages of Isabelle's Code Generator.

However, many of the features unique to Go are not needed by the generator; since the translation works as a shallows embedding into the target language, it suffices to use those features of Go which can be used to represent the various statements of Thingol. Only those will be presented in the following, along with—if necessary—discussion why alternative features or other possible solutions were not persued.

In effect, this leaves many of Go's more interesting features[1] entirely unused. The fragment used by the code generator could even be understood as a "functional subset" of the Go language, meaning that it picks only those features that closely align with those of the (functional) pre-existing code generation targets available in Isabelle as well as those of Thingol.

[1]e.g. Channels or any kind of parallelism, Methods, any kind of variable mutation, the `defer`-statement.

3.1. Syntax

Field name	A
Function name	f
Variable name	x
Structure type name	t_S
Interface type name	t_I
Type parameter	α
Type name	$t ::= t_S \mid t_I$
Type	$\tau, \gamma, \sigma ::=$
Parameter	α
Named structure type	$t_S\left[\overline{\tau}\right]$
Named interface type	$t_I\left[\overline{\tau}\right]$
Function head	$F ::= \left[\overline{\alpha\ \tau_I}\right]\texttt{(}\overline{x\ \tau}\texttt{)}\ \texttt{(}\overline{\tau}\texttt{)}$
Type literal	$T ::=$
Structure	$\texttt{struct}\left[\overline{\alpha\ \tau_I}\right]\{\overline{A\ \ \tau}\}$
Interface	$\texttt{interface}\left[\overline{\alpha\ \tau_I}\right]\{\}$
Declaration	$D ::=$
Type declaration	$\texttt{type}\ \ \tau\ \ T$
Function declaration	$\texttt{func}\ \ fF\ \ \{s\}$
Variable declaration	$\texttt{var}\ \ x\ \ \tau\ \ \texttt{=}\ \ e$
Expression	$d, e ::=$
Variable	x
Function call	$f[\overline{\alpha}](\overline{e})$
Structure literal	$t_s[\overline{\alpha}]\{\overline{e}\}$
Function abstraction	$\texttt{func}\ \ F\{s\}$
Select	$e.A$
Type conversion	$\tau_I(e)$
Zero value	$\texttt{*new(}\tau\texttt{)}$
Statement	$s ::=$
Expression	$\texttt{return}\ \ \overline{e}\texttt{;}$
Variable declaration	$x\ \ \texttt{:=}\ \ e\texttt{;}\ \ s$
if-Statement	$\texttt{if}\ \ \texttt{(}e\texttt{)}\ \ \texttt{\{}s\texttt{\};}\ \ s$
Type switch	$\texttt{switch}\ \ \texttt{(}x\texttt{:=}\ e\texttt{.(type))}\ \texttt{\{}\ \texttt{case}\ \tau\texttt{:}\ s\texttt{;}\ \texttt{default:}\ s\texttt{\}}$
Package name	n
Package url	$u ::= \text{"}u_{base}/n\text{"}$
Package	$P ::= \texttt{package}\ \ n\texttt{;}\ \ \texttt{import}\ \ \texttt{(}\overline{u}\texttt{);}\ \ \overline{D}$

The syntax fragment given above largely follows that of Featherweight Generic Go defined by Griesemer, Hu, et al. 2020, but differs in some important aspects:

1. methods are not used in favour of "normal" functions
2. Go's syntactic distinction between expressions and statements in terms is preserved. However, it is always possible to use an expression in place of a statement by using the $s :=$ `return` e`;` and to use a statement in place of an expression by wrapping it into an immediately-called function with an empty argument list $e :=$ `func () `τ` { `s` }()`. However, the latter additionally requires the type τ of the statement s.
3. Many more forms of statements and expressions are described, in so far as they are useful for translating Thingol's language features into Go.
4. The syntax used for generics differs in some aspects as Griesemer, Hu, et al. 2020 was itself meant as a proposal of generic types for Go, written before they were added to the language in Go 1.18 with a similar but not exactly the same design.

In the following, τ, γ and κ will always range over types, with τ_I denoting specifically an interface type, and τ_S a structure type. Type parameters are always written as α, expressions as e or d, and statements as s. The set of all declarations of a package will be referred to as $\overline{D}$. All packages live under the same base url u_{base}.

3.2. Declarations

A declaration D can be either a type declaration of a `struct` tuple or an `interface` type, a function declaration, or declare a constant. The order of declarations does not matter to the program; any constant defined in the same package may reference any other.

Structure types: a declaration of the form `type` t_S `struct`$\left[\overline{\alpha\ \ T_I}\right]\left\{\overline{A\ \ \tau}\right\}$ introduces a new type constructor with fields $\overline{A}$ of types $\overline{\tau}$ to the program. It may be polymorphic an take type arguments $\overline{\alpha}$ constrained by the interfaces $\overline{T_I}$; the $\overline{\alpha}$ can be freely referenced withing the $\overline{\tau}$ types.

However, in the fragment used by the code generator, no way to actually constrain an interface is included; thus in effect, all the $\overline{\alpha}$ are unconstrained type parameters which can be instantiated with any type, and all the $\overline{T_I}$ are the unconstrained `interface{}` (Section 4.7.1 will discuss why constraints by method are insufficient to represent type class constraints).

Interface types: a declaration of the form `type` t_I `interface`$[\overline{\alpha T_I}]$`{}` introduces a new interface type to the program.

In real-world Go code, such types are used to hold values of which nothing is known except that they fulfill the constraints given defined by the interface[2]. However, since in the given fragment all interfaces are the same unconstrained interface, all interface types, while distinct still from each other, are in effect "any" types which can hold values of arbitrary types. The type parameters $\overline{\alpha}$ thus become little more than decoration, used only to retain a sense of the translated Thingol constructions within the generated Go.

At this point it may seem worth asking: Why not use interfaces as more than annotated any-types? In principle, an interface in Go can contain two forms of constraints: contraints by *method*, and constraints by *type* (Go Team 2022). Unfortunately, neither of them is useful for our case: interface types are restricted to *basic interfaces*, that is, they can only be constrained by methods, not types. Unfortunately, as will be discussed in Section 4.7.1, we also cannot use methods to represent type classes. Thus only the bare unconstrained interface remains in our target fragment.

Functions: a declaration `func` $f F$ `{` s `}` introduces a new function called f with head F of the form $[\overline{\alpha\ \ \tau_I}]$`(`$\overline{x\ \ \tau}$`)` `(`$\overline{\gamma}$`)` to the program. The type parameters $\overline{\alpha}$ are nominally constrained by the interfaces types τ_I, and can be mentioned within both argument types $\overline{\tau}$ and the return types $\overline{\gamma}$

Unlike in Thingol, a function cannot have multiple equations, just one list of argument names contained in the head, which are all in scope for the only function body s. Type variables

An unusual feature of Go is its concepts of functions which return more than one value:

```go
func foo() (bool, int, string) {
  return false, 42, "bar"
}

func main() {
  x, y, z := foo()
}
```

Figure 3.1.: An example definition of a function which returns values of three different types, and a call to it from within `main`.

[2]e.g. most commonly that a given method is implemented for the inner value's type

At first glance this might seem analogous the tuples present in Standard ML, with `foo()` returning a single value of the tuple `(bool, int, string)`. But this is not the case, as Go has no concept of tuples. Instead, the function itself returns multiple values, which must be immediately assigned names (or discarded) at the function's call site. Thus a call like

```
func bad() {
  no_tuples := foo()
}
```

is not allowed given the above definition of the function `foo`.

Top-level variables: Variables can be declared at the top level of a program with a declaration of the form `var` x τ = e, binding the name x to the evaluated form of the expression e of type τ.

