

# 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 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, unbounded thread creation and 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 a decision procedure for liveness and safety in types which in turn statically ensures liveness and safety in Go programs. We have implemented a type inference and decision procedures 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, 42]*

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) [18]. 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, 47], instead of “messy chains of dozens of asynchronous callbacks spread over tens of source files” [4, 32].

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 [16, 33], nor can it detect deadlocks involving only a strict subset of a program’s goroutines (partial deadlocks).

**Statically Ensuring Liveness and Safety** 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 [41] and [22], 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 liveness and detects communication errors even in the presence of highly dynamic and unconstrained communication topologies, containing infinitely many recursive processes and channels, which can often be out of the scope of existing behavioural type-based analyses [12, 17, 24, 35] (see § 8 for related works).

#### 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). 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

```

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         chl := make(chan int)
16         go Filter(ch, chl, prime) // Chain filter.
17         ch = chl
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 `chl` 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  $\mathbf{t}_0$  given by:

$$\begin{aligned}
\mathbf{g}(x) &\triangleq \bar{x}; \mathbf{g}\langle x \rangle \\
\mathbf{f}(x, y) &\triangleq x; (\bar{y}; \mathbf{f}\langle x, y \rangle \oplus \mathbf{f}\langle x, y \rangle) \\
\mathbf{r}(x) &\triangleq x; (\text{new } b)(\mathbf{f}\langle x, b \rangle \mid \mathbf{r}\langle b \rangle) \\
\mathbf{t}_0() &\triangleq (\text{new } a)(\mathbf{g}\langle a \rangle \mid \mathbf{r}\langle a \rangle)
\end{aligned}$$

The type is described as a system of mutually recursive equations, where the distinguished name  $\mathbf{t}_0$  types the program entry point. This language is equivalent to a subset of CCS [29] with recursion, parallel and name creation, for which deciding liveness or safety is in general *undecidable* [7, 8]. The type  $\mathbf{t}_0$  specifies that `main` consists of the creation of a new channel `a` and the two parallel behaviours  $\mathbf{g}\langle a \rangle$  and  $\mathbf{r}\langle a \rangle$ . The type given by equation  $\mathbf{g}(x)$ , which types the generator, identifies the infinite output behaviour along the given channel `x`. The type equation  $\mathbf{f}(x, y)$  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  $\mathbf{r}(x)$  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, for which we can show that safety in the symbolic semantics implies general safety.

**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 lo-

cal 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  $\mathbf{r}\langle b \rangle$  in the equation  $\mathbf{r}(x)$  is *fenced*, since the recursive instances of  $\mathbf{r}$  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** Fenced types can be *symbolically* executed as CCS processes in a representative *finite-state* labelled transition system that soundly approximates their unrestricted semantics. By proving that liveness and safety in the symbolic semantics imply liveness and safety in the unbounded semantics, we are then able to produce a decision procedure for these properties on types. For instance, our analysis deems the type  $\mathbf{t}_0$  as safe.

**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  $\mathbf{f}(x, y)$  which just indicates an internal choice between two behaviours. The justification for the liveness of  $\mathbf{t}_0$  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 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 liveness and channel safety wrt. the symbolic semantics. This is then shown to be sound wrt. the general unbounded semantics (§ 4); We characterise the MiGo programs whose liveness and safety is 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 [27], 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, dubbed MiGo (mini-go). Beyond sending and receiving data values along channels, the Go language supports three key concurrency features:

$P, Q ::= \pi; P$ $\text{close } u; P$ $\text{select}\{\pi_i; P_i\}_{i \in I}$ $\text{if } e \text{ then } P \text{ else } Q$ $\text{newchan}(y:\sigma); P$ $P \mid Q \mid \mathbf{0}$ $X(\tilde{e}, \tilde{u})$ $(\nu c)P$ $c\langle\sigma\rangle::\tilde{v} \mid c^*\langle\sigma\rangle::\tilde{v}$	$u ::= a \mid x$ $\pi ::= u!\langle e \rangle \mid u?(y) \mid \tau$ $v ::= n \mid \text{true} \mid \text{false} \mid x$ $e ::= v \mid \text{not}(e) \mid \text{succ}(e)$ $D ::= X(\tilde{x}) = P$ $\mathbf{P} ::= \{D_i\}_{i \in I} \text{ in } P$ $\sigma ::= \text{bool} \mid \text{int} \mid \dots$
--	--

Figure 1. Syntax of MiGo.

**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*,  $\pi$  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  $\mathbf{P}$ , 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  $\sigma$ .  $\text{fn}(P)$  and  $\text{fv}(P)$  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  $\text{fv}(P) \subseteq \{\tilde{x}\}$  and  $\text{fn}(D) = \emptyset$ . 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  $\pi$  (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*  $\tau$ ) followed by process  $P$ ; a *close process*  $\text{close } u; P$  closes the channel  $u$  and continues as  $P$ ; a *selection process*  $\text{select}\{\pi_i; P_i\}_{i \in I}$  denotes a choice between the several  $P_i$  processes, where each  $P_i$  is guarded by a prefix  $\pi_i$ . Thus, the choice construct non-deterministically selects between any process  $P_i$  whose guarding action can be executed (note that a  $\tau$  action prefix can always be executed, cf. § 2.2); the standard *conditional process*  $\text{if } e \text{ then } P \text{ else } Q$ , *parallel process*  $P \mid Q$ , and the *inactive process*  $\mathbf{0}$  (often omitted); a *new channel process*  $\text{newchan}(y:\sigma); P$  creates a new channel with payload type  $\sigma$ , binding it to  $y$  in the continuation  $P$ ; *process call*  $X(\tilde{e}, \tilde{u})$  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  $\sigma$ , or a buffer for a *closed* channel  $c^*\langle\sigma\rangle::\tilde{v}$ . 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  $\mathbf{P}$  in MiGo, written  $\{D_i\}_{i \in I}$  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:

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

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:

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

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:

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

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  $n^{\text{th}}$  number of the Fibonacci sequence.

