Extending Isabelle/HOL's Code Generator with support for the Go programming language

Terru Stübinger[1,2][0009−0006−7411−2533] and Lars Hupel[1,2][0000−0002−8442−856X]

[1] Giesecke+Devrient, Prinzregentenstr. 161, 81677 München, Germany
[2] Technische Universität München, School of Computation, Information and Technology, Boltzmannstr. 3, 85748 Garching bei München, Germany
stuebinm@in.tum.de, lars.hupel@tum.de

Abstract. The Isabelle proof assistant includes a small functional language, which allows users to write and reason about programs. So far, these programs 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 targets, 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 as an add-on library that can be simply imported into existing theories.

Keywords: Theorem provers · Code generation · Go programming language.

1 Introduction

The interactive theorem prover *Isabelle* of the LCF tradition [12] is based on a small, well-established and trusted mathematical inference 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**).

Unlike most other languages, Isabelle's nature as a theorem prover means that it is easy to formalise and prove propositions about the programs written in Isabelle/HOL. 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 by first translating into an intermediate language called *Thingol*, which is shared between all targets; code of the intermediate language is then transformed into code in the individual

target languages via the principle of *shallow embedding*, that is, by representing constructs of the source language using only a well-defined subset of the target language [6, 7].

Go is a programming language introduced by Google in 2009. It is a general-purpose, garbage-collected, and statically typed language [4]. 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, and many large existing code bases have been written in it.

Contributions This paper extends Isabelle's Code Generation facility with support for Go. We demonstrate a translation scheme from programs in Thingol to programs in Go, and thereby allow Isabelle users to also generate Go code from their theory files. This scheme requires some novel approaches to the encoding of functional programming constructs, such as pattern matching, in an imperative programming language (§4).

This extension to the Code Generator is supplied as a stand-alone theory file that can easily be imported into existing developments,[3] making it immediately usable in other contexts.

The motivation for this work stems from G+D's internal use of both ecosystems: Isabelle for formalization purposes, and Go for the real-world implementation. This naturally lead to a formalization gap, which this project sought to close (§5).

Related work This paper describes the first attempt at translating Isabelle formalizations into a non-functional programming language. Prior work in leveraging imperative features in the Code Generator [2] has targeted the existing, functional programming languages, and thereby could reuse much of the existing infrastructure. There is also unpublished work on adding support for F# to the Code Generator [1], another functional language.

2 The intermediate language Thingol

Isabelle's Code Generation pipeline works in multiple stages. Crucially, all definitions made in Isabelle are first translated into an abstract intermediate language called *Thingol*, which is the last step 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.

Consequently, Thingol's design naturally reflects the features common to previous target languages, and is based on a simply-typed λ-calculus with ML-style polymorphism. Perhaps surprisingly, Thingol also supports type classes, which can be mapped easily to Haskell and Scala, but less easily to the other targets.

[3] Available at https://doi.org/10.5281/zenodo.8401392 or https://github.com/isabelle-prover/isabelle-go-codegen.

$$
\begin{aligned}
s &::= c_1 \cap \cdots \cap c_n \\
\tau &::= \kappa\ \ \overline{\tau} \ \mid\ \alpha :: s \\
t &::= f[\overline{\tau}] \\
&\quad \mid\ x :: \tau \\
&\quad \mid\ \lambda x :: \tau.\ \ (t :: \gamma) \\
&\quad \mid\ t_1\ \ t_2 \\
&\quad \mid\ \text{case } t :: \tau \text{ of } \boxed{p \to b}
\end{aligned}
$$

(a) Sorts, types, and terms

```
data κ ᾱ_k = f_1 τ̄_1 | ··· | f_n τ̄_n

fun f :: ∀ᾱ :: s̄_k.τ where
  f [ᾱ :: s̄_k] t̄_1 = t_1
  ···
class c ⊆ c_1 ∩ ··· ∩ c_m where
  g_1 :: ∀α :: c. τ_1
  ···
inst κ ᾱ :: s̄_k :: c where
  g_1 [κ ᾱ :: s̄_k] = t_1
  ···
```

(b) Statements in Haskell-like syntax [7]

Fig. 1: Thingol syntax overview

Thingol distinguishes between terms and declarations (Figure 1). Terms are simple λ-expressions with the addition of case expression for pattern matching on datatypes. Declarations are top-level items that introduce datatypes, functions, as well as type classes and their instances to a program.

While there is no formal semantics of Thingol, it can be thought of as a *Higher-Order Rewrite System* (HRS) [10,11]. It provides a convenient abstraction over the target languages' semantics. Because a HRS does not have a specified evaluation order, the Code Generator cannot guarantee total, but only partial correctness. Therefore, users—except when generating Haskell code—should be careful to avoid relying on lazy evaluation.

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 shallow 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 we did not pursue alternative features or solutions.

In effect, this leaves many of Go's most interesting features (e.g., channels or methods) 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.