Since the fragment of Go used by the Code Generator does not contain assignments to already-defined variables (nor any other form of mutable variables) the name x can be taken to be a top-level constant, which can be used throughout the program.

Unlike top-level functions, variables declared in this way do not take any type parameters and can thus not be polymorphic.

3.3. Expressions

Expressions e can have several forms: variables, function application, and function abstraction are familiar from the polymorphic lambda calculus[3]. The others may require a bit more explanation:

Structure literal: a literal of the form $t_s[\overline{\alpha}]\{\overline{e}\}$ gives a value of the `struct` type with name t_S applied to type arguments $\overline{\alpha}$, i.e. it produces a new value of the type $t_s[\overline{\alpha}]$ in which the fields are set to the evaluated forms of the expressions $\overline{e}$.

Structure literals must always fully satisfy the fields of the structure type; partial application is not allowed. Note that the field names present in the declaration of a `struct` type are absent: while they could be used, Go does not require them. In the interest of shorter code, they are therefore omitted from the code generator's fragment.

[3]As with top-level functions, function abstractions declared within expressions can, in principle, return multiple values. However, this is never done in code generated from Isabelle, and is therefore elided here.

Field Selection: an expression $e.A$ selects the field named A of an expression e which has a fitting `struct` type τ_S which was declared with one of the field names set to A, and returns the value of that field.

Notably, this is the only place outside the structure type's declaration that the field name is ever required to be used.

Type conversion: an expression $\tau_I(e)$ evaluates to a value of the interface type τ_I which contains the evaluated form of e as its inner value.

The original type σ of e is kept and will not be erased at run time; it can be recovered using a type switch statement, see Section 3.4.

Zero value: the expression `*new(`τ`)` will always produce a value of type τ "out of nothing."

This is possible since Go defines a well-known *zero value* for all types possible to define (Go Team 2022).

- for booleans, the zero value is `false`
- for interface types, it is the special value `nil`
- for structure types, it is constructed recursively: the zero value of a structure type's value has all its field set to their zero values

In this work, the precise value returned by `*new(`τ`)` will never be important; the code generator only uses it in situations where the value it returns is never used, but a value is still required to satisfy a language construct.

3.4. Statements

As with expressions, statements can be one of many possible forms. The following only introduces those needed for Isabelle's code generator. All statements of the defined fragment will necessarily always end in a `return` and thereby return from the current function (unless they also abort the entire program). This is done so it is always possible to embed a statement back into an expression by wrapping it into an immediately-called function abstraction `func () `τ` { `s` }()`.

All forms except for the type switch should be immediately familiar from similar languages.

return: evaluates one or more expressions $\overline{e}$, then returns from the current function. The number of expressions given must match the number of return types given in the function's head.

if-Statement: a statement of the form `if` (e) `{` s_1 `};` s_2 will evaluate the e which most be of the built-in type `bool`. If it evaluates to the built-in value `true`, then s_1 is evaluated; since all statements must end in a `return`, it will then return from the current function. Otherwise, s_2 is evaluated and does the same.

This statement could thus equivalently be written as `if` (e) `{` s `}` `else` `{` s_2 `}`, which acts the same way. The version without the explicit `else` branch is preferred to limit nesting of statements in the generated Go code.

Type switch: a statement `switch` $(x$ `:=` e`.(type))` `{` `case` σ: s_1; `default`: s_2 `}` can be thought of as the reverse operation of type conversions (Section 3.3): for an expression e of an interface type τ_I, s_1 will be executed if the inner value contained within the interface value has type σ. If not, the default branch s_2 is executed instead.

Within the statement s_1, the variable x is bound to the inner value of e.

3.5. Packages

A set of declarations $\overline{D}$ must always live in a package, which may import other packages and thereby make their declarations available to use (under names qualified with the other packages' names) within the $\overline{D}$.

Packages are referred to by URLs; in general, the intention is that a package referred to by a URL u can be found via an HTTP request to u.

Within the fragment of Go used by the code generator, all packages live as subpaths under a common base URL u_{base}. This allows for convenient mapping of Isabelle's theories to packages, and to define a go *module* (a concept entirely distinct from a Go package) to allow for local compilation of all generated packages.

3.6. Built-in Functions

Generally, the shallow embedding of Thingol code into Go produces code with almost no dependencies on pre-existing constructs in Go other than the bare language features; without the adaptation layer described in Section 5, the translation "brings everything it needs," even basic constructs such as boolean types or negation.

In fact, there are only two function names we assume exist on the Go side, both of which are built-in and in scope by default in every Go package that can be defined:

- the `panic` function, which will abort the program immediately with an error message, is used to abort translations of `case`-expressions in which no clause has matched, and to implement functions with no equations.
- the boolean operator for conjunction `&&` is used to reduce nesting in translations of pattern matches, as discussed in Section 4.5.

4. Code Generation

4.1. Names

4.1.1. Renamings

Identifiers in Go may contain any Unicode characters, but must start with a letter (Go Team 2022). Since this is a superset of the names allowed in Thingol, it might appear names can be kept unchanged. But a few adjustments do have to be made:

Reserved words: Names which happen to also be reserved words in Go (but not in Isabelle) must be renamed.

Top-level Identifiers: The initial letter of all top-level identifiers in all modules is transformed to be uppercase (if it is not already).

This is done since an identifier's first letter carries meaning in Go: if it is uppercase, it signifies this constant should be *public*, i.e. it will be exported and made available to other packages if they import the package the constant is defined in. Names starting with a lowercase letter are not exported.

Since the code generator will, whenever possible[1], preserve the module structure of the given Thingol code by creating an individual Go package for each Thingol module, this is done for *all* names, even those not explicitly listed in the **export_code** command.

Names bound within terms as well as type variables are not touched by this renaming.

Names reserved by the generator: To implement destructors for data types, the Go Generator creates additional top-level functions, whose names end in `_dest`. User-defined names which share this postfix must thus be renamed to avoid collisions.

This renaming scheme is applied by Isabelle's code generation framework during the translation to the intermediate Thingol representation. In the following sections, it is thus always safe to assume that all names of top-level constants start with an uppercase letter, and that no name will collide with a reserved word.

[1]Unfortunately, circular imports are not possible in Go.

4.1.2. Optional names

Variable names are optional in Thingol[2], and can be omitted in function arguments or in the patterns of a `case` expression, indicating that this value is discarded and not used.

Whenever this is the case, such names are translated to Go's blank identifier "_".

4.2. Types

$$\tau ::= \kappa \ \ \overline{\tau}_n \ \mid \ \alpha :: s$$

Given a type τ in Thingol, we can construct its translation $\text{type}(\tau)$ in Go as follows:

Bound names if τ is a bound name α, its name is kept[3] in Go, and $\text{type}(\tau)$ is thus the generic type variable with the same name.

If τ is a composite type $\kappa \ \ \overline{\tau}_n$ which applies a constant type with name κ to type arguments $\overline{\tau}_n$, its translation is a (potentially generic) type, with $\text{type}(\tau_i)$ as its generic type arguments. Note that if $n = 0$, i.e. there are no type arguments, then the type argument list must be omitted on the Go side.

$$\text{type}(\alpha :: s) = \alpha$$

$$\text{type}(\kappa) = \kappa$$

$$\text{type}(\kappa \ \ \overline{\tau}_i) = \kappa\texttt{[}\overline{\texttt{type}(\tau)_i}\texttt{]}$$

The only types integral to the code generator's functioning which require further special handling are function types, which are translated into equivalent types of function pointers in Go. Thus, a function type $\tau \to \gamma$ becomes `func (type(`τ`)) type(`γ`)` in Go. No effort is made to uncurry curried function types: while generally not idiomatic, curried functions in Go work as they do in Thingol[4].