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

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<sup>th</sup> element of the Fibonacci sequence is given below:

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

On the other hand, the following program should be deemed not live since the outputs from the recursive calls are never sent to the

$$\begin{array}{c}
\text{[SCOM]} \frac{e \downarrow v}{c!(e); P \mid c?(y); Q \mid c(\sigma)::\emptyset \rightarrow P \mid Q \{v/y\} \mid c(\sigma)::\emptyset} \\
\text{[SCLOSE]} \quad c?(y); P \mid c^*(\sigma)::\emptyset \rightarrow P \{v^\sigma/y\} \mid c^*(\sigma)::\emptyset \\
\text{[CLOSE]} \quad \text{close } c; P \mid c(\sigma)::\tilde{v} \rightarrow P \mid c^*(\sigma)::\tilde{v} \\
\text{[TAU]} \quad \tau; P \rightarrow P \\
\text{[NEWC]} \frac{c \notin \text{fn}(P)}{\text{newchan}(y;\sigma); P \rightarrow (\nu c)(P \{c/y\} \mid c(\sigma)::\emptyset)} \\
\text{[PAR]} \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad \text{[RES]} \frac{P \rightarrow P'}{(\nu c)P \rightarrow (\nu c)P'} \\
\text{[STR]} \frac{P \equiv Q \rightarrow Q' \equiv P'}{P \rightarrow P'} \quad \text{[SEL]} \frac{\pi_j; P_j \mid P \rightarrow R \quad j \in I}{\text{select}\{\pi_i; P_i\}_{i \in I} \mid P \rightarrow R} \\
\text{[IFT]} \frac{e \downarrow \text{true}}{\text{if } e \text{ then } P \text{ else } Q \rightarrow P} \quad \text{[IFF]} \frac{e \downarrow \text{false}}{\text{if } e \text{ then } P \text{ else } Q \rightarrow Q} \\
\text{[DEF]} \frac{P \{\tilde{v}, \tilde{c}/\tilde{x}\} \mid Q \rightarrow R \quad e_i \downarrow v_i \quad X(\tilde{x}) = P \in \{D_i\}_{i \in I}}{X(\tilde{c}, \tilde{c}) \mid Q \rightarrow R}
\end{array}$$

### Structural Congruence

$$\begin{array}{l}
P \mid Q \equiv Q \mid P \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad P \mid \mathbf{0} \equiv P \\
(\nu c)(\nu d)P \equiv (\nu d)(\nu c)P \quad (\nu c)\mathbf{0} \equiv \mathbf{0} \quad (\nu c)c(\sigma)::\tilde{v} \equiv \mathbf{0} \\
P \mid (\nu c)Q \equiv (\nu c)(P \mid Q) \quad (c \notin \text{fn}(P)) \quad (\nu c)c^*(\sigma)::\tilde{v} \equiv \mathbf{0}
\end{array}$$

**Figure 2.** Reduction Semantics.

initial  $\text{Fib}_{bad}$  call and so the answer is never returned to the main process (i.e.  $c?(u)$ ;  $\mathbf{0}$  can never fire).

$$\begin{array}{l}
\text{Fib}_{bad}(n, c) \triangleq \text{newchan}(c:\text{int}); \\
(\text{Fib}_{bad}(n-1, c') \mid \text{Fib}_{bad}(n-2, c') \mid c?(x); c?(y); c!(x+y))
\end{array}$$

## 2.2 Operational Semantics

The semantics of MiGo, written  $P \rightarrow Q$ , is defined by the reduction rules of Figure 2, together with the standard structural congruence  $P \equiv Q$ . 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  $v^\sigma$  of type  $\sigma$ . Rule [CLOSE] changes the state of a buffer from open ( $c(\sigma)::\tilde{v}$ ) to closed ( $c^*(\sigma)::\tilde{v}$ ). 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 [39].

## 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  $\tau$ -guarded branch. This notion of timeout makes the definition of liveness slightly challenging. Consider the following:

$$\begin{array}{ll}
P_1 \triangleq \text{select}\{a!(v), b?(x); \mathbf{0}, \tau; P_t\} & R_1 \triangleq a?(y); \mathbf{0} \\
P_2 \triangleq \text{select}\{a!(v), b?(x); \mathbf{0}\} & R_2 \triangleq c?(y); \mathbf{0}
\end{array}$$

A select with a branch guarded by  $\tau$  contains a branch that is always enabled by default (since  $\tau$  actions can always fire silently). Hence if the continuation of the  $\tau$  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,  $P_1 \mid R_2$  is not live unless  $P_t \mid R_2$  is live ( $i \in \{1, 2\}$ ).

Accounting for these features, we formalise safety and liveness properties, extending the notion of barbed process predicates [31]. 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  $P \downarrow_{\bar{a}}$  are dual; a synchronisation barb  $P \downarrow_{[a]}$ , indicating that  $P$  can perform a synchronisation on  $a$ ; a channel close barb  $P \downarrow_{\text{end}[a]}$ , denoting that  $P$  can close channel  $a$ ; and  $P \downarrow_{a^*}$ , 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  $\text{select}\{\pi_i; P_i\}_{i \in I}$  are those of all the processes that make up the external choice, provided that *all of them* can exhibit a barb.

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

$$c?(x) \downarrow_c \quad c!(e) \downarrow_{\bar{c}} \quad \frac{\pi \downarrow_o}{\pi; Q \downarrow_o} \quad \text{close } c; Q \downarrow_{\text{end}[c]} \quad c^*(\sigma)::\tilde{v} \downarrow_{c^*}$$

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

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

$$\frac{\forall i \in \{1, \dots, n\} : \pi_i \downarrow_{o_i}}{\text{select}\{\pi_i; P_i\}_{i \in \{1, \dots, n\}} \downarrow_{\{o_1 \dots o_n\}}} \quad \frac{P \downarrow_a \quad Q \downarrow_{\bar{a}} \text{ or } Q \downarrow_{a^*}}{P \mid Q \downarrow_{[a]}}$$

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

$P \downarrow_o$  if  $P \rightarrow^* P'$  and  $P' \downarrow_o$  with  $o \in \{c, \bar{c}, [c], \text{end}[c], c^*\}$ .

For example, we have that  $\neg(P_1 \downarrow_a)$  for any  $a$ , whereas  $P_2 \downarrow_{\bar{a}, b}$ . 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  $P_t \downarrow_{\bar{c}}$  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 are non- $\tau$ ), the state can also eventually synchronise on one of the action prefixes.

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

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

$$\begin{aligned}
T, S &:= \kappa; T \mid \oplus\{T_i\}_{i \in I} \mid \&\{\kappa_i; T_i\}_{i \in I} \mid (T \mid S) \mid \mathbf{0} \\
&\mid (\text{new } a)T \mid \text{end}[u]; T \mid \mathbf{t}_X(\bar{u}) \mid (\nu a)T \mid [a] \mid a^* \\
\mathbf{T} &:= \{\mathbf{t}_{X_i}(\bar{y}_i) = T_i\}_i \text{ in } S \quad \kappa := \bar{u} \mid u \mid \tau
\end{aligned}$$

**Figure 3.** Syntax of Types.

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  $\mathbf{P}$  is *channel safe* if for all  $Q$  such that  $\mathbf{P} \rightarrow^* (\nu \bar{c})Q$ , if  $Q \downarrow_{a^*}$  then  $\neg(Q \downarrow_{\text{end}[a]})$  and  $\neg(Q \downarrow_{\bar{a}})$ .

### 3. A Behavioural Typing System for MiGo

Go's channel types are related to those of the  $\pi$ -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  $\bar{u}$ , input  $u$  along channel  $u$ , or an explicit  $\tau$  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  $\kappa_i$  which drive the choice. Types include parallel composition of behaviours  $T \mid S$ , inaction  $\mathbf{0}$  and channel creation  $(\text{new } a)T$  (binding  $a$  in  $T$ ). The type  $\text{end}[u]; T$  denotes the closing of channel  $u$  followed by the behaviour  $T$ . The type variable  $\mathbf{t}_X(\bar{u})$  associated to a process variable  $X$ , denotes the behaviour bound to variable  $\mathbf{t}_X$  in the definition environment, with formal parameters instantiated to  $\bar{u}$ . The type  $\{\mathbf{t}_{X_i}(\bar{y}_i) = T_i\}_i$  in  $S$  codifies a set of (parameterised) mutually recursive type definitions  $\mathbf{t}_{X_i}(\bar{y}_i) = T_i$ , bound in  $S$ . This set of equations denoted by  $\mathbf{T}$  is the type assigned to top-level programs  $\mathbf{P}$ .

