Fencing off Go:
Liveness and Safety for Channel-Based Programming

Julien Lange Nicholas Ng Bernardo Toninho Nobuko Yoshida
Imperial College London, UK
{j.lange, nickng, b.toninho, n.yoshida}@imperial.ac.uk

Abstract

Go is a production-level statically typed programming language whose design features explicit message-passing primitives and lightweight threads, enabling (and encouraging) programmers to develop concurrent systems where components interact through communication more so than by lock-based shared memory concurrency. Go can only detect global deadlocks at runtime, but provides no compile-time protection against all too common communication mismatches or partial deadlocks.

This work develops a static verification framework for bounded liveness and safety in Go programs, able to detect communication errors and partial deadlocks in a general class of realistic concurrent programs, including those with dynamic channel creation and infinite recursion. Our approach infers from a Go program a faithful representation of its communication patterns as a behavioural type. By checking a syntactic restriction on channel usage, dubbed *fencing*, we ensure that programs are made up of finitely many different communication patterns that may be repeated infinitely many times. This restriction allows us to implement bounded verification procedures (akin to bounded model checking) to check for liveness and safety in types which in turn approximates liveness and safety in Go programs. We have implemented a type inference and liveness and safety checks in a tool-chain and tested it against publicly available Go programs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords Channel-based programming, Message-passing programming, Process calculus, Types, Safety and Liveness, Compile-time (static) deadlock detection

1. Introduction

> *Do not communicate by sharing memory; instead, share memory by communicating*
> *Go language proverb [5, 47]*

Go is a statically typed programming language designed with explicit concurrency primitives at the forefront, namely *channels* and *goroutines* (i.e. lightweight threads), drawing heavily from process calculi such as Communicating Sequential Processes (CSP) [20]. Concurrent programming in Go is mostly guided towards channel-based communication as a way to exchange data between goroutines, rather than more classical concurrency control mechanisms (e.g. locks). Channel-based concurrency in Go is lightweight, offering logically structured flows of messages in large systems programming [3, 52], instead of "messy chains of dozens of asynchronous callbacks spread over tens of source files" [4, 37].

On the other hand, Go inherits most problems commonly found in concurrent message-passing programming such as communication mismatches and deadlocks, offering very little in terms of compile-time assurances of correct structuring of communication. While the Go runtime includes a (sound) global deadlock detector, it is ultimately inadequate for complex, large scale applications that may easily be undermined by trivial mistakes or benign changes to the program structure [18, 38], nor can it detect deadlocks involving only a strict subset of a program's goroutines (partial deadlocks).

Liveness and Safety of Communication While Go's type system does ensure that channels are used to communicate values of the appropriate type, it makes no static guarantees about the liveness or channel safety (i.e. channels may be closed in Go, and sending on a closed channel raises a runtime error) of communication in well-typed code. Our work provides a framework for the *static* verification of liveness and absence of communication errors (i.e. safety) of Go programs by extracting *concurrent behavioural types* from Go source code (related type systems have been pioneered by [46] and [25], among others). Our types can be seen as an abstract representation of communication behaviours in the given program. We then perform an analysis on behavioural types, which checks for a bounded form of liveness and communication safety on types and study the conditions under which our verification entails liveness and communication safety of programs.

1.1 Overview

We present a general overview of the steps needed to perform our analysis of concurrent Go programs.

Prime Sieve in Go To illustrate the challenges in analysing Go programs, we begin by considering a rather concise implementation of a concurrent prime sieve (Listing 1, adapting the example from [41] to output infinite primes). This seemingly simple Go program includes intricate communication patterns and concurrent behaviours that are hard to reason about in general due to the combination of (1) unbounded iterative behaviour, (2) dynamic channel creation and (3) spawning of concurrent threads.

The program is made up of three functions: `Generate`, that given a channel `ch` continuously sends along the channel an increasing sequence of integers starting from 2 (encoded as a `for` loop without an exit condition); `Filter`, that given a channel

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009847

```
1  package main
2  func Generate(ch chan<- int) {
3    for i := 2; ; i++ { ch <- i } // Send sequence 2,3...
4  }
5  func Filter(in <-chan int, out chan<- int, prime int){
6    for { i := <-in // Receive value from 'in'.
7      if i%prime != 0 { out <- i } // Fwd 'i' if factor.
8    }
9  }
10 func main() {
11   ch := make(chan int) // Create new channel.
12   go Generate(ch) // Spawn generator.
13   for i := 0; ; i++ {
14     prime := <-ch
15     ch1 := make(chan int)
16     go Filter(ch, ch1, prime) // Chain filter.
17     ch = ch1
18   }
19 }
```

Listing 1. Concurrent Prime Sieve.

for inputs `in`, one for outputs `out`, and a `prime`, continuously forwards a number from `in` to `out` unless it is divisible by `prime`; and `main`, which assembles the sieve by creating a new synchronous channel `ch`, spawning the `Generator` thread (i.e. `go f(x)` spawns a parallel instance of `f`) with the channel `ch`, and then iteratively setting up a chain of `Filter` threads, where the first filter is connected to the generator and the next `Filter`, and so on. We note that with each iteration, a new synchronous channel `ch1` is created that is then used to link each filter instance. The program spawns an infinite parallel composition of `Filter` threads, each pair connected by a dynamically created channel; and the execution of `Generator` and the `Filter` processes in the sieve is non-terminating.

Types of Prime Sieve Our framework infers from the prime sieve program the type $\boldsymbol{t}_0$ given by:

$$\begin{array}{rcl} \boldsymbol{g}(x) & \triangleq & \overline{x}; \boldsymbol{g}\langle x\rangle \\ \boldsymbol{f}(x,y) & \triangleq & x; (\overline{y}; \boldsymbol{f}\langle x,y\rangle \ \oplus \ \boldsymbol{f}\langle x,y\rangle) \\ \boldsymbol{r}(x) & \triangleq & x; (\mathsf{new}\, b)(\boldsymbol{f}\langle x,b\rangle \mid \boldsymbol{r}\langle b\rangle) \\ \boldsymbol{t}_0() & \triangleq & (\mathsf{new}\, a)(\boldsymbol{g}\langle a\rangle \mid \boldsymbol{r}\langle a\rangle) \end{array}$$

The type is described as a system of mutually recursive equations, where the distinguished name $\boldsymbol{t}_0$ types the program entry point. This language is equivalent to a subset of CCS [34] with recursion, parallel and name creation, for which deciding liveness or safety is in general *undecidable* [7, 8]. The type $\boldsymbol{t}_0$ specifies that `main` consists of the creation of a new channel a and the two parallel behaviours $\boldsymbol{g}\langle a\rangle$ and $\boldsymbol{r}\langle a\rangle$. The type given by equation $\boldsymbol{g}(x)$, which types the generator, identifies the infinite output behaviour along the given channel x. The type equation $\boldsymbol{f}(x,y)$ specifies the filter behaviour: input along x and then either output on y and recurse or just recurse. Finally, we decompose the topology set-up as $\boldsymbol{r}(x)$ which inputs along x and then, through the creation of a new channel b, spawns a filter on x and b and recurses on b, creating an infinite parallel composition of filters throughout the execution of the program.

Our type-level analysis relies on the fact that types are able to accurately model a program's communication behaviour. The analysis proceeds in two steps: a simple syntactic check on channel usage in types, dubbed *fencing*, and a *symbolic* finite-state execution of *fenced* types.

Fenced Types Intuitively, if types are fenced, then during their execution there can only be a finite set of channels, or a *fence*, shared by finitely many types (threads). Moreover, fencing ensures that recursive calls under parallel composition eventually involve only local names, shared between finitely many threads. This guarantees that a program consists of finitely many different communication patterns (that may themselves be repeated infinitely many times).

For instance, the recursive call to $\boldsymbol{r}\langle b\rangle$ in the equation $\boldsymbol{r}(x)$ is *fenced*, since the recursive instances of $\boldsymbol{r}$ will not know the channel parameter x hence they cannot spawn threads which share x. This restriction enforces a "finite memory" property wrt. channel names, insofar as given enough recursive unfoldings all the parameters of the recursive call will be names local to the recursive definition.

Symbolic Semantics and Bounded Verification Fenced types can be *symbolically* executed as CCS processes in a representative *finite-state* labelled transition system that consists of a bounded version of the generally unrestricted type semantics. This enables us to produce decision procedures for *bounded* liveness and safety of types, which deems the type $\boldsymbol{t}_0$ as bounded live and safe. In the finite control case, the bounded correctness properties and their unbounded ones coincide. The approach is similar to bounded model checking, using a bound on the number of channels in a type to limit its execution.

From Type Liveness to Program Liveness The final step is to formally relate liveness and safety of types to their analogues in Go programs. For the case of safety, safety of types implies safety for programs. For the case of liveness, programs typically rely on data values to guide their control flow (e.g. in conditional branches) which are abstracted away at the type level. For instance, the `Filter` function only outputs if the received value `i` is not divisible by the given `prime`, but the type for the corresponding process is given by $\boldsymbol{f}(x,y)$ which just indicates an internal choice between two behaviours. The justification for the liveness of $\boldsymbol{t}_0$ is that the internal choice always has the *potential* to enable the output on y (assuming a fairness condition on scheduling). However, it is not necessarily the case that a conditional branch is "equally likely" to proceed to the `then` branch or the `else` branch. In § 5, we define the three classes of programs in which liveness of programs and their types coincide.

1.2 Contributions

We list the main contributions of our work: To define and then show the bounded liveness and safety properties entailed by our analysis, we formalise the message-passing concurrent fragment of the Go language as a process calculus (dubbed MiGo). The MiGo calculus mirrors very closely the intended semantics of the channel-based Go constructs and allows us to express highly dynamical and potentially complex behaviours (§ **2**); We introduce a typing system for MiGo which abstracts behaviours of MiGo as a subset of CCS process behaviours (§ **3**); We define a verification framework for our type language based on the notion of *fences* and *symbolic execution*, showing that for fenced types our symbolic semantics is finite control, entailing the decidability of *bounded* liveness and channel safety (i.e. the notions of liveness and channel safety wrt. the symbolic semantics – § **4**); We characterise the MiGo programs whose liveness and safety are derived from our analysis on types (§ **5**); We show that our results are systematically extended to asynchronous communication semantics (§ **6**); we describe the implementation of our analysis in a tool that we use to evaluate our approach against open-source, publicly available Go programs (§ **7**).

The full proofs, omitted examples and definitions are available in appendix [31], while our implementation and benchmark examples are available online [2].

2. MiGo: A Core Language for Go

This section introduces a core calculus that models the message passing concurrency features of the Go programming language,

$$
\begin{array}{rcl@{\qquad}rcl}
P, Q & := & \pi; P & u & := & a \mid x \\
 & | & \mathsf{close}\, u; P & \pi & := & u!\langle e\rangle \mid u?(y) \mid \tau \\
 & | & \mathsf{select}\{\pi_i; P_i\}_{i\in I} & v & := & \mathsf{n} \mid \mathsf{true} \mid \mathsf{false} \mid x \\
 & | & \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q & e & := & v \mid \mathsf{not}(e) \mid \mathsf{succ}(e) \\
 & | & \mathsf{newchan}(y{:}\sigma); P & D & := & X(\tilde{x}) = P \\
 & | & P \mid Q \mid \mathbf{0} & \boldsymbol{P} & := & \{D_i\}_{i\in I}\, \mathsf{in}\, P \\
 & | & X\langle \tilde{e}, \tilde{u}\rangle & \sigma & := & \mathsf{bool} \mid \mathsf{int} \mid \ldots \\
 & | & (\nu c)P & & & \\
 & | & c\langle\sigma\rangle{::}\tilde{v} \mid c^\star\langle\sigma\rangle{::}\tilde{v} & & &
\end{array}
$$

Figure 1. Syntax of MiGo.

dubbed MiGo (mini-go). Beyond sending and receiving data values along channels, the Go language supports three key concurrency features:

FIFO Queues. Message-passing in Go is achieved via an abstract notion of a lossless, order-preserving communication channel, implemented as a (bounded) FIFO-queue. When the bound on the queue size is 0, communication is *fully synchronous*, whereas with strictly positive bounds the communication is *asynchronous* (i.e. sending is non-blocking if a queue is not full and, dually, receiving is non-blocking if a queue is not empty). The bound is defined upon channel creation and cannot be changed dynamically.

Goroutines. Go supports lightweight threads, dubbed *goroutines*, which denote the spawning of a thread to execute a function concurrently with the main control flow of a program. This feature can be modelled by a combination of parallel composition and process definitions.

Select. The select construct in Go encodes a form of guarded choice, where each branch is guarded by an input or an output on some channel. When multiple branches can be chosen simultaneously, one is chosen at random (through pseudo-random number generation). It is also possible to encode a "timeout" branch in such a choice construct.

Our fencing-based analysis is oblivious to buffer sizes, hence we first focus on fully synchronous communication for ease of presentation, addressing bounded asynchrony in § 6.

2.1 Syntax of MiGo

The syntax of the calculus is given in Figure 1, where P, Q range over *processes*, π over communication *prefixes*, e, e' over *expressions* and x, y over *variables*. We write $\tilde{v}$ and $\tilde{x}$ for a list of expressions and variables, respectively (we use $\cdot$ as a concatenation operator). Programs are ranged over by $\boldsymbol{P}$, consisting of a collection of mutually recursive process definitions (ranged over by D), parameterised by a list of (expression and channel) variables. We omit a detailed enumeration of types such as booleans, floating-point numbers, etc., which are typed with payload types σ. $\mathsf{fn}(P)$ and $\mathsf{fv}(P)$ denote the sets of free names and variables. Process variables X are bound by *definitions* D of the form of $X(\tilde{x}) = P$ where $\mathsf{fv}(P) \subseteq \{\tilde{x}\}$ and $\mathsf{fn}(D) = \emptyset$. We use u to range over channel names a or variables x.