Of course, further adaptations can be made to the translation of types: for example, it can be desirable to translate the number and string types used in Isabelle to those habitually used in Go, greatly improving performance. Such adaptations are not built-in to the generator, but use the **code_printing** facilities provided by the code generator. A basic example of such adaptions for Isabelle/HOL is described in Chapter 5.

[2]i.e. they are represented by a value of `vname option`.

[3]of course transformed by the renaming scheme described in Section 4.1.

[4]This is in contrast to functions bound at a module's top-level, which the generator does uncurry, compare Section 4.6. Fortunately, we never need to explicitly print their types.

In the remainder of the chapter, we will informally equate Thingol types τ with their Go translation $\mathrm{type}(\tau)$ and write both simply as τ. It will of course always be clear from the language τ is used in which of the two is meant.

4.3. Terms

$$t ::= f[\overline{\tau}] \;\mid\; x :: \tau \;\mid\; \lambda x :: \tau.\;\; (t :: \rho) \;\mid\; t_1 \;\; t_2 \;\mid\; \text{case } t :: \tau \text{ of } [\overline{p \to b}]$$

Figure 4.1.: Definition of Thingol's terms.

Terms are not quite as easily dealt with: instead of just one translation schema $\mathrm{type}(\tau)$, we need two mutually recursive ones called expr and stmt.

The first is meant to produce Go expressions, the second (possibly multiple) Go statements ending in a `return`; the intended relationship between these two functions can be taken as:

$$\mathrm{stmt}(t) \equiv \texttt{return expr}(t)\texttt{;}$$

$$\mathrm{expr}(t) \equiv \texttt{func() } \tau \texttt{ \{stmt}(t)\texttt{\}()}$$

Of course, the above cannot be taken as an actual definition; it is merely meant to suggest the semantics of the code which both functions produce. In fact, in the interest of readability of the generated code, neither of the above equivalences is actually true.

In the following, often only one of stmt or expr will be defined explicitly for a language feature of Thingol. It should then be understood that the other is implicitly defined using the appropriate equivalence above.

4.3.1. Constants

$$t ::= f[\overline{\tau}_n]$$

Since even a bare constant $f[\overline{\tau}_n]$ (be it a function name or data constructor; in Thingol, these are treated the same) may carry type arguments $\overline{\tau}_n$ or make use of a type class instance, which will need to be translated into an explicit dictionary value passed to it as an argument (as will be discussed in Section 4.7.2), it is translated the same as an application with an empty argument list.

Conversely, top-level constants which are not functions must of course always be translated as functions into Go, even if these then have an empty argument list (see Section 4.6).

There is one exception: constants meant to abort the program if encountered are handled seperately and translated directly to an invocation of `panic` (see Section 5.2).

4.3.2. Bound Variables

A variable $x :: t$ with name x is translated to the variable with the same name in Go:

$$\text{expr}(x :: t) = x$$

Variables used in terms (and not patterns as in Section 4.5) must always have names; if the term _ is ever encountered, this implies an internal error in the earlier translation to Thingol, and code generation will abort.

4.3.3. Abstractions

The translation of a lambda-abstraction $\lambda(x :: \tau).\ (t :: \gamma)$ is again straightforward:

$$\text{expr}(\lambda(x :: \tau).\ (t :: \gamma)) = \texttt{func}\ \texttt{(}x\tau\texttt{)}\ \gamma\ \texttt{\{stmt(}t\texttt{)\}}$$

As with function types (see Section 4.2), no effort is made to uncurry nested abstractions.

4.3.4. Applications

Unfortunately, applications $t_1 \quad t_2$ are more tedious to translate:

Uncurrying: since the definitions of top-level functions are uncurried (see Section 4.6), it must first be checked if the term is a (potentially curried) application to a constant, i.e. if $t_1 \quad t_2$ has the shape $(\cdots((f[\overline{\tau}_i] \quad a_1) \quad a_2)\cdots) \quad a_n$ where f references a top-level function taking m arguments.

If it is not, then the translation is very simple:

$$\text{expr}(t_1 \quad t_2) = \texttt{expr(}t_1\texttt{)(expr(}t_2\texttt{))}$$

If it does, the translation becomes a lot more involved.

η-Expansion: if n is less than m, the entire term is η-expanded until the application is fully satisfied.

Immediate arguments: a_1 to a_m will be called the application's *immediate* arguments. Any remaining a_{m+1} to a_n are *curried* arguments, which are each applied separately.

In effect, this assumes that $f[\overline{\tau}_i]$, when called, will return a (curried) function; since the intermediate representation is itself well-typed, this is insured to always be the case.

Type arguments: although Go can often infer generic type parameters during compilation, this is not always the case. In particular, the way data types are translated make it impossible, since all data types are translated as equivalent unconstrained `interface{}` types (see Section 4.4).

Type class dictionaries: as described in Section 4.7.2, the dictionary-translation method used to represent Isabelle's type classes and locales in Go may introduce additional (value-level) parameters for a function, and corresponding additional parameters d_1 to d_r to each application of the same function. As described in Section 4.6, these are inserted in front of the user-defined parameters $\overline{a}_n$.

Functions: Altogether we arrive at the following scheme in case c is a top-level function:

$$\text{expr}(t_1 \;\; t_2) = f\,[\tau_1,\ldots,\tau_i]\,(d_1,\ldots d_r,a_1,\ldots,a_m)\,(a_{m+1})\ldots(a_n)$$

Data type constructors: if f references a data type constructor for type κ, the translation is slightly different

1. Since the individual constructor is represented as a `struct`, its arguments must be put within curly braces instead of brackets.
2. Since the data type δ itself is represented as an `interface` type which may contain any of a list of individual `struct` types corresponding to each of its constructors, the entire call must be wrapped into a type conversion to type φ:

$$\text{expr}(t_1 \;\; t_2) = \kappa(f\,[\tau_1,\ldots,\tau_i]\,\{d_1,\ldots d_r,a_1,\ldots,a_m\})\,(a_{m+1})\ldots(a_n)$$

4.4. Data Types

$$\texttt{data}\ \kappa\ \overline{\alpha}_k\ \texttt{=}\ f_1\ \texttt{of}\ \overline{\tau_1}\ \texttt{|}\ \cdots\ \texttt{|}\ f_n\ \texttt{of}\ \overline{\tau_n}$$

Figure 4.2.: Definition of Thingol's data types.

A data type κ defined in Thingol consists of a list of names for type parameters α_1 to α_k as well as a list of constructors f_i.

Since Go has no comparable concept of types with multiple (or no) constructors, each f_i will be translated into its own separate `struct` type φ_i. If the number of constructors is exactly 1, then c_i is also used to translate the type κ itself; if it is not, an additional interface type is generated for δ to represent values of any of the constructors.

As discussed in Section 3.2, all interface types in the code generator's fragment of Go are the same (up to their type parameter lists) and function only a kind of annotated "any"-type which can hold values of an arbitrary type.

Within Go, there is thus no language-based guarantee that a value x of type κ was actually constructed using one of the constructors f_i, i.e. that the inner value of x actually is of one of the types φ_i.

Since the generator operates only on (assumed to be type-correct) intermediate Thingol code, it will still never produce erroneous code; however, programmers must be careful not to pass values of a wrong type when writing Go code meant to interact with the generator's output.

4.4.1. Constructors

Constructing a `struct` type for an individual constructor is very simple: a constructor f with fields of types τ_1 to τ_i is translated into Go is simply a struct with the same name and fields:

$$\texttt{type}\ c\ \texttt{struct}\ \{\overline{A\ \ \tau_i}\}$$

where the $\overline{A}_i$ are newly-invented names for each of the fields, as no field-names are present in Thingol.

