A small-step semantics of a concurrent calculus with goroutines and deferred functions

Martin Steffen*

University of Oslo, Department of Informatics

Abstract. In this paper, we present a small-step operational semantics for a small concurrent language supporting *deferred* function calls and related constructs in the style of the Go programming language. For lexical scoping, the presence of higher-order functions, but also the presence of the `defer`-command, requires the notion of *closures* in the semantics.

1 Introduction

New programming languages appear all the time, most as variations and evolutions of earlier languages or with new combinations of established features. Many new designs remain obscure or establish a niche existence, some enjoy their days in the sun, some new general purpose languages even rise to prominence to stay, sometimes accompanied by considerable hype. A recent promising newcomer is Go [19,18,36,13], a language "backed" by Google, which gained quite some momentum after its inception and after going public in 2009. Syntactically, Go's bloodline, tracing back to C, is noticeable in its surface syntax as well as in *simplicity* and conciseness as advertised design principles of the language.

At its core, Go is a lexically scoped, concurrent, imperative language with higher-order functions, supporting object-oriented design. How to most profitably and elegantly combine object-orientation with concurrency is a long-standing question . See for instance [2] for an early discussion of the issue, where the essential design decision is whether objects as units of data coincide with the units of concurrency (in which case the objects are "active") or objects and threads/processes etc. are different. In, e.g., his PhD thesis, Frank de Boer [10] proposed and studied the "parallel object-oriented language" POOL, whose design is firmly in the "active objects" camp, where objects basically are processes, exchanging messages over channels. Many popular concurrent object-oriented languages follow such a design, including actor languages, agent languages etc. The alternative is multi-threading as supported perhaps most prominently by Java and related languages.

Go seems not to fit neatly into either camp. For a start, one may debate to which extent Go is object-oriented. Since the coinage of the term "object orientation" in Simula [9], being object-oriented has become a staple attribute of most modern languages in one way or the other, but unfortunately, there is not overly much consensus on what

* The work was partially supported by the Norwegian-German bilateral PPP project GoRETech ("Go Runtime Enforcement Techniques").

object-orientation exactly is. Whether or not Go is object-oriented is salomonically answered by the Go language FAQ as "yes and no". In general, the consensus opinion seems to be that Go is object-oriented but not entirely as you know it, and that at least that it supports object-oriented programming and design. Officially, there's no concept named "object" in the language, and classes and class inheritance as mechanisms of code reuse are missing. However, Go supports methods, which are functions with "receiver" as specific argument on which they are dynamically dispatched. The mechanism relies on interfaces, *structural* subtyping, there called "duck typing", as opposed to more conventional nominal subtyping disciplines (cf. [31]). In this paper, we ignore Go's static type system (and thereby its object-oriented features) concentrating on some aspects of non-local control flow and goroutines. For a very recent account of Go's type system and a formal calculus formalizing aspects of Go, see [30]. That work, however, does not capture deferred function calls, on which we concentrate in this paper.

Concerning concurrency, Go's primary feature is *asynchronous* function calls (resp. asynchronous method calls). The mechanism is baptized *goroutine* by the developers of the language (basically a lightweight form of threads with low overhead and lacking known thread synchronization mechanisms such as wait and signal). The second core concurrency construct is (typed) *channel* communication, in the tradition of languages like CSP [22,23] or Occam. Since (references to) channels can be sent over channels, Go allows "mobile channel" flexibility for communication as known from the π-calculus [32].

This paper concentrates on two aspects of Go, the structural, non-local control flow with Go's specific constructs of `defer`, `panic`, and `recover` and the notion of goroutines. Deferred function calls are to be executed when the surrounding function returns, and independently of whether that return is done following unexceptional control flow or while "panicking". The command `recover` can be used to exit panicking mode and return to normal execution. For lexical scoping, the small-step semantics uses a variant of closures, so-called capsules [25,26]. For the concurrent execution of multiple goroutines, we use simple *evaluation contexts* where the global configurations have to represent the parent-child relationship between goroutines.

2 A calculus with deferred functions and goroutines

After defining the abstract syntax of the calculus, we define a small-step semantics by structural, operational rules, where in a first step we concentrate on the behavior of one single goroutine (Section 2.2.1). Afterwards, Section 2.2.2 presents the global semantics, covering the concurrent execution of goroutines.