The type constructs  $(\nu a)T$ ,  $[a]$  and  $a^*$  denote the type representations of runtime channel bindings, open and closed buffers, respectively. We write  $\text{fn}(T)$  and  $\text{fv}(T)$  for the free names and variables of type  $T$ , respectively.

#### 3.2 Typing System

We first explain the two essential differences from (linear or session-based) type systems of the  $\pi$ -calculus [21, 22, 41]:

**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:

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

**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:

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

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 not compatible by the usual branch subtyping).

**Process Typing** The judgement  $(\Gamma \vdash P \blacktriangleright T)$  for processes is defined in Figure 4 where  $\Gamma$  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  $\Gamma$ . We write  $u:\text{ch}(\sigma)$  to denote that  $u$  stands for a channel with payload type  $\sigma$ . We omit the typing rules of expressions  $e$ , given that expressions only include basic data types. We write  $\text{dom}(\Gamma)$  for the set of channel bindings in  $\Gamma$ .

The rules implement a very close correspondence between processes and their respective types: Rule  $\langle \text{OUT} \rangle$  types output processes with the output prefix type  $\bar{u}; T$ , checking that the type of the object to be sent matches the payload type  $\sigma$  of channel  $u$ , and that the continuation  $P$  has type  $T$ . The rule  $\langle \text{IN} \rangle$  for inputs is dual. Rule  $\langle \text{SEL} \rangle$  types the select construct with the external choice type, whereas rule  $\langle \text{IF} \rangle$  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  $\tau$  are straightforward. Rule  $\langle \text{NEW} \rangle$  allocates a fresh type-level channel name with payload type  $\sigma$ . Rule  $\langle \text{VAR} \rangle$  matches a process variable with its corresponding type variable, checking that the specified arguments have the appropriate types.

**Program Typing** The judgement  $(\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T})$  is defined in Figure 4. A process declaration  $X(\bar{x}:\bar{\sigma}, \bar{y}:\text{ch}(\bar{\sigma}')) = P$  is matched with  $\mathbf{t}_X(\bar{y}) = T$ , connecting the process level variable  $X$  with the type variable  $\mathbf{t}$ , where  $P$  may use any of the parameters specified in the recursion variable. The typing rule for programs  $\langle \text{DEF} \rangle$  assigns a program the type  $\{\mathbf{t}_{X_i}(\bar{y}_i) = T_i\}_{i \in I}$  in  $S$ , checking that each definition is typed with  $\mathbf{t}_{X_i}(\bar{y}_i) = T_i$  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  $\langle \text{RES} \rangle$ , 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  $\langle \text{CBUFF} \rangle$  and  $\langle \text{BUFF} \rangle$ , respectively, noting that the set of active buffers is a singleton containing the appropriate buffer reference. The parallel rule  $\langle \text{PARR} \rangle$  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  $\mathbf{T}$  which is obtained from the program by collecting all the types for definitions and adding a distinguished unique definition  $\mathbf{t}_0() = S$  for the program entry point. We often write  $X_0$  to stand for the process variable of the program entry point.

## 4. Verification of Behavioural Types

This section introduces our main definition, *fencing*, and a technique to decide whether types are live and channel safe. 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 con-

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

$$\begin{array}{c}
\langle \text{OUT} \rangle \frac{\Gamma \vdash u:\text{ch}(\sigma) \quad \Gamma \vdash e:\sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash u!\langle e \rangle; P \blacktriangleright \bar{u}; T} \\
\langle \text{IN} \rangle \frac{\Gamma \vdash u:\text{ch}(\sigma) \quad \Gamma, x:\sigma \vdash P \blacktriangleright T}{\Gamma \vdash u?(x); P \blacktriangleright u; T} \quad \langle \text{TAU} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T} \\
\langle \text{CLOSE} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{close } u; P \blacktriangleright \text{end}[u]; T} \quad \langle \text{ZERO} \rangle \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}} \\
\langle \text{SEL} \rangle \frac{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}{\Gamma \vdash \text{select}\{\pi_i; P_i\}_{i \in I} \blacktriangleright \&\{\kappa_i; T_i\}_{i \in I}} \\
\langle \text{IF} \rangle \frac{\Gamma \vdash e:\text{bool} \quad \Gamma \vdash P \blacktriangleright S \quad \Gamma \vdash Q \blacktriangleright T}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } Q \blacktriangleright \oplus\{S, T\}} \\
\langle \text{NEW} \rangle \frac{\Gamma, y:\text{ch}(\sigma) \vdash P \blacktriangleright T \quad c \notin \text{dom}(\Gamma) \cup \text{fn}(T)}{\Gamma \vdash \text{newchan}(y:\sigma); P \blacktriangleright (\text{new } c)T \{c/y\}} \\
\langle \text{PAR} \rangle \frac{\Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \mid Q \blacktriangleright (T \mid S)} \\
\langle \text{VAR} \rangle \frac{\Gamma \vdash \tilde{e}:\tilde{\sigma} \quad \Gamma \vdash \tilde{u}:\text{ch}(\tilde{\sigma}')}{\Gamma, X(\tilde{\sigma}, \text{ch}(\tilde{\sigma}')) \vdash X(\tilde{e}, \tilde{u}) \blacktriangleright \mathbf{t}_X\langle \tilde{u} \rangle}
\end{array}$$

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

$$\langle \text{DEF} \rangle \frac{\Gamma, X_i(\tilde{\sigma}_i, \text{ch}(\tilde{\sigma}'_i)), \tilde{x}_i:\tilde{\sigma}_i, \tilde{y}_i:\text{ch}(\tilde{\sigma}'_i) \vdash P_i \blacktriangleright T_i \quad \Gamma, X_1(\tilde{\sigma}_1, \text{ch}(\tilde{\sigma}'_1)), \dots, X_n(\tilde{\sigma}_n, \text{ch}(\tilde{\sigma}'_n)) \vdash Q \blacktriangleright S}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i \in I} \text{ in } Q \blacktriangleright \{\mathbf{t}_{X_i}(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S}$$

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

$$\begin{array}{c}
\langle \text{INT} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash_{\emptyset} P \blacktriangleright T} \quad \langle \text{RES} \rangle \frac{\Gamma, c:\text{ch}(\sigma) \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B \setminus c} (\nu c)P \blacktriangleright (\nu c)T} \\
\langle \text{CBUFF} \rangle \frac{\Gamma \vdash a:\text{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a^*\langle \sigma \rangle::\tilde{v} \blacktriangleright a^*} \quad \langle \text{BUFF} \rangle \frac{\Gamma \vdash a:\text{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a\langle \sigma \rangle::\tilde{v} \blacktriangleright [a]} \\
\langle \text{PARR} \rangle \frac{\Gamma \vdash_B P \blacktriangleright T \quad \Gamma \vdash_{B'} Q \blacktriangleright S \quad B \cap B' = \emptyset}{\Gamma \vdash_{B \cup B'} P \mid Q \blacktriangleright (T \mid S)}
\end{array}$$

