

Concurrent Programming Languages

Channel-based Concurrency Module
Lecture 4: Applied Research Topics

19 November 2020

**MIEI - Integrated Masters in Comp. Science and Informatics
Specialization Block**

Bernardo Toninho

(with António Ravara and Carla Ferreira)



NOVALINCS

Last Lecture

- High-level coordination patterns:
 - Contexts
 - Request Replication
 - Worker Pools
 - Pipelines
 - Rate Limits

Today

- Last lecture (of the module!)
- Something a bit different:
 - Concurrency Research
 - Concurrency + Programming Languages
 - Types for Concurrency

Disclaimer

This is my particular, mildly opinionated take
on these topics.

It is **not** an exhaustive survey of the field.

But hopefully it will peak your interest...

Concurrency Research

- CS research in concurrency comes from a few different communities:
 - Systems
 - Theory / Algorithms
 - Verification / Formal methods
 - Programming Languages

A Systems Approach

- Developing and implementing new / better techniques and approaches to concurrent/distributed systems.
- Strong emphasis on performance, scale and scalability (quantitative).
- Examples from recent top venues:
 - A Scalable Off-Heap Allocated Key-Value Map
 - Wait-Free Universal Construct for Large Objects
 - Delegation Sketch: a Parallel Design with Support for Fast and Accurate Concurrent Operations
 - Avoiding Scheduler Subversion using Scheduler-Cooperative Locks
 - Revisiting Broadcast Algorithms for Wireless Edge Networks

A Theory Approach

- Developing new algorithms, complexity-theoretic models and establishing tighter asymptotic bounds.
- Many different models for “computation” (congested clique model, local model) with many variations (synchronous, asynchronous, fault models, topological properties).
- (Usually) not about implementation:
 - Synchronous Byzantine Lattice Agreement in $O(\log(f))$ Rounds
 - Almost-surely Terminating Asynchronous Byzantine Agreement Protocols with a Constant Expected Running Time
 - Exact Consensus under Global Asymmetric Byzantine Links
 - Silence
 - An $O(\log^{3/2} n)$ Parallel Time Population Protocol for Majority with $O(\log n)$ States

A Verification Approach

- Studying verification and analysis techniques for concurrent and distributed computation models (not quite program verification, despite the name).
- Not the same models as in “theory”, but typically quite abstract.
- Main techniques: Model checking, games, automata (**many** different kinds), algebras and co-algebras
 - Scalable Termination Detection for Distributed Actor Systems
 - Verification of Flat FIFO Systems
 - Reachability for Bounded Branching VASS
 - Partially Observable Concurrent Kleene Algebra
 - Elimination of Quantifiers and Undecidability in Spatial Logics for Concurrency

A PL Approach

- A combination of all of the above! PL research is a very broad and rich area (no bias whatsoever), even if we just zoom in on concurrency.
- On a spectrum between applied and foundational (usually corresponds to more systems-y or more verification-y).
- Typically targets some “minimal” model of a programming language (e.g. λ -calculus/PCF for functional languages; Featherweight Java/Scala for OO; π -calculus for channel-based concurrency).
- Main techniques: Type systems, static analysis, runtime instrumentation/checking, program logics.
 - Fast, Sound, and Effectively Complete Dynamic Race Prediction
 - Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation
 - A Separation Logic for Concurrent Randomized Programs
 - A Static Verification Framework for Message Passing in Go using Behavioural Types

A PL Approach

- Its not just about coming up with new programming languages.
- In fact, mostly **not** about that.
- About new techniques that make “programming better”:
 - Stronger / more precise / expressive type systems.
 - Analyses to rule out or flag certain “bad” programs.
 - Logics for program reasoning.

Sidenote

- Rust did not invent ownership types (Clarke, Potter, Noble in OOPSLA'98)
- Ownership + concurrency also not invented / unique to Rust.
- Most new PL features of “today” were invented 20+ years ago in academia.

How do “fancy types” help with concurrency?

- You’ve already seen the interaction between ownership types for mem. management and concurrency in Rust.
- Ownership types are about managing aliasing. So are data races.... good match!
- Ownership types are a form of so-called *affine types*.
- What about channels? Does “usage control” help?

How do “fancy types” help with concurrency?

- What can go wrong with channel-based concurrency?
Deadlocks!
- Usage control of channels and deadlocks — deadlocks arise from a mismatch in channel usage by peers!
- A lot of PL work on channel-based concurrency has been devoted to types (and related analyses) to prevent deadlocks.

Simple Types for Channels

- In their simplest form: channel types specify types of payloads (e.g. as found in Go).
- Prevents certain communication errors (e.g. expect an int, get a string).

Simple Types for Channels

- In their simplest form: channel types specify types of payloads (e.g. as found in Go).
- Prevents certain communication errors (e.g. expect an int, get a string).
- Doesn't prevent deadlocks or "orphan messages" (in asynchrony, sent messages may not be received).

Channel I/O Types [San98]

- Distinguish channel input and output capabilities:

`chan<- int` vs `<-chan int` vs `chan int`

- Go **does not** have I/O types, exactly.
- I/O types are governed by subtyping (e.g., `chan<-int ≤ chan int`).
- Different threads may have different types for a given channel.