2.1 Abstract syntax

The abstract syntax is given in Table 1. We elide types in this treatment, which will be covered in the technical report, so variable declarations and abstractions are untyped. The code is categorized into terms t and expressions e. A term t is either a value v, where values includes the truth values, the unit value, leaving further values such as integers etc. unspecified, as they are orthogonal to our semantics. A term $\texttt{var}\, x := e\, \texttt{in}\, t$

represents the sequential composition of first e followed by t, where the var-construct binds the local variable x in t, i.e., the construct is also used to represent local, lexical scopes. Furthermore, sometimes we write $\texttt{let}\ x = e\ \texttt{in}\ t$, if the variable x is not *written to* in t, i.e., is used in a single-assignment fashion, and additionally use sequential composition $t_1;t_2$ as abbreviation for $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$, if x is not mentioned free in t_2 at all, i.e., if $x \notin fv(t_2)$. Expressions include function applications and conditionals. New goroutines are created with the expression $\texttt{go}\ ((\lambda x.e_1)\ v)$. Values, which are evaluated expressions, are variables and function abstractions. We use $()$ in this calculus also to represent the absence of a value.

The constructs `defer`, `recover`, and `panic` are used for structured, non-local control flow: panic and return work similar to throwing and catching exceptions and deferred code is executed when the surrounding function call returns, independent from whether a goroutine is panicking or not. Their semantics is discussed in more detail in Section 2.2.1. The construct $\texttt{return}\ v$ is run-time syntax (hence underlined). Go itself has a "terminating statement" `return`, used to hand back results from callee to caller, if any. In our calculus, reducing a function application results in a value, which then is returned without a specific construct in the user-syntax. The `return` is inserted by the reduction rules to demarcate the boundaries of the function call's "stack-frames". This is necessary to appropriately capture the semantics of deferred code.

$$
\begin{array}{lll}
t ::= v \mid \texttt{var}\, x := e\, \texttt{in}\, t & \text{terms} & (1)\\
e ::= t \mid v\ v \mid \texttt{if}\, v\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{go}\ t & \text{expressions} & \\
\quad \mid \texttt{defer}\, ((\lambda x.t)\ v) \mid \texttt{recover} \mid \texttt{panic}\, v \mid \underline{\texttt{return}\, v} & & \\
v ::= x \mid () \mid \texttt{true} \mid \texttt{false} \mid \lambda x.t & \text{values} &
\end{array}
$$

Table 1: Abstract syntax

2.2 Operational semantics

Next we describe the small-step operational semantics of the calculus. The language offers higher-order functions and nested, lexical scopes. Thus function bodies can outlive their surrounding scope in which they are defined. As a consequence, lexical scoping for non-local variables requires a memory discipline more complex than a *stack*-based memory allocation and de-allocation. The phenomenon that a function definition can outlive its defining scope also occurs for deferred function calls, which are executed when the surrounding function returns and not at the place where the surrounding scope (which may be nested) ends. Similarly, goroutines, which are asynchronously executed function calls, have the same effect: when defined, variables refer lexically to a particular scope, but ultimate execution occurs "outside" that scope.

To represent such features, one conventionally uses *closures*. Closures [29] were first implemented in PAL [15,14] and first widely used in Scheme [35,34]. Go indeed

supports closures to enable static scoping. Generally speaking, a closure is a function, i.e., an abstraction together with providing values for the abstraction's free variables.

The semantics in this section concentrates on the local semantics of one goroutine. For simplicity we also ignore reference values, concentrating on specifying the order of reduction in the presence of deferred functions. Instead of using full closures, which would typically require the introduction of references or locations, we make use of so called *capsules* in the formulation of the rules in this presentation. Capsules [25] [26] have been recently introduced as a slightly simpler variant of closures to capture static binding in the presence of higher-order functions. We omit the treatment of references in this section; obviously, they are supported by Go, though. [25] [26] prove that modeling local state with capsules resp. with closures is equivalent.

A capsule environment, or environment for short, is used to model local state, here for one sequential piece of code. An environment is a partial, finite function from variables to *values*. We use $\gamma, \gamma_1, \gamma', \ldots$ for environments. By $dom(\gamma)$, we refer of the domain of γ. We use $\perp$ for the undefined value. Let's write $\bullet$ for the empty capsule environment. A binding from a variable x to a value v is written $[x \mapsto v]$, and in abuse of notation, we write $\gamma[x \mapsto v]$ if the mapping γ is updated by a new binding. That includes adding a new binding, resp., changing an already existing one for x. We also use the notation $[x_0 \mapsto v_1, \ldots, x_n \mapsto v_n]$ or $[\vec{x} \mapsto \vec{v}]$ when referring to a concrete capsule.

Capsules then are tuples consisting of a term t and an environment γ. We write $\gamma \vdash t$ for a capsule. As a standard invariant, it's required (and maintained by the rules) that all free variables of t are covered by the environment, i.e., $dom(\gamma) \supseteq fv(t)$. To model panicking code, we assume one specific variable p not used otherwise. Note that the environment can contain bindings to abstractions which is reminiscent to the notion of *higher-order store* [33].

2.2.1 Defer, panic, and recover Besides standard control-flow structures like loops and conditionals, Go supports various commands for *non-local* control flow. We concentrate on the following three ones, `defer`, `panic`, and `recover` (and ignore constructs like `goto` and `break`). Note that, resulting from a deliberate design decision, Go does *not* support exceptions, even if the behavior of `defer`, `panic`, and `recover` obviously represent some "exceptional" control flow.