The language constructs are as follows: a *prefixed* (or guarded) process $\pi; P$ denotes the behaviour π (a *send action* of e on u, $u!\langle e\rangle$, a *receive action* on u, bound to y, $u?(y)$, or an *internal action* τ) followed by process P; a *close process* $\mathsf{close}\, u; P$ closes the channel u and continues as P; a *selection* process $\mathsf{select}\{\pi_i; P_i\}_{i\in I}$ denotes a choice between the several P_i processes, where each P_i is guarded by a prefix π_i. Thus, the choice construct non-deterministically selects between any process P_i whose guarding action can be executed (note that a τ action prefix can always be executed, cf. § 2.2); the standard *conditional* process $\mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q$, *parallel* process $P \mid Q$, and the *inactive* process $\mathbf{0}$ (often omitted); a *new channel* process $\mathsf{newchan}(y{:}\sigma); P$ creates a new channel with payload type σ, binding it to y in the continuation P; *process call* $X\langle \tilde{e}, \tilde{u}\rangle$ denotes an instance of the process definition bound to X, with formal parameters instantiated to $\tilde{e}$ and $\tilde{u}$. Both *restriction* $(\nu c)P$ and *buffers* at channel c denote *runtime* constructs (i.e. not written explicitly by the programmer), where the former denotes the runtime handle c for a channel bound in P and a buffer for an *open* channel $c\langle\sigma\rangle{::}\tilde{v}$, containing messages $\tilde{v}$ of type σ, or a buffer for a *closed* channel $c^\star\langle\sigma\rangle{::}\tilde{v}$. A closed channel cannot be used to send messages, but may be used for receive operations an unbounded number of times.

Our representation of a Go program as a program $\boldsymbol{P}$ in MiGo, written $\{D_i\}_{i\in I}\, \mathsf{in}\, P$, consists of a set of mutually recursive process definitions which encode all the goroutines and functions used in the program, together with a process P that encodes the program entry point (i.e. the `main`).

2.1.1 Example – Prime Sieve in MiGo

To showcase the MiGo calculus, we present a concurrent implementation of the sieve of Eratosthenes that produces the infinite sequence of all prime numbers.

The implementation relies on a generator process $G(n, c)$ that outputs natural numbers and a filter process $F(n, i, o)$ that filters out divisible naturals. The code for the generator and filter processes are given below as definitions:

$$
\begin{array}{rcl}
G(n, c) & \triangleq & c!\langle n\rangle; G\langle n{+}1, c\rangle \\
F(n, i, o) & \triangleq & i?(x); \mathsf{if}\ (x\%n \neq 0)\ \mathsf{then}\ o!\langle x\rangle; F\langle n, i, o\rangle \\
 & & \qquad\qquad \mathsf{else}\ F\langle n, i, o\rangle
\end{array}
$$

Definition G stands for the generator process: given the natural number n and channel c, $G(n, c)$ sends the number n along c and recurses on $n{+}1$. Definition F stands for the filter: given a natural n and a pair of channels i and o, $F(n, i, o)$ inputs a number x along i and sends it on o if x is not divisible by n, followed by a recursive call. We then need a way to chain filters together, implementing the sieve:

$$R(c) \quad \triangleq \quad c?(x); \mathsf{newchan}(c'{:}\mathsf{int}); (F\langle x, c, c'\rangle \mid R\langle c'\rangle)$$

The process defined above inputs from the previous element in the chain (either a generator or a filter), creates a new channel which is then used to spawn a new filter process in parallel with a recursive call to R on the new channel. Putting all the components together we obtain the program:

$$\{G(n, c), F(n, i, o), R(c)\}\, \mathsf{in}\, \mathsf{newchan}(c{:}\mathsf{int}); (G(2, c) \mid R(c))$$

As we make precise in § 4.2, the execution of the processes is *fenced* insofar it is always the case that channels are shared (finitely) by a finite number of processes. For instance, name c above is only known to $G(k, c)$ and $F(2, c, c')$. This point is crucial to ensure the feasibility of our approach.

2.1.2 Example – Fibonacci in MiGo

We implement a parallel Fibonacci number generator that computes the n^{th} number of the Fibonacci sequence.

$$
\begin{array}{rcl}
Fib(n, c) & \triangleq & \mathsf{if}\ (n \leq 1)\ \mathsf{then}\ c!\langle n\rangle\ \mathsf{else}\ \mathsf{newchan}(c'{:}\mathsf{int}); \\
 & & (Fib\langle n{-}1, c'\rangle \mid Fib\langle n{-}2, c'\rangle \mid c'?(x); c'?(y); c!\langle x{+}y\rangle)
\end{array}
$$

The definition $Fib(n, c)$ above tests if the given number n is less than or equal to 1. If so, it sends n on c and terminates. Otherwise, the process creates a new channel c', which is then used to run two parallel copies of Fib for the two predecessors of n. The parallel instances are composed with inputs on c' twice which are then added and sent along c.

A sample program that produces the 10^{th} element of the Fibonacci sequence is given below:

$$\{Fib(n,c)\}\ \mathsf{in}\ \mathsf{newchan}(c{:}\mathsf{int});\ (Fib\langle 10,c\rangle \mid c?(u);\mathbf{0})$$

On the other hand, the following program should be deemed not live since the outputs from the recursive calls are never sent to the initial Fib_{bad} call and so the answer is never returned to the main process (i.e. $c?(u);\mathbf{0}$ can never fire).

$$Fib_{bad}(n,c) \triangleq \mathsf{newchan}(c'{:}\mathsf{int});\ (Fib_{bad}\langle n{-}1,c'\rangle \mid Fib_{bad}\langle n{-}2,c'\rangle \mid c'?(x);c'?(y);c!\langle x{+}y\rangle)$$

2.2 Operational Semantics

The semantics of MiGo, written $P \to Q$, is defined by the reduction rules of Figure 2, together with the standard structural congruence $P \equiv Q$ (which includes $\equiv_\alpha$). For process definitions, we implicitly assume the existence of an ambient set of definitions $\{D_i\}_{i\in I}$. Our semantics follows closely the semantics of the Go language: a channel is implemented at runtime by a buffer that is open or closed. Once a channel is closed, it may not be closed again nor can it be used for output. However, a closed channel can always be the subject of an input action, where the received value is a bottom element of the corresponding payload data type. For now, we impose a synchronous semantics (represented in Go with channel of size 0), see § 6 for the asynchronous semantics.

Rule [SCOM] specifies a synchronisation between a send and a receive. Rule [SCLOSE] defines inputs from closed channels, which according to the semantics of Go are always enabled, entailing the reception of a base value v^σ of type σ. Rule [CLOSE] changes the state of a buffer from open ($c\langle\sigma\rangle{::}\tilde{v}$) to closed ($c^\star\langle\sigma\rangle{::}\tilde{v}$). Rule [NEWC] creates a fresh channel c, instantiating it accordingly in the continuation process P and creating the buffer for the channel. Rule [SEL] encodes a mixed non-deterministic choice, insofar as any subprocess P_j that can exhibit a reduction may trigger the choice. The rule [DEF] replaces X by the corresponding process definition (according to the underlying definition environment), instantiating the parameters accordingly. The remaining rules are standard from process calculus literature [44].

2.3 Liveness and Channel Safety

We define a notion of liveness and channel safety for programs through barbs in processes (Definition 2.1). Liveness identifies the ability of communication actions to always eventually fire. Channel safety pertains to the semantics of channels in Go, where closing a channel more than once or sending a message on a closed channel raises a runtime error.

A common pattern in the usage of select in Go is to introduce a timeout (or default) branch, which we model as a τ-guarded branch. This notion of timeout makes the definition of liveness slightly challenging. Consider the following:

$$P_1 \triangleq \mathsf{select}\{a!\langle v\rangle,\ b?(x);\mathbf{0}\ ,\ \tau;P_t\} \qquad R_1 \triangleq a?(y);\mathbf{0}$$
$$P_2 \triangleq \mathsf{select}\{a!\langle v\rangle,\ b?(x);\mathbf{0}\} \qquad R_2 \triangleq c?(y);\mathbf{0}$$

A select with a branch guarded by τ contains a branch that is always enabled by default (since τ actions can always fire silently). Hence if the continuation of the τ prefix P_t is live, then P_1 is live. On the other hand, P_2 by itself *cannot* be live. For P_2 to be live, it must be composed with a process that can offer an input on a or an output on b, with respective live continuations. Hence $P_2 \mid R_1$ is live. However, $P_1 \mid R_i$ is not live unless $P_t \mid R_i$ is live ($i \in \{1,2\}$).

Accounting for these features, we formalise safety and liveness properties, extending the notion of barbed process predicates [36]. Most of the definitions are given in a standard way, with some specifics due to the ability to close channels: input barbs $P \downarrow_a$, denoting that process P is ready to perform an input action on the free channel name a; output barbs $P \downarrow_{\overline{a}}$ are dual; a synchronisation barb $P \downarrow_{[a]}$, indicating that P can perform a synchronisation on a; a channel close barb $P \downarrow_{\mathsf{end}[a]}$, denoting that P can close channel a; and $P \downarrow_{a^\star}$, denoting that P may send from closed channel a. We highlight the predicate $P \downarrow_{\tilde{o}}$, where $\tilde{o}$ is a set of barbs, which applies only for the select construct, stating that the barbs of $\mathsf{select}\{\pi_i;P_i\}_{i\in I}$ are those of all the processes that make up the external choice, provided that *all of them* can exhibit a barb.

$$[\text{SCOM}]\ \frac{e \downarrow v}{c!\langle e\rangle;P \mid c?(y);Q \mid c\langle\sigma\rangle{::}\emptyset \to P \mid Q\{^{v}/_{y}\} \mid c\langle\sigma\rangle{::}\emptyset}$$

$$[\text{SCLOSE}]\quad c?(y);P \mid c^\star\langle\sigma\rangle{::}\emptyset \to P\{^{v^\sigma}/_{y}\} \mid c^\star\langle\sigma\rangle{::}\emptyset$$

$$[\text{CLOSE}]\quad \mathsf{close}\,c;P \mid c\langle\sigma\rangle{::}\tilde{v} \to P \mid c^\star\langle\sigma\rangle{::}\tilde{v}$$

$$[\text{TAU}]\quad \tau;P \to P$$

$$[\text{NEWC}]\ \frac{c \notin \mathsf{fn}(P)}{\mathsf{newchan}(y{:}\sigma);P \to (\nu c)(P\{^{c}/_{y}\} \mid c\langle\sigma\rangle{::}\emptyset)}$$

$$[\text{PAR}]\ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad [\text{RES}]\ \frac{P \to P'}{(\nu c)P \to (\nu c)P'}$$

$$[\text{STR}]\ \frac{P \equiv Q \to Q' \equiv P'}{P \to P'} \qquad [\text{SEL}]\ \frac{\pi_j;P_j \mid P \to R \quad j \in I}{\mathsf{select}\{\pi_i;P_i\}_{i\in I} \mid P \to R}$$

$$[\text{IFT}]\ \frac{e \downarrow \mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to P} \qquad [\text{IFF}]\ \frac{e \downarrow \mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to Q}$$

$$[\text{DEF}]\ \frac{P\{^{\tilde{v},\tilde{c}}/_{\tilde{x}}\} \mid Q \to R \quad e_i \downarrow v_i \quad X(\tilde{x}) = P \in \{D_i\}_{i\in I}}{X(\tilde{e},\tilde{c}) \mid Q \to R}$$

Structural Congruence

$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathbf{0} \equiv P$$
$$(\nu c)(\nu d)P \equiv (\nu d)(\nu c)P \qquad (\nu c)\mathbf{0} \equiv \mathbf{0} \qquad (\nu c)c\langle\sigma\rangle{::}\tilde{v} \equiv \mathbf{0}$$
$$P \mid (\nu c)Q \equiv (\nu c)(P \mid Q)\ (c \notin \mathsf{fn}(P)) \qquad (\nu c)c^\star\langle\sigma\rangle{::}\tilde{v} \equiv \mathbf{0}$$

Figure 2. Reduction Semantics.

Definition 2.1 (Barbs)**.** We define the predicates $\pi \downarrow_o$, $P \downarrow_o$ and $P \downarrow_{\tilde{o}}$ with $o, o_i \in \{a, \overline{a}, [a], \mathsf{end}[a], a^\star\}$.

$$c?(x) \downarrow_c \qquad c!\langle e\rangle \downarrow_{\overline{c}} \qquad \frac{\pi \downarrow_o}{\pi;Q \downarrow_o} \qquad \mathsf{close}\,c;Q \downarrow_{\mathsf{end}[c]} \qquad c^\star\langle\sigma\rangle{::}\vec{v} \downarrow_{c^\star}$$

$$\frac{P \downarrow_o}{P \mid Q \downarrow_o} \qquad \frac{P \downarrow_o \quad a \notin \mathsf{fn}(o)}{(\nu a)P \downarrow_o} \qquad \frac{P \downarrow_o \quad P \equiv Q}{Q \downarrow_o}$$

$$\frac{Q\{^{\tilde{e},\tilde{a}}/_{\tilde{x}}\} \downarrow_o \quad X(\tilde{x}) = Q}{X\langle\tilde{e},\tilde{a}\rangle \downarrow_o}$$

$$\frac{\forall i \in \{1,..,n\}\ :\ \pi_i \downarrow_{o_i}}{\mathsf{select}\{\pi_i;P_i\}_{i\in\{1,..,n\}} \downarrow_{\{o_1 \ldots o_n\}}} \qquad \frac{P \downarrow_a \quad Q \downarrow_{\overline{a}}\ \text{or}\ Q \downarrow_{a^\star}}{P \mid Q \downarrow_{[a]}}$$

$$\frac{P \downarrow_a \quad \pi_i \downarrow_{\overline{a}}}{P \mid \mathsf{select}\{\pi_i;Q_i\}_{i\in I} \downarrow_{[a]}} \qquad \frac{P \downarrow_{\overline{a}}\ \text{or}\ P \downarrow_{a^\star} \quad \pi_i \downarrow_a}{P \mid \mathsf{select}\{\pi_i;Q_i\}_{i\in I} \downarrow_{[a]}}$$

$P \Downarrow_o$ if $P \to^* P'$ and $P' \downarrow_o$ with $o \in \{c, \overline{c}, [c], \mathsf{end}[c], c^\star\}$.

