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

Terru Stübinger[1,2] and Lars Hupel[1,2]

[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 [14] 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.

Many 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, functions, and Haskell-style type classes and instances.

Isabelle's nature as a theorem prover further makes it easy to formalise and prove propositions about such programs. 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*, shared between all targets; from this language, code is then transformed into 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, thus side-stepping the issue of finding a complete formal description of a target language's behaviour [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. For that, we demonstrate a translation scheme from programs in Thingol to programs in Go (§4). We provide this facility as a stand-alone theory file that can easily be imported into existing developments. We provide our development as an entry in the *Archive of Formal Proofs (AFP)*—a repository of Isabelle proof libraries—, making it immediately usable in other contexts [18].

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

Related work This paper describes the first attempt at translating Isabelle formalisations 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.

Shallow embeddings of C in proof assistant are already well known; for example in F* [15], Isabelle [17], and Why3 [16]. Those tools are not designed to export arbitrary code, but require developers to use a restricted subset of the host language. Instead, they are mainly geared towards low-level programming; with some providing C-style memory management. Our work focuses instead on translating the full functional host language into a high-level imperative language, therefore avoiding the need to (re)write host language code specifically for the purpose.

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 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, which

```
datatype Nat = Zero | Suc Nat
datatype α list = Nil | Cons α (α list)

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

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

class monoid ⊆ semigroup where
  zero :: α

instance Nat :: semigroup where
  a + Zero = a
  Zero + a = a
  (Suc a) + b = Suc (a + b)

instance Nat :: monoid where
  zero = Zero

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

Fig. 1. An example program (omitting the definition of `fold` for brevity)

instead use a dictionary construction (§4.5). The supported fragment of type classes and instance corresponds to Haskell98, with the exception of constructor classes (which would require a more expressive type system) [8,10].

Thingol's terms are simple λ-expressions with the addition of case expression for pattern matching on data types. A Thingol program is a list of declarations, i.e. top-level items which introduce data types, functions, type classes, and their instances.

While there is no formal semantics of Thingol, it can be thought of as a *Higher-Order Rewrite System* (HRS) [12,13]. 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. (This restriction applies to all supported target languages.)

Reusing Thingol has two immediate benefits: we can leverage the entire entire existing pipeline as well as its existing code adaptations, and are not forced to reimplement some tedious translation of Isabelle's more advanced features. Additionally, creating a custom intermediate language would not help to meaningfully address the functional–imperative mismatch between Isabelle/HOL and Go, but only shift the complexity elsewhere.

3 The target fragment of Go

Go, being an imperative language, differs in many aspects from the already-existing target languages of Isabelle's Code Generator. Conversely, many of Go's unique features are not needed by the generator. Since the translation works as a shallow embedding into the target language, it suffices to use the fragment which can be used to represent the various statements of Thingol. Consequently, we will focus on this fragment only, but discuss—if necessary—why we did not pursue alternative features or solutions.

This approach leaves many of Go's most interesting features (e.g. channels or methods) entirely unused. The fragment we use can be understood as a "functional subset" of the Go language, meaning that it comprises 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.

3.1 Syntax

The syntactic fragment used by the Code Generator[3] 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, whereas Featherweight Generic Go does not. We retain this 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 modern Go's syntax for generics, which differs from the one used by Featherweight Generic Go, which pre-dates the introduction of generics in Go 1.18 and was meant as a proposal demonstrating a possible design.

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. A program as a whole is simply a list $\overline{D}$ of such declarations (note that we use overlines such as $\overline{\alpha}$ to mean syntactic repetition).

Structure types A declaration of the form $\texttt{type}\ t_S\left[\overline{\alpha\ c}\right]\ \texttt{struct}\{\overline{A\ \tau}\}$ 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 $\overline{\tau}$. Since Go's syntax demands a constraint c for each type variable α, we always use `any`, which allows any type to be substituted for α.

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.

[3] See the appendix of this paper for the full listing: arxiv.org/abs/2310.02704

Therefore, when encountering non-trivial sum types in Thingol, we must encode them accordingly (see §4.2 for details).

Interface types A declaration of the form `type` $t_I\left[\overline{\alpha\ c}\right]$ `interface{}` introduces a new (empty) interface type to the program. While Go supports non-empty interfaces containing methods, we do not use this feature (see §4.5).

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

This implies that empty interfaces correspond to a "top" type that can denote arbitrary values. Go defines the unconstrained interface `any` as an alias to this empty interface type, which we use extensively in the translation scheme of data types, for reasons that will be explained later (§4.2). Additionally, we also use them for the translation of type classes (§4.5).

Functions A declaration `func` $f\left[\overline{\alpha\ c}\right]$ `(`$\overline{x\ \tau}$`) (`$\overline{\gamma}$`) {` 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 that its functions may return more than one value (note that we have return types $\overline{\gamma}$ instead of just a single return type γ):

```
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; 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 $t_s\left[\overline{\alpha}\right]$ 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 not 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

Unlike in Isabelle (and in Thingol) where “everything is an expression”, Go has the same syntactic distinction between expression and statements that is common across imperative languages: an assignment `x := 42;` is a statement, not an expression, and cannot be used in places where an expression is expected.