The local steps are straightforward and are given as a small-step SOS between capsules. Rule R-VAR restructures a nested var-construct. As the construct generalizes sequential composition, the rule expresses associativity of that construct. Thus it corresponds to transforming $(e_1;t_1);t_2$ into $e_1;(t_1;t_2)$. Note that the grammar insists that, e.g., in an application, both the function and the arguments are values, analogously when acquiring a lock, etc. This form of representation is known as *a-normal form* [17]. Together with the rest of the rules, which perform a case distinction on the first non-var expression in a var-construct, a deterministic left-to-right evaluation is ensured.

Rule R-RED is the basic evaluation step, replacing in the continuation term t the local variable by the value v (where $[x \leftarrow v]$ is understood as capture-avoiding substitution). The `var`-construct introduces a new variable with an initial value v. To allow imperative update, a fresh variable y' is used to store the value in the environment, and

$$\gamma \vdash \texttt{var}\, x_2 := (\texttt{var}\, x_1 := e_1\ \texttt{in}\, t_1)\ \texttt{in}\, t_2 \ \rightarrow\ \gamma \vdash \texttt{var}\, x_1 := e_1\ \texttt{in}\, (\texttt{var}\, x_2 := t_1\ \texttt{in}\, t_2) \quad \text{R-VAR}$$

$$\frac{y' \text{ fresh}}{\gamma \vdash \texttt{var}\, y := v\ \texttt{in}\, t \rightarrow \gamma[y' \mapsto v] \vdash\ t[y \leftarrow y']} \text{ R-RED}$$

$$\gamma \vdash\ x := v; t \ \rightarrow\ \gamma[x \mapsto v] \vdash\ t \quad \text{R-ASSIGN}$$

$$\gamma \vdash \texttt{var}\, x := \texttt{if true then}\, e_1\, \texttt{else}\, e_2\ \texttt{in}\, t \rightarrow \gamma \vdash \texttt{var}\, x :=\ e_1\ \texttt{in}\, t \quad \text{R-IF}_1$$

$$\frac{x' \text{ fresh}}{\gamma \vdash \texttt{var}\, y := (\lambda x.e_1)v\ \texttt{in}\, t \rightarrow \gamma[x' \mapsto v] \vdash \texttt{var}\, y := (\texttt{var}\, x_0 := e_1[x \leftarrow x']\ \texttt{in}\ \underline{\texttt{return}}\, x_0)\ \texttt{in}\, t} \text{ R-APP}$$

$$\gamma \vdash \texttt{defer}\, e_1; (\texttt{var}\, x_0 := e_2\ \texttt{in}\, (\texttt{var}\, y := \underline{\texttt{return}}\, d\ \texttt{in}\, t)) \rightarrow \quad \text{R-DEFER}$$
$$\gamma \vdash \texttt{var}\, x_0 := e_2\ \texttt{in}\, (\texttt{var}\, y := \underline{\texttt{return}}\, e_1; d\ \texttt{in}\, t)$$
$$\gamma \vdash \texttt{panic}\, (v); (\texttt{var}\, x_0 := e_1\ \texttt{in}\, (\texttt{var}\, y := e_2\ \texttt{in}\, t)) \rightarrow \quad \text{R-PANIC}$$
$$\gamma[p \mapsto v] \vdash\ (\texttt{var}\, x_0 := ()\ \texttt{in}\, (\texttt{var}\, y := e_2\ \texttt{in}\, t))$$
$$\gamma \vdash \texttt{var}\, x := \texttt{recover}\, ()\ \texttt{in}\, t \rightarrow \gamma[p \mapsto \bot] \vdash \texttt{var}\, x := \gamma(p)\ \texttt{in}\, t \quad \text{R-RECOVER}$$

$$\frac{\gamma(p) = \bot \qquad y' \text{ fresh}}{\gamma \vdash \texttt{var}\, y := \underline{\texttt{return}}\, v\ \texttt{in}\, t \rightarrow \gamma[y' \mapsto v] \vdash\ t[z' \leftarrow v]} \text{ R-RETURN}_1$$

$$\frac{\gamma(p) = v \neq \bot \qquad y' \text{ fresh}}{\gamma \vdash \texttt{var}\, y := \underline{\texttt{return}}\, v\ \texttt{in}\, t \rightarrow \gamma[p \mapsto \bot] \vdash \texttt{panic}\, (v); t[y' \leftarrow ()]} \text{ R-RETURN}_2$$

Table 2: Transition steps

y is replaced by y' in the continuation of the code. In case the variable is not updated in t, i.e., in a functional, single-assignment setting, the behavior can more simply but equivalently covered by a simple substitution:

$$\gamma \vdash \texttt{let}\, x{:}T = v\ \texttt{in}\, t \rightarrow \gamma \vdash\ t[x \leftarrow v]$$

Sometimes, we will use the function let-construct and the simplified substitution rule when possible. In contrast to R-RED, the assignment treated in R-ASSIGN does not introduce a fresh variable but simply updates the value for an already existing one. Since the assignment does not return a (non-trivial) value, we use sequential composition as syntactic sugar for simplicity for the formulation of the rules. The treatment of conditionals is standard (the rule for `false`, symmetric to R-IF$_1$, is omitted).