We also exclude any treatment of Go's package system, which is typically used to structure modular programs. For the purposes of our implementation, the user can choose between only one flat package containing the entire program,

Field name	A	Declaration	$D ::=$
Function name	f	Type declaration	`type` $t[\overline{\alpha\ c}]\ T$
Variable name	x, y	Function declaration	`func` $f[\overline{\alpha\ c}]\ F$ `{`s`}`
Structure type name	t_S	Expression	$d, e ::=$
Interface type name	t_I	Variable	x
Type parameter	α	Function call	$f[\overline{\alpha}](\overline{e})$
Type name	$t ::= t_S \mid t_I$	Structure literal	$t_s[\overline{\alpha}]\{\overline{e}\}$
Type	$\tau, \gamma, \sigma ::=$	Function abstraction	`func` F `{`s`}`
Parameter	α	Field selection	$e.A$
Structure Type	$t_S[\overline{\tau}]$	Type conversion	$\tau_I(e)$
Interface Type	$t_I[\overline{\tau}]$	Null pointer	`nil`
Function head	$F ::=$`(`$\overline{x\ \tau}$`)` $(\overline{\tau})$	Statement	$s ::=$
Type literal	$T ::=$	Expression	`return` $\overline{e}$`;`
Structure	`struct{`$\overline{A\ \tau}$`}`	Variable declaration	x `:=` e`;` s
Interface	`interface{}`	if-Statement	`if (`e`) {`s`};` s
Interface constraint	$c ::=$ `any`	Type assertion	x`,` y `:=` e`.(`τ`);` s

Fig. 2: A fragment of Go's syntax

or one package per Isabelle theory.[4] We only take into account that top-level names must start with an upper-case letter if they are to be accessible to other Go packages, which requires occasional renaming.

3.1 Syntax

The syntax fragment given in Figure 2 is inspired by that of Featherweight Generic Go [5], but differs in some important aspects:

1. Methods are not included; instead we use "ordinary" top-level functions.
2. Go distinguishes syntactically between expressions and statements in Go, whereas Featherweight Go does not. We retain the distinction and discuss conversion between them in §3.4.
3. Type parameters can be declared with an interface constraint. However, in our fragment the only available constraint is the unconstrained `any`, as Go's other constraints are not useful for our translation (§4.5).
4. We use Go 1.18's syntax for generics, which differs from the proposal put forward in Featherweight Generic Go.

3.2 Declarations

A (top-level) declaration D can define either a new type or function. Within one package, the order of declarations does not matter; any item may reference any other.

[4] Though in this case we require the theories' dependencies to be acyclic.

Structure types A declaration of the form `type` $t_S\,[\overline{\alpha\ c}]$ `struct{`$\overline{A\ \tau}$`}` introduces a new type constructor with fields $\overline{A}$ of types $\overline{\tau}$ to the program. It may be polymorphic and take type arguments $\overline{\alpha}$ which can be freely referenced within the $\overline{\tau}$ types. Since it is not possible to omit any constraint c for the type variables α, we use the (unrestricted) `any` constraint.

Note that there is no analogous construct to Thingol's sum types; that is, it is not possible to a have a structure type which has more than one constructor.

Interface types A declaration of the form `type` $t_I\,[\overline{\alpha\ c}]$ `interface{}` introduces a new (empty) interface type to the program. While Go supports non-empty interfaces containing methods, we do not use this feature.

Unlike interfaces in typical object-oriented languages such as Java, Go's interfaces are structural in nature: at runtime, any `struct` value conforms to an interface if (and only if) the `struct` implements a superset of the declared methods of the interface.

This implies that empty interfaces correspond to a "top" type that can denote arbitrary values. This perhaps odd choice will become obvious when introducing the translation scheme of data types (§4.2). Similarly, non-empty interfaces are not useful for the translation of type classes (§4.5).

Functions A declaration `func` $f\,[\overline{\alpha\ c}]$`(`$\overline{x\ \tau}$`)` $(\overline{\gamma})$ `{` s `}` introduces a new function f to the program. The type parameters $\overline{\alpha}$ can be referenced within both argument types $\overline{\tau}$ and the return types $\overline{\gamma}$.

Unlike in Thingol, a function cannot have multiple equations nor perform pattern matching on its arguments. Instead there is only one list of argument names $\overline{\alpha}$, which are in scope for the (unique) function body s.

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

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

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

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 `no_tuples := foo()` is not allowed.

3.3 Expressions

Expressions e can have several forms: variables, function application, and function abstraction are familiar from the λ-calculus. 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 $\overline{e}$. 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. We omit them in the interest of shorter code.

Field selection An expression $e.A$ selects the field named A of an expression e, which must have a fitting `struct` type τ_S that was declared with a field name A, and returns the value of that field. This is the only place outside a structure type's declaration that field names are 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 runtime; it can be recovered using a type assertion statement (see the next section). This expression can also be thought of as an "upcast".

3.4 Statements

This section introduces the statements that are used by the Code Generator. All statements of the defined fragment will always end in a `return`, and thereby return from the current function. We utilize this convention so that it is always possible to embed a statement into an expression by wrapping it into an immediately-called function abstraction `func () τ { s }()`. All forms except for the type assertion should appear familiar from similar languages.

Return The `return` keyword 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) { s1 }; s2` will evaluate e, which must have a boolean type. 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.

This statement could equivalently be written as `if (e) { s } else { s2 }`, which has the same semantics. The version without the explicit `else` branch is preferred to limit nesting of statements in the generated code.

Type assertion A statement of the form `x, y := e.(σ)` can be thought of as the inverse operation of type conversions, i.e., a "downcast". For an expression e of an interface type τ_I, the assertion checks whether the inner value contained within the interface value has type σ. The result of the check is assigned to the boolean variable y. If it was successful, x will be bound to that inner value; if not, it will be `nil`, Go's null pointer. Note that the type of x is σ.

4 Translation scheme

In this section, we will discuss the concrete translation schemes employed for Thingol programs. In the interest of brevity, we omit "uninteresting" steps, i.e., purely syntactic mappings, and focus on the non-trivial steps.