**Figure 4.** Typing Rules (Processes and Programs).

sists 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 if the symbolic execution of a type is live (resp. channel safe), then the type is live (resp. channel safe) under the general semantics; and that liveness and channel safety are decidable for the 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, ranged over by  $\alpha$  and  $\beta$ , have the form:

$$\alpha, \beta := \bar{a} \mid a \mid \tau \mid [a] \mid \overline{\text{end}[a]} \mid \overline{\text{end}[a]} \mid a^*$$

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

$$\begin{array}{c}
|\text{SND}| \quad \bar{a}; T \xrightarrow{\bar{a}} T \quad |\text{RCV}| \quad a; T \xrightarrow{a} T \quad |\text{TAU}| \quad \tau; T \xrightarrow{\tau} T \\
|\text{SEL}| \frac{j \in I}{\oplus\{T_i\}_{i \in I} \xrightarrow{\tau} T_j} \quad |\text{BRA}| \frac{\kappa_j; T_j \xrightarrow{\alpha} T_j}{\&\{\kappa_i; T_i\}_{i \in I} \xrightarrow{\alpha} T_j} \\
|\text{PAR}| \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \quad |\text{COM}| \frac{T \xrightarrow{\beta} T' \quad S \xrightarrow{\alpha} S' \quad \beta = \bar{a}, a^*}{T \mid S \xrightarrow{[a]} T' \mid S'} \\
|\text{NEW}| \quad (\text{new } a)T \xrightarrow{\tau} (\nu a)(T \mid [a]) \quad |\text{END}| \quad \text{end}[a]; T \xrightarrow{\overline{\text{end}[a]}} T \\
|\text{BUF}| \quad [a] \xrightarrow{\overline{\text{end}[a]}} a^* \quad |\text{CLOSE}| \frac{T \xrightarrow{\overline{\text{end}[a]}} T' \quad S \xrightarrow{\overline{\text{end}[a]}} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \\
|\text{CLD}| \quad a^* \xrightarrow{a^*} a^* \quad |\text{RES-1}| \frac{T \xrightarrow{\alpha} T' \quad \text{fn}(\alpha) \neq \{a\}}{(\nu a)T \xrightarrow{\alpha} (\nu a)T'} \quad |\text{RES-2}| \frac{T \xrightarrow{[a]} T'}{(\nu a)T \xrightarrow{\tau} (\nu a)T'} \\
|\text{EQ}| \frac{T \equiv_{\alpha} T' \quad T \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''} \quad |\text{DEF}| \frac{T \{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \quad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}(\tilde{a}) \xrightarrow{\alpha} T'}
\end{array}$$

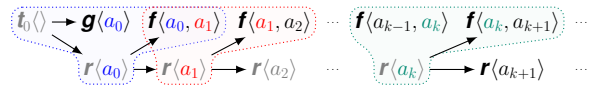
**Figure 5.** LTS Semantics of Types.

Rule  $|\text{SND}|$  (resp.  $|\text{RCV}|$ ) allows a type to emit a send (resp. receive) action on a channel  $a$ . Rules  $|\text{SEL}|$  and  $|\text{BRA}|$  model internal and (mixed) external choices, respectively. Rule  $|\text{COM}|$  allows two types to synchronise on a dual action (send with receive, or closed channel with receive). Rule  $|\text{NEW}|$  creates a new (open) channel for  $a$ . Rule  $|\text{END}|$ , together with rule  $|\text{CLOSE}|$ , allows a type to request the closure of channel  $a$ . Rule  $|\text{BUF}|$  models a transition from an open channel to a closed one, and rule  $|\text{CLD}|$  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  $|\text{CLOSE}|$ ,  $|\text{PAR}|$ , and  $|\text{COM}|$ . We write  $\rightarrow$  for  $\xrightarrow{\tau} \cup \equiv$  where  $\equiv$  is defined as for processes (see appendix [27]).

#### 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,  $\mathbf{t}_0$  is the parent of  $\mathbf{g}\langle a_0 \rangle$  and  $\mathbf{r}\langle a_0 \rangle$ .

We observe that there are only finitely many kinds of concurrent threads (i.e.  $\mathbf{t}_0$ ,  $\mathbf{g}\langle a \rangle$ ,  $\mathbf{f}\langle a, b \rangle$ , and  $\mathbf{r}\langle a \rangle$ ). Also given any name in the diagram, e.g.  $a_1$ , it is shared only by finitely many threads. For instance,  $a_1$  is “known” only to  $\mathbf{f}\langle a_0, a_1 \rangle$ ,  $\mathbf{f}\langle a_1, a_2 \rangle$ , and  $\mathbf{r}\langle a_1 \rangle$ . 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.  $\mathbf{r}\langle a_1 \rangle$ , 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 execut-

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

$$\begin{array}{c} \frac{\tilde{y} \neq \varepsilon}{G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} \mathbf{t}(\tilde{u})} \text{[AXIOM-FS]} \quad \frac{\mathbf{s}(\tilde{u}) \in G}{G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} \mathbf{s}(\tilde{u})} \text{[DEF-}\in\text{]} \\ \frac{\tilde{u} \prec \tilde{z} \quad \mathbf{t}(\tilde{x}) = T \quad \& \notin T}{G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} \mathbf{t}(\tilde{u})} \text{[AXIOM-INF]} \\ \frac{G; \varepsilon; \tilde{z} \cdot \tilde{y} \vdash_{\mathbf{t}} T_1 \quad G; \varepsilon; \tilde{z} \cdot \tilde{y} \vdash_{\mathbf{t}} T_2}{G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} T_1 \mid T_2} \text{[PAR]} \\ \frac{\mathbf{t} \neq \mathbf{s} \quad \mathbf{s}(\tilde{u}) \notin G \quad G \cdot \mathbf{s}(\tilde{u}); \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} T \{ \tilde{u}/\tilde{x} \} \quad \mathbf{s}(\tilde{x}) = T}{G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} \mathbf{s}(\tilde{u})} \text{[DEF-}\notin\text{]} \\ \frac{\forall i \in I : \Delta \vdash_{\mathbf{t}} T_i}{\Delta \vdash_{\mathbf{t}} \mathbf{0}} \text{[END]} \quad \frac{\forall i \in I : \Delta \vdash_{\mathbf{t}} T_i}{\Delta \vdash_{\mathbf{t}} \oplus \{T_i\}_{i \in I}} \text{[SEL]} \quad \frac{\forall i \in I : \Delta \vdash_{\mathbf{t}} T_i}{\Delta \vdash_{\mathbf{t}} \& \{T_i\}_{i \in I}} \text{[BRA]} \\ \frac{\Delta \vdash_{\mathbf{t}} T}{\Delta \vdash_{\mathbf{t}} \text{end}[u]; T} \text{[END]} \quad \frac{\Delta \vdash_{\mathbf{t}} T}{\Delta \vdash_{\mathbf{t}} (\text{new } a)T} \text{[RES]} \quad \frac{\Delta \vdash_{\mathbf{t}} T}{\Delta \vdash_{\mathbf{t}} \kappa; T} \text{[PREF]} \end{array}$$

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

ing within that fence approximate a form of finite control. We use judgements of the form  $G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} T$  to guarantee that a type definition  $\mathbf{t}(\tilde{x}) = T$  is fenced; where  $G$  records previously encountered recursive calls,  $\tilde{y}$  represents the names that  $\mathbf{t}$  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  $\varepsilon$  for the empty environment. The judgement  $G; \tilde{y}; \tilde{z} \vdash_{\mathbf{t}} T$  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,  $yz a \prec x y z$ , but  $x a z \not\prec x y z$ .