The next rule R-APP deals with function calls. Parameter passing is done call-by-value as given in R-APP, where the environment is updated to $\gamma[x' \mapsto v]$. The body of the function is treated by an appropriate substitution $e_1[x \leftarrow x']$. Besides that, the rule

introduces a scope for a new variable x_0 used to store the result of the function body before passing it back to the caller. In a situation where x_0 is not mentioned in the function body e_1, the expression $\texttt{var}\ x_0 := e_1[x \leftarrow x']\ \texttt{in}\ x_0$ corresponds to an (equivalent) η-expansion of the (instantiated) function body $e_1[x \leftarrow x']$. It should be noted that the run-time syntax $\underline{\texttt{return}}\ x_0$ does *not* completely correspond to Go's terminating statement `return`. In our reduction semantics, the return syntax is used to demarcate the end of the top-level stack frame for the function instance currently being executed. The variable x_0 and the η-expanded form of the post-configuration in R-APP is introduced to capture the semantics of *deferred* code (see also R-DEFER).

Deferred code, more precisely, deferred function applications, is executed "when" the function in which the code is deferred, returns. In Go, the signature of a function can specify a named return parameter. For instance, a function taking an integer argument and returning an integer in a specified parameter `x` carries the signature `int (x int)`. The return parameter corresponds to the var-bound variable in rule R-APP. Deferred code, which is executed at the end of the function body, can access and change this return parameter. In the reduction rules, we omitted the type information; using a named return parameter introduces that variable with the function body as scope and is covered by the rule R-APP in that e_1 (and potential deferred code therein) can access and change x_0

The defer-statement is treated in R-DEFER. Defer allows code to be executed "later", exactly at the point where the surrounding function or method returns. An analogous `defer`-command has been introduced recently in Apple's Swift-language [24] as well. Concretely, only function applications, including partial applications, can be deferred in Go, but the rule abstractly mentions just an expression e_1. Note also that while deferred functions are allowed in Go to have a non-trivial return type, the value they eventually may return plays no role. The only way the deferred code can influence the outcome of the surrounding function (besides recovering from a panic) is by side-effects, which includes changing the surrounding function's return value, making use of named return parameters.

A further subtle point about deferred code is what happens if more than one piece of code is being deferred when executing a method body. The discipline adopted is that all of the deferred code will be executed upon return in a LIFO manner. In other words, each stack frame can be thought of being equipped with an "extra stack" of deferred code. Thus, deferred code follows a stack-discipline: within the stack-frame of the surrounding function, code deferred first will be executed last.

Once deferred, the deferred code is *"guaranteed"*[1] to be executed and thus a main purpose of the code is similar to code in the `finally`-clause of exception handling as in languages like Java: it can be used to "clean up" data structures, to close open connections or files, even in case something unexpected happens. There are some high-level differences between `finally`-clauses and deferred function calls. One is that try-catch-

[1] There's an exception to this guarantee, though. Deferred code is executed independent from whether the goroutine panics or not, but it's executed only *if* the enclosing stack frame returns. Divergence may prevent that, and another reason for failing to return is that the goroutine containing the deferred code may be terminated due to the fact that its parent goroutine terminates. See Section 2.2.2.

finally lexically indicates a block of code to which the `finally`-clause belongs to, and once entering the try-catch-finally statement, the final clause is (almost) guaranteed to be executed independent of how the try-block is exited. In contrast, a `defer`-statement may be defined inside a nested block inside a function body, but its execution is delayed until the surrounding function *body* is exited, not the immediately surrounding scope. As a result, the deferred code may typically outlive its immediately surrounding scope much in the same way that nested functions in a higher-order language may outlive their scope. As a consequence, to model or implement the mechanism adequately in a language with lexical scoping, *closures* (or here capsules) are needed.

In general, the mechanism of deferred calls offers greater flexibility compared to finally-clauses as in Java, as deferring code is done at run-time whereas try-catch-finally blocks are statically given. See for instance [3] for a calculus treating exceptions or [1] for a compositional Hoare-style proof system for a Java-like object-oriented language with *exceptions* à la Java.

Example 1. The code in Listing 1.1 illustrates lexical scoping and the need of closures for deferred functions: x in the body of the deferred function refers to the definition of x with value 7. However, this x is updated in the *same scope* later, the value being actually printed in the deferred way is 8. The closure therefore treats its non-local variables "by reference".[2] The increment `x++` at the end of the function body refers clearly to the var-definition in the first line of the function body and hence has nothing to do with the variable being printed.