For example, we have that $\neg(P_1 \downarrow_a)$ for any a, whereas $P_2 \downarrow_{\overline{a},b}$. Note that $(P_1 \mid R_1) \downarrow_{[a]}$ and $(P_2 \mid R_1) \downarrow_{[a]}$; and if $P_t \downarrow_o$ then $P_1 \Downarrow_o$ and if $P_t \downarrow_{\overline{c}}$ then $P_1 \mid R_2 \Downarrow_{[c]}$.

We may now define liveness and channel safety: a program is live if, for all the reachable process states, (a) if the state can perform an input or output action on a channel, the state can also eventually perform a synchronisation on that channel; and, (b) if a state can perform a set of actions (i.e. a select where all its guards

$$
\begin{array}{rcl}
T, S & := & \kappa; T \mid \oplus\{T_i\}_{i\in I} \mid \&\{\kappa_i; T_i\}_{i\in I} \mid (T \mid S) \mid \mathbf{0} \\
& \mid & (\mathsf{new}\, a)T \mid \mathsf{end}[u]; T \mid \boldsymbol{t}_X\langle\tilde{u}\rangle \mid (\nu a)T \mid \lfloor a \rfloor \mid a^\star \\
\boldsymbol{T} & := & \{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_i \;\mathsf{in}\; S \qquad \kappa := \overline{u} \mid u \mid \tau
\end{array}
$$

Figure 3. Syntax of Types.

are non-τ), the state can also eventually synchronise on one of the action prefixes.

Definition 2.2 (Liveness)**.** The program $\boldsymbol{P}$ satisfies liveness if for all Q such that $\boldsymbol{P} \to^* (\nu\tilde{c})Q$:

(a) If $Q \downarrow_a$ or $Q \downarrow_{\overline{a}}$ then $Q \Downarrow_{[a]}$.
(b) If $Q \downarrow_{\tilde{a}}$ then $Q \Downarrow_{[a_i]}$ for some $a_i \in \{\tilde{a}\}$.

Channel safety states that in all reachable program states, channels are closed at most once and no process performs outputs on closed channels, as specified by the semantics of the Go language.

Definition 2.3 (Channel Safety)**.** The program $\boldsymbol{P}$ is *channel safe* if for all Q such that $\boldsymbol{P} \to^* (\nu\tilde{c})Q$, if $Q \downarrow_{a^\star}$ then $\neg(Q \Downarrow_{\mathsf{end}[a]})$ and $\neg(Q \Downarrow_{\overline{a}})$.

3. A Behavioural Typing System for MiGo

Go's channel types are related to those of the π-calculus, where the type of a channel carries the type of the objects that threads can send and receive along the channel. Our typing system augments Go's channel types by also serving as a behavioural abstraction of a valid MiGo program, where types take the form of CCS processes with name creation.

3.1 Syntax of Types

The syntax of types T, S is given in Figure 3, mirroring closely that of MiGo processes: The type $\kappa; T$ denotes an output $\overline{u}$, input u along channel u, or an explicit τ action (often used to encode timeouts in external choices), followed by the behaviour denoted by type T. The type $\oplus\{T_i\}_{i\in I}$ represents an internal choice between the behaviours T_i, whereas $\&\{\kappa_i; T_i\}_{i\in I}$ denotes an external choice between behaviours T_i, respectively guarded by prefixes κ_i which drive the choice. Types include parallel composition of behaviours $T \mid S$, inaction $\mathbf{0}$ and channel creation $(\mathsf{new}\, a)T$ (binding a in T). The type $\mathsf{end}[u]; T$ denotes the closing of channel u followed by the behaviour T. The type variable $\boldsymbol{t}_X\langle\tilde{u}\rangle$ associated to a process variable X, denotes the behaviour bound to variable $\boldsymbol{t}_X$ in the definition environment, with formal parameters instantiated to $\tilde{u}$. The type $\{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_i$ in S codifies a set of (parameterised) mutually recursive type definitions $\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i$, bound in S. This set of equations denoted by $\boldsymbol{T}$ is the type assigned to top-level programs $\boldsymbol{P}$.

The type constructs $(\nu a)T$, $\lfloor a \rfloor$ and $a^\star$ denote the type representations of runtime channel bindings, open and closed buffers, respectively. We write $\mathsf{fn}(T)$ and $\mathsf{fv}(T)$ for the free names and variables of type T, respectively; and $\mathsf{n}(T)$ denotes the set of bound and free names.

3.2 Typing System

We first explain the two essential differences from (linear or session-based) type systems of the π-calculus [24, 25, 46]:

Sharing of Channels. We do not enforce linear (disjoint) channel usages, allowing processes to have races. For instance, the process below (with shared y) is typable:

$$\mathsf{newchan}(y{:}\mathsf{bool}); (y!\langle\mathsf{true}\rangle; \mathbf{0} \mid y!\langle\mathsf{false}\rangle; \mathbf{0} \mid y?(x); \mathbf{0})$$

Conditionals. We do not enforce the same types of both branches of the conditional. This design choice stems from the fact that most real programs make use of conditionals precisely to identify points where behaviours need to be different. For instance, consider the following definition:

$$X(c) = c?(x); \mathsf{if}\; x \geq 0 \;\mathsf{then}\; X\langle c\rangle \;\mathsf{else}\; \mathbf{0}$$

The recursive process defined by $X(c)$ receives a potentially unbounded number of positive integers, stopping when the received value x is less than 0. To type such a commonplace programming pattern, we allow the branches in the conditional to hold different types (which are also incompatible by the usual branch subtyping).

Process Typing The judgement ($\Gamma \vdash P \blacktriangleright T$) for processes is defined in Figure 4 where Γ is a typing environment that maintains information about channel payload types, types of bound communication variables and recursion variables, P is a process and T a behavioural type.

We write $\Gamma \vdash \mathcal{J}$ for $\mathcal{J} \in \Gamma$ and $\Gamma \vdash e : \sigma$ to state that the expression e is well-typed according to the types of variables in Γ. We write $u{:}\mathsf{ch}(\sigma)$ to denote that u stands for a channel with payload type σ. We omit the typing rules of expressions e, given that expressions only include basic data types. We write $\mathsf{dom}(\Gamma)$ for the set of channel bindings in Γ.

The rules implement a very close correspondence between processes and their respective types: Rule ⟨OUT⟩ types output processes with the output prefix type $\overline{u}; T$, checking that the type of the object to be sent matches the payload type σ of channel u, and that the continuation P has type T. The rule ⟨IN⟩ for inputs is dual. Rule ⟨SEL⟩ types the select construct with the external choice type, whereas rule ⟨IF⟩ types the conditional as a binary internal choice between the type S corresponding to P and the type T corresponding to Q.

The typing rules for close, zero, parallel and τ are straightforward. Rule ⟨NEW⟩ allocates a fresh type-level channel name with payload type σ. Rule ⟨VAR⟩ matches a process variable with its corresponding type variable, checking that the specified arguments have the appropriate types.

Program Typing The judgement ($\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$) is defined in Figure 4. A process declaration $X(\tilde{x}{:}\tilde{\sigma}, \tilde{y}{:}\mathsf{ch}(\tilde{\sigma}')) = P$ is matched with $\boldsymbol{t}_X(\tilde{y}) = T$, connecting the process level variable X with the type variable $\boldsymbol{t}$, where P may use any of the parameters specified in the recursion variable. The typing rule for programs ⟨DEF⟩ assigns a program the type $\{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_{i\in I}$ in S, checking that each definition is typed with $\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i$ and that the main process Q has type S.

Runtime Process Typing The judgement ($\Gamma \vdash_B P \blacktriangleright T$) types a process created after execution of a program (called runtime process). B is a set of channels with associated runtime buffers to ensure their uniqueness. Runtime channel bindings are typed by rule ⟨RES⟩, where given a process of type T that can use the buffered channel c, we type $(\nu c)P$ with $(\nu c)T$ removing c from the set s since it is local to P (and T). Closed and open buffers are typed by rules ⟨CBUFF⟩ and ⟨BUFF⟩, respectively, noting that the set of active buffers is a singleton containing the appropriate buffer reference. The parallel rule ⟨PARR⟩ ensures that the buffers of both processes do not overlap (hence only a single buffer for each name exists in the context).

Notation 3.1. In the remainder of this paper, we refer to the type of a program as a system of type equations $\boldsymbol{T}$ which is obtained from the program by collecting all the types for definitions and adding a distinguished unique definition $\boldsymbol{t}_0() = S$ for the program entry point. We often write X_0 to stand for the process variable of the program entry point.

$\boxed{\Gamma \vdash P \blacktriangleright T}$

$$\langle \text{OUT} \rangle \frac{\Gamma \vdash u{:}\mathsf{ch}(\sigma) \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash u!\langle e\rangle; P \blacktriangleright \overline{u}; T}$$

$$\langle \text{IN} \rangle \frac{\Gamma \vdash u{:}\mathsf{ch}(\sigma) \quad \Gamma, x{:}\sigma \vdash P \blacktriangleright T}{\Gamma \vdash u?(x); P \blacktriangleright u; T} \quad \langle \text{TAU} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T}$$

$$\langle \text{CLOSE} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{close}\, u; P \blacktriangleright \mathsf{end}[u]; T} \quad \langle \text{ZERO} \rangle \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}}$$

$$\langle \text{SEL} \rangle \frac{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}{\Gamma \vdash \mathsf{select}\{\pi_i; P_i\}_{i\in I} \blacktriangleright \&\{\kappa_i; T_i\}_{i\in I}}$$

$$\langle \text{IF} \rangle \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \blacktriangleright S \quad \Gamma \vdash Q \blacktriangleright T}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q \blacktriangleright \oplus\{S\,,\, T\}}$$

$$\langle \text{NEW} \rangle \frac{\Gamma, y{:}\mathsf{ch}(\sigma) \vdash P \blacktriangleright T \quad c \notin \mathsf{dom}(\Gamma) \cup \mathsf{fn}(T)}{\Gamma \vdash \mathsf{newchan}(y{:}\sigma); P \blacktriangleright (\mathsf{new}\, c)T\,\{^c/_y\}}$$

$$\langle \text{PAR} \rangle \frac{\Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \mid Q \blacktriangleright (T \mid S)}$$

$$\langle \text{VAR} \rangle \frac{\Gamma \vdash \tilde{e}{:}\tilde{\sigma} \quad \Gamma \vdash \tilde{u}{:}\mathsf{ch}(\tilde{\sigma}')}{\Gamma, X(\tilde{\sigma}, \mathsf{ch}(\tilde{\sigma}')) \vdash X\langle \tilde{e}, \tilde{u}\rangle \blacktriangleright \boldsymbol{t}_X\langle \tilde{u}\rangle}$$

$\boxed{\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}}$

$$\langle \text{DEF} \rangle \frac{\begin{array}{c}\Gamma, X_i(\tilde{\sigma}_i, \mathsf{ch}(\tilde{\sigma}_i')), \tilde{x}_i{:}\tilde{\sigma}_i, \tilde{y}_i{:}\mathsf{ch}(\tilde{\sigma}_i') \vdash P_i \blacktriangleright T_i \\ \Gamma, X_1(\tilde{\sigma}_1, \mathsf{ch}(\tilde{\sigma}_1')), ..., X_n(\tilde{\sigma}_n, \mathsf{ch}(\tilde{\sigma}_n')) \vdash Q \blacktriangleright S\end{array}}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i\in I}\ \mathsf{in}\ Q \blacktriangleright \{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_{i\in I}\ \mathsf{in}\ S}$$

$\boxed{\Gamma \vdash_B P \blacktriangleright T}$

$$\langle \text{INT} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash_\emptyset P \blacktriangleright T} \quad \langle \text{RES} \rangle \frac{\Gamma, c{:}\mathsf{ch}(\sigma) \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B\setminus c} (\nu c)P \blacktriangleright (\nu c)T}$$

$$\langle \text{CBUFF} \rangle \frac{\Gamma \vdash a{:}\mathsf{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a^\star\langle\sigma\rangle{::}\tilde{v} \blacktriangleright a^\star} \quad \langle \text{BUFF} \rangle \frac{\Gamma \vdash a{:}\mathsf{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a\langle\sigma\rangle{::}\tilde{v} \blacktriangleright \lfloor a \rfloor}$$

$$\langle \text{PARR} \rangle \frac{\Gamma \vdash_B P \blacktriangleright T \quad \Gamma \vdash_{B'} Q \blacktriangleright S \quad B \cap B' = \emptyset}{\Gamma \vdash_{B\cup B'} P \mid Q \blacktriangleright (T \mid S)}$$

Figure 4. Typing Rules (Processes and Programs).

4. Bounded Verification of Behavioural Types

This section introduces our main definition, *fencing*, and a bounded verification of liveness and channel safety for types. Our development consists of the following steps:

Step 1. Define a syntactic restriction on types, dubbed a *fence*, guaranteeing that whenever *fenced* types model a program that spawns infinitely many processes, the program actually consists of finitely many communication patterns (which may be repeated infinitely many times).

Step 2. Define a *symbolic semantics* for types, which generates a representative *finite-state* labelled transition system (LTS) whenever types validate the fencing predicate.

Step 3. Prove that liveness and channel safety are decidable for the bounded *symbolic executions* of fenced types.