We comment on the key rules of Figure 6. Rule [PAR] identifies multi-threaded types by moving the  $\tilde{y}$  environment to the  $\tilde{z}$  environment. The axiom [AXIOM-FS] states that  $\mathbf{t}(\tilde{u})$  is fenced if  $\tilde{y} \neq \varepsilon$ , i.e. the type corresponding to  $\mathbf{t}$  does not contain any parallel composition. While axiom [AXIOM-INF] requires that  $\tilde{u} \prec \tilde{z}$  hold, i.e. any recursive call over  $\mathbf{t}$  uses strictly fewer (non newly created) names, and enforces that the selection operator ( $\&$ ) does not appear in (the expansion of) the body of  $\mathbf{t}$ .

The second part of the premise guarantees that after a certain number of recursive calls, the type  $\mathbf{t}$  will have completely “forgotten” the names it started executing with; hence moving outside of the fence. Rules [DEF- $\in$ ] and [DEF- $\notin$ ] deal with recursive calls to different type variables.

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

A consequence of fencing is that for each equation  $\mathbf{t}(\tilde{x}) = T$  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  $\mathbf{t}$  recurses on; or (2)  $T$  is *multi-threaded* (i.e. there is a parallel composition within  $T$ ), hence for each occurrence of a  $\mathbf{t}$ -recursion, at least one of the parameters must be forgotten. In the prime sieve example, the types  $\mathbf{g}(x)$  and  $\mathbf{f}(x, y)$  always recurse on the same parameters, but they do not include parallel composition. While the type  $\mathbf{r}(x)$  has a parallel composition, its parameter  $x$  is “forgotten” at the  $\mathbf{r}(b)$  recursive call (i.e.  $b \prec x$ ).

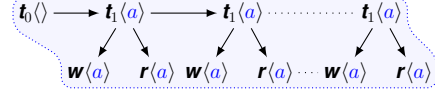
## 4.2.2 Examples on Fencing

**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.

$$\begin{array}{l} \mathbf{w}(x) \triangleq \bar{x}; \mathbf{w}\langle x \rangle = T_w \quad \mathbf{t}_1(x) \triangleq \mathbf{w}\langle x \rangle \mid \mathbf{r}\langle x \rangle \mid \mathbf{t}_1\langle x \rangle = T_1 \\ \mathbf{r}(x) \triangleq x; \mathbf{r}\langle x \rangle = T_r \quad \mathbf{t}_0() \triangleq (\text{new } a)(\mathbf{t}_1\langle a \rangle) = T_0 \end{array}$$

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

The topology induced by the type is given as:

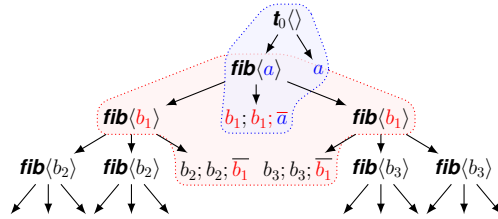


We can observe that there are infinitely many instances of types that “know” the name  $a$ .

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

$$\begin{array}{l} \mathbf{fib}(x) \triangleq \bar{x} \oplus (\text{new } b)(\mathbf{fib}\langle b \rangle \mid b; b; \bar{x} \mid \mathbf{fib}\langle b \rangle) = T_{\mathbf{fib}} \\ \mathbf{t}_0() \triangleq (\text{new } a)(\mathbf{fib}\langle a \rangle \mid a) = T_0 \end{array}$$

Observe that  $\varepsilon; \varepsilon; \varepsilon \vdash_{\mathbf{t}_0} T_0$ , holds trivially since there is no recursive call on  $\mathbf{t}_0$ . The judgement  $\varepsilon; x; \varepsilon \vdash_{\mathbf{fib}} T_{\mathbf{fib}}$  also holds since we have  $b \prec x$ . Essentially, the equation  $\mathbf{fib}$  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  $\mathbf{t}_0$ ); while the  $\{b_1\}$ -fence includes five components: one instance of  $\mathbf{fib}(x)$  as well as two of its recursive children and three instances of the non-recursive component of  $\mathbf{fib}$ .

## 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.  $\mathbf{t}(\tilde{u})$ , by its corresponding definition. The overall purpose of the symbolic semantics is that for any  $\mathbf{T}$  such that  $\text{Fenced}(\mathbf{T})$ , the symbolic LTS of  $\mathbf{T}$  is *finite state*.

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 $\geq$ ] replace rule [RES-1], and rules [R2<] and [R2 $\geq$ ] replace [RES-2].

In a term  $N \triangleleft T$ ,  $N$  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

$$\begin{array}{c}
\frac{N \triangleleft T \{\bar{a}/\bar{x}\} \xrightarrow{\alpha}_k N \triangleleft T' \quad \mathbf{t}(\bar{x}) = T \quad \bar{a} \cap N \neq \emptyset}{|\text{DEF}| \quad N \triangleleft \mathbf{t}(\bar{a}) \xrightarrow{\alpha}_k N \triangleleft T'} \\
\frac{N \uplus \{a\} \triangleleft T \xrightarrow{\alpha}_k N \uplus \{a\} \triangleleft T' \quad \text{fn}(\alpha) \neq \{a\} \quad |N| < k}{|\text{R1}<| \quad N \triangleleft (\nu a)T \xrightarrow{\alpha}_k N \triangleleft (\nu a)T'} \\
\frac{N \uplus \{a\} \triangleleft T \xrightarrow{[a]}_k N \uplus \{a\} \triangleleft T' \quad |N| < k}{|\text{R2}<| \quad N \triangleleft (\nu a)T \xrightarrow{\tau}_k N \triangleleft (\nu a)T'} \\
\frac{N \triangleleft T \xrightarrow{\alpha}_k N \triangleleft T' \quad \text{fn}(\alpha) \neq \{a\} \quad |N| \geq k}{|\text{R1}\geq| \quad N \triangleleft (\nu a)T \xrightarrow{\alpha}_k N \triangleleft (\nu a)T'} \\
\frac{N \triangleleft T \xrightarrow{[a]}_k N \triangleleft T' \quad |N| \geq k}{|\text{R2}\geq| \quad N \triangleleft (\nu a)T \xrightarrow{\tau}_k N \triangleleft (\nu a)T'}
\end{array}$$

**Figure 7.** Symbolic Semantics for Types.

unfolding of a type is such that there is no clash with the names in  $N$ .