Listing 1.1: Defer, static scope, and mutable "non-local" variables

```
func main() {
    var x = 0
    {   var x = 7              // local, nested scope
        defer func () {
            fmt.Println(x)     // = 8
        } ()
        x = x+1
    }
    x++
}
```

□

The built-in function `panic` can be used to cause a "panic", which roughly corresponds to throwing an exception. Besides that, panics can occur due to a number of "natural causes" such as attempts to dereference null pointers, out-of-bounds access to arrays and slices, deadlocks, and many more situations. A panic causes the standard execution of a method or function to stop and the control flow to jump directly to the

[2] In other languages, an alternative semantics for closures exists as well, where, when building the closure, the non-local variables obtain their meaning passing them "by value" instead. Of course, a by-value treatment would make it impossible for deferred code to change the return value, after the main body has been exited, for instance due to a panic. Passing by value can be achieved here by handing over the value *explicitly* as an extra formal parameter, effectively using a "λ-lifted" version of the deferred code. Indeed, λ-lifting is a transformation used to give semantics to higher-order functions under lexical scoping [27] and an alternative to closures.

end of the function body. Before returning to the caller, any code previously deferred in the function body will be executed in LIFO fashion. In R-PANIC, x_0 is the designated callee-site variable to hand over the result of the function to the caller. Since no "non-exceptional" value is being returned in an (unrecovered) panicking call, x_0 is irrelevant and set to the corresponding types initial value. Omitting the type information, written summarily as $()$ in R-PANIC. The value of the panic is remembered in γ using the "reserved variable" p. Note that executing deferred code at the end of a panicking function can execute a second panic, which will overwrite the previous one. At each point in time there is at most one panic active. To recover from a panic means to resume the standard mode of execution and a function body having recovered returns as value of the declared type to its caller (as opposed to propagating the panic). R-RECOVER simply retrieves the value of the previously caused panic from p and unsets it.

Example 2 (Defer stack). The function f in the code from Listing 1.2 invokes two function calls in a deferred manner. Instead of deferring $(\lambda().z := z+1)\ ()$, the derivation below just uses $z := z+1$ for simplicity.

Listing 1.2: Stacked defers

```
func f () int {  // alternative: func f () (z int)
        var z = 1
        defer func () {
                z = z+1
        } ()
        defer func () {
                z = z+2
        } ()
        return z                        // 1
}
```

At the beginning of the reduction, in (2) below, we assume that the environment γ_1 contains the definition for the function f. The function in Listing 1.2 does not make use of named return parameters (the return type is just integers), hence the deferred abstraction cannot access it. Therefore, for illustration, the derivation treats x_0 via a let-binding and handing back the value is done via substitution in step from (12) to (13). Note in passing that if the updates to z were not done inside the deferred code, but the function would simply do $z := z+1; z := z+2; z$, then the returned value via z would be 4, not 1.

Similarly, if f would declare z as return parameter in its signature (in which case, z could not declared again via `var` in the same scope), the function would return 4. In the deviation below, the step from (4) to (5) would use variable z (and `var`) instead of the let-bound variable x_0 as shown below.

For reference, the environments in the reductions are, where the γ_1 at the start contains already the definition for the function named f, a binding which remains unchanged:

$$\gamma_2 = \gamma_1[z' \mapsto 1]$$
$$\gamma_3 = \gamma_1[z' \mapsto 3]$$
$$\gamma_4 = \gamma_1[z' \mapsto 4]$$