Likewise we will no discuss the treatment of variable names, which we preserve wherever possible; the only change made is to ensure that those names which should be exported are made upper-case (as required by Go). In some cases the translation will introduce new names; as Isabelle's Code Generator already provides a way to generate guaranteed-unused and therefore "fresh" names when needed, their choice will also not be discussed.

4.1 Types, terms and statements

We define three translations $\mathrm{type}(\tau)$, $\mathrm{expr}(t)$, and $\mathrm{stmt}(t)$. The first is a straightforward syntactic mapping of 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. For now, we exclude any mapping of common types (e.g. integers) to built-in Go types; this topic will be revisited later (§4.6).

The other two translations—expr and stmt—are used for converting Thingol terms into Go expressions and statements, respectively. Which one is used thus depends on what Go expects in each particular context; for example, terms used as function arguments use expr; a term which is a function body uses stmt. Semantically, expr and stmt are related roughly as follows:

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

Abstractions The translation of a λ-abstraction $\lambda(x :: \tau).\ \ (t :: \gamma)$ demonstrates the distinction between expressions and statements:

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

Although curried abstractions are unusual in Go, no effort is made to uncurry them (this is in constrast with the treatment of top-level functions, which are uncurried §4.4).

Applications of top-level functions Applications t are more tedious to translate. Since definitions of top-level functions are uncurried (§4.4), we first have to check if t is a call to such a function, i.e., if t has the shape $\big(\cdots((f[\overline{\tau}_i]\ \ a_1)\ \ a_2)\cdots\big)\ \ a_n$, where f references a top-level function or data type constructor which takes m arguments.

If so, we have to consider three cases:

$n = m$ Fully-satured application; all arguments are passed into f
$n < m$ Unsatured application; need to η-expand

$n > m$ Over-satured application. This occurs if f returns another function, with a_1 to a_m being the immediate arguments to f and any remaining a_{m+1} to a_n as curried arguments. The latter will be passed individually.

As will be described later (§4.5), translating Isabelle's type classes through the dictionary construction may introduce additional (value-level) parameters for a top-level function, and corresponding additional arguments d_1 to d_r to each application of the same function. These are inserted before the user-defined parameters.

Altogether, we arrive at the following scheme when f references a function:

$$\mathrm{expr}(t) = f\texttt{[}\tau_1\texttt{,}\ldots\texttt{,}\tau_i\texttt{]}\texttt{(}d_1\texttt{,}\ldots d_r\texttt{,}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{)}\texttt{(}a_{m+1}\texttt{)}\ldots\texttt{(}a_n\texttt{)}$$

Finally, if f references a data type constructor of a type τ rather than a function, the case $n > m$ cannot occur. However, we must wrap the constructor into a type conversion to type τ, and use slightly different syntax for passing the arguments:

$$\mathrm{expr}(t) = \kappa\texttt{(}f\texttt{[}\tau_1\texttt{,}\ldots\texttt{,}\tau_i\texttt{]}\texttt{\{}d_1\texttt{,}\ldots d_r\texttt{,}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{\}}\texttt{)}$$

Lambda applications If an application $t = t_1\ \ t_2$ is not a call to a top-level function, then the translation is straightforward:

$$\mathrm{expr}(t_1\ \ t_2) = \texttt{expr(}t_1\texttt{)(expr(}t_2\texttt{))}$$

4.2 Data types

Recall that a data type κ defined in Thingol consists of type parameters $\overline{\alpha}_i$ and constructors $\overline{f}_i$. Each f_i gets translated into its own separate `struct` type.

As was discussed in § 3, Go knows no sum types, thus the translation has to simulate their behaviour using other means. For a data type, we generate an additional unconstrained interface type δ, meant to represent any constructor f_i of κ.

Since δ is left unconstrained, there is no language-based guarantee on the Go side that a value x of type δ was actually constructed using one of the constructors f_i, i.e., that the inner value of x actually is of one of the `struct` types. We assume that the generator operates only already type-correct intermediate code, so it will still never go wrong. However, programmers writing wrapper code to interact with the generator's output must be careful not to pass values of a wrong type, lest they produce run-time errors.

If the data type κ has exactly one constructor f_1, then no additional interface type δ is generated.

Constructors Defining a `struct` type for an individual constructor is straightforward. A constructor f with fields of types τ_1 to τ_i is translated into Go as a `struct` with the same name and fields: `type` f `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. Note that those generated field names are entirely unimportant (access happens only through destructors, and the names are not required when constructing a value); the only requirement imposed on them is that each $\overline{A}_i$ of the same `struct` are distinct.

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. Those destructors are synthetic and not present in the Thingol representation. Their sole purpose is to unpack and return the individual fields in a `struct` type, exploiting Go's multiple return types.

```
func f_dest (q f) (τ1, ..., τn) {
  return q.A1, ..., q.An
}
```

We need those destructors for technical reasons (§4.3). Note that they ignore the interface type κ and operate on the individual structure types $\overline{f}$ directly; the pattern match translation thus has to unpack the inner value before invoking the destructor.