4.4.2. Destructors

Along with each constructor's `struct` type, the translation generates a *destructor* function f_dest which will be used as a helper function in the translation of Thingol's `case`-expressions described in Section 4.5.

```
func f_dest (p κ) (bool, τ1, ..., τn) {
  switch q := p.(type) {
    case c:
      return true, q.A1, ..., q.An
    default:
      return false, *new(τ1), ..., *new(τn)
  }
}
```

The above uses two somewhat unusual features of Go which are not present in Thingol which were mentioned in Chapter 3:

- Functions with multiple return types: as is idiomatic in Go, the first return value indicates whether the operation was successful (here that means the interface type's inner value matched the constructor's structure type). If it did, the function will return all the fields of this value's constructors, which are then be bound to local names at the destructor function's call site.

- If the constructor could not be matched, the first return value will be `false`. However, to satisfy the function's signature, all the other return types must equally be given values, even if these will never be used. Thus for all other return types, the function returns its zero value, which can always be synthesised "out of nothing".

4.4.3. Field Names

Since Go allows constructing a value of a structure type without explicitly giving the field names explicitly (compare Section 4.3.4), and Thingol's only way to access a field's values is to bind it to a new name using a `case`-expression, which is translated using the destructor functions $\overline{f\_\text{dest}_i}$ (see Section 4.5), the names $\overline{A_i}$ are never used in code other than the structure type's declaration and its destructor function.

They are thus entirely unimportant; the only requirement imposed on them is that each $\overline{A_i}$ of the same `struct` is distinct and starts with an upper-case letter, to ensure that the field is accessible when imported into another Go package.

4.5. Case expressions

Thingol's `case`-expressions implement pattern matching on a value, in a way which will be immediately obvious to readers familiar with other functional languages such as Standard ML or Haskell: they inspect a term t called the expression's *scrutinee* and match it against a series of constructor patterns $\overline{p}_n$. As a whole, the term then evaluates to the term b_i of the first matching pattern p_i. If none of the clauses match, the program will abort as a whole.

$$\texttt{case}\ t :: \tau\ \texttt{of}\ \left[\overline{p \to b}_n\right]$$

Figure 4.3.: Thingol's case expressions.

Each clause $p_i \to b_i$ consists of two terms: the first is its*pattern*, the second the term to be evaluated if p_i matches the target value t.

Types of Patterns: The $\overline{p}_n$ may only contain three kinds of terms:

- a variable name x.
- a data type constructor c of a data type κ by itself.
- a fully satisfied application of a data type constructor c (but not a variable name) with arity k to other patterns $s_1, \ldots, s_k$.

The latter two kinds we will call *proper patterns*.

Since Go has no comparable feature, a proper pattern in an `case`-expression is translated into a series of (possibly nested) `if`-conditions and calls to destructor functions. The body of the innermost `if`-condition is stmt(t), which ends in a `return`-statement; thus if the pattern could be matched, further patterns will not be executed[5]. If the pattern did not match, execution will continue with either the next block of `if`-conditions generated from the next clause, or encounter a final catch-all call to Go's built-in `panic` function, which aborts the program in case no clause could be matched.

4.5.1. Let-Bindings

The simplest `case`-expression consists of a single pattern p which itself consists of a single variable name x. In effect, it simply binds t to a new name, as a `let`-binding would in other languages.

Thus we can directly generate the corresponding variable declaration in Go:

$$\text{expr}(\text{case } t :: \tau \text{ of } x \to b) = \{x \;\; \texttt{:=} \;\; \texttt{expr}(t)\texttt{;} \;\; \text{stmt}(b)\}$$

4.5.2. Proper Patterns

$$p = f\ [\overline{\tau}_i] \quad [\overline{s}_k]$$

A proper pattern consisting of either just a constructor f or of f applied to sub-patterns s_1 to s_k deconstructs a value of type κ.

Constructors: If all sub-patterns $\overline{s}_k$ are variable bindings, then checking whether the constructor matches the given value using the deconstructor function f_dest is straightforward:

$$\text{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}m\texttt{,}A_1\texttt{,}\ldots\texttt{,}A_k\texttt{:=}\ f\texttt{\_dest(}t\texttt{);}\ \texttt{if}\ \texttt{(}m\texttt{)}\ \texttt{\{stmt(}b\texttt{)\}\}}$$

[5] of course, using `return` in this manner implies that a `case`-expression must always either stand at the tail of a function, or else be wrapped into its own function if it does not.

For any of the sub-patterns $\overline{s}_k$ which are variable names, we can set the corresponding $\overline{A}_k$ to the same name. Nothing more needs to be done for these; within the pattern's body, these are now properly in scope (if any of the variables use the blank name, it is mapped to the blank identifier "_" in Go, which likewise introduces no new variable).

Proper sub-patterns: It then remains to check that all the sub-patterns $\overline{s}_k$ which are proper patterns likewise match their target value which is now bound to the (newly invented) corresponding variable names $\overline{A}_k$.

These are translated in the same way, but "pushed inwards" into the body of the `if`-statement generated above:

$$\text{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}m\texttt{,}A_1\texttt{,}\ldots\texttt{,}A_k\texttt{:= }f\texttt{\_dest(}t\texttt{); if (}m\texttt{) \{}\mathit{inner}\texttt{\}\}}$$

where

$$\mathit{inner} = \text{stmt}(\overline{\text{case } A \text{ of } s \to}_k \ \ b)$$

i.e. the sub-patterns are treated as if they were further nested case-expression withing the outer one's body.

Within the innermost `if`, the body b of the pattern's clause is put as $\text{stmt}(b)$, to ensure it returns from the current function if it was reached.

4.5.3. Reducing the number of nested `if`

The above is already enough to translate arbitrary patterns, but at the price of potentially exponential code blow-up: even a single pattern consisting of just a constructor and k fields, none of which are proper patterns, will still produce k levels of nested `if`-statements; if instead the fields themselves are again data type constructors with sub-patterns, the number of nested levels quickly increases further.

While this blow-up is unavoidable in general, it can still be reduced: for s_i which consist of just a data type constructor (i.e. a constructor which has no fields), its call to the destructor function and `if` can be replaced by just a check for equality, inserted into the next-outer `if`.

$$\text{check}(s_i) = (s_i \ \texttt{==} \ A_i)$$

$$\text{checks}(p) = \texttt{check}(s_i) \ \texttt{\&\&} \ \ldots$$

$$\text{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}m\texttt{,}A_1\texttt{,}\ldots\texttt{,}A_k\texttt{:= }f\texttt{\_dest(}t\texttt{); if (}m\texttt{ \&\& checks(}f[\overline{\tau}_i][\overline{s}_k]\texttt{)) \{}\mathit{inner}\texttt{\}\}}$$

The worst-case of potential code blow-up for deep patterns of course remains the same; however, in most "real-world" code, such situations are rare, and the above helps to limit the size of the generated code.

4.6. Top-level Functions

Unlike lambdas that occur within terms, top-level functions in Thingol can have multiple clauses and pattern-match on their arguments, neither of which is supported in Go. It is thus necessary to translate them differently: all equations of the same function will have to be merged, with the pattern matching on their parameters again "pushed inwards" into the then combined, single function body.

Further, treating them differently from in-term lambda expression also allows the generator to uncurry them, creating code that is much closer to an idiomatic style in Go.

```
fun $f$ :: $\forall\overline{\alpha :: s_k}.\tau_1 \to \cdot \to \tau_m \to \gamma$ where
  $f$ $[\overline{\alpha :: s_k}]$ $\overline{p_1}_m$ = $t_1$
| ...
| $f$ $[\overline{\alpha :: s_k}]$ $\overline{p_n}_m$ = $t_n$
```

Figure 4.4.: A top-level function in Thingol, with multiple equations.

Type Arguments: while kept almost as they are in Thingol, all information on the sorts $\overline{s}_k$ of type parameters $\overline{\alpha}_k$ (that is, their type class constraints) is lost entirely; every parameter is translated as an unconstrained `interface{}`.