$$\gamma_1 \vdash \texttt{var}\ y := f\ ()\ \texttt{in}\ t \ \rightarrow \quad (2)$$
$$\gamma_1 \vdash \texttt{var}\ y := (\lambda().\ \texttt{var}\ z := 1\ \texttt{in}\ ((\texttt{defer}\ z := z+1); ((\texttt{defer}\ z := z+2); z)))\ ()\ \texttt{in}\ t \ \rightarrow \quad (3)$$
$$\gamma_1 \vdash \texttt{var}\ y := (\texttt{let}\ x_0 = (\texttt{var}\ z := 1\ \texttt{in}\ ((\texttt{defer}\ z := z+1); ((\texttt{defer}\ z := z+2); z)))\ \texttt{in}\ \texttt{return}\ x_0)\ \texttt{in}\ t \ \rightarrow \quad (4)$$
$$\gamma_1 \vdash \texttt{let}\ x_0 = (\texttt{var}\ z := 1\ \texttt{in}\ ((\texttt{defer}\ z := z+1); ((\texttt{defer}\ z := z+2); z)))\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ x_0\ \texttt{in}\ t) \ \rightarrow \quad (5)$$
$$\gamma_1 \vdash \texttt{var}\ z := 1\ \texttt{in}\ (\texttt{let}\ x_0 = (((\texttt{defer}\ z := z+1); ((\texttt{defer}\ z := z+2); z)))\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ x_0\ \texttt{in}\ t)) \ \rightarrow \quad (6)$$
$$\gamma_2 \vdash \texttt{let}\ x_0 = (((\texttt{defer}\ z' := z'+1); ((\texttt{defer}\ z' := z'+2); z')))\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ x_0\ \texttt{in}\ t) \ \rightarrow \quad (7)$$
$$\gamma_2 \vdash \texttt{defer}\ z' := z'+1; (\texttt{let}\ x_0 = ((\texttt{defer}\ z' := z'+2); z'))\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ x_0\ \texttt{in}\ t) \ \rightarrow \quad (8)$$
$$\gamma_2 \vdash \texttt{let}\ x_0 = ((\texttt{defer}\ z' := z'+2); z'))\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ (z' := z'+1); x_0\ \texttt{in}\ t \ \rightarrow \quad (9)$$
$$\gamma_2 \vdash \texttt{defer}\ z' := z'+2; (\texttt{let}\ x_0 = z'\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ (z' := z'+1); x_0\ \texttt{in}\ t) \ \rightarrow \quad (10)$$
$$\gamma_2 \vdash \texttt{let}\ x_0 = z'\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ z' := z'+2; (z' := z'+1; x_0)\ \texttt{in}\ t \ \rightarrow \quad (11)$$
$$\gamma_2 \vdash \texttt{let}\ x_0 = 1\ \texttt{in}\ (\texttt{let}\ y = \texttt{return}\ z' := z'+2; (z' := z'+1; x_0)\ \texttt{in}\ t) \ \rightarrow \quad (12)$$
$$\gamma_2 \vdash \texttt{let}\ y = \texttt{return}\ z' := z'+2; (z' := z'+1; 1)\ \texttt{in}\ t \ \rightarrow \quad (13)$$
$$\gamma_2 \vdash\ z' := z'+2; (\texttt{let}\ y = \texttt{return}\ z' := z'+1; 1\ \texttt{in}\ t) \ \rightarrow \quad (14)$$
$$\gamma_3 \vdash \texttt{let}\ y = \texttt{return}\ z' := z'+1; 1\ \texttt{in}\ t \ \rightarrow \quad (15)$$
$$\gamma_3 \vdash\ z' := z'+1; (\texttt{let}\ y = \texttt{return}\ 1\ \texttt{in}\ t) \ \rightarrow \quad (16)$$
$$\gamma_4 \vdash \texttt{let}\ y = \texttt{return}\ 1\ \texttt{in}\ t \ \rightarrow \quad (17)$$
$$\gamma_4 \vdash\ t[y \leftarrow 1]\ . \quad (18)$$

□

2.2.2 Goroutines and concurrent execution Concurrency is built into the core of Go, where the unit of concurrency is called *goroutine*, a pun on the notion of coroutines [8]. Coroutines are already a very old concept, originally introduced as a generalization of subroutines, namely roughly as a procedure that can repeatedly yield "intermediate" results, and for non-pre-emptive multitasking. Note in passing that the first object-oriented language Simula [9] [37] supported coroutines already, and a restricted form known as generators or semi-coroutines has been used in various languages. See e.g. [4] for a recent semantical account of a calculus with coroutines (using a small-step semantics as in the presentation here), and including a type and effect system. Further semantical studies and calculi treating coroutines include [6,7,28,12].

Syntactically, starting a goroutine is similar to deferring code. In both cases, a (function or method) *application* is deferred resp. started asynchronously with the command `go`. In both cases, while the function may have a return type and return value, it's not handed back to the caller of the deferred code[3] resp. the spawner of the new goroutine. For example, in Listing 1.2 above illustrating stacked defers, one can replace the two `defer`-commands by two `go`-commands, letting the functions run asynchronously with the parent goroutine. Since three goroutines are then running concurrently, sharing variable z, the result from f is non-deterministic, depending on the scheduling. However, when a goroutine terminates, all its children terminate, as well. For that example, it means: if the parent goroutine, executing the main function and f terminates before the two child goroutines modify the shared variable z, their update to z will not be-

[3] With the exception that deferred code can be used to change the value of return parameters declared in the function's signature.

come effective.[4] It it should be noted that if a goroutine is terminated due to its parent's termination, this also prevents the coroutine's already deferred code from happening: deferred code is guaranteed to happen —panicking or not— upon *return* from a call, but this form of aborting of a goroutine precludes any further returns from being executed.

For instance, the deferred function g in Listing 1.3 may or may not be executed, even if the defer-statement itself happens.