Example As a simple example, consider the definition of natural numbers in Isabelle, here reusing Thingol pseudo-syntax (Figure 1):

```
data nat = Zero | Suc nat
```

Translated into Go, this produces the following output:

```
type Nat any;
type Zero struct { };
type Suc struct { A Nat; };
func Suc_dest(p Suc)(Nat) {
  case return q.A;
}
```

Note the use of the unconstrained interface type `Nat` to represent a faux sum type that is supposed to only contain `Zero` and `Suc` values. Constructing the number 1 would look as follows: `Nat(Suc{Nat(Zero{}))`. While type-correct according to Go, the value `Nat(Suc{nil})` could easily cause run-time exceptions in other parts of the generated code, particular where it simulates a pattern match on it (§4.3). Programmers must thus be careful not to introduce such values when hand-writing wrapper code. Furthermore, the translation omits the destructor for `Zero`, because the structure has no fields that could be unpacked.

A slightly more involved example is the polymorphic list datatype. In Isabelle, it is defined as follows:

```
data α list = Nil | Cons α (α list)
```

The resulting Go code now contains generic annotations:

```
type List[a any] interface {};
type Nil[a any] struct { };
type Cons[a any] struct { A a; Aa List[a]; };
func Cons_dest[a any](p Cons[a])(a, List[a]) {
  return p.A, p.Aa
}
```

4.3 Case expressions

Thingol's case expressions implement pattern matching on a value, in a way which will be immediately familiar to anyone acquainted with other functional languages such as Standard ML or Haskell: they inspect a term t (the expression's *scrutinee*) and match it against a series of clauses $\overline{p_i \to b_i}$. Each clause contains a pattern p_i and a term t_i that is to be evaluated if the pattern matches the scrutinee. Syntactically, patterns are a subset of terms; they can only be composed of variables and fully-satisfied applications of data type constructors to sub-patterns $f\ \ \overline{p}_i$ constructed of the same subset.

Since Go has no comparable feature, a data type pattern in a case expression is translated into a series of (possibly nested) `if`-conditions and calls to destructor functions. The bodies of the innermost `if`-condition then correspond to the translated terms t_i, which must be in statement-form, i.e., ending in a `return`-statement. Thus, if the pattern could be matched, further patterns will not be executed. Naturally, using `return` in this manner implies that a case expression must always either be in tail position, or else be wrapped into an anonymous function if it does not (§3).

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 of an incomplete pattern where no clause could be matched (incomplete patterns are admissable in Isabelle's logic, see Hupel [8] for a detailed description). This `panic` can also be encountered if an external caller exploited the lossy conversion of sum types as described above and supplied, e.g., a `nil` value as a scrutinee.

Taken together, an entire case expression is translated as a linear sequence of individual clauses, followed by a `panic`:

$$\text{stmt}(\text{case } t :: \tau \text{ of } [\overline{p \to b}]) = \overline{\text{stmt}(p \to b);}\ \texttt{panic("Match failed");}$$

Let us now consider the concrete translation for variable and constructor patterns.

Variable pattern We assign the scrutinee t to the variable x to make it available in the scope of b.

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

Constructor pattern The pattern is of the form $f[\overline{\tau}_i][\overline{s}_k]$. If all sub-patterns $\overline{s}_k$ are variable patterns, the translation is once again straightforward:

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

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

$$\begin{aligned}\text{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}&m\texttt{,}A_1\texttt{,}\ldots\texttt{,}A_k\texttt{:= }f\texttt{\_dest(}t\texttt{); if (}m\texttt{) \{}\text{inner}\texttt{\}\}}\\ \text{inner} = \text{stmt}(&\text{case } A_1 \text{ of } s_1 \to (\ldots \to (\text{case } A_k \text{ of } s_k \to b)))\end{aligned}$$

In other words, the sub-patterns are treated as if they were further nested case expressions. This results in a total nesting depth of one level per constructor.

Within the innermost `if`, the body b of the pattern's clause is translated as statement to ensure it returns from the current function.

Optimizing the nesting level The translation described in this section can translate arbitrary patterns, but comes 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. But if the fields themselves are again data type constructors with sub-patterns, the number of nested levels quickly increases further.

In real-world applications, we can reduce the blow-up by optimizing constructor patterns without arguments. Instead of calling a destructor function, we can emit an equality check, since there are no fields to extract. Multiple equality checks can be joined together using Go's conjunction operator `&&`.

Example Consider a function `hd2` that takes a list and returns (optionally) the second element of the list. Using the Thingol pseudo-syntax, this can be defind as follows (assuming standard definition of the option type):

```
fun hd2 :: ∀α.α list ⇒ α option where
  hd2 xs = case xs of Nil ⇒ None
                    | Cons x Nil ⇒ None
                    | Cons x (Cons y xs) ⇒ Some y
```

This is translated into Go as follows:

```
func Hd2[a any] (x0 List[a]) Option[a] {
  if (x0 == (List[a](Nil[a]{}))) {
    return (Option[a](None[a]{}));
  }
  q, m := x0.(Cons[a]);
  if (m) {
    _, c := Cons_dest(q);
    if (c == (List[a](Nil[a]{}))) {
      return (Option[a](None[a]{}));
    }
  }
  q, m := x0.(Cons[a]);
  if (m) {
    _, p := Cons_dest(q);
    q, m := p.(Cons[a]);
    if (m) {
      ya, _ := Cons_dest(q);
      return (Option[a](Some[a]{ya}));
    }
  }
```

```
  panic("match failed");
}
```

This piece of generated code benefits from the optimization described above (in the first and second clauses). Also, observe that some bound variables are unused and have to be generated as `_`, because unused variables are a compile error in Go.

4.4 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.

Merging multiple clauses Thingol allows Haskell-style function definition comprising multiple clauses. But in Go, all parameters of functions must be simple variables. 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.