Instead, type classes are translated via the explicit dictionary-construction method (see Section 4.7.2).

Merging multiple clauses: All parameters of functions must have names in Go, and be annotated with their type. Thus if any of the parameters patterns $\overline{p_i}$ is a proper pattern, a fresh name x_i for it is invented. Likewise, if a parameter is a variable binding instead of a proper pattern, but has multiple different names in two clauses, the name x_i used in the first clause is picked as the name of the parameter in Go.

Additionally, further arguments $\overline{d}_j$ with types $\overline{\delta}_j$ not present in Thingol may be introduced by the dictionary construction used to represent type classes, as described in Section 4.7.2.

Pattern matching: The combined function body then consists of a pattern match translation as described in Section 4.5[6]. Each equation is then treated as a clause of an imagined case-expression; since functions can pattern match on multiple parameters,

[6]The already-existing Scala target uses a similar transformation.

we again "push inwards" and translate as if a nested series of `case`-expressions were present.

$$\begin{array}{l}
\texttt{func}\ f\texttt{[}\overline{\alpha_k}\texttt{]}\texttt{(}\overline{d\ \delta_j}\texttt{,}\ \overline{x\ \tau_m}\texttt{)}\ \gamma\ \texttt{\{} \\
\quad \texttt{stmt}(\overline{\texttt{case}\ x\ \texttt{of}\ p \rightarrow}_{m}\ t_1) \\
\texttt{\}}
\end{array}$$

Figure 4.5.: A function with only one equation, translated using pattern matching.

4.6.1. Top-level Variables:

The attentive reader may have wondered what happens with Isabelle definitions such as

definition *a* :: *nat* **where**
  a = *10*

or more generally

definition $t :: \tau$ **where**
  $t = \ldots$

that is, top-level declarations which define constants that are *not* functions. Unfortunately, there is no better solution[7] than to treat these as top-level functions which take no arguments at all (i.e. an empty list), which must be called explicitly each time they are used.

While this changes nothing of the semantics of the translated program, it does incur a (potentially significant) runtime cost.

4.6.2. Empty Functions

A particular oddity of Thingol is that it allows functions which have *no clauses at all*. An example familiar to anyone who has worked with Isabelle/HOL is **HOL.undefined**.

In such cases the generator will produce a function with an empty argument list, containing nothing but a call to Go's built-in `panic()`, which will abort the program.

[7]Go does allow top-level variable declarations, but only for monomorphic types, and it disallows function calls in their definitions.

4.7. Type Classes

4.7.1. Why Interfaces are insufficient

It will initially appear tempting to translate type classes directly into `interface`s, since they share many of the same features, and are sometimes considered near-analogous features (for example in Ellis et al. 2022).

Strikingly, both can be used to constrain type arguments by requiring a set list of functions to be available for a polymorphic type, regardless of which concrete type it will be instantiated with.

Unfortunately, this is not possible, as functions in an `interface` are limited to *methods*, which must dispatch on an (explicit) value (Go Team 2022)—thus they must always take at least one value-level argument of the type the interface is associated with. A type class in Isabelle such as

class *foo* =
  fixes *foo* :: *unit* ⇒ *'a*

or even

class *bar* =
  fixes *bar* :: (*'a* ⇒ *'a*) ⇒ *unit*

is therefore not easily translatable using interfaces, as the methods foo and bar lack a parameter of type *'a* to dispatch on.

4.7.2. The Dictionary Translation

Instead, we use Thingol's provision for languages with no feature equivalent to Isabelle's type classes. This resolves all type class constraints during translation and replaces the implicit dictionaries of functions given by type classes by explicit values of dictionaries, which are represented as one data type per type class.

This translation is already integrated into Thingol, and is also used in the generator for Standard ML (Haftmann and Nipkow 2010); thus only relatively few things are left for the Go generator to do:

1. declare a data type for each type class, called its *dictionary* type
2. translate type class constraints on functions into explicit function arguments of dictionary types
3. translate type class instances into either a value of the type class's dictionary type, or, if the instance itself takes type class constraints, to a function producing such a value when given values of dictionary types representing these constraints.

4. any time a top-level function is used, the already-resolved type class constraints must be given as explicit arguments (see Section 4.3).

4.7.3. A Dictionary type

```
class c ⊆ c_1 ∩ ··· ∩ c_m where
  g_1 :: ∀α :: c.  τ_1,   ..., g_n :: ∀α :: c.  τ_n
```

We can translate a class c into a `struct` type which has one field for each super class $\overline{c}_m$, and one field per each class parameter $\overline{g}_n$. Types of the fields $\overline{c}_m$ are themselves dictionary types which represent the super classes; since these are translated the same way into top-level type names, it suffices to put each class's name.

```
type c[a] struct {
  c_1  c_1[a],
  ...
  c_m  c_m[a],
  g_1  τ_1,
  ...
  g_n  τ_n,
}
```

The types τ_j for class parameters are again uncurried in the same manner as those of top-level functions:

- if τ_j is already a function type, it is uncurried
- if it is not already a function type, `func()` τ_j is used instead

This allows translating the individual class parameters used in instances in the same way as other top-level functions (see Section 4.6), as their types will match.

4.7.4. A Dictionary value

```
inst $\kappa$ $\overline{\alpha :: s_k}$ :: $c$ where
  $g_1$ [$\kappa$  $\overline{\alpha :: s_k}$] = $t_1$,  ...,  $g_n$[$\kappa$  $\overline{\alpha :: s_k}$] = $t_n$
```

Figure 4.6.: An instance gives values for the instance parameters g_j for a concrete constructor κ. Through the $\overline{s}_k$ it can implicitly depend on instances for the types $\overline{\alpha}_k$. Additionally, in instance requires instances for κ of all superclasses $\overline{c}_m$ of c.

All terms in instance parameters are translated as their own top-level functions as described in Section 4.6; to translate an instance, it thus only remains to assemble an appropriate value of its dictionary type c_κ.

Top-level binding: If the type constructor κ takes a non-zero number of arguments $\overline{\alpha}_k$, then the instance cannot be translated as a single constant, as its value would not be monomorphic; instead the instance is translated as a function, which must be given explicit type arguments to produce the required (then monomorphic) value. Further, if the sorts $\overline{s_k}$ of the $\overline{\alpha}_k$ contain any type class constraints $\overline{\delta}_j$, these must be added as arguments to that function, since they are required to satisfy type class constraints of instance parameters.

Instance Parameters: The fields $\overline{g}_n$ are set to their corresponding function names. If any of these functions themselves take dictionary arguments, these must be partially applied (and thus the whole term η-expanded) to yield a term of the required type $\overline{\tau}_n$.
The only dictionary types which can occur here will be of the types $\overline{\delta}_j$, thus fitting values are always available.

Super classes: Conversely, the fields $\overline{c}_m$ must be set to a value of the super class's dictionary type $\overline{c\_\kappa}_m$. Since the corresponding instance is translated in the same way, such a value will be available either as a top-level constant c_m_κ, or (if this instance was translated into a function) by calling the top-level function c_m_κ with appropriate arguments.

```
var c_κ = c[κ] {
  g₁: g₁, …, gₙ: gₙ,
  c₁: c₁_κ, …, cₘ: cₘ_κ,
}
```

```
func c_κ[$\overline{\alpha \text{ interface\{\}}}$]($\overline{x\ \delta_j}$) c[κ[$\overline{\alpha}$]] {
  return c {
    g₁: g₁, …, gₙ: gₙ,
    c₁: c₁_κ, …, cₘ: cₘ_κ,
  }
}
```

Figure 4.7.: Translation of class instances as top-level variable and function.