Listing 1.3: Termination and defer

```
func g () {
    defer func () {
    } ()
}

func f ()  {
    go func () {
        defer g()
    } ()
    return
}
func main() {
    go f()
}
```

The semantics therefore needs to account for the parent-child relationship between goroutines. We write $\langle t \rangle$ to denote a goroutine (without child goroutine), where t is the term being executed, and use $\|$ for the parallel composition. Let G stand for a "set" of goroutines running in parallel, as given as follows.

$$G ::= \circ \mid \langle t \parallel G \rangle \tag{19}$$

As usual, parallel composition $\|$ is assumed commutative and associative, and we use $\circ$, representing the empty set of goroutines, as neutral element, i.e., $\langle t \parallel \circ \rangle \equiv \langle t \rangle$. We use $\equiv$ for the induced congruence. Obviously, when stipulating that $\|$ is associative, we mean the arrangement of elements inside $\langle t \parallel \ldots \rangle$, the parenthetic structure using the angle brackets represents the parent-child relationship between goroutines and is not associative.

To formulate the steps for configurations of the form $\gamma \vdash G$, we use *evaluation contexts* [16] (also known as reduction contexts [20]) to specify the redex inside G. Since the evaluation strategy is rather trivial —non-deterministically reducing one term of one goroutine in G— the definition of the contexts is likewise rather simple. An evaluation context is basically a syntactic entity, here G, with exactly one hole (written $[\,\cdot\,]$):

$$E ::= [\,\cdot\,] \mid \langle [\,\cdot\,] \parallel G \rangle \mid \langle t \parallel E \rangle \,. \tag{20}$$

Then $E[t]$ represents the context E with t taking the place of the hole. The global small-step transition relation is then given inductively by the rules of Table 3. Rule R-CONTEXT lifts a local steps to the global level, using the evaluation contexts. Evaluating the go-command spawns asynchronously a new goroutine. The parent-child relationship is captured in that the new goroutine $\langle t \rangle$ runs within the same enclosing

[4] Running the example as is, where the main goroutine does not do much else than spawning two child goroutines, it is practically guaranteed that the parent (and with it the child goroutines) terminates before the children start affecting z.

angle brackets. Note that goroutines don't carry an identity. Such an identity could be used by the spawning goroutine to obtain back an eventual result, if any, from the asynchronously running code. Such a generalization would correspond to the notion of *futures* [21][5]. Cf. also De Boer's paper of a proof-theoretic account of a calculus with asynchronous communication using futures [11]). Of course, the functionality of first-class futures can easily be implemented in Go using channels.

$$\frac{\gamma_1 \vdash\ t_1 \to \gamma_2 \vdash\ t_2}{\gamma_1 \vdash\ E[t_1] \to \gamma_2 \vdash\ E[t_2]}\ \text{R-CONTEXT}$$

$$\frac{}{\gamma \vdash\ E[\langle(\mathsf{go}\ (t_1);t_2) \parallel G\rangle] \to \gamma \vdash\ E[\langle t_2 \parallel \langle t_1\rangle \parallel G\rangle]}\ \text{R-GO}$$

$$\frac{}{\gamma \vdash\ E[\langle v \parallel G\rangle] \to \gamma \vdash\ E[\circ]}\ \text{R-TERM}$$

$$\frac{G_1 \equiv G_1' \qquad G_1 \equiv G_1' \qquad \gamma \vdash\ \langle G_1\rangle \to \gamma \vdash\ \langle G_2\rangle}{\gamma \vdash\ \langle G_1'\rangle \to \gamma \vdash\ \langle G_2'\rangle}\ \text{R-STRUCT}$$

Table 3: Global transition relation

3 Conclusion, discussion, and future work

We presented an operational semantics, in particular capturing concurrency and non-standard control-flow using deferred functions, as they have been introduced in the language Go. Concentrating on the mentioned features, the paper obviously left out many others that deserve study. These include references and reference types, which can be treated in a standard manner, namely by introducing references or locations allocated on a heap; their treatment is orthogonal to the aspects covered here. Other interesting data structures include arrays and slices and their iterators.

Concentrating on the run-time behavior and the operational semantics, the presentation leaves out basically all typing aspects. Go claims to be strongly typed. While being strongly typed is nearly as vague an attribute for a programming language as being "modern" or "high-level", Go certainly is light-years ahead when it comes to imposing typing restrictions with meaningful semantics guarantees, compared to its spiritual predecessor C (from which Go otherwise borrows many syntactic conventions). Rather unconventional for most mainstream object-oriented (typed) languages is to do away with *nominal* typing and nominal subtyping (not to mention to do away with classes, class types, and inheritance, ...). Based on record types (or struct types) and interfaces, Go adopts what is known as *structural* (sub-)typing, as alternative to *nominal* subtyping.

Nominal subtyping was introduced already in the first object-oriented language Simula [9], and ever since has nominal subtyping been the basis for subtype polymorphism for most general mainstream, class-based object-oriented languages, including Smalltalk, Java, C^{++} and more.