Channel I/O Types [San98]

- Distinguish channel input and output capabilities:

`chan<- int` vs `<-chan int` vs `chan int`

- Go **does not** have I/O types, exactly.
- I/O types are governed by subtyping (e.g., `chan<- int` \leq `chan int`).
- Different threads may have different types for a given channel.
- Provides a more fine-grained control of channel usage but... still doesn't help much.

Advanced Types for Channels

- Ownership types in Rust control the number of times a variable is “used”:

```
let s1 = String::from("hello");  
let s2 = s1;
```

```
println!("{}", s1);
```

```
let s1 = String::from("hello");  
let s2 = &s1;
```

```
println!("{}", s1);
```

- Lets explore a similar idea but for channels.

Linear Types for Channels [KPT99]

- Control the number of times a resource (i.e. a channel) is used.
- A channel **capability** of linear type must be used **exactly once**.

```
func f(c <-lchan int) {  
    c <- 1  
    c <- 2  
}  
func g(c lchan int) {  
    c <- 1 //Bad!  
}
```

Linear Types for Channels [KPT99]

- Control the number of times a resource (i.e. a channel) is used.
- A channel capability of linear type must be used **exactly once**.

```
func f(c <-lchan int) {  
    c <- 1  
    c <- 2  
}  
func g(c lchan int) {  
    c <- 1 //Bad!  
}
```

```
func h(c, d <-lchan int) {  
    c <- 1 //Bad! 'd' not used.  
}  
func i(c lchan int) {  
    go (c <- 1) //out cap. used  
    fmt.Println(<-c)  
}
```

Linear Types for Channels [KPT99]

- Control the number of times a resource (i.e. a channel) is used.
- A channel capability of linear type must be used **exactly once**.
- Type system expresses **obligations** on linear channels.
- Certain bad behaviors are ruled out by typing.
- Can linear channels used concurrently deadlock?

Linear Types for Channels [KPT99]

- Certain bad behaviors are ruled out by typing.
- Can linear channels used concurrently deadlock?

```
func f(c,d <-lchan int) {  
    c <- 1  
    d <- 2  
}  
func g(c,d lchan<- int) {  
    <- d  
    <- c  
}
```

```
func main() {  
    c := make(lchan int)  
    d := make(lchan int)  
    go f(c,d)  
    go g(c,d)  
    ...  
}
```

Linear Types for Channels [KPT99]

- Certain bad behaviors are ruled out by typing.
- Can linear channels used concurrently deadlock?

```
func f(c,d <-lchan int) {  
    c <- 1  
    d <- 2  
}  
func g(c,d lchan<- int) {  
    <- d  
    <- c  
}
```

```
func main() {  
    c := make(lchan int)  
    d := make(lchan int)  
    go f(c,d)  
    go g(c,d)  
    ...  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
    // c:!int;?int;End  
    c <- 23 //c:?int;End  
    fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
    // c:?int;!int;End  
    <- c // c:!int;End  
    c <- 42 // c:End  
}
```


Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Channel types as protocol descriptions.
- Takes advantage of duality of input and output

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Session types are linear and “stateful”.
- Duality ensures compatibility of endpoints.
- Linearity ensures actions must take place in the right order.

```
type T = !int;?int;End
```

```
func f(c schan T) {  
  // c:!int;?int;End  
  c <- 23 //c:?int;End  
  fmt.Println(<-c) //c:End  
}
```

```
dual(T) = ?int;!int;End
```

```
func g(c schan dual(T)) {  
  // c:?int;!int;End  
  <- c // c:!int;End  
  c <- 42 // c:End  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
type T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
    for {  
        case c of  
            Add: n := <-c; m := <- c; c<- n+m  
            Sym: n := <-c; c <- -1*n  
            Quit: break  
    }  
}}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
stype T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```


Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
stype T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
type T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
stype T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
stype T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```

Session Types [HVK98]

- Channel types specify a **sequence** of interactions.
- Also allows for a form of **labelled choice**.
- Types denote finite-state automata of **behaviors** (CFSM).