4.8. Unused Variables

Unlike most other languages which will (at most) warn the programmer if a variable was declared but then never used in subsequent code, Go strictly prohibits any unused variable declarations, and will abort compilation if any are present. There is no option to relax this behaviour (Go Team 2023); however, it suffices to "use" a variable n by assigning it once to the blank identifier, which otherwise has no effect:

```
n := ...
_ = n
```

Figure 4.8.: Go will consider n as used in the above.

Thus code generation must always be careful to either never introduce such an unused variable (lest its output will be useless without manual edits), or if it does, additionally generate such an assignment to the blank identifier.

Luckily, in almost all cases, unused name bindings are already marked as such by in the generated intermediate Thingol[8].

This is not the case for argument names of functions. Here the translation has to traverse the function's body once to collect all variables used therein, and then replaces all unused argument names by the bank identifier in the generated code.

[8]i.e. their variable name will be set to `NONE`.

4.9. Packages

The structure of Isabelle Theory files is preserved in Thingol as *modules*, which the generator maps to Go's *packages*. Perhaps confusingly, Go also has a concept of a *module*, which is not related: while a package defines constants in its own namespace which can be imported into other packages, a module groups several packages under a common URL.

Again there is not much for the Go Generator to do: the graph of imports is already resolved in Thingol. The generator merely has to produce one Go source file for each of Thingol's modules and translate the list of imports to import statements in Go.

Since during translation all top-level names were changed to be upper case and thus public (see Section 4.1), no further specification of which names precisely are to be imported from a package have to be made.

The only slight complication is that Go does not allow importing paths relative to the current source file, i.e. all imports must be referred to by an absolute URL, with local packages being assigned a URL by the Go module of which they are part.

Thus if constants from several Isabelle different theories are exported into separate Go, as in the following example, invoked in an imagined theory file `Addition.thy`:

fun *add* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
  add $x\ y = x + y$

export-code *add* **in** *Go* (*go-module example.org/isabelle*)

the generator will additionally create an appropriate `go.mod` file, with the common base URL of the created packages either set via the **go_module** option passed to the generator or (if the option is not given) to the default dummy value `isabelle/exported`.
The above will thus produce a `go.mod` file as follows:

```
module "example.org/isabelle"

go 1.19
```

Figure 4.9.: An example `go.mod` file.

The generated module then contains two pacakges: one is `example.org/isabelle/Addition`, corresponding to our source file, which itself imports the definition of natural numbers from a second exported package `example.org/isabelle/Nat`:

```go
package Addition

import (
  "example.org/isabelle/Nat"
)

func Add (x Nat.Nata, y Nat.Nata) Nat.Nata {
  return Nat.Plus_nat(x, y);
}
```

Figure 4.10.: The `Addition` package, as generated into the file `Addtion/exported.go`. The `Nat` package will be generated into the file `Nat/exported.go`. The `go.mod` file ensures that the Go tools will understand the relationship between both.

5. Native Code Adaptations

To improve readability, efficiency and especially interaction with other code in the target language that is not generated from Isabelle, the Code Generator supports an *adaptation layer* which can be used to explicitly map specific types, data constructors, and functions directly to corresponding constructs of the target language through a list of *printing rules*.

Of course, such adaptations imply that any guarantee of correct-by-translation code is necessarily abandoned: it is trivial to write a printing rule which causes the generator to emit code which does not behave as it should, or even code that is syntactically invalid in the target language!

However, compared to the pre-existing target languages, the capabilities of such printing rules are more narrow when generating Go code: the **code_printing** machinery does not know about the distinction between expressions and statements in Go. Adding such support would require reworking how such rules are parsed, and how they interact with the Standard ML code which implements the existing target languages.

Thus nothing of the kind was done, and printing rules in Go are (for now) limited to target language constructs which use only expressions, not statements.

5.1. Isabelle/HOL

Thus the following only describes a very basic adaptation layer to use code generated from Isabelle/HOL theories in Go.

Booleans: the type `HOL.Bool` is printed as Go's built-in `bool` type, and constants `HOL.False` and `HOL.False` as `true` and `false` respectively. Boolean operators can then be printed as their Go counterparts:

- `HOL.conj`, `HOL.disj`, `HOL.equal`, and `HOL.Not` as `&&`, `||`, `==`, and `!` respectively.
- Since Go has no built-in operator for boolean implication, terms `HOL.implies` a b are printed as `(expr(`a`) || !(expr(`b`)))`
- By contrast, Isabelle's if-then-else expressions are not handled specially, and are still translated by using the pattern match translation given in Section 4.5,

as translating them using `if`-`else`-statements would require printing rules that generated Go statements.

Unit: the unit type is printed as the empty struct type `struct{}`, and its only value to the empty struct's only value `struct{}{}`.

Strings & Characters: the type `String.literal` is printed as the built-in type `string`:

```
code-printing
  type-constructor String.literal ⇀ (Go) string
| constant STR '''' ⇀ (Go)
| constant plus :: String.literal ⇒ - ⇒ - ⇀
    (Go) infix 6 +
```

Literal string values can then be printed as their (appropriately escaped, so as to not interfere with Go's syntax) versions.

```
setup ‹
  fold Literal.add-code [Go]
›
```

Fundamental operations must then also be adapted: equality is mapped to the built-in equality operator ==, string concatenation to the built-in +, etc.

5.1.1. Comparison to other target languages

The above-described adaptations are usable for many small-to-medium sized formalisations which are limit themselves to relatively basic types.

But compared to the adaptation layers for other target languages which come included in Isabelle, its scope is very limited, and for many types there are no printing rules at all:

- While it would be possible to translate tuples as anonymous structure types with exactly two fields, doing so in Go would require type annotations in Go: for example, a pair `(false,"foo")` could be translated as `struct{A:bool,B:string}{false,"foo"}`. Unfortunately, code printing rules do not currently support printing types.
- For now, even support for mapping integers to the arbitrary-precision integers provided by the Go `math/big` package is missing, as it is not strictly required for the code generator tests discussed in Chapter 6. However, adding such support should be easily possible.

- Lists are likewise not supported, as Go lacks a suitable built-in type for linked lists.
- The same holds for all of the formalisations included in `HOL-Library` as well as `Imperative-HOL`.

Unfortunately, this means the adaptation layer is in a certain sense *incomplete*: it is possible to write formalisations using Isabelle/HOL which can be successfully exported to any of the pre-existing target languages, but for which translation to Go will either produce very inefficient code or even—for example if it uses a function that was marked as **[[code drop]]**, and is represented via a printing rule in other languages, but not yet in Go—fail outright.

5.2. Program-terminating Functions

A special provision is made in the generator to handle *program-termination* functions, that is, functions which never return but instead abort the program. An example is `String.Code.abort` or `HOL.undefined`.

The usual way of printing these into a target language is to replace them with whatever built-in equivalent exists, for example Haskell's `error` or `undefined`. This is especially convenient since (in all previous target languages) these functions have an unconstrained polymorphic type, and can thus be used anywhere within a term.

Unfortunately, not only is this not true of Go's built-in `panic`, but no suitable function can easily exist in Go: while functions can return an "unconstrained" generic type, using such a function would require a type annotation, which a printing rule unfortunately cannot provide.

Worse, the built-in `panic` returns *no values at all*, which is distinct from a function returning a unit value of a unit type: Go does not allow such functions in expressions at all, only in statements.

To work around this, the generator takes an optional argument **panic_on**, to which a list of function names can be supplied which, if called, should abort the program. Instead of the usual translation scheme for applications of constants described in Section 4.3, it will then generate a direct call to `panic` as its own statement, along with (if necessary) a series of assignments to the blank identifier as described in Section 4.8 for any variables used in the function's arguments, to ensure they will not become unused.