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

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

$$\begin{array}{c}
\{a\} \triangleleft (\nu b)(\mathbf{g}\langle a \mid \mathbf{f}\langle a, b \mid \mathbf{r}\langle b \rangle) \\
\rightarrow_1^{[a]} \{a\} \triangleleft (\nu b)(\mathbf{g}\langle a \mid \bar{b}; \mathbf{f}\langle a, b \mid \mathbf{r}\langle b \rangle)
\end{array}$$

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

$$\begin{array}{c}
\{a, b\} \triangleleft \mathbf{g}\langle a \mid \bar{b}; \mathbf{f}\langle a, b \mid \mathbf{r}\langle b \rangle \\
\rightarrow_2^{[b]} \{a, b\} \triangleleft \mathbf{g}\langle a \mid \mathbf{f}\langle a, b \mid (\text{new } c)(\mathbf{f}\langle b, c \mid \mathbf{r}\langle c \rangle)
\end{array}$$

#### 4.4 Liveness and Channel Safety for Types

Following § 2.3, we define liveness and channel safety properties for types. The formal definitions of liveness and channel safety rely on barbs  $T \Downarrow_o$ , which are identical to those of processes (see appendix [27]).

We write  $T \Downarrow_o$  if  $N \triangleleft T \xrightarrow{*}_n N \triangleleft T'$  and  $T' \Downarrow_o$ , with  $N = \text{fn}(T)$ ,  $n = |N|$ , and  $o \in \{a, \bar{a}, [a], \text{end}[a], a^*\}$ . Observe that the predicate  $T \Downarrow_o$  is defined wrt. the symbolic semantics.

**Definition 4.3** (Liveness). The system  $\mathbf{T}$  satisfies *k-liveness* if for all  $T$  such that  $\emptyset \triangleleft \mathbf{t}_0 \langle \rangle \xrightarrow{*}_k \emptyset \triangleleft (\nu \bar{a})T$ ,

- (a) If  $T \Downarrow_a$  or  $T \Downarrow_{\bar{a}}$  then  $T \Downarrow_{[a]}$ .
- (b) If  $T \Downarrow_{\bar{a}}$  then  $T \Downarrow_{[a_i]}$  for some  $a_i \in \bar{a}$ .

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

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

$$\mathbf{t}_1(x) \triangleq (\text{new } b)(\mathbf{t}_1\langle b \mid b; \bar{x} \rangle \quad \mathbf{t}_0() \triangleq (\text{new } a)(\mathbf{t}_1\langle a \mid a \rangle)$$

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 4.4** (Channel Safety). The system  $\mathbf{T}$  satisfies *k-channel safety* if for all  $T$  such that  $\emptyset \triangleleft \mathbf{t}_0 \langle \rangle \xrightarrow{*}_k \emptyset \triangleleft (\nu \bar{a})T$ , if  $T \Downarrow_{a^*}$  then  $\neg(T \Downarrow_{\text{end}[a]})$  and  $\neg(T \Downarrow_{\bar{a}})$ .

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

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

$$\mathbf{f}(x, y) \triangleq x; \bar{y}; x; \bar{y}; x; \bar{y}; x; \bar{y}; x; \bar{y}; \mathbf{0}$$

with  $\mathbf{g}(x)$ ,  $\mathbf{r}(x)$ , and  $\mathbf{t}_0 \langle \rangle$  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  $\mathbf{f}(x, y)$ ; while a bound of 3 enables spawning another filter process, hence to explore all the states of  $\mathbf{f}(x, y)$ .

#### 4.5 Decidability and Soundness of the Verification

We show that our symbolic semantics is sound wrt. the general semantics of types, entailing the decidability of liveness and channel safety (**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  $\text{Fenced}(\mathbf{T})$ , then the set  $\{[T] \mid \emptyset \triangleleft \mathbf{t}_0 \langle \rangle \xrightarrow{*}_k \emptyset \triangleleft T\}$  is finite, for any finite  $k$ .*

The crux of the proof is to show that the number of occurrences of a type variable  $\mathbf{t}$  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  $\mathbf{t}(\bar{a})$  only if  $\bar{a} \cap N \neq \emptyset$ .

**Theorem 4.1** (Decidability). *For all  $\mathbf{T}$  s.t.  $\text{Fenced}(\mathbf{T})$ , it is decidable whether or not  $\mathbf{T}$  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).

The *soundness* result mainly follows from the fact that the fencing predicate implies that any infinite-state system of types always produces, essentially, a finite number of (possibly infinitely wide) parallel compositions of *finite-state* sub-terms, where each of these sub-terms can only communicate with a finite number of neighbours. Moreover, by disallowing infinite-state types from using the external choice construct  $\&$ , we ensure that a system of such types cannot deterministically maintain any form of persistent multi-alternative state between its infinitely many sub-terms.

We proceed by showing that there is a bound  $k \in \mathbb{N}$  for which the *k-symbolic semantics* guarantees liveness and safety in the general semantics for types. It is then enough to check liveness only for a finite fragment of any infinitely wide parallel composition.

**Theorem 4.2** (Soundness of Liveness). *For all  $\mathbf{T}$  s.t.  $\text{Fenced}(\mathbf{T})$ , there exists a finite  $k \geq 0$  such that if  $\mathbf{T}$  is *k-live*, then  $\mathbf{T}$  is *live*.*

**Theorem 4.3** (Soundness of Safety). *For all  $\mathbf{T}$  s.t.  $\text{Fenced}(\mathbf{T})$ , there exists a finite  $k \geq 0$  such that if  $\mathbf{T}$  is *live* and *k-channel safe*, then  $\mathbf{T}$  is *channel-safe*.*

A sufficient bound  $k$  is a (linear) factor of the size (of the syntax tree) of the one-time expansion of  $\mathbf{T}$ .

## 5. Properties of MiGo

We now make precise the properties our behavioural type analysis ensures on MiGo programs. We show that if a program  $\mathbf{P}$  is typed by a safe type, then  $\mathbf{P}$  is safe according to Definition 2.3. For liveness, we identify the classes of programs for which liveness of types implies program liveness.



## 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  $P \rightarrow P'$ . Then there exists  $T'$  such that  $\Gamma \vdash_{B'} P' \blacktriangleright T'$  with  $T \rightarrow T'$ .*

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  $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$  and  $\mathbf{T}$  is safe. Then  $\mathbf{P}$  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 4.3) 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  $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$ . We write  $\mathbf{P} \in \text{May}\Downarrow$  if for all  $X_0 \langle \rangle \rightarrow^* P, P \rightarrow^* \mathbf{0}$ .