```
stype T = Choice{Add: ?int;?int;!int;T,  
                  Sym: ?int;!int;T, Quit:End}  
func f(c schan T) {  
  for {  
    case c of  
      Add: n := <-c; m := <- c; c<- n+m  
      Sym: n := <-c; c <- -1*n  
      Quit: break  
    }  
  }  
}
```

Session Types [HVK98]

- What about deadlocks?
- If two threads use **a single** session channel **dually**, no deadlocks!

Session Types [HVK98]

- If two threads use **a single** (no higher-order channels) session channel **dually**, no deadlocks! (Theorem)
- Great but... a bit weak / restrictive.
- Active area of research:
 - Behavioral Types for deadlock-freedom (e.g. [Koba02,IK04,KS08,CV09,Pado14,BTP19,LP19, etc.])
 - Multiparty Session Types [HYC08, etc]
 - Behavioral Types + Model-Checking [CRR02,LNTY17, etc.]
 - “Logical” session types [CP10,TCP13,LM16,BTP18,DP20,etc.]

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Global types specify the conversation from a global perspective:

$B \rightarrow S : \text{ItemId.}$

$S \rightarrow B : \text{Quote.}$

$B \rightarrow S : \{ \text{Ok} : S \rightarrow \text{Sh} : \{ \text{Ok} : S \rightarrow \text{Sh} : \text{Address.}$

$\text{Sh} \rightarrow S : \text{Receipt.}$

$S \rightarrow B : \text{Receipt.}$

end,

$\text{Quit} : \dots \}$

$\text{Quit} : \dots \}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Global types specify the conversation from a global perspective:

$B \rightarrow S : \text{ItemId.}$

$S \rightarrow B : \text{Quote.}$

$B \rightarrow S : \{Ok : S \rightarrow Sh : \{Ok : S \rightarrow Sh : \text{Address.}$

$Sh \rightarrow S : \text{Receipt.}$

$S \rightarrow B : \text{Receipt.}$

end,

$Quit : \dots \}$

$Quit : \dots \}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Global types specify the conversation from a global perspective:

$B \rightarrow S : \text{ItemId.}$

$S \rightarrow B : \text{Quote.}$

$B \rightarrow S : \{Ok : S \rightarrow Sh : \{Ok : S \rightarrow Sh : \text{Address.}$

$Sh \rightarrow S : \text{Receipt.}$

$S \rightarrow B : \text{Receipt.}$

end,

$Quit : \dots \}$

$Quit : \dots \}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Global types specify the conversation from a global perspective:

$B \rightarrow S : \text{ItemId.}$

$S \rightarrow B : \text{Quote.}$

$B \rightarrow S : \{Ok : S \rightarrow Sh : \{Ok : S \rightarrow Sh : \text{Address.}$

$Sh \rightarrow S : \text{Receipt.}$

$S \rightarrow B : \text{Receipt.}$

end,

Quit : ... }

Quit : ... }

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Global types specify the conversation from a global perspective:

$B \rightarrow S : \text{ItemId.}$

$S \rightarrow B : \text{Quote.}$

$B \rightarrow S : \{Ok : S \rightarrow Sh : \{Ok : S \rightarrow Sh : \text{Address.}$

$Sh \rightarrow S : \text{Receipt.}$

$S \rightarrow B : \text{Receipt.}$

end,

$Quit : \dots \}$

$Quit : \dots \}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Endpoint types are algorithmically derived from global types:

$$G \upharpoonright B = S!(\text{ItemId}); S?(Quote); S!\{Ok : S?(Receipt).\text{end}, Quit : \dots\}$$
$$G \upharpoonright S = B?(ItemId); B!(Quote); B?\{Ok : Sh!\{Ok : Sh!(Address); Sh?(Receipt); B!(Receipt), Quit : \dots\},$$
$$G \upharpoonright Sh = S?\{Ok : S?(Address); S!(Receipt); \text{end}, Quit : \dots\}$$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Endpoint types are algorithmically derived from global types:

$G \upharpoonright B = S!(\text{ItemId}); S?(Quote); S!\{Ok : S?(Receipt).\text{end}, Quit : \dots\}$

$G \upharpoonright S = B?(ItemId); B!(Quote); B?\{Ok : Sh!\{Ok : Sh!(Address); Sh?(Receipt); B!(Receipt), Quit : \dots\},$
 $Quit : \dots\}$

$G \upharpoonright Sh = S?\{Ok : S?(Address); S!(Receipt); \text{end}, Quit : \dots\}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Endpoint types are algorithmically derived from global types:

$G \upharpoonright B = S!(\text{ItemId}); S?(Quote); S!\{Ok : S?(Receipt).\text{end}, Quit : \dots\}$

$G \upharpoonright S = B?(ItemId); B!(Quote); B?\{Ok : Sh!\{Ok : Sh!(Address); Sh?(Receipt); B!(Receipt), Quit : \dots\},$
 $Quit : \dots\}$

$G \upharpoonright Sh = S?\{Ok : S?(Address); S!(Receipt); \text{end}, Quit : \dots\}$

Multiparty Session Types [HYC08]

- A framework for deadlock-free communication between many parties/endpoints using multiple channels.
- Generalizing **duality** (two endpoints) to **multiparty compatibility** (many endpoints).
- Endpoint types are algorithmically derived from global types:

$G \upharpoonright B = S!(\text{ItemId}); S?(Quote); S!\{Ok : S?(Receipt).\text{end}, Quit : \dots\}$

$G \upharpoonright S = B?(ItemId); B!(Quote); B?\{Ok : Sh!\{Ok : Sh!(Address); Sh?(Receipt); B!(Receipt), Quit : \dots\},$
 $Quit : \dots\}$

$G \upharpoonright Sh = S?\{Ok : S?(Address); S!(Receipt); \text{end}, Quit : \dots\}$

Multiparty Session Types [HYC08]

$$G \upharpoonright B = S!(\text{ItemId}); S?(\text{Quote}); S!\{Ok : S?(\text{Receipt}).\text{end}, \text{Quit} : \dots\}$$
$$G \upharpoonright S = B?(\text{ItemId}); B!(\text{Quote}); B?\{Ok : Sh!\{Ok : Sh!(\text{Address}); Sh?(\text{Receipt}); B!(\text{Receipt}), \text{Quit} : \dots\}, \\ \text{Quit} : \dots\}$$
$$G \upharpoonright Sh = S?\{Ok : S?(\text{Address}); S!(\text{Receipt}); \text{end}, \text{Quit} : \dots\}$$

- Global types are essentially message sequence charts.
- For “well-formed” global types, if endpoints adhere to endpoint types, deadlock-freedom guaranteed.
- No “orphan messages”.
- Ongoing research on relaxing “well-formedness” / increasing expressiveness.

Model-Checking Types [CRR02]

- What is model-checking? [Emerson et al. 80,86, ...]
 - An automated method for verifying if (concurrent) finite-state systems satisfy a given *temporal* property.
 - Finite-state systems modeled as ~finite-state automata (more precisely, Kripke structures).
 - What are temporal properties? Formulas in Linear Temporal Logic (LTL).
 - $M \models \varphi$ (i.e., system satisfies formula) is decidable.

Model-Checking Types [CRR02]

- What is LTL? [Pnueli77]
 - Propositional Logic ($A \wedge B, \neg A$) +
 - “In the ne**X**t state, A is true” (**X** A)
 - “A is true **U**ntil B becomes true” (A **U** B)
 - “A is **G**lobally (always) true” (**G** A)
 - “A is true at some point in the **F**uture” (**F** A)

Model-Checking Types [CRR02]

- Safety properties (“something bad won’t happen”)
 - **G** $\neg(\text{reactor_temp} > 1000)$
 - **G** $\neg(\text{crit_region1} \wedge \text{crit_region2})$
- Liveness properties (“something good will happen”)
 - **G** (sending \Rightarrow **F** received)
 - **F** ($x > 5$)
- Fairness (“something good always will happen in the future”):
 - **G** (**F** crit_region)

Model-Checking Types [CRR02]

- What is model-checking good for?
 - Provide a **model** of your system as a finite state “automata”.
 - Provide a **specification** in the form of an LTL formula.
 - Model-checking can decide whether the model satisfies the spec., and if it doesn't, can provide a counter-example.
 - Variants exist with “richer” models (process models) and slightly richer logics (modal μ -calculus — can talk about state changes via actions).
 - Sounds great but... model-checking LTL is **PSPACE**-Complete, μ -calculus is **PSPACE**-Complete. But is still usable in practice! (lots of work to make it so).

Model-Checking Types [CRR02]

- What does it have to do with types?