6. Evaluation

Viewed as a whole, the process of generating code from an Isabelle formalisation can seem a little unsatisfying: usually, we begin by writing pristine code in Isabelle, where we can formalise and prove (almost) any properties of the code's behaviour that had intended. We can then be sure that the code is, as-written, actually *correct*, in the sense that even if it might show undesired behaviour, at least the theorems we *have* proved over it will always hold true.

But then in the next step, this certainty is lost: any confidence in the generated code's complexity rest solely on the correctness of the translation, i.e. on whether or not a particular part of the code generator contains a bug. In an ideal world, we would have a formal proof that the translation is itself correct—but the lack of a clear semantics for either Thingol or Go make it impossible to use anything but informal reasoning in its place.

Thus this section will look at several ways of attempting to empirically *test* the generator's output: via a unit-test approach on an already-existing code base which had previously been used to generate Scala code in Section 6.1, via the `HOL_Codegenerator-Test` session in Section 6.2, and finally by using Isabelle's **nitpick** tool to automatically generate test cases from theorems which were proven to hold over the Isabelle code in Section 6.3. The last is a novel and still experimental approach, and not specific to Go as a target language.

6.1. A Case Study

The code generator was used to translate an internal theory development at *Giesecke +Devrient advance52* based on Isabelle/HOL into Go. It had previously been used to generate Scala code; additional wrapper code had also been written in Scala, along with 10 unit tests checking for basic functionality.

As a simple evaluation of Go code generated from the same Isabelle theories, these unit tests and the necessary wrapper code were re-written in Go, where they produced equivalent results, and no bugs in the code generator or unintended behaviour of the code it produced were found.

However, the task of porting the wrapper code from Scala proved to be error-prone: many explicit type annotations are needed in the code (in particular, every usage of a data type constructor requires at least one, compare Section 4.4), and not all mistaken type annotations will cause compilation of the wrapper code to fail. Instead, if a data type's constructor is annotated with a different `interface` type, the assumption underlying the translation of case-expressions will fail, resulting in an "match failed" error at runtime.

6.2. HOL Codegenerator-Test

Isabelle's distribution contains a `HOL_Codegenerator-Test` session which is used as a self-check for the various target languages of the code generator. It contains two forms of checks:

- An integration with the **value** command to evaluate terms within a running Isabelle session using the code generator with a different target language. Additionally, a **test_code** command is provided, which executes individual test cases in the target language.
- A single **export_code** command meant to "export all of HOL" to the target language, as a stress-test for the code generator. This does not include any checking for correctness of the generated code; the only tested property is that the generator's output is accepted by the target language's compiler.

6.2.1. Integration into Isabelle

Until now, everything described in this work could exist outside of the Isabelle distribution itself, and indeed had been written as an entirely out-of-tree development. However, using the `HOL_Codegenerator-Test` session requires the target language to be available as part of `HOL` itself.

Thus a version of the Isabelle distribution, forked to include the Go target language and rudimentary support for interacting with the Go compiler[1] has also been created as part of this work.

6.2.2. test_code

The **test_code** command defined in `HOL-Library.Code_Test` can be used for quickly checking that neither the code generator nor an adaptation-layer printing rule went wrong in a particular case.

As an example, the invocation

[1]Assumed to have been installed independently; no Go component for Isabelle has been written.

test-code $14 + 7 * -12 = (140\ div\ -2 :: integer)$ **in** *Go*

Would fail if the given arithmetic equation would not hold true on the Go side. **test_code** accepts any term of type `bool`, which describes an assertion. It then checks that the assertion still holds "on the other side", i.e. that after translation into a target language the term still evaluates to `true`.

Further, the `HOL_Codegenerator-Test` contains a simple test for each language, which exports a *gcd* function computing the greatest common divisor of two natural numbers, and checks several test cases for it.

test-code
$gcd\ 15\ 10 = (5 :: integer)$
$gcd\ 15\ (-\ 10) = (5 :: integer)$
$gcd\ (-\ 10)\ 15 = (5 :: integer)$
$gcd\ (-\ 10)\ (-\ 15) = (5 :: integer)$
$gcd\ 0\ (-\ 10) = (10 :: integer)$
$gcd\ (-\ 10)\ 0 = (10 :: integer)$
$gcd\ 0\ 0 = (0 :: integer)$
in *Go*

All of them succeeded when exported to Go. A command to automatically generate test cases from Isabelle theorems extending on the **test_code** command is further given in Section 6.3.

6.2.3. Exporting HOL

The theory `HOL-Codegenerator_Test.Generate` contains a catch-all **export_code** command which will attempt to export large parts of `HOL-Library` into all target languages supported by the generator. This are no tests checking that the produced code shows the desired behaviour; it is merely meant as a stress-test to check that a target language's compiler will at least accept the code generator's output.

Unfortunately, this check currently still fails for the Go target, as it lacks an adaptation layer of comparable scope to that provided for the other target languages, as was discussed in Section 5.1.1.

6.3. Theorem-based Tests

The **nitpick** tool is a Kodkod-based counterexample and model finder for Isabelle/HOL (Blanchette and Nipkow 2010) included in the Isabelle distribution. It is commonly used during development of Isabelle theories to quickly check if a stated theory has a counterexample and hence cannot be proved.

However, it can also be used to generate models of theorems that are known to be correct, i.e. to simply create a list of variable assignments which satisfy a proposition.

Combining this with the **test_code** command discussed in Section 6.2, it is further possible to use these generated models as test cases of the code generator:

lemma $(x + y) = (y + (x :: \mathit{int}))$ **nitpick-codegen** *Go*

The **nitpick_codegen** command above will generate possible variable assigments. For the user, this is visible as output in Isabelle/jedit:

```
Nitpicking formula. . .
checking that 0 + 0 = 0 + 0 in Go . . .
checking that 1 + 0 = 0 + 1 in Go . . .
checking that 0 + 1 = 1 + 0 in Go . . .
checking that 0 + 0 = 0 + 0 in Go . . .
checking that 1 + 0 = 0 + 1 in Go . . .
checking that 1 + 1 = 1 + 1 in Go . . .
checking that 2 + 0 = 0 + 2 in Go . . .
checking that 0 + 2 = 2 + 0 in Go . . .
checking that 0 + 0 = 0 + 0 in Go . . .
checking that 0 + 1 = 1 + 0 in Go . . .
```

While this approach should be considered as merely an experimental proof of concept, it can already be helpful when working with real-world code. Particularly when writing code printing rules in the adaptation layer, it is useful in much the same way as **nitpick** is when writing new propositions: it allows to quickly establish that something "obvious" has gone wrong if it fails. Of course, it can never any guarantee of correctness — many kinds of unsoundness in printing rules will not be found[2].

Of course, this approach is necessarily limited in scope, since not everything that might appear in an Isabelle proposition will be executable. Its usefulness is further be limited, as **nitpick** often produces only relatively trivial variable assignments, and often (as seen above) also generates duplicates.

[2]For example, consider attempting to map Isabelle/HOL's `int` type to Go's built-in `int`, which is either 32 or 64 bit wide. This would cause unsoundness, but **nitpick_codegen** will never try numbers large enough to encounter an overflow.

7. Conclusion

I have presented a translation from Thingol by shallow embedding into a fragment of Go, and implemented it as a target language for Isabelle's code generation framework. The new target language has been used with success to port an existing Isabelle formalisation targeting Scala to also target Go.

It can be used either as an entirely out-of-tree development with a recent version of Isabelle, or alternatively as an in-tree extension of Isabelle itself. In the latter case, it is then also possible to use the `HOL_Codegenerator-Test` session and its commands, particularly **value** and **test_code**, with Go.

Finally, a short sketch of an automated testing scheme making use of the Isabelle formalisation to automatically generate test cases has also been shown.