4.1 Types as Processes: Semantics

The semantics of our types is given by the labelled transition system (LTS), extending that of CCS, defined in Figure 5. The labels,

$$|\text{SND}| \quad \overline{a}; T \xrightarrow{\overline{a}} T \quad |\text{RCV}| \quad a; T \xrightarrow{a} T \quad |\text{TAU}| \quad \tau; T \xrightarrow{\tau} T$$

$$|\text{SEL}| \frac{j \in I}{\oplus\{T_i\}_{i\in I} \xrightarrow{\tau} T_j} \qquad |\text{BRA}| \frac{\kappa_j; T_j \xrightarrow{\alpha} T_j}{\&\{\kappa_i; T_i\}_{i\in I} \xrightarrow{\alpha} T_j}$$

$$|\text{PAR}| \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \qquad |\text{COM}| \frac{T \xrightarrow{\beta} T' \quad S \xrightarrow{a} S' \quad \beta = \overline{a}, a^\star}{T \mid S \xrightarrow{[a]} T' \mid S'}$$

$$|\text{NEW}| \ (\mathsf{new}\, a)T \xrightarrow{\tau} (\nu a)(T \mid \lfloor a \rfloor) \quad |\text{END}| \ \mathsf{end}[a]; T \xrightarrow{\mathsf{end}[a]} T$$

$$|\text{BUF}| \ \lfloor a \rfloor \xrightarrow{\overline{\mathsf{end}}[a]} a^\star \qquad |\text{CLOSE}| \frac{T \xrightarrow{\mathsf{end}[a]} T' \quad S \xrightarrow{\overline{\mathsf{end}}[a]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$$

$$|\text{CLD}| \ a^\star \xrightarrow{a^\star} a^\star$$

$$|\text{RES-1}| \frac{T \xrightarrow{\alpha} T' \quad \mathsf{fn}(\alpha) \neq \{a\}}{(\nu a)T \xrightarrow{\alpha} (\nu a)T'} \qquad |\text{RES-2}| \frac{T \xrightarrow{[a]} T'}{(\nu a)T \xrightarrow{\tau} (\nu a)T'}$$

$$|\text{EQ}| \frac{T \equiv_\alpha T' \quad T' \xrightarrow{\alpha} T''}{T \xrightarrow{\alpha} T''} \qquad |\text{DEF}| \frac{T\,\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \quad \boldsymbol{t}(\tilde{x}) = T}{\boldsymbol{t}\langle\tilde{a}\rangle \xrightarrow{\alpha} T'}$$

Figure 5. LTS Semantics of Types.

ranged over by α and β, have the form:

$$\alpha, \beta ::= \ \overline{a} \mid a \mid \tau \mid [a] \mid \mathsf{end}[a] \mid \overline{\mathsf{end}}[a] \mid a^\star$$

Labels denote send and receive actions ($\overline{a}$ and a), silent transitions τ, synchronisation over a channel $[a]$, the request and acceptance of channel closure ($\mathsf{end}[a]$ and $\overline{\mathsf{end}}[a]$), and send actions from a closed channel $a^\star$. We write $\boldsymbol{t}(\tilde{x}) = T$ if $\boldsymbol{t}(\tilde{x}) = T$ is in $\boldsymbol{T}$.

Rule |SND| (resp. |RCV|) allows a type to emit a send (resp. receive) action on a channel a. Rules |SEL| and |BRA| model internal and (mixed) external choices, respectively. Rule |COM| allows two types to synchronise on a dual action (send with receive, or closed channel with receive). Rule |NEW| creates a new (open) channel for a. Rule |END|, together with rule |CLOSE|, allows a type to request the closure of channel a. Rule |BUF| models a transition from an open channel to a closed one, and rule |CLD| models the perpetual ability of a closed channel to emit send actions. The other rules are standard from CCS. In Figure 5, we omit the symmetric rules for |CLOSE|, |PAR|, and |COM|. We define structural congruence rules over types as follows:

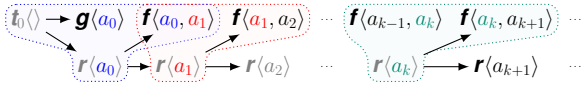
$$\begin{array}{l} T \mid S \equiv S \mid T \quad T \mid (T' \mid S) \equiv (T \mid T') \mid S \quad T \mid \mathbf{0} \equiv T \\ (\nu a)(\nu b)T \equiv (\nu b)(\nu a)T \ \ (\nu a)\mathbf{0} \equiv \mathbf{0} \ \ (\nu a)a^\star \equiv \mathbf{0} \ \ (\nu a)\lfloor a \rfloor \equiv \mathbf{0} \\ T \mid (\nu a)S \equiv (\nu a)(T \mid S) \ (a \notin \mathsf{fn}(T)) \quad T \equiv_\alpha T' \ \Rightarrow T \ \equiv T' \end{array}$$

We write $\rightarrow$ for $\xrightarrow{\tau} \cup \equiv$ and $T \rightarrow^* \xrightarrow{\alpha}$ if there exist T' and T'' such that $T \rightarrow^* T' \xrightarrow{\alpha} T''$.

4.2 Fenced Types and Fencing Predicate

This section develops **Step 1** by defining the fencing predicate. We illustrate the key intuitions with an example.

Recall the prime sieve program (§ 2.1.1) given in § 1.1, as inferred by the type system of § 3. We represent the execution wrt. the semantics of § 4.1, via the diagram below.



In the diagram, a node represents an instance of a concurrently executing type (or thread) and the arrows represent the parent-child relation. For instance, $\boldsymbol{t}_0\langle\rangle$ is the parent of $\boldsymbol{g}\langle a_0\rangle$ and $\boldsymbol{r}\langle a_0\rangle$.

We observe that there are only finitely many kinds of concurrent threads (i.e. $\boldsymbol{t}_0\langle\rangle$, $\boldsymbol{g}\langle a\rangle$, $\boldsymbol{f}\langle a, b\rangle$, and $\boldsymbol{r}\langle a\rangle$). Also given any name in

$\boxed{G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} T}$

$$\lfloor \text{AXIOM} \rfloor \frac{\tilde{y} \neq \varepsilon \;\vee\; \tilde{u} \prec \tilde{z}}{G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} \boldsymbol{t}\langle \tilde{u} \rangle} \qquad \lfloor \text{DEF-}\in \rfloor \frac{\boldsymbol{s}\langle \tilde{u} \rangle \in G}{G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} \boldsymbol{s}\langle \tilde{u} \rangle}$$

$$\lfloor \text{END} \rfloor \frac{}{\Delta \vdash_{\boldsymbol{t}} \boldsymbol{0}} \qquad \lfloor \text{PAR} \rfloor \frac{G\,;\varepsilon\,;\tilde{z}\cdot\tilde{y} \vdash_{\boldsymbol{t}} T_1 \quad G\,;\varepsilon\,;\tilde{z}\cdot\tilde{y} \vdash_{\boldsymbol{t}} T_2}{G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} T_1 \mid T_2}$$

$$\lfloor \text{DEF-}\notin \rfloor \frac{\boldsymbol{t} \neq \boldsymbol{s} \;\; \boldsymbol{s}\langle \tilde{u} \rangle \notin G \;\; G \cdot \boldsymbol{s}\langle \tilde{u} \rangle\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} T\,\{^{\tilde{u}}/_{\tilde{x}}\} \quad \boldsymbol{s}(\tilde{x}) = T}{G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} \boldsymbol{s}\langle \tilde{u} \rangle}$$

$$\lfloor \text{SEL} \rfloor \frac{\forall i \in I \,:\, \Delta \vdash_{\boldsymbol{t}} T_i}{\Delta \vdash_{\boldsymbol{t}} \oplus\{T_i\}_{i\in I}} \qquad \lfloor \text{BRA} \rfloor \frac{\forall i \in I \,:\, \Delta \vdash_{\boldsymbol{t}} T_i}{\Delta \vdash_{\boldsymbol{t}} \&\{T_i\}_{i\in I}}$$

$$\lfloor \text{END} \rfloor \frac{\Delta \vdash_{\boldsymbol{t}} T}{\Delta \vdash_{\boldsymbol{t}} \mathsf{end}[u]; T} \quad \lfloor \text{RES} \rfloor \frac{\Delta \vdash_{\boldsymbol{t}} T}{\Delta \vdash_{\boldsymbol{t}} (\mathsf{new}\, a) T} \quad \lfloor \text{PREF} \rfloor \frac{\Delta \vdash_{\boldsymbol{t}} T}{\Delta \vdash_{\boldsymbol{t}} \kappa; T}$$

Figure 6. Rules for Fencing, where Δ stands for $G\,;\tilde{y}\,;\tilde{z}$.

the diagram, e.g. a_1, it is shared only by finitely many threads. For instance, a_1 is "known" only to $\boldsymbol{f}\langle a_0, a_1\rangle$, $\boldsymbol{f}\langle a_1, a_2\rangle$, and $\boldsymbol{r}\langle a_1\rangle$. Note that this situation does not change during the execution of the program because (1) types cannot exchange names in communication and (2) the grand-children of, e.g. $\boldsymbol{r}\langle a_1\rangle$, do not know a_1, hence they cannot spawn further threads sharing that name.

Intuitively, we can observe that despite the fact that the prime sieve generates an unbounded number of fresh channels and threads, it consists of finitely many different communication patterns (i.e. the coloured regions above). It is this general pattern that we capture with *fencing*.

4.2.1 Fencing Types

We introduce a judgement which ensures that a system of types is *fenced*: given a finite set of names called *fence*, the types executing within that fence approximate a form of finite control. We use judgements of the form $G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} T$ to guarantee that a type definition $\boldsymbol{t}(\tilde{x}) = T$ is *fenced*; where G records previously encountered recursive calls, $\tilde{y}$ represents the names that $\boldsymbol{t}$ can use if T is *single-threaded*, and $\tilde{z}$ represents the names that a sub-term of T can use if T is a *multi-threaded* type. We write ε for the empty environment. The judgement $G\,;\tilde{y}\,;\tilde{z} \vdash_{\boldsymbol{t}} T$ holds if it can be inferred by the rules of Figure 6, which use a relation on sequences of names that ensures a strictly decreasing usage of names, defined below.

Definition 4.1 ($\prec$-relation)**.** We write $\tilde{u} \prec \tilde{x}$ iff (1) $\tilde{x} = x_1 \cdots x_n$, (2) $\tilde{u} = x_{k+1} \cdots x_n \cdot a_1 \cdots a_k$, with $k \geq 1$, and (3) $\forall 1 \leq i \leq n \,:\, \forall 1 \leq j \leq k \,:\, x_i \neq a_j$.

The relation $\prec$ enforces that types featuring parallel composition have a "finite memory" wrt. the names over which they can recurse. For instance, $yza \prec xyz$, but $xaz \not\prec xyz$.

We comment on the key rules of Figure 6. Rule $\lfloor \text{PAR} \rfloor$ identifies multi-threaded types by moving the $\tilde{y}$ environment to the $\tilde{z}$ environment. The axiom $\lfloor \text{AXIOM} \rfloor$ states that $\boldsymbol{t}\langle \tilde{u} \rangle$ is fenced if either (i) $\tilde{y} \neq \varepsilon$, i.e. the type corresponding to $\boldsymbol{t}$ does not contain any parallel composition; or (ii) $\tilde{u} \prec \tilde{z}$ holds, i.e. any recursive call over $\boldsymbol{t}$ uses strictly fewer (non newly created) names. The second part of the premise guarantees that after a certain number of recursive calls, the type $\boldsymbol{t}$ will have completely "forgotten" the names it started executing with; hence moving outside of the fence. Rules $\lfloor \text{DEF-}\in \rfloor$ and $\lfloor \text{DEF-}\notin \rfloor$ deal with recursive calls to different type variables.

Definition 4.2 (Fenced types)**.** We say $\boldsymbol{T}$ is *fenced* (denoted by $\mathsf{Fenced}(\boldsymbol{T})$) if for all $\boldsymbol{t}(\tilde{x}) = T$ in $\boldsymbol{T}$, either $\tilde{x} = \varepsilon$ or $\varepsilon\,;\tilde{x}\,;\varepsilon \vdash_{\boldsymbol{t}} T$ holds.

A consequence of fencing is that for each equation $\boldsymbol{t}(\tilde{x}) = T$ such that $\tilde{x} \neq \varepsilon$, either (1) T is *single-threaded* (i.e. there is no parallel composition within T), hence there is no restriction as to the parameters $\boldsymbol{t}$ recurses on; or (2) T is *multi-threaded* (i.e. there is a parallel composition within T), hence for each occurrence of a $\boldsymbol{t}$-recursion, at least one of the parameters must be forgotten. In the prime sieve example, the types $\boldsymbol{g}(x)$ and $\boldsymbol{f}(x, y)$ always recurse on the same parameters, but they do not include parallel composition. While the type $\boldsymbol{r}(x)$ has a parallel composition, its parameter x is "forgotten" at the $\boldsymbol{r}\langle b\rangle$ recursive call (i.e. $b \prec x$).

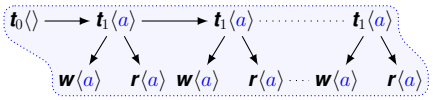
4.2.2 Examples on Fencing

No Fence We now give an example that does *not* validate the fencing predicate. The types below model a program that spawns a reader and a writer on a channel a infinitely many times.

$$\boldsymbol{w}(x) \triangleq \overline{x};\boldsymbol{w}\langle x\rangle = T_w \qquad \boldsymbol{t}_1(x) \triangleq \boldsymbol{w}\langle x\rangle \mid \boldsymbol{r}\langle x\rangle \mid \boldsymbol{t}_1\langle x\rangle = T_1$$
$$\boldsymbol{r}(x) \triangleq x;\boldsymbol{r}\langle x\rangle = T_r \qquad \boldsymbol{t}_0() \triangleq (\mathsf{new}\, a)(\boldsymbol{t}_1\langle a\rangle) = T_0$$

We have: $\varepsilon\,;x\,;\varepsilon \vdash_{\boldsymbol{w}} T_w$, $\varepsilon\,;x\,;\varepsilon \vdash_{\boldsymbol{r}} T_r$, and $\varepsilon\,;\varepsilon\,;\varepsilon \vdash_{\boldsymbol{t}_0} T_0$ hold (since they do not feature any parallel composition). However, $\varepsilon\,;x\,;\varepsilon \vdash_{\boldsymbol{t}_1} T_1$ does *not* hold. This is due to the fact that the axiom $\lfloor \text{AXIOM} \rfloor$ cannot be applied (i.e. $\neg(x \prec x)$).