```
 $G \upharpoonright B = S!(\text{ItemId});$   
 $S?(\text{Quote});$   
 $S!\{Ok : S?(\text{Receipt}).\text{end},$   
 $\quad \text{Quit} : \dots\}$ 
```

```
func Buyer(sellerChan schan TBuyerSeller) {  
    //Determining item logic..  
    sellerChan <- itemId  
    quote := <- sellerChan  
    if quote < ... {  
        sellerChan <- Ok  
        receipt := <- selerChan  
    } else  
        //Quit logic  
}
```

Model-Checking Types [CRR02]

- In this “world”, types for endpoints are very descriptive:

```
 $G \upharpoonright B =$  S!(ItemId);  
S?(Quote);  
S!{Ok : S?(Receipt).end,  
    Quit : ... }
```

```
func Buyer(sellerChan schan TBuyerSeller) {  
    //Determining item logic..  
    sellerChan <- itemId  
    quote := <- sellerChan  
    if quote < ... {  
        sellerChan <- Ok  
        receipt := <- selerChan  
    } else  
        //Quit logic  
}
```

Model-Checking Types [CRR02]

$G \upharpoonright B =$ S!(ItemId);
S?(Quote);
S!{*Ok* : S?(Receipt).end,
 Quit : ... }

```
func Buyer(sellerChan schan TBuyerSeller) {  
    //Determining item logic..  
    sellerChan <- itemId  
    quote := <- sellerChan  
    if quote < ... {  
        sellerChan <- Ok  
        receipt := <- selerChan  
    } else  
        //Quit logic  
}
```

- Such rich types can reasonably be used as **models**.

Model-Checking Types [CRR02]

$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(Quote); \\ S!\{Ok : S?(Receipt).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$	$G \upharpoonright S = \begin{array}{l} B?(ItemId); \\ B!(Quote); \\ B!\{Ok : Sh!(Address); \\ \quad Sh?(Receipt); \\ \quad B!(Receipt), \\ \quad \text{Quit} : \dots\} \end{array}$	$G \upharpoonright Sh = \begin{array}{l} S!\{Ok : S?(Address); \\ \quad S!(Receipt); \\ \quad \text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$
--	--	---

- We can engineer formulae that denote properties of interest:
 - $G(\langle S! \rangle T \Rightarrow F(\langle S! ? \rangle T))$ “Eventual reception for S”
 - $G(\langle S! \rangle T \vee \langle S? \rangle T \vee \langle B! \rangle T \vee \dots) \Rightarrow \langle * \rangle T$ “No global deadlocks”
 - Can also do “no partial deadlocks”, but its a bit verbose...
- Useful if we can extract/infer such types from code, but has limitations wrt data dependent behaviors and termination.

This is all great but...
how about in practice?

Session Types in Practice

- Session types rely on linear typing, which is absent from most general purpose languages.
- Without linearity, compile-time correctness is compromised.
- **Options:**
 - Forego linearity, relying on dynamic checks.
 - Extend the host language's type system.
 - Encode linearity in the host language's type system in some way.

Session Types in Practice

- Foregoing linearity:
- We still want to provide correctness guarantees.
- **Idea:** Encode the session behavior using the language's type structure

Session Types in Practice

$$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(Quote); \\ S!\{Ok : S?(Receipt).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$$

Session Types in Practice

$$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(Quote); \\ S!\{Ok : S?(Receipt).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$$