Future Work

There are multiple interesting directions into which this work could be extended in the future:

Given Go's nature as a non-functional programming language which generally encourages an imperative programming style, an obvious extension to the Go target language would be to integrate it with Imperative/HOL (Bulwahn et al. 2008). Apart from producing more easily readable code, this also has the potential to greatly increase performance without sacrificing verified correctness by making use of other work in this direction, such as the Imperative Collections Framework presented by Lammich 2019.

The adaptation layer's power could be vastly improved by working on lifting some of the current limitations of printing rules discussed in Section 5. In particular, allowing placeholders in printing rules to distinguish between expressions and statements (thus allowing the intuitive translation of `if-then-else` expressions to `if`-statements), as well as making it possible to refer to a placeholder's type (i.e. to allow constructs where the target language requires explicit type annotations, such as polymorphic return types in Go) would allow a much more powerful adaptation layer comparable to what is currently provided for other languages, and will most likely be a precondition to let the `HOL-Codegenerator_Test.Generate` theory's catch-all **export_code** succeed for the Go target (recall Sections 5.1.1 and 6.2.3).

Likewise, the generation of test cases from theorems presented in Section 6.3 could be extended from the sketch of an implementation given there, and implemented as its own tool that can be used independently from any particular target language.

Finally, the most obvious task left would be to integrate Go as a target language for code generation into Isabelle itself, assuming there is interest in this from the wider Isabelle community and the Isabelle maintainer. Initial work on this has already been done, but many of the practical questions (such as how to invoke the Go compiler, how to best provide a Go component setup analogous to the `isabelle ghc_setup` or `isabelle ocaml_setup` tools) have not been resolved yet.

A. List of changes made to Isabelle

In the course of this work, several changes were made to Thingol itself. The two most important ones both add more information present in Isabelle to the intermediate language:

- Function Abstractions in terms are now decorated with their return types, which was not previously the case (changeset `#5016262a2384`).
- Likewise, in the dictionary construction now always includes the target type of any super instances referenced by a dictionary value (changeset `#a6a81f848135`).

Both represent minor incompatibilities in the Standard ML code of Isabelle compared to previous versions, as the definitions of the `Code_Thingol.iterm` and `Code_Thingol.plain_dict` types have changed slightly.

Other existing target languages, as well as other code within Isabelle and the AfP (changeset `#1d9151a74b81`) working with these types, were updated accordingly, and now simply ignore any values given in the new field.

One additional further change was made, fixing a bug:

- The **export_code** command did not work correctly when used with the **file** option if the target language generated files in subdirectories. Since none of the existing target languages required this, this had never been noticed, as it had not previously been an issue (changeset `#a11e25bdd247`).

At time of writing, all these changes are already part of the main Isabelle Mercurial repository on `https://isabelle.sketis.net/repos/isabelle`, and therefore included in any recent development checkout.

B. User's Guide

In general, using the new Go target should feel no different from using one of the pre-existing targets; users are therefore encouraged to refer to Haftmann, Bulwan, and Nipkow 2022 for basic usage.

As a trivial example, a trivial use of addition on natural numbers

fun *add* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  add x y = *x* + *y*

export-code *add* **in** *Go* **module-name** *Addition*

will produce the following Go code:

```
package Addition

import (
)

// sum type which can be Zero_nat, Suc
type Nat interface {};
type Zero_nat struct { };
type Suc struct { A Nat; };
func Elim_Zero_nat(p Nat)(bool) {
  switch p.(type) {
    case Zero_nat:
      return true;
    default:
      return false
  }
}
func Elim_Suc(p Nat)(bool, Nat) {
  switch q := p.(type) {
    case Suc:
      return true, q.A;
    default:
      return false, *new(Nat)
  }
}
```

```
func Plus_nat (x0 Nat, n Nat) Nat {
  {
    m, ma := Elim_Suc(x0);
    if (m) {
      nb := n;
      return Plus_nat(ma, (Nat(Suc{nb})));
    }
  };
  {
    if (x0 == (Nat(Zero_nat{}))) {
      nb := n;
      return nb;
    }
  };
  panic("match␣failed");
}

func Add (x Nat, y Nat) Nat {
  return Plus_nat(x, y);
}
```

However, there are some things that are worth keeping in mind when working on an Isabelle formalisation meant for generation of Go code:

- As was mentioned in Section 4.6.1, Isabelle definitions living at the top level will be translated as constant functions which take no arguments, which are called each time the constant is used. If the constant's definition is complex and it is used often, this may unnecessarily increase runtime.
- When writing wrapper code that creates values of data types, it is necessary to explicitly add a type conversion statement to convert the structure type which represents the constructor to the interface type which represents the entire data type. This can be easily forgotten, but will otherwise lead to pattern matches failing at runtime.

B.1. List of Target-specific Options

The **export_code** command supports target-language specific options. For Go, the full list of accepted options is as follows:

- **go_module** should take a string that is a valid URL. Generated packages will be assumed to live under this URL; a `go.mod` file for use with the `go` tools will be generated accordingly. If not given, a dummy value is used instead.

- **gen_stringer** takes no further argument. If present, it indicates that the code generator should produce instances of the `fmt.Stringer` interface for all structure types which represent data types. The functions for printing in the `fmt` package (such as `fmt.Printf`) will then produce syntax closer to Isabelle's for values of these structure types.
- **panic_on** takes a list of function names which should abort the program entirely. This behaves similarly to the `[[code abort]]` and `[[code drop]]` attributes, but the precise implementation differs (compare Section 5.2)

Bibliography

Blanchette, J. C. and T. Nipkow (2010). "Nitpick: A counterexample generator for higher-order logic based on a relational model finder." In: *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1*. Springer, pp. 131–146.

Bulwahn, L., A. Krauss, F. Haftmann, L. Erkök, and J. Matthews (2008). "Imperative functional programming with Isabelle/HOL." In: *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*. Springer, pp. 134–149.

Ellis, S., S. Zhu, N. Yoshida, and L. Song (2022). "Generic go to go: dictionary-passing, monomorphisation, and hybrid." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pp. 1207–1235.

Go Team (2022). *The Go Programming Language Specification. Version of December 15, 2022.* URL: `https://go.dev/ref/spec`.

Go Team (2023). *FAQ: Can I stop these complaints about my unused variable/import?* URL: `https://go.dev/doc/faq#unused_variables_and_imports`.

Griesemer, R., R. Hu, W. Kokke, J. Lange, I. L. Taylor, B. Toninho, P. Wadler, and N. Yoshida (Nov. 2020). "Featherweight Go." In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: `10.1145/3428217`.

Griesemer, R., R. Pike, K. Thompson, I. Taylor, R. Cox, J. Kim, and A. Langley (2009). *Hey! ho! let's go*. URL: `https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html`.

Haftmann, F. (2009). "Code generation from specifications in higher-order logic." PhD thesis. Technische Universität München.

Haftmann, F., L. Bulwan, and T. Nipkow (2022). *Code Generation from Isabelle/HOL theories*. URL: `https://isabelle.in.tum.de/doc/codegen.pdf`.

Haftmann, F. and T. Nipkow (2010). "Code generation via higher-order rewrite systems." In: *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings 10*. Springer, pp. 103–117.

Hupel, L. (2019). "Certifying Dictionary Construction in Isabelle/HOL." In: *Fundamenta Informaticae* 170.1-3, pp. 177–205.

Lammich, P. (2019). "Refinement to imperative HOL." In: *Journal of Automated Reasoning* 62.4, pp. 481–503.

Nipkow, T. and G. Klein (2014). *Concrete semantics: with Isabelle/HOL*. Springer. ISBN: 3319105418.

Stack Overflow (2023). *2023 Developer Survey*. URL: `https://survey.stackoverflow.co/2023`.