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

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

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

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:

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

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

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

**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:

$$l := \epsilon \mid n \cdot L \mid n \cdot R$$

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

$$\begin{array}{c} \text{[IFTM]} \frac{e \downarrow \text{true}}{\text{if } {}^n e \text{ then } P \text{ else } Q \xrightarrow{n \cdot L} P} \quad \text{[IFFM]} \frac{e \downarrow \text{false}}{\text{if } {}^n e \text{ then } P \text{ else } Q \xrightarrow{n \cdot R} Q} \end{array}$$

**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

$P \xrightarrow{l_1} P_1 \xrightarrow{l_2} \dots$ , with  $\vec{l} = \{l_1 l_2 \dots\}$ . 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:

$$\mathbb{C}_r := [] \mid (P \mid \mathbb{C}_r) \mid (\mathbb{C}_r \mid P) \mid (\nu a)\mathbb{C}_r$$

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  $P \in \text{Inf}$ , iff  $\text{mark}(P) \rightarrow^* \mathbb{C}_r[\text{if}^n e \text{ then } Q_1 \text{ else } Q_2] = R$ , for some  $n$ , and  $R$  has an infinite trace where  $n\cdot\text{L}$  or  $n\cdot\text{R}$  appears infinitely often. We say that such an  $n$  is an *infinite conditional mark* and write  $\text{InfCond}(P)$  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  $\Gamma \vdash P \blacktriangleright T$  and  $T$  is live and  $P \notin \text{Inf}$ . Then  $P$  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  $\text{if}^n e \text{ then } Q \text{ else } R$ , such that  $n \in \text{InfCond}(P)$ , with  $\text{if}^* e \text{ then } Q \text{ else } R$ . The reduction semantics of  $\text{if}^* e \text{ then } Q \text{ else } R$  is defined as follows:

$$[\text{IFT}^*] \text{if}^* e \text{ then } P \text{ else } Q \rightarrow P \quad [\text{IFE}^*] \text{if}^* e \text{ then } P \text{ else } Q \rightarrow Q$$

**Definition 5.7** (Alternating Conditionals). We say that  $P$  has *alternating conditional branches*, written  $P \in \text{AC}$ , iff  $P \in \text{Inf}$  and if  $P \rightarrow^* (\nu \tilde{c})Q$  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  $\text{InfCond}(P)$  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  $P^* \in \text{Inf}$  then  $P^* \in \text{AC}$ ; (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  $P$  is well-typed with some live type  $T$  and  $P \in \text{AC}$  holds, then  $P$  must itself be live.

**Theorem 5.3** (Liveness). *Suppose  $\Gamma \vdash P \blacktriangleright T$  and  $T$  is live and  $P \in \text{AC}$ . Then  $P$  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 admits more live programs than synchronous communication. Consider the following example:

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

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,  $\text{newchan}(y:\sigma, n); P$ . 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  $c^*\langle\sigma, n\rangle::\tilde{v}$  (also replacing  $c\langle\sigma\rangle::\emptyset$  and  $c^*\langle\sigma\rangle::\tilde{v}$  by  $c\langle\sigma, 0\rangle::\emptyset$  and  $c^*\langle\sigma, n\rangle::\tilde{v}$  for synchronous channels). We add the reduction rules for asynchronous communication:

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

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

In all other rules that use buffers we add the buffer bound straightforwardly. The type system is fundamentally unchanged, now accounting for buffer bounds:

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

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

In contrast with the types and rules in Figure 4,  $(\text{new}^n a)T$  and  $[a]$  are replaced by  $(\text{new}^n a)T$  and  $[a]_k^n$ , 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:

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

$$\frac{P \Downarrow_{\tilde{a}} \quad Q \Downarrow_{\bullet a}}{P \mid Q \Downarrow_{[a]}} \quad \frac{P \Downarrow_{a \bullet} \quad \pi_i \Downarrow_{a}}{P \mid \text{select}\{\pi_i; Q_i\}_{i \in I} \Downarrow_{[a]}}$$

The barbs for asynchronous types,  $T \Downarrow_o$ , which are identical to those of processes are in appendix [27].

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

$$\begin{array}{c}
|\text{NEW}| \frac{(\text{new}^n a)T \xrightarrow{\tau} (\nu a)(T \mid [a]_0^n)}{|\text{NEW}|} \quad |\text{BUF}| \frac{[a]_k^n \xrightarrow{\text{end}[a]} a^*}{|\text{BUF}|} \\
|\text{IN-B}| \frac{k < n}{[a]_k^n \xrightarrow{a} [a]_{k+1}^n} \quad |\text{OUT-B}| \frac{k \geq 1}{[a]_k^n \xrightarrow{a} [a]_{k-1}^n} \\
|\text{PUSH}| \frac{T \xrightarrow{\bar{a}} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{[a]} T' \mid S'} \quad |\text{POP}| \frac{T \xrightarrow{a} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{[a]} T' \mid S'}
\end{array}$$

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  $\mathbf{T}$  s.t.  $\text{Fenced}(\mathbf{T})$ , it is decidable whether or not  $\mathbf{T}$  is  $k$ -live (resp. channel safe), for any  $k \geq 0$ .*

**Theorem 6.2** (Channel Safety and Liveness in Types – Asynchrony). *For all  $\mathbf{T}$  s.t.  $\text{Fenced}(\mathbf{T})$ , there exists a finite  $k \geq 0$  such that if  $\mathbf{T}$  is  $k$ -live (resp.  $k$ -channel safe), then  $\mathbf{T}$  is live (resp. channel-safe).*

**Theorem 6.3** (Process Channel Safety and Liveness – Asynchrony). *Suppose  $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$ .*

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

With the revised semantics, the program

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

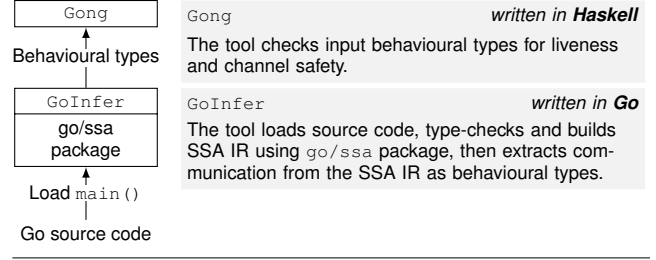
is correctly deemed as live, with the typing given by:

$$\{\mathbf{t}_P(x, y) = (\bar{x}; y \mid \bar{y}; x)\} \text{ in } (\text{new}^1 x)(\text{new}^1 y)\mathbf{t}_P\langle x, y \rangle$$

## 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 liveness and channel safety of the types. An outline of our verification tool chain is shown in Figure 8.

**Type Inference** Our type inference tool `GoInfer` is written in Go, using the `go/ssa`<sup>1</sup> 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  $\mathbf{T}$  by converting each SSA block into an individual type equation. The analysis and conversions are context-sensitive, for example, channels created in different



**Figure 8.** Workflow of our verification tool chain.

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  $\mathbf{T}$  representing a Go program’s concurrent behaviour and performs liveness and channel safety checks. Our representation of  $\mathbf{T}$  makes use of the `unbound` package [46] to deal with the binding structure of types. First, it checks if  $\mathbf{T}$  is fenced. If so, we generate all  $\rightarrow_k$ -reachable terms, where  $k$  is heuristically-computed. Finally, each of these terms are checked for liveness and 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 [33], on concurrent programs adapted from [17, 28], and open-source Go programs from developer guides [36, 38] and GitHub [1].

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 [33] to demonstrate the extra expressiveness of our approach. If a program is “Static”, it has no dynamic spawning of goroutines (a requirement for the usage of the tool of [33]).