```
type BState1 struct {  
    SellerChan chan interface{}  
}  
type BState2 struct { ... }  
type BState3 struct { ... }
```

Session Types in Practice

$$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(\text{Quote}); \\ S!\{Ok : S?(\text{Receipt}).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$$

```
type BState1 struct {  
    SellerChan chan interface{  
}
```

```
type BState2 struct { ... }
```

```
...
```

```
func (b *BState1) SendItemIdToS(itemId int) *BState2 {  
    b.SellerChan <- itemId  
    return &BState2{b.SellerChan}  
}
```

Session Types in Practice

```
type BState1 struct {
    SellerChan chan interface{
}
type BState2 struct { ... }
...
func (b *BState1) SendItemIdToS(itemId int) *BState2 {
    b.SellerChan <- itemId
    return &BState2{b.SellerChan}
}
func (b *BState2) RecvQuoteFromS() (*Quote, *BState3) {
    quote := <- b.SellerChan
    return quote.(*Quote) , &BState3{b.SellerChan}
}
```

$G \upharpoonright B = S!(\text{ItemId});$
 $S?(Quote);$
 $S!\{Ok : S?(Receipt).end,$
 $Quit : \dots\}$

Session Types in Practice

$$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(\text{Quote}); \\ S!\{Ok : S?(\text{Receipt}).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$$

```
func BeginSession() (*SState1,*BState1,*ShState1) {  
    Bands := make(chan interface{})  
    SandSh := make(chan interface{})  
    return &SState1{Bands,SandSh},  
        &BState1{Bands},  
        &ShState1{SandSh}  
}
```

Session Types in Practice

$$G \upharpoonright B = \begin{array}{l} S!(\text{ItemId}); \\ S?(Quote); \\ S!\{Ok : S?(Receipt).\text{end}, \\ \quad \text{Quit} : \dots\} \end{array}$$

```
func BeginSession() (*SState1,*BState1,*ShState1) {  
    Bands := make(chan interface{})  
    SandSh := make(chan interface{})  
    return &SState1{Bands,SandSh},  
        &BState1{Bands},  
        &ShState1{SandSh}  
}
```

Session Types in Practice

```
func BeginSession() (*SState1,*BState1,*ShState1) {
    Bands := make(chan interface{})
    SandSh := make(chan interface{})
    return &SState1{Bands,SandSh},
        &BState1{Bands},
        &ShState1{SandSh}
}

func Buyer(b *BState1, threshold float32) *Receipt {
    q, b := b.SendItemIdToS(93).RecvQuoteFromS()
    if (q.price < threshold) {
        b := b.ChooseOkToS()
        receipt, b := b.RecvReceiptFromS(); b.EndSession();
        return receipt
    } else {
        b := b.ChooseQuitToS()
        ...
    }
}
```

$G \upharpoonright B = S!(\text{ItemId});$
 $S?(\text{Quote});$
 $S!\{Ok : S?(\text{Receipt}).\text{end},$
 $Quit : \dots\}$

Session Types in Practice

- Encode session states as Go structs.
- Possible actions as available methods.
- Actions always produce the corresponding next state (fluent API).
- API “enforces” the state change, but some programmer cooperation is required...
- What about linearity?

Session Types in Practice

```
type BState1 struct {  
    SellerChan chan interface{}  
    used boolean  
}  
type BState2 struct { ... }  
...  
func (b *BState1) SendItemIdToS(itemId int) *BState2 {  
    if b.used { panic() }  
    else {  
        b.used = true  
        b.SellerChan <- itemId  
        return &BState2{b.SellerChan}  
    }  
}
```

Session Types in Practice

- What about linearity?
 - Can dynamically enforce that each state is used at most once.
 - Cannot enforce that states are used, but, if used then protocol is followed.
- Programming this encoding by hand is rather tedious so....

Session Types in Practice

[CHJNY19]

```
1 global protocol Pget(role M, role F, role S) {
2   Head from F[1] to S; Res from S to F[1]; // (1) Obtain metadata from Server
3   Meta from F[1] to M; Job from M to F[1,K]; // (2) Allocate Fetcher download tasks
4   Get from F[1,K] to S; Res from S to F[1,K]; // (3) Perform downloads
5   Data from F[1,K] to M; Sync@A from F[1,K] to M; // (4) Gather data and control channels
6 } // Sync@A is the local type projection of Sync onto A, i.e., a delegation
7 global protocol Sync(role A, role B) { choice at A { Done from A to B; } // Choice: terminate B (i.e., Fi) or ...
8                                     or { ... } }
```

- Roles are parameterized to allow for more explicit representation of concurrent topologies.
- If protocol is “well-formed”, generate API.

Session Types in Practice

[CHJNY19]

State type (with nested peer/action types)

State	Peer(s)	I/O action
M_1	F_1	Receive
M_2	F_1toK	Scatter
M_3	F_1toK	Gather
M_4	F_1toK	GatherAndSpawn

Method name and signature (parameters, result type)

Message label/values, aux. functions	Successor
Meta(a *Meta)	*M_2
Job(a []Job)	*M_3
Data(a []Data)	*M_4
Sync_A(run func(*A_1) End_A)	End_M