Pattern matching The combined function body then consists of a pattern match translation as described earlier.[5] Each equation is then treated as a clause of a synthetic case-expression; since functions can pattern match on multiple parameters, we again push inwards and translate as if a nested series of case-expressions were present.

Example The following Thingol definition is semantically equivalent to the example from the previous section, but written using multiple equations. Due to the transformation applied by the Code Generator, the generated Go code is identical.

```
fun hd2 :: ∀α.α list ⇒ α option where
  hd2 Nil = None
  hd2 (Cons x Nil) = None
  hd2 (Cons x (Cons y xs) ys) = Some y
```

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

Special case: top-level constants Unsurprisingly, Thingol accepts top-level definitions that are not functions, for example:[6]

```
fun a :: nat where
  a = 10
```

For those, we have to battle yet another Go restriction: Go admits top-level variable declarations, but only for monomorphic types, and it disallows function calls in their definitions. Therefore, we must treat such Thingol definitions as if they were nullary functions. While this changes nothing of the semantics of the translated program, it does incur a (potentially significant) runtime cost: constants will be evaluated each time they are used, instead of only once when the program is initialized.

4.5 Dictionary construction

On the surface, Isabelle's Haskell-style type classes and Go's interfaces share many of the same features, and are sometimes considered to be near-analogous [3]. However, translating type classes into interfaces does not work. This is caused by an implementation concern: Go directly compiles methods into virtual tables for dynamic dispatch. An `interface` in Go declares multiple *methods*, where each method type must take the generic value as zeroth (i.e. implicit) parameter. Isabelle (and Haskell) do not have such a restriction, as can be observed from the following examples, which are valid in Isabelle:

```
class foo where
  foo :: unit ⇒ α

class bar where
  bar :: (α ⇒ α) ⇒ unit
```

Naively translated into Go, both would be rejected by the its compiler. The first class declares a function that does not take an α parameter at all, whereas the second class' function does not take a simple α parameter (but a parameter whose type contains α).

As a practical example, consider that while the class for semigroups would be admissible as an `interface` (having a single method `(+) :: α ⇒ α ⇒ α`), monoids would not be (`unit :: α` does not even have any parameters).

To avoid the additional complexity of treating all these cases separately, we resort to using a dictionary construction [7, 8] in all cases. Since the existing SML target of the Code Generator has to deal with the same issue, all required infrastructure is already in place: Thingol's terms come with enough annotations to resolve all type class constraints during translation and replace the implicit instance arguments of functions making use of type classes by explicit dictionary values, which we represent as one data type per type class.

[6] Readers familiar with Isabelle syntax may be surprised about this notation; while Isabelle/HOL distinguishes between the **fun** and **definition** keywords, Thingol has no such distinction.

Thus only relatively few things are left to do in Go:

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

Example Consider the following definition of a semigroup together with a function operating on it:

```
class semigroup where
  (+) :: α ⇒ α ⇒ α where

class monoid ⊆ semigroup where
  zero :: α

fun sum :: α :: monoid list ⇒ α where
  sum xs = fold (+) xs zero
```

The generated code looks as follows (ignoring the `list` data type):

```
type Semigroup[a any] struct {
  Plus func(a, a) a
}

type Monoid[a any] struct {
  Semigroup_monoid Semigroup[a]
  Zero func () a
}

func Sum[a any] (a_ Monoid[a], xs List[a]) a {
  return Fold[a, a](
    func (aa a) func(a) a {
      return func (b a) a { return a_.Semigroup_monoid.Plus(aa, b); };
    },
    xs,
    a_.Zero()
  );
}
```

4.6 Mapping high-level constructs

So far, the shallow embedding we have presented produced code with no dependencies on the Go side, with only the built-in constructs `panic` and `&&` used.

All higher-level constructs used by programs (such as lists, numbers) must thus be "brought along" from Isabelle, and are translated wholesale exactly as they are defined in their formalisations. While this guarantees correctness, it is highly impractial for real-world applications: for example, natural numbers as defined in Isabelle/HOL (unary Peano representation, §4.2) require linear memory and quadratic runtime even for simple oprations like addition.

Luckily, the Code Generator already has a solution for this conundrum in the form of *printing rules*, which can map Isabelle's types and constants to user-supplied names in the target language. We have set up printing rules mapping:

- Isabelle/HOL's booleans to booleans in Go
- numbers to arbitrary-precision integers (via Go's `math/big` package)
- strings of the `String.literal` type to strings in Go

Unfortunately, linked lists cannot be mapped, because Go does not feature a standard implementation of linked lists.

5 Evaluation

Even though Go is a very different programming language compared to the other targets Haskell, Scala, OCaml, and SML, we have achieved almost full feature parity for the translation described in this paper. This means that almost any Isabelle construct can be cleanly mapped to a corresponding encoding in Go. We have confirmed that in two case studies:

Existing formalisation At G+D, we use Isabelle for a substantial formalization of various graph algorithms powering a financial transaction system. The purpose of the formalization is to provide real-world security guarantees, such as inability to clone money. We have previously used the Code Generator to produce Scala code as a reference implementation, combined with some hand-written wrapper code and basic unit tests.

As a simple evaluation of Go code generated from the same Isabelle theories, we re-wrote the unit tests and the necessary wrapper code in Go. We obtained equivalent results and could not find bugs in the Code Generator or unintended behaviour of the code it produced. 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), and not all incorrect 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 a "match failed" error at runtime.