The topology induced by the type is given as:

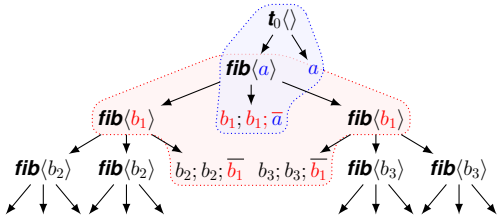


We can observe that there are infinitely many instances of types that "know" the name a.

Fibonacci Below, we give the types for the Fibonacci example, cf. § 2.1.2.

$$\begin{array}{lcll} \boldsymbol{fib}(x) & \triangleq & \overline{x} \oplus (\mathsf{new}\, b)(\boldsymbol{fib}\langle b\rangle \mid b;b;\overline{x} \mid \boldsymbol{fib}\langle b\rangle) & = T_{\boldsymbol{fib}} \\ \boldsymbol{t}_0() & \triangleq & (\mathsf{new}\, a)(\boldsymbol{fib}\langle a\rangle \mid a) & = T_0 \end{array}$$

Observe that $\boldsymbol{t}_0()$ is trivially fenced since it has no parameter (cf. Definition 4.2). The judgement $\varepsilon\,;x\,;\varepsilon \vdash_{\boldsymbol{fib}} T_{\boldsymbol{fib}}$ also holds since we have $b \prec x$. Essentially, the equation $\boldsymbol{fib}$ validates the fencing predicate because each recursively spawned child does not have access to the parameter x. We illustrate the behaviour of this type in the diagram below.



Two fences are highlighted in the diagram: the $\{a\}$-fence includes four parallel types (including the initial $\boldsymbol{t}_0$); while the $\{b_1\}$-fence includes five components: one instance of $\boldsymbol{fib}(x)$ as well as two of its recursive children and three instances of the non-recursive component of $\boldsymbol{fib}$.

4.3 Symbolic Semantics

For **Step 2**, we introduce a symbolic semantics for types, which is parameterised by a bound on the number of free names that can be used when unfolding a recursive call, e.g. $\boldsymbol{t}\langle \tilde{u}\rangle$, by its corresponding definition. The overall purpose of the symbolic semantics is that for any $\boldsymbol{T}$ such that $\mathsf{Fenced}(\boldsymbol{T})$, the symbolic LTS of $\boldsymbol{T}$ is *finite state*.

$$|\textsc{def}|\frac{N \lhd T\,\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha}_k N \lhd T' \quad \boldsymbol{t}(\tilde{x}) = T \quad \tilde{a} \cap N \neq \emptyset}{N \lhd \boldsymbol{t}\langle \tilde{a} \rangle \xrightarrow{\alpha}_k N \lhd T'}$$

$$|\textsc{r1}<|\frac{N \uplus \{a\} \lhd T \xrightarrow{\alpha}_k N \uplus \{a\} \lhd T' \quad \mathsf{fn}(\alpha) \neq \{a\} \quad |N| < k}{N \lhd (\nu a)T \xrightarrow{\alpha}_k N \lhd (\nu a)T'}$$

$$|\textsc{r2}<|\frac{N \uplus \{a\} \lhd T \xrightarrow{[a]}_k N \uplus \{a\} \lhd T' \quad |N| < k}{N \lhd (\nu a)T \xrightarrow{\tau}_k N \lhd (\nu a)T'}$$

$$|\textsc{r1}\geq|\frac{N \lhd T \xrightarrow{\alpha}_k N \lhd T' \quad \mathsf{fn}(\alpha) \neq \{a\} \quad |N| \geq k}{N \lhd (\nu a)T \xrightarrow{\alpha}_k N \lhd (\nu a)T'}$$

$$|\textsc{r2}\geq|\frac{N \lhd T \xrightarrow{[a]}_k N \lhd T' \quad |N| \geq k}{N \lhd (\nu a)T \xrightarrow{\tau}_k N \lhd (\nu a)T'}$$

Figure 7. Symbolic Semantics for Types.

The symbolic semantics for types is given in Figure 7, where we show only the interesting new rules. The other rules are essentially those of Figure 5, with the additional parameters k and N as expected. Rule |DEF| replaces its counterpart from Figure 5, while rules [R1<] and |R1≥| replace rule |RES-1|, and rules |R2<| and |R2≥| replace |RES-2|.

In a term $N \lhd T$, N can be seen as a subset of the free names of T. Whenever a new name is encountered, e.g. through $(\nu a)T$, a is recorded in N as long as N has less than k elements. Rule |DEF| states that a recursive call can only be unfolded if some of its parameters are in N. Note that, in rule |DEF|, we assume that the unfolding of a type is such that there is no clash with the names in N.

For $k \geq 0$, we write $N \lhd T \rightarrow_k^* \xrightarrow{\alpha}_k$ if there exist T' and T'' such that $N \lhd T \rightarrow_k^* N \lhd T' \xrightarrow{\alpha}_k N \lhd T''$.

We consider a fragment of the prime sieve example and show its behaviour according to the symbolic semantics, with $k = 1$. We have:

$$\begin{aligned}&\{a\} \lhd (\nu b)(\boldsymbol{g}\langle a \rangle \mid \boldsymbol{f}\langle a, b \rangle \mid \boldsymbol{r}\langle b \rangle)\\ &\quad \rightarrow_1^* \xrightarrow{[a]}_1 \{a\} \lhd (\nu b)(\boldsymbol{g}\langle a \rangle \mid \overline{b}; \boldsymbol{f}\langle a, b \rangle \mid \boldsymbol{r}\langle b \rangle)\end{aligned}$$

at this point the process is stuck. The sub-term $\overline{b}; \boldsymbol{f}\langle a, b \rangle$ awaits to synchronise on b, however the dual action on b is "hidden" in the unfolding of $\boldsymbol{r}\langle b \rangle$, which cannot be unfolded by rule |DEF| since $b \notin \{a\}$ and, since $k = 1$, b cannot be added to the set of names. If we set the bound to $k = 2$:

$$\begin{aligned}&\{a, b\} \lhd \boldsymbol{g}\langle a \rangle \mid \overline{b}; \boldsymbol{f}\langle a, b \rangle \mid \boldsymbol{r}\langle b \rangle\\ &\quad \rightarrow_2^* \xrightarrow{[b]}_2 \{a, b\} \lhd \boldsymbol{g}\langle a \rangle \mid \boldsymbol{f}\langle a, b \rangle \mid (\mathsf{new}\, c)(\boldsymbol{f}\langle b, c \rangle \mid \boldsymbol{r}\langle c \rangle)\end{aligned}$$

4.4 Liveness and Channel Safety for Types

Following § 2.3, we define liveness and channel safety properties for types. The definitions of liveness and channel safety rely on barbs. The predicate $T \downarrow_{\overline{a}}$ (resp. $T \downarrow_a$) denotes a type ready to send (resp. receive) over channel a. Barb $T \downarrow_{\mathsf{end}[a]}$ denotes a type ready to close channel a and barb $T \downarrow_{a^\star}$ denotes a closed channel. Barb $T \downarrow_{[a]}$ denotes a synchronisation over channel a. Barb $T \downarrow_{\tilde{o}}$ denotes a type that is waiting to synchronise over the actions in $\tilde{o}$.

Definition 4.3 (Type Barbs)**.** We define the predicates $\kappa \downarrow_o$, $T \downarrow_o$ and $T \downarrow_{\tilde{o}}$ with $o, o_i \in \{a, \overline{a}, [a], \mathsf{end}[a], a^\star\}$.

$$a \downarrow_a \qquad \overline{a} \downarrow_{\overline{a}} \qquad \frac{\kappa \downarrow_o}{\kappa; T \downarrow_o} \qquad \mathsf{end}[a]; T \downarrow_{\mathsf{end}[a]} \qquad a^\star \downarrow_{a^\star}$$

$$\frac{T \downarrow_o}{T \mid T' \downarrow_o} \qquad \frac{T \downarrow_o \quad a \notin \mathsf{fn}(o)}{(\nu a)T \downarrow_o} \qquad \frac{T \downarrow_o \quad T \equiv T'}{T \downarrow_o}$$

$$\frac{T\,\{\tilde{a}/\tilde{x}\} \downarrow_o \quad \boldsymbol{t}(\tilde{x}) = T}{\boldsymbol{t}\langle \tilde{a} \rangle \downarrow_o}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \kappa_i \downarrow_{o_i}}{\&\{\kappa_i; T\}_{i \in \{1,\ldots,n\}} \downarrow_{\{o_1 \ldots o_n\}}} \qquad \frac{T \downarrow_a \quad T' \downarrow_{\overline{a}} \text{ or } T' \downarrow_{a^\star}}{T \mid T' \downarrow_{[a]}}$$

$$\frac{T \downarrow_a \quad \kappa_i \downarrow_{\overline{a}}}{T \mid \&\{\kappa_i; S_i\}_{i \in I} \downarrow_{[a]}} \qquad \frac{T \downarrow_{\overline{a}} \text{ or } T \downarrow_{a^\star} \quad \kappa_i \downarrow_a}{T \mid \&\{\kappa_i; S_i\}_{i \in I} \downarrow_{[a]}}$$

Given $k \in \mathbb{N}$, we write $T \Downarrow_o^k$ if $N \lhd T \rightarrow_n^* N \lhd T'$ and $T' \downarrow_o$, with $N = \mathsf{fn}(T)$, $n = k + |N|$ and $o \in \{a, \overline{a}, [a], \mathsf{end}[a], a^\star\}$. Observe that the predicate $T \Downarrow_o^k$ is defined wrt. the symbolic semantics. We write $T \Downarrow_o$ if $T \Downarrow_o^\infty$.

Definition 4.4 (Liveness)**.** The system $\boldsymbol{T}$ satisfies *k-liveness* if for all T such that $\emptyset \lhd \boldsymbol{t}_0\langle\rangle \rightarrow_k^* \emptyset \lhd (\nu \tilde{a})T$,

(a) If $T \downarrow_a$ or $T \downarrow_{\overline{a}}$ then $T \Downarrow_{[a]}^k$.
(b) If $T \downarrow_{\tilde{a}}$ then $T \Downarrow_{[a_i]}^k$ for some $a_i \in \tilde{a}$.

If $\boldsymbol{T}$ is ∞-live, then we say that $\boldsymbol{T}$ is *live*.

Consider the type equations below, where the reduction of $\boldsymbol{t}_0\langle\rangle$ leads to terms that are *not* k-live, for any $k \geq 0$.

$$\boldsymbol{t_1}(x) \triangleq (\mathsf{new}\, b)(\boldsymbol{t_1}\langle b \rangle \mid b; \overline{x}) \quad \boldsymbol{t}_0() \triangleq (\mathsf{new}\, a)(\boldsymbol{t_1}\langle a \rangle \mid a)$$

Intuitively, the system is not live since it is not possible to find a synchronisation for, e.g. the receive action on a, within bounded unfolding.

The definition of channel safety follows the same structure as that of liveness.

Definition 4.5 (Channel Safety)**.** The system $\boldsymbol{T}$ satisfies *k-channel safety* if for all T such that $\emptyset \lhd \boldsymbol{t}_0\langle\rangle \rightarrow_k^* \emptyset \lhd (\nu \tilde{a})T$, if $T \downarrow_{a^\star}$ then $\neg(T \Downarrow_{\mathsf{end}[a]}^k)$ and $\neg(T \Downarrow_{\overline{a}}^k)$.

If $\boldsymbol{T}$ is ∞-safe, then we say that $\boldsymbol{T}$ is *channel safe*.

Example We illustrate the need of a sufficiently large bound to detect liveness errors with an example. Consider a variation of the Prime Sieve example with a non-recursive filter:

$$\boldsymbol{f}(x, y) \quad \triangleq \quad x; \overline{y}; x; \overline{y}; x; \overline{y}; x; \overline{y}; \mathbf{0}$$

with $\boldsymbol{g}(x)$, $\boldsymbol{r}(x)$, and $\boldsymbol{t}_0()$ as in § 1.1. The system above is 2-live, but not 3-live. A bound of 2 is too small to allow the symbolic semantics to explore all the states of $\boldsymbol{f}(x, y)$; while a bound of 3 enables spawning another filter process, hence to explore all the states of $\boldsymbol{f}(x, y)$.

4.5 Decidability of the Bounded Verification

We show that k-liveness and k-channel safety are decidable, noting that these are defined wrt the symbolic semantics (**Step 3**).

The *decidability* result mainly follows from fencing, which ensures finite control over the symbolic semantics. The key idea is to show that the set of non-equivalent terms reachable by the symbolic semantics is finite. This is formalised below.

Lemma 4.1 (Finite Control)**.** *If* $\mathsf{Fenced}(\boldsymbol{T})$*, then the set* $\{[T]_\equiv \mid \emptyset \lhd \boldsymbol{t}_0\langle\rangle \rightarrow_k^* \emptyset \lhd T\}$ *is finite, for any finite* k*.*

The crux of the proof is to show that the number of occurrences of a type variable $\boldsymbol{t}$ is bounded in the maximal unfolding of a term up-to a given set of names N. That is, an unfolding which unfolds a term $\boldsymbol{t}\langle \tilde{a} \rangle$ only if $\tilde{a} \cap N \neq \varnothing$.