```

1 func mainM(req HttpReq, K int) {
2   proto := Pget.New()
3   M := proto.M.Kgt1.New(K) // API for K>1
4   ss1 := shm.Listen(8888+1); defer ss1.close()
5   go mainF1(req, 8888+1)
6   M.F_1.Accept(ss1)
7   for i := 2; i <= K; i++ {
8     ssi := shm.Listen(8888+i); defer ssi.close()
9     go mainF_2toK(req, 8888+i)
10    M.F_2toK.Accept(i, ssi) // Supported by K>1 API
11    M.run(runM) // runM: func(*M_1) End_M
12 } }

```

```

14 func runM(m *M_1) End_M {
15   var meta Meta; var data Data
16   // F[1]?Meta. F[1,K]!Job. F[1,K]?Data. F[1,K]?Sync@A
17   return m.F_1 .Receive .Meta(&meta).
18             F_1toK.Scatter .Job(split(&meta)).
19             F_1toK.Reduce .Data(&data, agg).
20             F_1toK.GatherAndSpawn.Sync_A(runA)
21 }
22
23 func runA(a *A_1) End_A {
24   return a.B.Send.Done() // Just do Done, for brevity
25 }

```


Session Types in Practice

[CHJNY19]

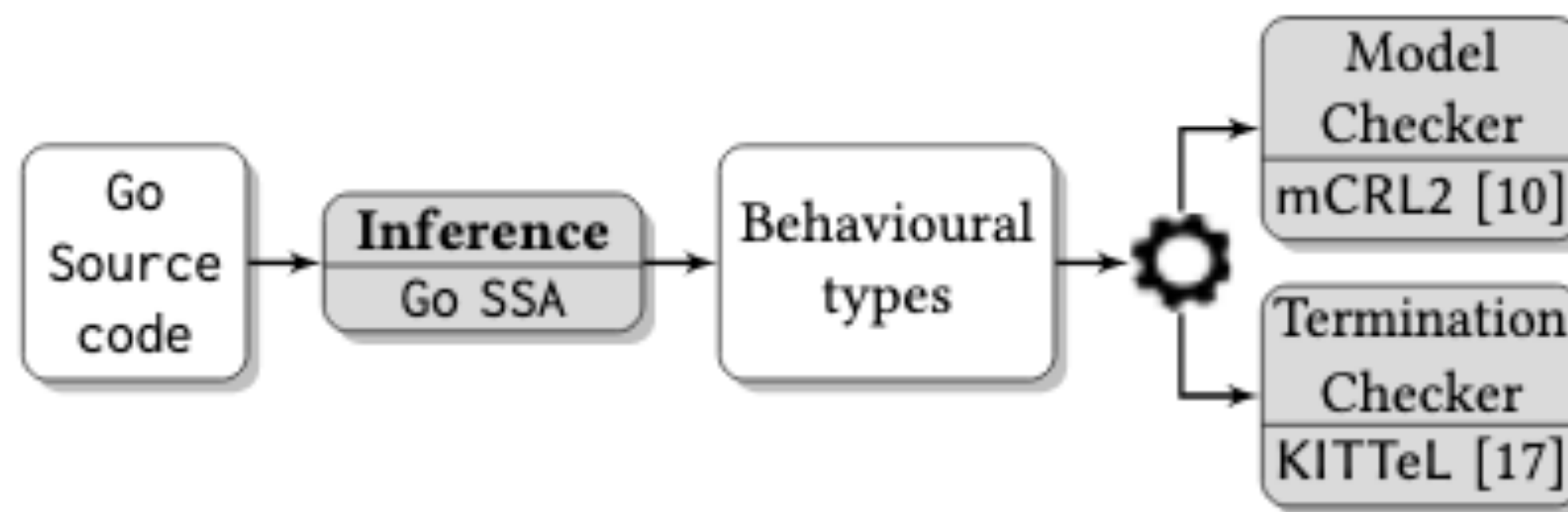
- Based on earlier work [HY17] for Java, which is more faithful to what was shown earlier (e.g., no parameterized roles).
- Communication substrate can be channels or actual sockets.
- Similar (but simpler) approaches exist for Scala [SDHY17], Python [DHHNY15] and F# [NHYA18].

Model Checking Go Types

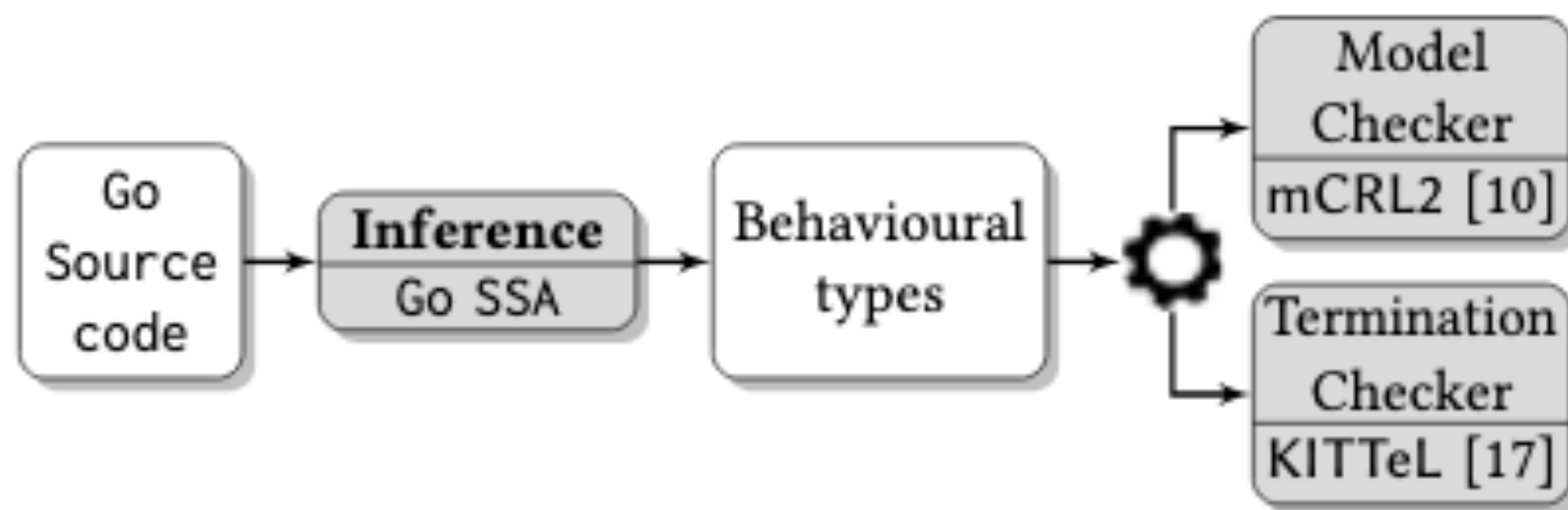
[LNTY17,LNTY18]

- Previous works presuppose you have the global specification and want to write the endpoints.
- Often we already have the program and want to verify its properties.
- Can we **extract** from a Go program a type-based abstraction (i.e. a model) and then verify it?

Model Checking Go Types

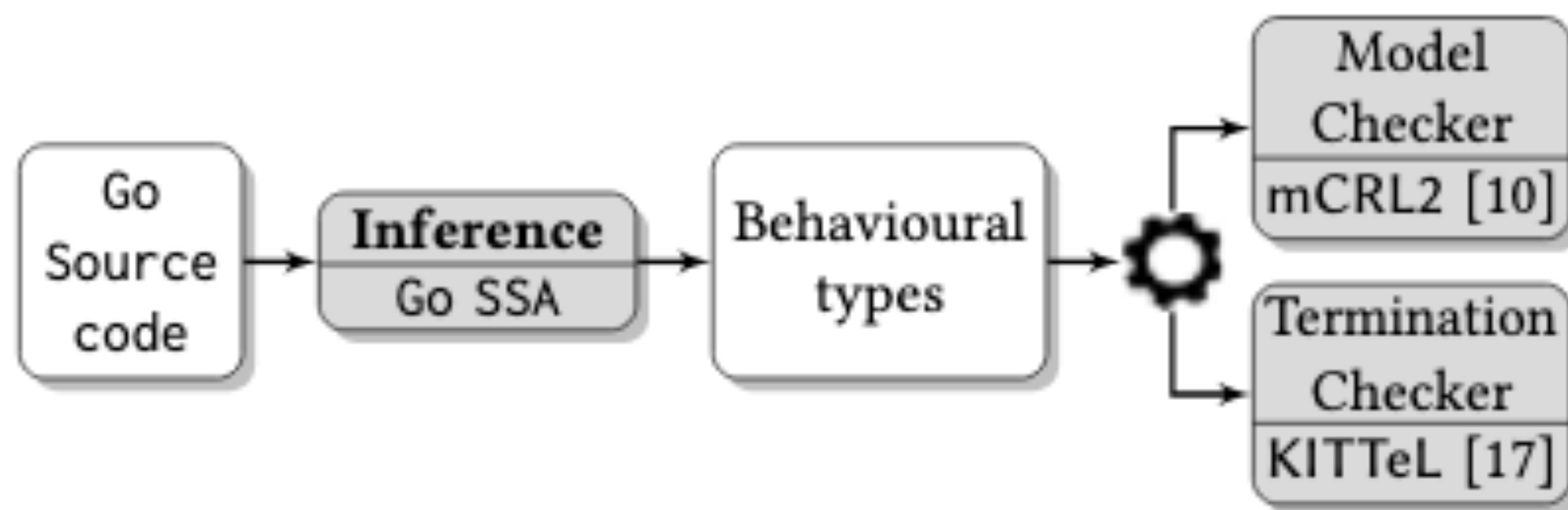


Model Checking Go Types