Another awkward source of problems when integrating the generated Go code with a larger code base is that Go's standard library lacks basic functional data

structures, such as lists on tuples (§4.6). This means that the generated code is unidiomatic and relies on manual conversions, e.g. between arrays and lists.

HOL-Codegenerator_Test Isabelle's distribution contains a Code Generator test session which is used as a self-check for the various target languages of the Code Generator. For this paper, a single export command is relevant, which is meant to export a considerable chunk of Isabelle/HOL's library as a stress-test for the Code Generator. This has worked as expected, with the entirety of the stress-test successfully compiling in Go.

Red-black trees As an example, we have translated a portion of Isabelle's implementation of red-black trees to Go. We reproduce a small subset (balancing function) in the appendix (§A) and refer to GitHub[7] for the full code.

Trusted code base Just like for the other target language, our implementation is part of the *trusted code base*, i.e., bugs in the Code Generator may lead to bugs in the generated program, and will not be caught by Isabelle's kernel. We did not have to enlarge that trusted code base, therefore promising similar correctness like the other targets. More ambitious code printing however may change that picture, since such rules may have to assume more constructs in Go.

6 Conclusion

We 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 that was only targeting Scala to additionally target Go. The implementation is readily usable with a standard Isabelle2023 installation and requires merely importing an additional theory file. The suite of existing tests of Isabelle's Code Generator is also supported.

Future work The two most promising areas of future work are: leveraging Go's imperative nature by tightly integrating it with Imperative/HOL [2]; and generating more idiomatic Go code through custom code printing rules. Both can be implemented using similar mechanisms. However, substantial changes to Isabelle's code generation infrastructure are required, because Go demands more type annotations than other target languages.

Acknowledgements The authors would like to thank Florian Haftmann for his contributions to the development. We appreciate the comments suggested by Cornelius Diekmann, greatly improving the presentation in this paper. This work has been partially supported by the Federal Ministry of Education and Research (BMBF), Verbundprojekt CONTAIN (13N16582).

[7] https://github.com/isabelle-prover/isabelle-go-codegen

References

1. Brucker, A.D.: New Code Generator Target: F#, https://mailman46.in.tum.de/pipermail/isabelle-dev/2022-August/017633.html
2. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: 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. pp. 134–149. Springer (2008)
3. Ellis, S., Zhu, S., Yoshida, N., Song, L.: Generic go to go: dictionary-passing, monomorphisation, and hybrid. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1207–1235 (2022)
4. Go Team: The Go Programming Language Specification, https://go.dev/ref/spec
5. Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I.L., Toninho, B., Wadler, P., Yoshida, N.: Featherweight Go. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428217
6. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technische Universität München (2009)
7. Haftmann, F., Nipkow, T.: 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. pp. 103–117. Springer (2010)
8. Hupel, L.: Certifying Dictionary Construction in Isabelle/HOL. Fundamenta Informaticae **170**(1-3), 177–205 (2019)
9. Krauss, A.: Automating recursive definitions and termination proofs in higher-order logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/doc/681651/document.pdf
10. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical Computer Science **192**(1), 3–29 (1998). https://doi.org/10.1016/S0304-3975(97)00143-6
11. Nipkow, T.: Higher-order rewrite systems. In: Hsiang, J. (ed.) Rewriting Techniques and Applications. pp. 256–256. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
12. Nipkow, T., Klein, G.: Concrete semantics with Isabelle/HOL. Springer (2014), http://concrete-semantics.org

A Example: Red-black trees

A.1 Isabelle specification

First, let us consider a small subset of the definition of red-black trees in Isabelle/HOL. Isabelle's syntax admits abbreviations that are expanded both on the left-hand and right-hand sides of defining equations, leading to a very compact definition of balancing.

```
datatype 'a tree =
  Leaf |
  Node ('a tree) 'a ('a tree)

datatype color = Red | Black
```

```
type_synonym 'a rbt = ('a * color) tree

abbreviation R where R l a r == Node l (a, Red) r
abbreviation B where B l a r == Node l (a, Black) r

fun baliL :: 'a rbt => 'a => 'a rbt => 'a rbt where
baliL (R (R t1 a t2) b t3) c t4 = R (B t1 a t2) b (B t3 c t4) |
baliL (R t1 a (R t2 b t3)) c t4 = R (B t1 a t2) b (B t3 c t4) |
baliL t1 a t2 = B t1 a t2
```

A.2 Scala translation (for reference)