Starting with POOL, a "parallel object-oriented language" [10], Frank de Boer provided semantic studies, proof theories, and verification methods for numerous language features related to object-orientation and concurrency (features like channel communication, multi-threading, objects and object creation, inheritance, futures, active objects ...). It would be interesting to make use of the proof techniques he and his colleagues developed to apply to Go with its new take on combining established language features into an interesting design. While mentioning POOL and as a personal remark: during the early stages of my own PhD, I was working in a group interested in formal methods for concurrency and object-orientation. POOL and its proof theory was one of the works we carefully scrutinized and which influenced our own work as it was one of the few solid theoretical studies on this topic available at that time. Though my concrete thesis work afterwards digressed into type theory for functional (non-concurrent) object-oriented calculi, later my interest repeatedly came back to study aspects of concurrency and object-orientation, an interest which has been sparked by work like Frank's about POOL.

Acknowledgments I am also grateful for the thorough, insightful, and detailed feedback from the anonymous reviewers which helped improving the paper.

References

1. Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A deductive proof system for multithreaded Java with exceptions. *Fundamenta Informaticae*, 82(4):391–463 (73 pages), 2008. An extended version of the 2005 conference contribution to FSEN'05 and a reworked and shortened version of the University of Kiel, Dept. of Computer Science technical report 0303.
2. Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
3. D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. *SIGPLAN Notices*, 36:16–30, October 2001.
4. Konrad Anton and Peter Thiemann. Typing coroutines. In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Trends in Functional Programming*, volume 6546 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2011.
5. H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.
6. D. Belsnes and B. M. Østvold. Mixing threads and coroutines, 2005. Unpublished manuscript.
7. J. Berdine, Peter O'Hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15(2–3):181–208, 2002.
8. E. M. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.
9. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67, common base language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

10. Frank S. de Boer. *Reasoning about Dynamically Evolving Process Structures. A Proof Theory for the Parallel Object-Oriented Language POOL.* PhD thesis, Free University of Amsterdam, 1991.
11. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.
12. A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10, 2004.
13. Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.
14. Arthur Evans Jr. PAL — a language designed for teaching programming linguistics. In *Proceedings of the 1968 23rd ACM National Conference*, pages 395–403, New York, NY, USA, 1968. ACM.
15. Arthur Evans Jr. PAL: Pedagogic algorithmic language: A reference manual and primer. Unpublished report, Department of Electrical Engineering, MIT, February 1968.
16. Martin Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
17. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247. ACM, June 1993. In *SIGPLAN Notices* 28(6).
18. The Go programming language specification. `https://golang.org/ref/spec`, August 2015.
19. Google. The Go programming language. `www.golang.org`, 2014.
20. Andrew D. Gordon, Paul D. Hankin, and Søren B. Lassen. Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings of FSTTCS '97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, December 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.
21. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
22. Charles A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–677, 1978.
23. Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
24. Apple Inc. Swift. A modern programming language that is safe, fast, and interactive. `https://developer.apple.com/swift/`, October 2015.
25. Jean Baptise Jeannin. Capsules and closures: a small-step approach. *Electronic Notes in Theoretical Computer Science*, 276:191–293, 2011. Proceedings of the Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).
26. Jean Baptise Jeannin and Dexter Kozen. Computing with capsules. Technical report, Computing and Information Science, Cornell University, January 2011.
27. Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Second Functional Programming Languages and Computer Architecture (Nancy, France)*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer-Verlag, 1985.
28. J. Laird. A calculus of coroutines. In *Proceedings of ICALP 2004*, volume 3142 of *Lecture Notes in Computer Science*, pages 882–893. Springer-Verlag, 2004.
29. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

30. Haiyang Liu and Zongyan Qiu. Go model and object oriented programming. In Alberto Pardo and S. Doaitse Swierstra, editors, *Proceedings of the 19th Brazilian Symposium SBLP 2015, Belo Horizonte, Brazil, September 24–25, 2015*, volume 9325 of *Lecture Notes in Computer Science*, pages 59–74, September 2015.
31. Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In Jan Vitek, editor, *ECOOP 2008: Object-Oriented Programming*, number 5124 in Lecture Notes in Computer Science, pages 260–285. Springer-Verlag, 2008.
32. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.
33. Bernhard Reus and Thomas Streicher. About Hoare logics for higher-order store. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Proceedings of ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2005.
34. G. L. Steele and G. J. Sussman. Scheme: an interpreter for the extended lambda calculus. AI Memo 349, MIT Artificial Intelligence Laboratory, 1975.
35. G. L. Steele and G. J. Sussman. Scheme: an interpreter for the extended lambda calculus. *Higher-Order and Symbolic Computation*, 11:405–439, 1998.
36. Mark Summerfield. *Programming in Go*. Addison-Wesley, 2012.
37. A. Wang and O.-J. Dahl. Coroutine sequencing in a block structured environment. *BIT Numerical Methods*, 11(4):425–449, 1971.