Theorem 4.1 (Decidability)**.** *For all* $\boldsymbol{T}$ *s.t.* $\mathsf{Fenced}(\boldsymbol{T})$*, it is decidable whether or not* $\boldsymbol{T}$ *is* k*-live (resp. channel safe), for any* $k \geq 0$*.*

Theorem 4.1 follows from the fact that checking k-liveness (resp. channel safety) is decidable over any finite LTS (finiteness is guaranteed by Lemma 4.1).

Remark 4.1. It is not always possible to compute a finite k such that k-liveness (resp. k-channel safety) implies general liveness (resp. channel safety) of fenced types. However, if the types are *finite control* (i.e., parallel composition does not appear under recursion), then liveness and channel safety are indeed decidable, see e.g. [14].

5. Properties of MiGo

We now make precise the properties our behavioural type analysis ensures on MiGo programs. We show that if a program $\boldsymbol{P}$ is typed by a safe type, then $\boldsymbol{P}$ is safe according to Definition 2.3. For liveness, we identify the classes of programs for which liveness of types implies program liveness. We note that type liveness and safety refers to ∞-liveness and ∞-safety, respectively. Moreover, we recall that our bounded analysis on types implies its ∞-counterpart only in the finite control fragment.

5.1 Type and Channel Safety in MiGo

The typing system of § 3 ensures that channel payloads always have the expected type. This property is made precise by a standard subject reduction result, stating that the semantics of types simulates the semantics of processes.

Theorem 5.1 (Subject Reduction)**.** *Let $\Gamma \vdash_B P \blacktriangleright T$ and $P \to P'$. Then there exists T' such that $\Gamma \vdash_{B'} P' \blacktriangleright T'$ with $T \to T'$.*

We prove that type safety implies program safety, using a correspondence between barbs. Hereafter we write $P \Downarrow_o$ including the case $o = \tilde{a}$.

Lemma 5.1. *Suppose $\Gamma \vdash_B P \blacktriangleright T$. If $P \Downarrow_o$ then $T \Downarrow_o$.*

Theorem 5.2 (Process Channel Safety)**.** *Suppose $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ and $\boldsymbol{T}$ is safe. Then $\boldsymbol{P}$ is safe.*

5.2 Liveness of Limited Programs

The development in § 4 performs an analysis on our abstract representation of processes (i.e. the *types*), verifying liveness (Definition 4.4) for fenced types. Our goal is to ensure liveness of a general class of *typable programs*. We divide programs into three classes to discuss the issue of liveness.

The first class is a set of programs which have a path to terminate. In this class, a program that is typable with a live type can always satisfy liveness.

Definition 5.1 (May Converging Program)**.** Let $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$. We write $\boldsymbol{P} \in \mathsf{May}\Downarrow$ if for all $X_0\langle\rangle \to^* P$, $P \to^* \mathbf{0}$.

Proposition 5.1. *Assume $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ and $\boldsymbol{T}$ is live.* (1) *Suppose there exists P such that $X_0\langle\rangle \to^* P \not\to$. Then $P \equiv \mathbf{0}$; and* (2) *If $\boldsymbol{P} \in \mathsf{May}\Downarrow$, then $\boldsymbol{P}$ is live.*

We note that the above statement does not restrict the programs to be finite. A program with infinite reduction sequences can satisfy liveness by Proposition 5.1. For instance:

$$\{D_1, D_2\} \,\mathsf{in}\, \mathsf{newchan}(b); \mathsf{newchan}(c); (X_1\langle b, c\rangle \mid X_2\langle b, c\rangle)$$

$$\begin{array}{rrcl} \text{with} & D_1 & = & X_1(b,c) \triangleq \mathsf{select}\{c!\langle v\rangle; X_1\langle b,c\rangle\ ,\ b!\langle w\rangle; \mathbf{0}\} \\ & D_2 & = & X_2(b,c) \triangleq \mathsf{select}\{c?(x); X_2\langle b,c\rangle\ ,\ b?(y); \mathbf{0}\} \end{array}$$

is live, as is its corresponding type.

The second class is a set of programs which do not contain *infinitely occurring conditional branches* (we discuss at length the issues raised by the interplay of conditional branching and recursion in § 5.3). If such a process is assigned a live type, then it is itself live. For example, any program which does not include conditionals or one with conditionals containing only finite processes in both branches belong to this class. Consider a program obtained from the one above by replacing $\mathbf{0}$ in D_1 and D_2 by X_1 and X_2, respectively. Despite this program executing forever, both program and type liveness hold. This class of programs is made precise in Proposition 5.2, along with the issue of infinitely occurring conditionals, which are explained below.

5.3 Liveness of Infinitely Occurring Conditionals

As the third class, we consider infinitely running programs that contain recursive variables in conditional branches. The behaviours of conditionals in a program rely on data to decide which branch is taken. On the other hand, at the type level, this information is abstracted as an internal choice. This causes a mismatch between program and type behaviours.

Revisiting the prime sieve example of § 2.1.1, consider the definition of the filter process:

$$F(n,i,o) \quad \triangleq \quad i?(x); \mathsf{if}\ (x\%n \neq 0)\ \mathsf{then}\ o!\langle x\rangle; F\langle n,i,o\rangle \ \mathsf{else}\ F\langle n,i,o\rangle$$

whose type is given as: $\mathbf{t}_F(i,o) = i; \oplus\{\overline{o}; \mathbf{t}_F\langle i,o\rangle,\ \mathbf{t}_F\langle i,o\rangle\}$.

Our analysis on types does indeed determine the types of the prime sieve as live, even in the absence of terminating reduction sequences. In $\mathbf{t}_F(i,o)$, we have an internal choice between a branch that recurses back to $\mathbf{t}_F$ and another that outputs along o and recurses back to $\mathbf{t}_F$. Thus, if we compose a call $\mathbf{t}_F$ with a type that denotes an infinite sequence of inputs along o, we deem such a composition as live since all the inputs *can* eventually be synchronised with an output from $\mathbf{t}_F$, given that the semantics of internal choice state that we may indeed move to either branch.

However, the type $\boldsymbol{T}$ of the prime sieve program is an abstract approximation of the actual prime sieve implementation, where the test $x\%n \neq 0$ is not obviously guaranteed to ever succeed given that it depends on received data (which is sent by either the generator process G or a previous filter process). Thus, the interplay of conditional branching and infinite recursion may in general cause a disconnect between the semantics of the types and those of the concrete processes. For instance, if the test $x\%n \neq 0$ is replaced by false in the prime sieve example, its type is live while the program is not. In the remainder of this section, we make precise the conditions under which the semantics of infinite processes and types simulate one another, thus implying liveness (even in the presence of infinite branching).

To achieve our liveness results, we proceed as follows:

Step 1. Define the notion of *infinite conditional* (Inf), identifying a class of programs where conditional branches are executed infinitely often.

Step 2. Fill the gap between internal choices of types and conditionals by defining a $*$-conditional ($\mathsf{if}\ *\ \mathsf{then}\ P\,\mathsf{else}\,Q$) which non-deterministically reduces to either P or Q (as the internal choice $\oplus\{T,S\}$), allowing us to identify the subclass of Inf, dubbed alternating conditionals (AC), where programs simulate their non-deterministic conditional counterparts.

Step 3. Prove that liveness of types implies liveness of programs in AC.

Alternating and Non-deterministic Conditionals For **Step 1**, we begin by defining a notion of *infinite conditional* (Definition 5.5) in programs. Intuitively, we identify programs that reduce forever and where conditional branches appearing under recursion have their branches taken infinitely often.

Definition 5.2 (Marked Programs)**.** Given a program $\boldsymbol{P}$ we define its *marking*, written $\mathsf{mark}(\boldsymbol{P})$, as the program obtained by deterministically labelling every occurrence of a conditional of the form $\mathsf{if}\ e\ \mathsf{then}\ Q\ \mathsf{else}\ R$ in $\boldsymbol{P}$, as $\mathsf{if}^n\ e\ \mathsf{then}\ Q\ \mathsf{else}\ R$, such that n is distinct natural number for all conditionals in $\boldsymbol{P}$.

Definition 5.3 (Marked Reduction Semantics)**.** We define a marked reduction semantics, written $P \xrightarrow{l} Q$, stating that program P reduces to Q in a single step, performing action l. The grammar of action labels is defined as:

$$l \coloneqq \epsilon \mid n{\cdot}\mathsf{L} \mid n{\cdot}\mathsf{R}$$

where ϵ denotes an unmarked action, $n{\cdot}\mathsf{L}$ denotes a conditional branch marked with the natural number n in which the then branch is chosen, and $n{\cdot}\mathsf{R}$ denotes a conditional branch in which the else branch is chosen. We write $P \to Q$ for $P \xrightarrow{\epsilon} Q$. The marked reduction semantics replace rules [IFT] and [IFF] with:

$$[\text{IFTM}]\ \frac{e \downarrow \mathsf{true}}{\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{n\cdot\mathsf{L}} P} \qquad [\text{IFFM}]\ \frac{e \downarrow \mathsf{false}}{\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{n\cdot\mathsf{R}} Q}$$

Definition 5.4 (Trace)**.** We define an execution trace $\mathcal{T}$ of a process P as the potentially infinite sequence of action labels $\vec{l}$ such that $P \xrightarrow{l_1} P_1 \xrightarrow{l_2} \ldots$, with $\vec{l} = \{l_1\, l_2 \ldots\}$. We write $\mathbb{T}_P$ for the set of all possible traces of a process P.

A trace of the marked reduction semantics identifies exactly which branches were selected during the potentially infinite execution of a program.

We now define infinitely recurring conditionals. We use a reduction context $\mathbb{C}_r$ given by:

$$\mathbb{C}_r \quad \coloneqq \quad [\,] \mid (P \mid \mathbb{C}_r) \mid (\mathbb{C}_r \mid P) \mid (\nu a)\mathbb{C}_r$$

We write $\mathbb{C}_r[P]$ for the process obtained by replacing P for the hole $[\,]$ in $\mathbb{C}_r$.

Definition 5.5 (Infinite Conditional)**.** We say that P has infinite conditional branches, written $P \in \mathsf{Inf}$, iff $\mathsf{mark}(P) \to^* \mathbb{C}_r[\mathsf{if}^n\ e\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2] = R$, for some n, and R has an infinite trace where $n{\cdot}\mathsf{L}$ or $n{\cdot}\mathsf{R}$ appears infinitely often. We say that such an n is an *infinite conditional mark* and write $\mathsf{InfCond}(P)$ for the set of all such marks.

The following statement implies that even programs which contain only infinite executions can be live if none of its conditionals appear in traces infinitely often (i.e. our second class of programs).

Proposition 5.2 (Liveness for Finite Branching)**.** *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is live and* $\boldsymbol{P} \notin \mathsf{Inf}$. *Then* $\boldsymbol{P}$ *is live.*

The main purpose of Definition 5.7 is to identify infinitely running processes where the behaviour of conditional branching approximates that of non-deterministic internal choice (i.e. the type-level semantics of internal choice). To make this relationship precise, we define a mapping from MiGo programs to programs where conditional branching is replaced by a form of non-deterministic branching. This step corresponds to **Step 2**.

Definition 5.6. The mapping $(P)^*$ replaces all occurrences of $\mathsf{if}^n\ e\ \mathsf{then}\ Q\ \mathsf{else}\ R$, such that $n \in \mathsf{InfCond}(P)$, with $\mathsf{if} * \mathsf{then}\ Q\ \mathsf{else}\ R$. The reduction semantics of $\mathsf{if} * \mathsf{then}\ Q\ \mathsf{else}\ R$ is defined as follows:

$$[\text{IFT}*]\ \mathsf{if} * \mathsf{then}\ P\ \mathsf{else}\ Q \to P \quad [\text{IFF}*]\ \mathsf{if} * \mathsf{then}\ P\ \mathsf{else}\ Q \to Q$$

Definition 5.7 (Alternating Conditionals)**.** We say that P has *alternating conditional branches*, written $P \in \mathsf{AC}$, iff $P \in \mathsf{Inf}$ and if $P \to^* (\nu \tilde{c})Q$ then $Q^* \Downarrow_o$ implies $Q \Downarrow_o$.

Recall that o ranges over any barbs, including $\tilde{a}$. Moreover, observe that the mapping P^* only affects conditionals that are executed infinitely often (i.e. those whose behaviour may fail to be captured by the type-level analysis). We do not require conditionals that are not in $\mathsf{InfCond}(P)$ to necessarily match the barbs of their non-deterministic counterpart, since their behaviour is already over-approximated by the corresponding types.

Proposition 5.3 ($*$-properties)**.** *Suppose* $\Gamma \vdash_B P \blacktriangleright T$. *Then* (1) *if* $P^* \in \mathsf{Inf}$ *then* $P^* \in \mathsf{AC}$*;* (2) *If* $P \Downarrow_o$*, then* $P^* \Downarrow_o$*;* (3) *if* $P^* \Downarrow_o$ *then* $T \Downarrow_o$.

Liveness for Infinite Conditionals We now have defined the conditions under which programs simulate the behaviour of their types. More precisely, when a program $\boldsymbol{P}$ is well-typed with some live type T and $\boldsymbol{P} \in \mathsf{AC}$ holds, then $\boldsymbol{P}$ must itself be live.

Theorem 5.3 (Liveness)**.** *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is live and* $\boldsymbol{P} \in \mathsf{AC}$. *Then* $\boldsymbol{P}$ *is live.*

To summarise, we identified three significant classes of programs for which type liveness implies liveness: those with at least one terminating path (Definition 5.1 and Proposition 5.1) such as Fibonacci, cf. § 2.1.2; those for which their infinite traces do not contain infinite occurrences of a given conditional (Proposition 5.2) such as Dining Philosophers, cf. § 7.1; and, those with infinite traces containing infinite occurrences of conditional branches (Definition 5.7 and Theorem 5.3) such as Prime Sieve, cf. § 2.1.1.