As a point of reference, consider the generated Scala code. While the datatype definitions are rather compact. But hidden behind the scenes, HOL's implementation of recursive functions will always expand pattern matching to disambiguate patterns. The advantage is that patterns are no longer sensitive about the order in which they are applied. The disadvantage however is that the number of patterns may explode, depending on their depth. This is well-documented behaviour and is independent of code generation [9].

```
object RbtTest {

abstract sealed class color
final case class Red() extends color
final case class Black() extends color

abstract sealed class tree[A]
final case class Leaf[A]() extends tree[A]
final case class Node[A](a: tree[A], b: A, c: tree[A]) extends tree[A]

def baliL[A](x0: tree[(A, color)], c: A, t4: tree[(A, color)]): tree[(A, color)]
  =
  (x0, c, t4) match {
  case (Node(Node(t1, (a, Red()), t2), (b, Red()), t3), c, t4) =>
    Node[(A, color)](Node[(A, color)](t1, (a, Black()), t2), (b, Red()),
                     Node[(A, color)](t3, (c, Black()), t4))
  case (Node(Leaf(), (a, Red()), Node(t2, (b, Red()), t3)), c, t4) =>
    Node[(A, color)](Node[(A, color)](Leaf[(A, color)](), (a, Black()), t2),
                     (b, Red()), Node[(A, color)](t3, (c, Black()), t4))
  case (Node(Node(v, (vc, Black()), vb), (a, Red()), Node(t2, (b, Red()), t3)),
        c, t4)
    => Node[(A, color)](Node[(A, color)](Node[(A, color)](v, (vc, Black()), vb),
  (a, Black()), t2),
                        (b, Red()), Node[(A, color)](t3, (c, Black()), t4))
  case (Leaf(), a, t2) => Node[(A, color)](Leaf[(A, color)](), (a, Black()), t2)
  case (Node(Leaf(), (v, Black()), vb), a, t2) =>
    Node[(A, color)](Node[(A, color)](Leaf[(A, color)](), (v, Black()), vb),
                     (a, Black()), t2)
  case (Node(Leaf(), va, Leaf()), a, t2) =>
    Node[(A, color)](Node[(A, color)](Leaf[(A, color)](), va,
                                      Leaf[(A, color)]()),
                     (a, Black()), t2)
  case (Node(Leaf(), va, Node(v, (ve, Black()), vd)), a, t2) =>
    Node[(A, color)](Node[(A, color)](Leaf[(A, color)](), va,
                                      Node[(A, color)](v, (ve, Black()), vd)),
                     (a, Black()), t2)
  case (Node(Node(vc, (vf, Black()), ve), (v, Black()), vb), a, t2) =>
```

```
    Node[(A, color)](Node[(A, color)](Node[(A, color)](vc, (vf, Black()), ve),
                                      (v, Black()), vb),
                     (a, Black()), t2)
  case (Node(Node(vc, (vf, Black()), ve), va, Leaf()), a, t2) =>
    Node[(A, color)](Node[(A, color)](Node[(A, color)](vc, (vf, Black()), ve),
                                      va, Leaf[(A, color)]()),
                     (a, Black()), t2)
  case (Node(Node(vc, (vf, Black()), ve), va, Node(v, (vh, Black()), vg)), a,
        t2)
    => Node[(A, color)](Node[(A, color)](Node[(A,
        color)](vc, (vf, Black()), ve),
  va, Node[(A, color)](v, (vh, Black()), vg)),
                        (a, Black()), t2)
  case (Node(v, (vc, Black()), vb), a, t2) =>
    Node[(A, color)](Node[(A, color)](v, (vc, Black()), vb), (a, Black()), t2)
}

}
```