However, we constrain our fragment to only include sequences of statements that end in a `return`. This enables easy embedding of a statement into an expression: wrapping it into an immediately-called lambda `func ()` τ `{` s `}()` suffices. Note that a sequence of statements interspersed with `;` is treated syntactically as a single statement.

The remainder of this section introduces the statement forms of our fragment. All but the type assertion should feel familiar from similar languages.

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

If statement A statement of the form `if (`e`) {` s_1 `};` s_2 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 end in `return`, it will then return from the current function. Otherwise, s_2 is evaluated. The form `if (`e`) {` s `} else {` s_2 `}` would be semantically equivalent within our fragment; we avoid it to reduce nesting 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 boolean variable y will indicate if the check was successful. If so, x will be bound to that inner value; otherwise, 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. For brevity, we omit purely syntactic mappings, and focus on the non-trivial steps.

The translation scheme attempts to preserve names as far as possible. Isabelle's Code Generator already provides (re)naming infrastructure, such as generating guaranteed-unused "fresh" names where necessary. In addition to that, some functions and data types require upper-case names, to match Go's rules for exported symbols.

4.1 Types, terms and statements

We define three translations $\textsc{type}(\tau)$, $\textsc{expr}(t)$, and $\textsc{stmt}(t)$. 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 $\textsc{type}(\tau)$ and write both simply as τ. For now, we exclude any mapping of common types (e.g. integers) to built-in Go types; we will revisit this topic later (§4.6).

The other two translations—$\textsc{expr}$ and $\textsc{stmt}$—are used for converting Thingol terms into Go expressions and statements. Which one is used thus depends on what Go expects in each particular context; for example, terms used as function arguments use $\textsc{expr}$; a term which is a function body uses $\textsc{stmt}$. Semantically, for any term t, $\textsc{expr}$ and $\textsc{stmt}$ satisfy the following equivalences:

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

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

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

Although curried abstractions are unusual in Go, no effort is made to uncurry them (unlike top-level functions, which we do uncurry §4.4).

Applications of top-level functions Applications t are more tedious: Definitions of top-level functions are uncurried (§4.4), so 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 taking m arguments.

If so, we have to consider three cases:

1. Fully-satured application ($n = m$); all arguments are passed into f
2. Unsatured application ($n < m$); need to η-expand
3. Over-satured application ($n > m$). 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), the dictionary construction used to encode Isabelle's type classes may introduce additional (value-level) parameters to top-level functions, also adding corresponding additional arguments d_1 to d_r to each of their applications. These are inserted before the user-defined parameters.

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

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

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:

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

Lambda applications If an application $t = t_1\ t_2$ is not a call to a top-level function, then the translation is straightforward: $\textsc{expr}(t_1\ t_2) = \textsc{expr}(t_1)\texttt{(}\textsc{expr}(t_2)\texttt{)}$.

4.2 Data types

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 by other means. For a data type, we generate a new unconstrained interface type δ, meant to represent any constructor f_i of κ.

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. Thus the type `Nat` (Figure 1) becomes:

```
type Nat any;
type Zero struct { };
type Suc struct { A Nat; };
```

With that, we can construct the number 1 as `Nat(Suc{Nat(Zero{})})`. The interface type δ (here `Nat`) acts as a faux sum type: the translation promises that (as long as its input program was type-correct) it will never contain anything but values of types `Zero` and `Suc`. On the Go side, there is no such guarantee: it sees `Nat` as unconstrained, and would happily allow such values as `Nat(Suc{nil})` or even `Suc{"wrong"}`, leading to runtime exceptions elsewhere in the generated code, especially in translated pattern matches (§4.3).