While a reasonable percentage of real-world programs are in the first two classes, our empirical observations show that a substantial amount of infinitely running programs (with infinitely occurring conditionals) that are not in AC have redundant or erroneous conditionals.

6. Bounded Asynchrony in MiGo

Our framework extends with relative ease to the asynchronous communication variant of the Go language. As mentioned in § 2, communication channels in Go are implemented as *bounded* FIFO queues, where by default the buffer bound is 0 – synchronous communication. For bounds greater than 0, communication is then potentially asynchronous – sends do not block if the buffer is not full and inputs do not block if the buffer is not empty.

Asynchrony significantly affects a program's liveness. Consider the following example:

$$P(x, y) \quad \triangleq \quad x!\langle 1\rangle; y?(z) \mid y!\langle 2\rangle; x?(z)$$

A program that instantiates $P(x, y)$ with synchronous communication channels will necessarily not be live since the output and input actions in P are mismatched. However, with asynchronous channels, the output actions become non-blocking and the program is indeed *live* – the output on x on the left-hand side can fire asynchronously, exposing the input on y which may then fire. Similarly for the output on y and input on x on the right-hand side.

Processes and Typing To account for the buffer bounds in the syntax of MiGo we add a bound n to channel creation, $\mathsf{newchan}(y{:}\sigma, n); P$. This number must be equal or greater to zero and must be a literal. We also carry this information in runtime buffers: $c\langle\sigma, n\rangle{::}\tilde{v}$ and $c^\star\langle\sigma, n\rangle{::}\tilde{v}$ (also replacing $c\langle\sigma\rangle{::}\emptyset$ and $c^\star\langle\sigma\rangle{::}\tilde{v}$ by $c\langle\sigma, 0\rangle{::}\emptyset$ and $c^\star\langle\sigma, n\rangle{::}\tilde{v}$ for synchronous channels). We add the reduction rules for asynchronous communication:

$$[\text{OUT}]\ \frac{|\tilde{v}| < n \quad e \downarrow v}{c!\langle e\rangle; P \mid c\langle\sigma, n\rangle{::}\tilde{v} \to P \mid c\langle\sigma, n\rangle{::}v \cdot \tilde{v}}$$

$$[\text{INA}]\quad c?(y); P \mid c\langle\sigma, n\rangle{::}\tilde{v} \cdot v \to P\,\{{}^{v}/_{y}\} \mid c\langle\sigma, n\rangle{::}\tilde{v}$$

In all other rules that use buffers we add the buffer bound straightforwardly. The type system is fundamentally unchanged, now ac-

counting for buffer bounds:

$$\langle \text{NEW} \rangle \frac{\Gamma, y{:}\mathsf{ch}(\sigma, n) \vdash_s P \blacktriangleright T \quad c \notin \mathsf{dom}(\Gamma) \cup s \cup \mathsf{fn}(T)}{\Gamma \vdash_s \mathsf{newchan}(y{:}\sigma, n); P \blacktriangleright (\mathsf{new}^n\, c)T\,\{^c/_y\}}$$

$$\langle \text{BUFF} \rangle \frac{|\tilde{v}| = k}{\Gamma, a{:}\mathsf{ch}(\sigma, n) \vdash_{\{a\}} a\langle \sigma, n\rangle{::}\tilde{v} \blacktriangleright \lfloor a \rfloor_k^n}$$

In contrast with the types and rules in Figure 4, $(\mathsf{new}\, a)T$ and $\lfloor a \rfloor$ are replaced by $(\mathsf{new}^n\, a)T$ and $\lfloor a \rfloor_k^n$, respectively, where k stands for the number of elements in the buffer.

Liveness and safety of types are defined as in § 4.4, with two extra rules for the definition of type barbs, pertaining to buffers. In particular we need barbs for writing to a non-full buffer ($P \downarrow_{\bullet a}$) and reading from a non-empty buffer ($P \downarrow_{a\bullet}$), combined with the following additional rules:

$$\frac{|\tilde{v}| < n}{a\langle \sigma, n\rangle{::}\tilde{v} \downarrow_{\bullet a}} \qquad \frac{|\tilde{v}| \geq 1}{a\langle \sigma, n\rangle{::}\tilde{v} \downarrow_{a\bullet}}$$

$$\frac{P \downarrow_{\overline{a}} \quad Q \downarrow_{\bullet a}}{P \mid Q \downarrow_{[a]}} \qquad \frac{P \downarrow_{a\bullet} \quad \pi_i \downarrow_a}{P \mid \mathsf{select}\{\pi_i; Q_i\}_{i \in I} \downarrow_{[a]}}$$

The barbs for asynchronous types, $T \downarrow_o$, are given below:

$$\frac{k < n}{\lfloor a \rfloor_k^n \downarrow_{\bullet a}} \qquad \frac{k \leq 1}{\lfloor a \rfloor_k^n \downarrow_{a\bullet}} \qquad \frac{T \downarrow_{\overline{a}} \quad T' \downarrow_{\bullet a}}{T \mid T' \downarrow_{[a]}} \qquad \frac{T \downarrow_{a\bullet} \quad \kappa_i \downarrow_a}{T \mid \&\{\kappa_i; S_i\}_{i \in I} \downarrow_{[a]}}$$

Verification of Types The changes to the semantics of types are straightforward. It is based on the LTS of § 4.1, where rule |NEW| and |BUF| are replaced by their counterparts below, and four additional rules |IN-B|, |OUT-B|, |PUSH| and |POP|.

$$|\text{NEW}| \quad (\mathsf{new}^n\, a)T \xrightarrow{\tau} (\nu a)(T \mid \lfloor a \rfloor_0^n) \qquad |\text{BUF}| \quad \lfloor a \rfloor_k^n \xrightarrow{\overline{\mathsf{end}}[a]} a^\star$$

$$|\text{IN-B}| \frac{k < n}{\lfloor a \rfloor_k^n \xrightarrow{\bullet_a} \lfloor a \rfloor_{k+1}^n} \qquad |\text{OUT-B}| \frac{k \geq 1}{\lfloor a \rfloor_k^n \xrightarrow{a^\bullet} \lfloor a \rfloor_{k-1}^n}$$

$$|\text{PUSH}| \frac{T \xrightarrow{\overline{a}} T' \quad S \xrightarrow{\bullet_a} S'}{T \mid S \xrightarrow{[a]} T' \mid S'} \qquad |\text{POP}| \frac{T \xrightarrow{a^\bullet} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{[a]} T' \mid S'}$$

Observe that since types abstract away from values and channels are attributed a unique payload type, the semantics does not model message ordering.

With all the technical machinery in place for the *bounded* asynchronous setting, we replicate our main results. The proofs are essentially identical to those in the synchronous setting. Indeed, asynchrony affects our analysis only in the size of the models to be checked (larger buffer sizes give larger LTSs). The symbolic semantics executes the types up-to a limited number of channels, which is orthogonal to the number of message a buffer can store, cf. Figure 7.

Theorem 6.1 (Decidability – Asynchrony)**.** *For all* $\boldsymbol{T}$ *s.t.* $\mathsf{Fenced}(\boldsymbol{T})$*, it is decidable whether or not* $\boldsymbol{T}$ *is* k*-live (resp. channel safe), for any* $k \geq 0$*.*

Theorem 6.2 (Process Channel Safety and Liveness – Asynchrony)**.** *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$*.*

1. *If* $\boldsymbol{T}$ *is channel safe, then* $\boldsymbol{P}$ *is channel safe.*
2. *If* $\boldsymbol{T}$ *is live and either* $\boldsymbol{P} \in \mathsf{May}\Downarrow$*,* $\boldsymbol{P} \notin \mathsf{Inf}$ *or* $\boldsymbol{P} \in \mathsf{AC}$*, then* $\boldsymbol{P}$ *is live.*

With the revised semantics, the program

$$\{P(x, y)\}\ \mathsf{in}\ \mathsf{newchan}(x{:}\mathsf{int}, 1); \mathsf{newchan}(y{:}\mathsf{int}, 1); P\langle x, y\rangle$$

is correctly deemed as live, with the typing given by:

$$\{\boldsymbol{t}_P(x, y) = (\overline{x}; y \mid \overline{y}; x)\}\ \mathsf{in}\ (\mathsf{new}^1 x)(\mathsf{new}^1 y)\boldsymbol{t}_P\langle x, y\rangle$$

7. Implementation

We have implemented our static analysis as a verification tool-chain consisting of two parts: First, we analyse Go source code and infer behavioural types (§ 3) based on a program's usage of concurrency primitives. The types are passed to a tool that implements the verification outlined in § 4, checking bounded liveness and channel safety of the types. An outline of our verification tool chain is shown in Figure 8.

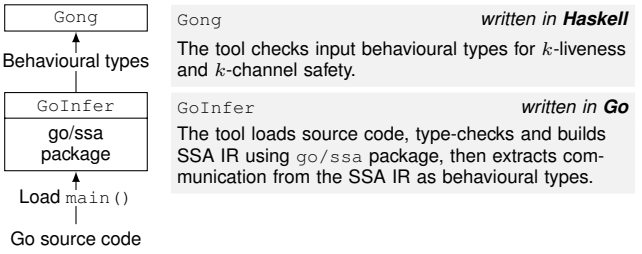


Figure 8. Workflow of our verification tool chain.

Type Inference Our type inference tool `GoInfer` is written in Go, using the `go/ssa`[1] package from Go project's extra tools. The package builds Go source code in Static Single Assignment (SSA) representation, and provides an API to access the resulting SSA IR programmatically. Starting from the program entry point, i.e. the `main()` function in the `main` package, we transform the SSA IR into a system of type equations $\boldsymbol{T}$ by converting each SSA block into an individual type equation. The analysis and conversions are context-sensitive, for example, channels created in different instances of a function are different, and loops are unrolled if it is possible to determine the bounds statically. We note that our analysis is agnostic wrt. aliasing since we do not rely on linearity of channels. In addition to inference, our tool can also check for trivial conditionals that do not belong to any of the three classes of programs defined in § 5.

Verification Our proof-of-concept verification tool `Gong`, written in Haskell, inputs a system of type equations $\boldsymbol{T}$ representing a Go program's concurrent behaviour and performs liveness and channel safety checks on the bounded symbolic semantics. Our representation of $\boldsymbol{T}$ makes use of the `unbound` package [51] to deal with the binding structure of types. First, it checks if $\boldsymbol{T}$ is fenced. If so, we generate all $\rightarrow_k$-reachable terms, where k is heuristically-computed. Finally, each of these terms are checked for k-liveness and k-channel safety by identifying their barbs and successors.

7.1 Evaluation

We tested our tool-chain on the examples from the paper, from works on static deadlock detection in Go [38], on open-source Go programs from developer guides [41, 43] and GitHub [1], on classic concurrency examples [17, 33], and on concurrent programs translated to idiomatic Go from from [32].

Table 1 summarises the experimental results. The column "Go programs" shows the names of the programs. In the columns "Number of channels", the number of channels given for programs with bounded loops is precise since bounded loops are unrolled and we can statically count the number of channels; for programs with recursion, we count the channels that appear in the source code. The columns "Time" show the inference and verification times in seconds. We include a comparison with the tool from [38] to demonstrate the extra expressiveness of our approach. If a program

[1] `http://golang.org/x/tools/go/ssa`

Table 1. Go programs verified by our tool chain.

Examples	# chans LoC	unbuf.	buf.	Live	Safe	Analysis Time (ms)	[38] Static	Safe
`sieve` †	19	2	0	✓	✓	209.55	×	
`fib` †	23	2	0	✓	✓	14638.4	×	
`fib-async` †	23	1	1	✓	✓	32173.8	×	
`fact` †	19	2	0	✓	✓	206.63	×	
`dinephil` [17, 33]	56	3	0	✓	✓	646921.76	×	
`jobsched`	41	0	1	✓	✓	48.12	×	
`concsys` [1]	112	2	0	×	✓	323.75	×	
`fanin` [38, 43]	36	3	0	✓	✓	89.14	✓	✓
`fanin-alt` [38]	37	3	0	×[1]	✓	209.02	✓	✓
`mismatch` [38]	26	2	0	×	✓	26.59	✓	×
`fixed` [38]	25	2	0	✓	✓	24.58	✓	✓
`alt-bit` [35]	74	0	2	✓	✓	405.78	✓	✓
`forselect`	40	3	0	✓	✓	31.01	✓	✓
`cond-recur`	32	2	0	✓	✓	34.08	✓	✓

[1]: testing for channel close state is not supported in this version
†: examples that are *not* finite control

The benchmarks were compiled with ghc 7.10.3 and go1.6.2 executed on Intel Core i5 @ 3.20GHz with 8GB RAM.

is "Static", it has no dynamic spawning of goroutines (a requirement for the usage of the tool of [38]).

`forselect` is a pattern described in [42] where an infinite `for` loop and a `select` statement with two cases are combined to repeatedly receive (or send). In our example we spawn two goroutines, where each goroutine has a for-select loop with compatible channel communication. In the for-select loop, one of the select cases receives (or sends) a message then continues to the next iteration of the infinite loop; the other case breaks out of the loop upon sending (or receiving) a message from the other goroutine so that both goroutines exit the loop together. The exit condition is non-deterministic (because of select), but the program is both live and safe. `cond-recur` is similar to `forselect`, where one of the two goroutines contains a for-select loop, but the other has an ordinary for-loop so that the exit condition of the for-loop is deterministic.