A.3 Go translation

The generated Go code has roughly four times the size as the reference Scala code. This is explained by the constant overhead in the datatype definitions, as well as the overhead for encoding pattern matching.

```
package RbtTest

import (
)

// sum type which can be Red, Black
type Color any;
type Red struct { };
type Black struct { };

type Prod[a, b any] struct { A a; Aa b; };
func Pair_dest[a, b any](p Prod[a, b])(a, b) {
  return p.A, p.Aa;
}

// sum type which can be Leaf, Node
type Tree[a any] any;
type Leaf[a any] struct { };
type Node[a any] struct { A Tree[a]; Aa a; Ab Tree[a]; };

func Node_dest[a any](p Node[a])(Tree[a], a, Tree[a]) {
  return p.A, p.Aa, p.Ab
}

type Prod[a, b any] struct { A a; Aa b; };
func Pair_dest[a, b any](p Prod[a, b])(a, b) {
  return p.A, p.Aa;
}

func BaliL[a any] (x0 Tree[Prod[a, Color]], c a, t4 Tree[Prod[a, Color]]) Tree[Prod[a, Color
    ]] {
  {
    q, m := x0.(Node[Prod[a, Color]]);
    if m {
      q, p, t3a := Node_dest(q);
      r, m := q.(Node[Prod[a, Color]]);
      if m {
        t1a, r, t2a := Node_dest(r);
        _ = r;
        ab, d := Pair_dest(r);
```

```
      if d == (Color(Red{})) {
        _ = p;
        ba, e := Pair_dest(p);
        if e == (Color(Red{})) {
          cb := c;
          t4b := t4;
          return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
                 Color]]{t1a, Prod[a, Color]{ab, Color(Black{})}, t2a}), Prod[a, Color]{ba,
                Color(Red{})}, Tree[Prod[a, Color]](Node[Prod[a, Color]]{t3a, Prod[a, Color]{
                cb, Color(Black{})}, t4b})});
        }
      }
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    d, q, p := Node_dest(q);
    if d == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) {
      _ = q;
      ab, e := Pair_dest(q);
      if e == (Color(Red{})) {
        r, m := p.(Node[Prod[a, Color]]);
        if m {
          t2a, r, t3a := Node_dest(r);
          _ = r;
          ba, f := Pair_dest(r);
          if f == (Color(Red{})) {
            cb := c;
            t4b := t4;
            return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[
                   a, Color]]{Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{}), Prod[a, Color]{ab,
                    Color(Black{})}, t2a}), Prod[a, Color]{ba, Color(Red{})}, Tree[Prod[a,
                   Color]](Node[Prod[a, Color]]{t3a, Prod[a, Color]{cb, Color(Black{})}, t4b})
                   });
          }
        }
      }
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    r, q, p := Node_dest(q);
    s, m := r.(Node[Prod[a, Color]]);
    if m {
      va, s, vba := Node_dest(s);
      _ = s;
      vca, d := Pair_dest(s);
      if d == (Color(Black{})) {
        _ = q;
        ab, e := Pair_dest(q);
        if e == (Color(Red{})) {
          t, m := p.(Node[Prod[a, Color]]);
          if m {
            t2a, t, t3a := Node_dest(t);
            _ = t;
            ba, f := Pair_dest(t);
            if f == (Color(Red{})) {
              cb := c;
              t4b := t4;
              return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[
                     Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a, Color]]{va, Prod[a,
                     Color]{vca, Color(Black{})}, vba}), Prod[a, Color]{ab, Color(Black{})},
                     t2a}), Prod[a, Color]{ba, Color(Red{})}, Tree[Prod[a, Color]](Node[Prod[a
                     , Color]]{t3a, Prod[a, Color]{cb, Color(Black{})}, t4b})});
```

```
          }
        }
      }
    }
  }
};
{
  if x0 == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) {
    ab := c;
    t2a := t4;
    return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Leaf[Prod[a,
         Color]]{}), Prod[a, Color]{ab, Color(Black{})}, t2a});
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    d, p, vba := Node_dest(q);
    if d == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) {
      _ = p;
      va, e := Pair_dest(p);
      if e == (Color(Black{})) {
        ab := c;
        t2a := t4;
        return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
             Color]]{Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{}), Prod[a, Color]{va, Color
             (Black{})}, vba}), Prod[a, Color]{ab, Color(Black{})}, t2a});
      }
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    e, vaa, d := Node_dest(q);
    if e == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) && d == (Tree[Prod[a, Color]](
         Leaf[Prod[a, Color]]{})) {
      ab := c;
      t2a := t4;
      return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
           Color]]{Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{}), vaa, Tree[Prod[a, Color]](
           Leaf[Prod[a, Color]]{})}), Prod[a, Color]{ab, Color(Black{})}, t2a});
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    d, vaa, p := Node_dest(q);
    if d == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) {
      q, m := p.(Node[Prod[a, Color]]);
      if m {
        vb, q, vda := Node_dest(q);
        _ = q;
        vea, e := Pair_dest(q);
        if e == (Color(Black{})) {
          ab := c;
          t2a := t4;
          return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
               Color]]{Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{}), vaa, Tree[Prod[a,
              Color]](Node[Prod[a, Color]]{vb, Prod[a, Color]{vea, Color(Black{})}, vda})})
              , Prod[a, Color]{ab, Color(Black{})}, t2a});
        }
      }
    }
  }
};
```

```
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    q, p, vba := Node_dest(q);
    r, m := q.(Node[Prod[a, Color]]);
    if m {
      vca, r, vea := Node_dest(r);
      _ = r;
      vfa, d := Pair_dest(r);
      if d == (Color(Black{})) {
        _ = p;
        va, e := Pair_dest(p);
        if e == (Color(Black{})) {
          ab := c;
          t2a := t4;
          return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
                 Color]]{Tree[Prod[a, Color]](Node[Prod[a, Color]]{vca, Prod[a, Color]{vfa,
                Color(Black{})}, vea}), Prod[a, Color]{va, Color(Black{})}, vba}), Prod[a,
                Color]{ab, Color(Black{})}, t2a});
        }
      }
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    p, vaa, d := Node_dest(q);
    if d == (Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})) {
      q, m := p.(Node[Prod[a, Color]]);
      if m {
        vca, q, vea := Node_dest(q);
        _ = q;
        vfa, e := Pair_dest(q);
        if e == (Color(Black{})) {
          ab := c;
          t2a := t4;
          return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
                 Color]]{Tree[Prod[a, Color]](Node[Prod[a, Color]]{vca, Prod[a, Color]{vfa,
                Color(Black{})}, vea}), vaa, Tree[Prod[a, Color]](Leaf[Prod[a, Color]]{})}),
                Prod[a, Color]{ab, Color(Black{})}, t2a});
        }
      }
    }
  }
};
{
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    q, vaa, p := Node_dest(q);
    r, m := q.(Node[Prod[a, Color]]);
    if m {
      vca, r, vea := Node_dest(r);
      _ = r;
      vfa, d := Pair_dest(r);
      if d == (Color(Black{})) {
        s, m := p.(Node[Prod[a, Color]]);
        if m {
          vb, s, vga := Node_dest(s);
          _ = s;
          vha, e := Pair_dest(s);
          if e == (Color(Black{})) {
            ab := c;
            t2a := t4;
            return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[
                   a, Color]]{Tree[Prod[a, Color]](Node[Prod[a, Color]]{vca, Prod[a, Color]{
                   vfa, Color(Black{})}, vea}), vaa, Tree[Prod[a, Color]](Node[Prod[a, Color
```

```
                ]]{vb, Prod[a, Color]{vha, Color(Black{})}, vga})}), Prod[a, Color]{ab,
                Color(Black{})}, t2a});
          }
        }
      }
    }
  }
 };
 {
  q, m := x0.(Node[Prod[a, Color]]);
  if m {
    va, p, vba := Node_dest(q);
    _ = p;
    vca, d := Pair_dest(p);
    if d == (Color(Black{})) {
      ab := c;
      t2a := t4;
      return Tree[Prod[a, Color]](Node[Prod[a, Color]]{Tree[Prod[a, Color]](Node[Prod[a,
           Color]]{va, Prod[a, Color]{vca, Color(Black{})}, vba}), Prod[a, Color]{ab, Color
           (Black{})}, t2a});
    }
  }
 };
 panic("match failed");
}
```