Destructors Along with each constructor's `struct` type, we generate a synthetic function f`_dest` not present in the Thingol program, to be used as a destructor in the translation of Thingol's case expressions (§4.3). Their sole purpose is to unpack and return the individual fields in a `struct` type, exploiting Go's multiple return types.

```
func f_dest (p f) (τ_1, ..., τ_n) {
  return p.A_1, ..., p.A_n
}
```

Destructors are omitted when there are no fields to unpack. For `Nat`, we need only one:

```
func Suc_dest(p Suc)(Nat) { return p.A; }
```

Example Slightly more involved is the α `list` data type (Figure 1). It is polymorphic, and thus requires use of Go's generics:

```
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 from other functional languages such as Standard ML or Haskell: they inspect a *scrutinee* t and match it against a series of clauses $\overline{p_i \rightarrow b_i}$. 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 [9] 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`:

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

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

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:

$$\begin{aligned}\textsc{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}&q, m\texttt{:=}\ t\texttt{.(}f[\overline{\tau}_i]\texttt{);}\\ &\texttt{if (}m\texttt{) \{}A_1, \ldots, A_k\ \texttt{:=}\ f\texttt{\_dest(}t\texttt{);}\ \textsc{stmt}(b)\texttt{\}\}}\end{aligned}$$

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

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

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 the function `hd2` (Figure 1), which takes a list and returns (optionally) the second element of the list. It 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 optimisation described above (in its first two clauses). Also, observe that since unused variables are a compile error in Go, unused bound names above have been generated as _ instead.

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 contains a pattern match translation, as described above.[4] Each equation is treated as a clause of a synthetic case-expression; for functions matching on multiple parameters, we again push inwards and translate as if a nested series of case-expressions were present.

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

Example Consider this definition for `hd2'`, which is semantically equivalent to `hd2`, but written using multiple equations:

```
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
```

The generated Go code is identical.

Special case: top-level constants Thingol accepts top-level definitions that are not functions, for example: `definition a :: nat where a = 10`. Unfortunately, Go admits top-level variable declarations only for monomophic types, and further 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, due to an implementation concern: Go directly compiles methods into virtual tables for dynamic dispatch. A Go interface declares multiple *methods*, where each method type must take the generic value as its zeroth (i.e. implicit) parameter. Thingol has no such restriction. Consider, for example:

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

As Go interfaces, both are invalid: `foo` declares a function whose parameter types do not mention α at all, while `bar`'s function does not take a simple α parameter (but a parameter whose type contains α).

To avoid the additional complexity of treating all these cases separately, we resort to using a dictionary construction [7, 9] in all cases. Since the existing SML target of the Code Generator has to deal with the same issue, the 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.

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 The class declarations (Figure 1) are translated as follows:

```
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 impractical 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 operations 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 as easily mapped by default, because Go does not feature a standard implementation of linked lists.

5 Evaluation

Even though Go greatly differs from the existing targets, we have achieved almost full feature parity with the translation described in this paper. Isabelle constructs that are not (cleanly) mapped are:

– infinite data types, which can be defined e.g. via `codatatype` in Isabelle, but are rejected by Go's compiler;
– some low-level string operations that operate on byte values of characters.

Trusted code base All target language generators are part of Isabelle's *trusted code base*, i.e. bugs inside its own code may lead to bugs in the generated program, and cannot be checked for by Isabelle's kernel. Fortunately, our implementation is "just another module" to the core infrastructure; up until Thingol everything remains unchanged, in line with the other language targets.

However, future (more ambitious) code printing may require changes in Thingol: If code printing shall assume more constructs of Go, it would be useful for Thingol itself to have some concept of the syntactic distinction between expressions and statements.

Code style The generated Go code is not idiomatic, but neither is the generated code for the other languages. Even though the semantics of SML, OCaml and others may more closely resemble the intention of Isabelle users, the generated code in those languages is also littered with syntactic artifacts. This is evidenced by the fact that neither SML nor OCaml support type classes, and Scala code hardly uses type classes in the way that Haskell does (typically prefering object-orientation). Therefore, we do not envision a future need to align the style of the generated code more closely with the preferred style of hand-written Go code.

The main challenge arises from interfacing between generated and hand-written Go code, both of which would be present in a typical application. For instance, constructing values for the translated `datatype` definitions or using curried functions in Go is unfortunately verbose, and can easily introduce errors.

We therefore recommend to write wrapper code that exposes a "cleaner" interface, ready to be consumed by the real-world application. The wrapper must be written carefully: many explicit type annotations are needed in the code, and not all incorrect type annotations will cause compilation to fail. In particular, if a data type's constructor is annotated with a wrong `interface` type, the assumption underlying the translation of case-expressions will fail, resulting in a "match failed" error at runtime (§4.3).

Another awkward source of problems when integrating the generated code with a larger code base is that Go's standard library lacks common functional data structures, such as lists or tuples (§4.6). Hand-written code would need to deal with the necessary conversions (e.g. from a Go array into a linked list). To some extent, this can be alleviated by leveraging third-party libraries for functional data structures, which are unfortunately not popular in the Go community.