`forselect` is a pattern described in [37] 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 [33, 40]. The work [33] extracts *Communicating Finite State Ma-*

<sup>1</sup><http://golang.org/x/tools/go/ssa>

**Table 1.** Go programs verified by our tool chain.

Examples	# chans					Analysis Time (ms)	[33]	
	LoC	unbuf.	buf.	Live	Safe		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]	56	3	0	✓	✓	646921.76	×	
jobsched	41	0	1	✓	✓	48.12	×	
concsys [1]	112	2	0	×	✓	323.75	×	
fanin [33, 38]	36	3	0	✓	✓	89.14	✓	✓
fanin-alt [33]	37	3	0	×	✓	209.02	✓	✓
mismatch [33]	26	2	0	×	✓	26.59	✓	×
fixed [33]	25	2	0	✓	✓	24.58	✓	✓
alt-bit [30]	74	0	2	✓	✓	405.78	✓	✓
forselect	40	3	0	✓	✓	31.01	✓	✓
cond-recur	32	2	0	✓	✓	34.08	✓	✓

<sup>1</sup>: testing for channel close state is not supported in this version

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

*chines* [6] whose representation corresponds to *session types* [19, 41] from Go source code, and synthesises from them a global choreography using the tool developed in [28]. 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 [20] 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 [33].

The work of [40] 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 significantly limited when compared to ours, in both theory and practice: (1) their approach relies on regular expressions, hence they cannot analyse programs with recursion appearing under parallel compositions. This prevents their work from being able to verify our main examples (see Table 1 in § 7.1); (2) they do not support asynchrony (buffered channels), closing of channels, or select with non-trivial case bodies; and (3) conditionals can easily generate false positives in their analysis. For instance, the following program which is verified as live in our approach (this program belongs to  $\text{May}\downarrow$  in our theory) is judged as a deadlock in their approach (implemented as `cond-recur` in Table 1):

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

and (4) 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 [21] for a broad survey) have been developed for general concurrent program analyses [22], such as deadlock-freedom [17, 24], lock-freedom [25, 35] and resource usages [26].

All of the behavioural type systems above rely on some form of dependency tracking mechanisms to ensure the correct interplay of communication actions. A limitation of this class of techniques is

that subtle changes in channel usage (that may not have a significant effect on a program’s outcome) can produce dramatically different analysis outcomes (see discussion in [35]). Our approach based on fencing is better suited in covering a wider range of Go programs (and other channel-based programming languages) since it only limits names in recursive call sites but does not depend on the ordering of communications or circularities of channels, provided the programs are in one of the classes of Section 5.

The work [17] 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 [24] and [35]. Their approach uses an extension of the typing system of [24] 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 [24] and [35].

However, our prime sieve example is deemed “deadlock” by the analysis of [17] (and subsequently [24] and [35]). Intuitively, their analysis detects a circularity in the definition of  $r(x)$  due to the mutual dependencies between the parameter  $x$  and the bound name  $b$ , despite the fact that the dependencies never actually generate a deadlock. Note that this is not due to the output in the filter process depending on the outcome of the conditional test ([23] and [25] study type-based techniques for potentially data-driven information flow and termination-checking, respectively). Even if we remove the conditional branch, changing the type of  $f(x, y)$  to  $x; \bar{y}; f\langle x, y \rangle$ , their analysis judges the program as “deadlock”. Our analysis deems both the original and this revised version of the prime sieve as live. All of the deadlock-free processes in [17] are also identified as live by our analysis, modulo some minor syntactic rearrangements due to fencing (Table 1 in § 7.1). To the best of our knowledge, no implementation of their analysis exists. In addition, their deadlock-freedom property does not match with our notion of liveness (which is essentially lock freedom in [25, 35]). For instance, their analysis accepts the program  $Fib_{bad}$  in § 2.1.2 as a deadlock-free process since a program that loops non-productively is deadlock-free in their definition.

The work of [35] studies an extension of [24, 25] that ensures deadlock freedom and lock freedom of the linear  $\pi$ -calculus, with a form of channel polymorphism. By relying on linearity, the system in [35] immediately rules out many examples that are captured by our work. The *factorial* and *diningphilosopher* examples of [17] denote patterns that are untypable in [35], but can be verified in our tool. Our tool can also verify programs morally equivalent to most examples discussed in [35], 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 [19]) or more (multiparty [20]) 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, 14]. 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, 45] 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 [35]. However, to ensure logical consistency general recursion is disallowed. In the presence of general recursion [43], progress is weakened; ensuring all typable processes are deadlock-free but not necessarily lock-free. The work of [44] 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 [34] 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 communication is given. Our fencing predicate allows for potentially unbounded processes and channels, and our analysis ensures liveness and safety properties.

**Conclusion and Future Work** Since the early 1990s, behavioural type theories which formalise “types as concurrent processes” [22] have been studied actively in models for concurrency [21]. 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 symbolic 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).

## Acknowledgments

We thank Naoki Kobayashi for finding a flaw in an earlier version of Section 5.3. We also would like to thank Elena Giachino and Raymond Hu for fruitful discussions on this work, as well as the anonymous referees for their comments. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1 and EP/N027833/1; and by EU FP7 612985 (UPSCALE).

## 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. Gabbriellini, and G. Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP’03*, pages 133–144, 2003.
- [8] N. Busi, M. Gabbriellini, 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] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL’02*, pages 45–57, 2002.
- [13] 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.
- [14] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 26(2):238–302, 2016.
- [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] B. Fitzpatrick. go 1.5.1 linux/amd64 deadlock detection failed, 2015. <https://github.com/golang/go/issues/12734#issuecomment-142859447>.
- [17] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *CONCUR*, volume 8704 of *LNCS*, pages 63–77. Springer, 2014.
- [18] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [19] 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.
- [20] 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.
- [21] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniérou, 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.
- [22] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
- [23] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.
- [24] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR’06*, volume 4137 of *LNCS*, pages 233–247, 2006.
- [25] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *TOPLAS*, 32(5):16:1–16:49, May 2008.
- [26] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the p-calculus. *Logical Methods in Computer Science*, 2(3), 2006.
- [27] J. Lange, N. Ng, B. Toninho, and N. Yoshida. Full version of this paper. Available at <https://arxiv.org/abs/1610.08843>.

- [28] 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.
- [29] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, 1980.
- [30] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [31] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *ICALP*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.
- [32] 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.
- [33] N. Ng and N. Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CC 2016*, pages 174–184. ACM, 2016.
- [34] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL '94*, pages 84–97. ACM, 1994.
- [35] L. Padovani. Deadlock and Lock Freedom in the Linear  $\pi$ -Calculus. In T. A. Henzinger and D. Miller, editors, *CSL-LICS'14*, pages 72:1–72:10. ACM Press, 2014.
- [36] Rob Pike. Go Concurrency Patterns, 2012. <https://talks.golang.org/2012/concurrency.slide>.
- [37] Sameer Ajmani. Advanced Go Concurrency Patterns, 2013. <https://talk.golang.org/2013/advconc.slide>.
- [38] Sameer Ajmani. Go Concurrency Patterns: Pipelines and cancellation, 2014. <https://blog.golang.org/pipelines>.
- [39] D. Sangiorgi and D. Walker. *The  $\pi$ -Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- [40] K. Stadtmüller, M. Sulzmann, and P. Thiemann. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*, 2016. to appear.
- [41] 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.
- [42] The Go Authors. Effective Go. [https://golang.org/doc/effective\\_go.html](https://golang.org/doc/effective_go.html).
- [43] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP'13*, pages 350–369, 2013.
- [44] B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC'14*, pages 159–175, 2014.
- [45] P. Wadler. Proposition as Sessions. In *ICFP'12*, pages 273–286, 2012.
- [46] S. Weirich and B. Yorgey. Unbound library. <https://hackage.haskell.org/package/unbound>.
- [47] 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>.