8. Related Work and Conclusion

Static Deadlock Detection in Synchronous Go There are two recent works on static deadlock detection for synchronous Go [38, 45]. The work [38] extracts *Communicating Finite State Machines* [6] whose representation corresponds to *session types* [21, 46] from Go source code, and synthesises from them a global choreography using the tool developed in [32]. If the choreography is well-formed, a program does not have a (partial) deadlock. This approach is seriously limited due to the lack of expressiveness of (multiparty) session types [23] and its synthesis theory. The approach expects all goroutines to be spawned before any communication happens at runtime. This is due to the fact that the synthesis technique requires all session participants to be present from the start of the global interaction, meaning that their work cannot handle most programs with dynamic patterns, such as spawning new threads after communication started. The analysis is also limited to unbuffered channels and does not support asynchrony. For instance, our prime sieve example cannot be verified by their tool, and is in fact used to clarify the limitations of their approach. Moreover, the work is limited to the tool implementation, no theoretical property nor formalisation is studied in [38].

The work of [45] uses the notion of *forkable behaviours* (i.e. a regular expression extended with a fork construct) to capture spawning behaviours of synchronous Go programs, developing a tool based on this approach to directly analyse Go programs. Their technique is sound but has some significant theoretical and practical limitations: (1) their analysis does not support asynchrony (buffered channels), closing of channels or usages of the select construct with non-trivial case bodies; (2) while their liveness analysis (when restricted to synchrony) targets the *sound* fragment of our analysis, they are more conservative in their approach. For instance, the following program which is verified as live in our approach (this program belongs to $\mathsf{May}\Downarrow$ in our theory) is judged as a deadlock in their approach (implemented as `cond-recur` in Table 1):

$$\begin{array}{rl} \{X(a,b) = & \mathsf{if}\ e_1\ \mathsf{then}\ a!\langle e_2\rangle.X\langle a,b\rangle\ \mathsf{else}\ b?(z);\mathbf{0}, \\ Y(a,b) = & \mathsf{select}\{a?(z);Y\langle a,b\rangle,\ b!\langle\rangle;\mathbf{0}\}\}\ \mathsf{in} \\ & \mathsf{newchan}(a);\mathsf{newchan}(b);(X\langle a,b\rangle \mid Y\langle a,b\rangle) \end{array}$$

Finally, (3) it is unclear how their tool can deal with the ambiguity of context sensitive inter-procedural analysis given their use of the `oracle` tool and the syntactic approach taken in the implementation.

Behavioural Types Behavioural type-based techniques (see [24] for a broad survey) have been developed for general concurrent program analyses [25], such as deadlock-freedom [19, 27], lock-freedom [29, 40], resource usages [30] and information flow analysis [26].

All of the type-based techniques above differ from ours in that we perform an analysis on types akin to *bounded* model-checking, whereas their works take a type-checking based approach to deadlock (or lock) freedom. Their techniques are sound against all possible inputs of processes, but often too conservative. Our approach is sound only for some subsets of possible inputs, but less conservative. A potential limitation of their techniques is that subtle changes in channel usage (that may not have a significant effect on a program's outcome) can produce significantly different analysis outcomes (see discussion in [19] and [40]). Moreover, the dependency tracking can be quite intricate and hard to implement in a real language setting. Our fencing-based approach is more easily implemented as a *post-hoc* analysis, covering a wide range of Go programs, since it only limits names in recursive call sites and does not explicitly depend on the ordering of communications or on computing circularities of channels (provided the programs are in one of the classes of § 5).

The work [19, 28] develops a deadlock detection analysis of asynchronous CCS processes with recursion and new name creation. The analysis is able to reason about infinite-state systems that create networks with an arbitrary number of processes, going beyond those of [27] and [40]. Their approach uses an extension of the typing system of [27] as a way to extract a so-called lam term from a (typed) process. Lam terms track dependencies between channel usages as pairs of level names. Given a lam term, the authors develop a sound and complete decision procedure for circularities in dependencies. By separating this decision procedure from the type system, their system is able to accurately analyse deadlock-free processes that are not possible in [27] and [40].

We first point out that the deadlock-freedom property of [19, 28] does not match with our notion of liveness (which is closer to lock freedom in [29, 40]). For instance, their analysis accepts program Fib_{bad} in § 2.1.2 as a deadlock-free process since a program that loops non-productively is deadlock-free (but not lock-free).

While their analysis can soundly verify unfenced types, which is by construction outside the scope of our work, we note that the reduction of deadlock-freedom to circularity of lams in their work excludes some natural communication patterns that are finite-control, which can be soundly checked by our type-level analysis. Consider the following finite control program (described in [28]), which can be directly interpreted as a MiGo type:

$$\begin{array}{rl} \{A(x,y) = & x?();y!\langle\rangle;A\langle x,y\rangle, \\ B(x,y) = & x!\langle\rangle;y?();B\langle x,y\rangle\}\ \mathsf{in} \\ & \mathsf{newchan}(a);\mathsf{newchan}(b);(A\langle a,b\rangle \mid B\langle a,b\rangle) \end{array}$$

The program above consists of two threads that continuously send and receive along the two channels a and b. This program, despite being finite control (and deadlock-free) is excluded by their analysis. As described in [28], this happens due to their current formulation of the type system assigning a finite number of levels in recursive channel usages, which entails that finite-control systems that use channels infinitely often (such as the one above) can be assigned circular lams, despite being deadlock-free. Our approach, by not relying on such notions of circularity can tackle these finite-control cases in a sound manner.

The work of [40] studies a variation of [27, 29] that ensures deadlock freedom and lock freedom of the linear π-calculus, with a form of channel polymorphism. By relying on linearity, the system in [40] rules out many examples that are captured by our work (although it can in principle analyse unfenced types). The `fib` and `diningphilosopher` examples denote patterns that are untypable in [40], but can be verified in our tool. Our tool can also verify programs morally equivalent to most examples discussed in [40], see Table 1 in § 7.1.

Session Types The work on session types is another class of behavioural typing systems that rely crucially on linearity in channel types to ensure certain compatibility properties of structured communication between two (binary [21]) or more (multiparty [23]) participants. Progress (deadlock-freedom on linear channels) is guaranteed within a *single* session, but not for multiple interleaved sessions. Several extensions to ensure progress on multiple sessions have been proposed, e.g. [11–13]. Our main examples are not typable in these systems for the same reasons described in the above paragraph. Their systems do not ensure progress of shared names, which are key in our examples.

A different notion of liveness called request-response is proposed in [15] based on binary session types. Their liveness means that when a particular label of a branching type (a request) is selected, a set of other labels (its responses) is eventually selected. The system requires *a priori* assumptions that a process must satisfy lock-freedom and annotations of response labels in types.

The works of [9, 10, 50] based on linear logic ensure progress in the presence of multiple session channels, but the typing discipline disallows modelling of process networks with cyclic patterns (such as prime sieve). In these works, progress denotes both deadlock and lock-freedom in the sense of [40]. However, to ensure logical consistency general recursion is disallowed. In the presence of general recursion [48], progress is weakened; ensuring all typable processes are deadlock-free but not necessarily lock-free. The work of [49] studies a restricted form of corecursion that ensures both deadlock- and lock-freedom in the context of logic-based processes. However, since the typing discipline ensures termination of all computations it is too restrictive for a more practical setting such as ours.

Effect Systems The work [39] introduces a type and effect system for a fragment of concurrent ML (including dynamic channel creations and process spawning) with a predicate on types which guarantees that typed programs are limited to a finite communication topology. Their types are only used to check whether a program has a *finite* communication topology, that is, whether a program uses a bounded number of channels and a bounded number of processes. No analysis wrt. safe or live communication is given, which is the ultimate goal of our work.

Conclusion and Future Work Since the early 1990s, behavioural type theories which formalise "types as concurrent processes" [25] have been studied actively in models for concurrency [24]. Up to this point, there have been few opportunities to apply these techniques directly to a real production-level language. The Go language opens up such a possibility. This work proposes a static verification framework for liveness and safety in Go programs based on a bounded execution of *fenced* behavioural types. We develop a tool that analyses Go code by directly inferring behavioural types with no need for additional annotations.

In future work we plan to extend our approach to account for channel passing, and also lock-based concurrency control, enabling us to verify *all* forms of concurrency present in Go. The results of § 4 suggest that it should be possible to encode our analysis as a model checking problem, allowing us to: (1) exploit the performance enhancements of state of the art model checking techniques; (2) study more fine-grained variants of liveness; (3) integrate model checking into the analysis of conditionals to, in some scenarios, decide the program class (viz. § 5.3). Another interesting avenue of future work is to explore the integration of type-checking based approaches into our framework, including those aimed at liveness and termination-checking (such as [16, 22, 26, 29]). These techniques eliminate false positives arising due to issues on divergence of processes, which are related to our classification of § 5, hence would be useful to identify a set of processes which conform, e.g. Proposition 5.1. This would enable a more fine-grained analysis, taking advantage of the strong soundness properties of this line of work. Moreover, the latter mentioned works could be applied in order to soundly approximate the program classification studied in § 5.3.

Acknowledgments

We gratefuly acknowledge Naoki Kobayashi for finding flaws in Sections 4 and 5 in earlier versions of this work; as well as for detailed comments on Sections 7 and 8. The present version aims at correcting these errors. In particular, we have revised the definition of liveness and safety for types (Definitions 4.4 and 4.5) and removed theorems related to soundness of our analysis of general liveness and safety for types. We have revised several remarks in the comparison between our work and [19, 26, 27, 29] (§ 7 and 8).

We also would like to thank Elena Giachino, Raymond Hu and Luca Padovani for fruitful discussions on this work, as well as the anonymous referees for their comments. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1 and EP/N027833/1; and by EU FP7 612985 (UPSCALE) and COST Action IC1405 (RC).

References

[1] Collection of Golang concurrency patterns. `https://github.com/stillwater-sc/concurrency`.

[2] Tool chain. `http://mrg.doc.ic.ac.uk/tools/gong`.

[3] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *NSDI 2015*, 2015.

[4] D. G. Anderson. Experience with ePaxos: Systems Research using Go. 2013. `https://da-data.blogspot.co.uk/2013/10/experience-with-epaxos-systems-research.html`.

[5] Andrew Gerrand. Share Memory By Communicating. `https://blog.golang.org/share-memory-by-communicating`.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, April 1983.

[7] N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP'03*, pages 133–144, 2003.

[8] N. Busi, M. Gabbrielli, and G. Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *ICALP'04*, pages 307–319, 2004.

[9] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[10] L. Caires, F. Pfenning, and B. Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[11] M. Carbone, O. Dardha, and F. Montesi. Progress as compositional lock-freedom. In *COORDINATION*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

[12] M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.

[13] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 26(2):238–302, 2016.

[14] M. Dam. Model checking mobile processes. *Inf. Comput.*, 129(1):35–51, Aug. 1996.

[15] S. Debois, T. T. Hildebrandt, T. Slaats, and N. Yoshida. Type-checking liveness for collaborative processes with bounded and unbounded recursion. *Logical Methods in Computer Science*, 12(1), 2016.

[16] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in impure concurrent languages. In *CONCUR'10*, volume 6269 of *LNCS*, pages 328–342. Springer, 2010.

[17] E. W. Dijkstra. Cooperating sequential process. *Programming Languages*, pages 43–112, 1965.

[18] B. Fitzpatrick. go 1.5.1 linux/amd64 deadlock detection failed, 2015. `https://github.com/golang/go/issues/12734#issuecomment-142859447`.

[19] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *CONCUR*, volume 8704 of *LNCS*, pages 63–77. Springer, 2014.

[20] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[21] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[22] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007.

[23] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008. A full version in *JACM*: 63(1-9):1–67, 2016.

[24] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, Apr. 2016.

[25] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[26] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.

[27] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.

[28] N. Kobayashi and C. Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.

[29] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *TOPLAS*, 32(5):16:1–16:49, May 2008.

[30] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the p-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[31] J. Lange, N. Ng, B. Toninho, and N. Yoshida. Fencing off go: Liveness and safety for channel-based programming (extended version), 2016. Available at `https://arxiv.org/abs/1610.08843`.

[32] J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In S. K. Rajamani and D. Walker, editors, *POPL'15*, pages 221–232. ACM Press, 2015.

[33] J. Magee and J. Kramer. *Concurrency: State Models &Amp; Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[34] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, 1980.

[35] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[36] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *ICALP*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.

[37] I. Moraru, D. G. Andersen, and M. Kaminsky. There is More Consensus in Egalitarian Parliaments. In *SOSP'13*, pages 358–372, New York, NY, USA, 2013. ACM.

[38] N. Ng and N. Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CC 2016*, pages 174–184. ACM, 2016.

[39] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL '94*, pages 84–97. ACM, 1994.

[40] L. Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In T. A. Henzinger and D. Miller, editors, *CSL-LICS'14*, pages 72:1–72:10. ACM Press, 2014.

[41] Rob Pike. Go Concurrency Patterns, 2012. `https://talks.golang.org/2012/concurrency.slide`.

[42] Sameer Ajamni. Advanced Go Concurrency Patterns, 2013. `https://talk.golang.org/2013/advconc.slide`.

[43] Sameer Ajmani. Go Concurrency Patterns: Pipelines and cancellation, 2014. `https://blog.golang.org/pipelines`.

[44] D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[45] K. Stadmüller, M. Sulzmann, and P. Thiemann. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*, volume 10017 of *LNCS*, 2016.

[46] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[47] The Go Authors. Effective Go. `https://golang.org/doc/effective_go.html`.

[48] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP'13*, pages 350–369, 2013.

[49] B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC'14*, pages 159–175, 2014.

[50] P. Wadler. Proposition as Sessions. In *ICFP'12*, pages 273–286, 2012.

[51] S. Weirich and B. Yorgey. Unbound library. `https://hackage.haskell.org/package/unbound`.

[52] M. Welsh. Rewriting a large production system in Go. 2013. `http://matt-welsh.blogspot.co.uk/2013/08/rewriting-large-production-system-in-go.html`.