5.1 Case studies

We conducted two case studies that have confirmed our approach.

Existing formalisation At Giesecke+Devrient, we use Isabelle for a substantial formalisation of various graph algorithms powering a financial transaction system. The purpose of the formalisation 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. Note that no adaptations of the Isabelle formalisation were necessary, which proves that the Go backend works as a drop-in replacement for the other targets.

Starting from this, we can narrow the formalisation gap mentioned in the introduction. It allows us to link the Isabelle/HOL reference implementation with the real-world production implementation in Go.

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 test suite successfully compiling in Go.

As a consequence, our approach enables the vast majority of Isabelle users to generate Go code without having to rewrite their formalisation. In particular—because we map to a functional fragment of Go—there is no need for users to reach for a deep embedding of an imperative language.

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 Isabelle2024 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 code that utilizes more of Go's standard libraries 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 and Cornelius Diekmann for their contributions to the development. This work has been partially supported by the Federal Ministry of Education and Research (BMBF), Verbundprojekt CONTAIN (13N16582).

Availability The artifact for this paper is available in the *Archive of Formal Proofs (AFP)* [18] and under the DOI https://doi.org/10.5281/zenodo.11608252.

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). https://doi.org/10.1007/978-3-540-71067-7_14
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). https://doi.org/10.1145/3563331
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). https://doi.org/10.1007/978-3-642-12251-4_9
8. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) Types for Proofs and Programs. pp. 160–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74464-1_11
9. Hupel, L.: Certifying Dictionary Construction in Isabelle/HOL. Fundamenta Informaticae **170**(1-3), 177–205 (2019). https://doi.org/10.3233/FI-2019-1859
10. Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
11. 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
12. 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
13. Nipkow, T.: Higher-order rewrite systems. In: Hsiang, J. (ed.) Rewriting Techniques and Applications. pp. 256–256. Springer Berlin Heidelberg, Berlin, Heidelberg (1995). https://doi.org/10.1016/S0304-3975(97)00143-6

14. Nipkow, T., Klein, G.: Concrete semantics with Isabelle/HOL. Springer (2014), http://concrete-semantics.org
15. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F*. Proc. ACM Program. Lang. **1**(ICFP) (aug 2017). https://doi.org/10.1145/3110261
16. Rieu, R.: Development and verification of arbitrary-precision integer arithmetic libraries. Ph.D. thesis, Université Paris-Saclay (Nov 2020), https://theses.hal.science/tel-03032942
17. Schirmer, N.: A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment. Archive of Formal Proofs (February 2008), https://isa-afp.org/entries/Simpl.html, Formal proof development
18. Stübinger, T., Hupel, L.: Go Code Generation for Isabelle. Archive of Formal Proofs (January 2024), https://isa-afp.org/entries/Go.html, Formal proof development

A Syntax of functional Go

Overlines (such as $\overline{\alpha}$) denote syntactic repetition.

Field name	A
Function name	f
Variable name	x, y
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
Structure Type	$t_S\,[\overline{\tau}]$
Interface Type	$t_I\,[\overline{\tau}]$
Function head	$F ::= (\overline{x \;\; \tau})\; (\overline{\tau})$
Type literal	$T ::=$
Structure	$\texttt{struct\{}\overline{A \;\; \tau}\texttt{\}}$
Interface	`interface{}`
Interface constraint	$c ::=$ `any`
Declaration	$D ::=$
Type declaration	$\texttt{type}\;\; t[\overline{\alpha \;\; c}] \;\; T$
Function declaration	$\texttt{func}\;\; f[\overline{\alpha \;\; c}] \;\; F \;\; \texttt{\{} \;\; s \;\; \texttt{\}}$
Expression	$d, e ::=$
Variable	x
Function call	$f[\overline{\alpha}](\overline{e})$
Structure literal	$t_s[\overline{\alpha}]\{\overline{e}\}$
Function abstraction	$\texttt{func}\;\; F \;\; \texttt{\{} \;\; s \;\; \texttt{\}}$
Field selection	$e.A$
Type conversion	$\tau_I(e)$

Null pointer	`nil`
Statement	s ::=
Expression	`return` $\overline{e}$`;`
Variable declaration	x `:=` e`;` s
if-Statement	`if (`e`) {` s `};` s
Type assertion	x`,` y `:=` e`.(`τ`);` s

B Example: Red-black trees

B.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
```

B.2 Scala translation (for reference)

As a point of reference, consider the generated Scala code. Note that the data type definitions in Isabelle/HOL 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 [11].

```
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)
}

}
```

B.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 data type 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 { };
```

```
// 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");
  }
```