```
1 func prod(ch chan int) {
2     for i := 0; i < 5; i++ {
3         ch <- i // Send i to ch
4     }
5     close(ch) // No further values accepted at ch
6 }
7 func cons(ch1, ch2 chan int) {
8     for {
9         select {
10            case x := <-ch1: print(x) // Either input from ch1
11            case x := <-ch2: print(x) // or input from ch2
12        }
13    }
14 }
15 func main() {
16     ch1, ch2 := make(chan int), make(chan int)
17     go prod(ch1)
18     go prod(ch2)
19     cons(ch1, ch1)
20 }
```

Model Checking Go Types



```
1 func prod(ch chan int) {
2   for i := 0; i < 5; i++ {
3     ch <- i // Send i to ch
4   }
5   close(ch) // No further values accepted at ch
6 }
7 func cons(ch1, ch2 chan int) {
8   for {
9     select {
10    case x := <-ch1: print(x) // Either input from ch1
11    case x := <-ch2: print(x) // or input from ch2
12    }
13  }
14 }
15 func main() {
16   ch1, ch2 := make(chan int), make(chan int)
17   go prod(ch1)
18   go prod(ch2)
19   cons(ch1, ch1)
20 }
```

$$\{ \mathit{prod}(ch) = \overline{ch}; \mathit{prod}\langle ch \rangle \oplus \mathit{close}\ ch$$
$$\mathit{cons}(ch1, ch2) = \&\{ch1; \mathit{cons}\langle ch1, ch2 \rangle, ch2; \mathit{cons}\langle ch1, ch2 \rangle\}$$
$$\mathit{main}() = (\mathit{new}\ ch1, ch2); (\mathit{prod}\langle ch1 \rangle \mid \mathit{prod}\langle ch2 \rangle \mid \mathit{cons}\langle ch1, ch1 \rangle) \}$$
$$\text{in } \mathit{main}\langle \rangle$$

Model Checking Go Types

```

1  func prod(ch chan int) {
2      for i := 0; i < 5; i++ {
3          ch <- i // Send i to ch
4      }
5      close(ch) // No further values accepted at ch
6  }
7  func cons(ch1, ch2 chan int) {
8      for {
9          select {
10             case x := <-ch1: print(x) // Either input from ch1
11             case x := <-ch2: print(x) // or input from ch2
12          }
13      }
14  }
15  func main() {
16      ch1, ch2 := make(chan int), make(chan int)
17      go prod(ch1)
18      go prod(ch2)
19      cons(ch1, ch1)
20  }

```

$$\begin{aligned}
 &\{ \mathit{prod}(ch) = \overline{ch}; \mathit{prod}\langle ch \rangle \oplus \text{close } ch \\
 &\quad \mathit{cons}(ch1, ch2) = \&\{ch1; \mathit{cons}\langle ch1, ch2 \rangle, ch2; \mathit{cons}\langle ch1, ch2 \rangle\} \\
 &\quad \mathit{main}() = (\text{new } ch1, ch2); (\mathit{prod}\langle ch1 \rangle \mid \mathit{prod}\langle ch2 \rangle \mid \mathit{cons}\langle ch1, ch1 \rangle) \} \\
 &\qquad\qquad\qquad \text{in } \mathit{main}\langle \rangle
 \end{aligned}$$

$$\Psi(\phi) \stackrel{\text{def}}{=} \nu x. (\phi \wedge [A]x) \qquad\qquad\qquad [\text{Always}]$$

$$\Phi(\phi) \stackrel{\text{def}}{=} \mu y. (\phi \vee \langle A \rangle y) \qquad\qquad\qquad [\text{Eventually}]$$

$$\psi_t \stackrel{\text{def}}{=} \langle A \rangle \top \qquad\qquad\qquad [\text{No terminal}]$$

$$\psi_c \stackrel{\text{def}}{=} \mu y. [A]y \qquad\qquad\qquad [\text{No cycle}]$$

$$\psi_g \stackrel{\text{def}}{=} (\bigwedge_{a \in \mathcal{A}} \downarrow_a \vee \downarrow_{\bar{a}}) \implies \langle A \rangle \top \qquad\qquad\qquad [\text{No global deadlock}]$$

$$\psi_{l_a} \stackrel{\text{def}}{=} (\bigwedge_{a \in \mathcal{A}} \downarrow_a \vee \downarrow_{\bar{a}}) \implies \Phi(\langle \tau_a \rangle \top) \qquad\qquad\qquad [\text{Liveness (a)}]$$

$$\psi_{l_b} \stackrel{\text{def}}{=} (\bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}) \implies \Phi(\langle \{\tau_a \mid a \in \tilde{a}\} \rangle \top) \qquad\qquad\qquad [\text{Liveness (b)}]$$

$$\psi_s \stackrel{\text{def}}{=} (\bigwedge_{a \in \mathcal{A}} \downarrow_{a^*}) \implies \neg(\downarrow_{\bar{a}} \vee \downarrow_{c1o a}) \qquad\qquad\qquad [\text{Channel safety}]$$

$$\psi_e \stackrel{\text{def}}{=} (\bigwedge_{a \in \mathcal{A}} \downarrow_{a^*}) \implies \Phi(\langle \tau_a \rangle \top) \qquad\qquad\qquad [\text{Eventual reception}]$$

Model Checking Go Types

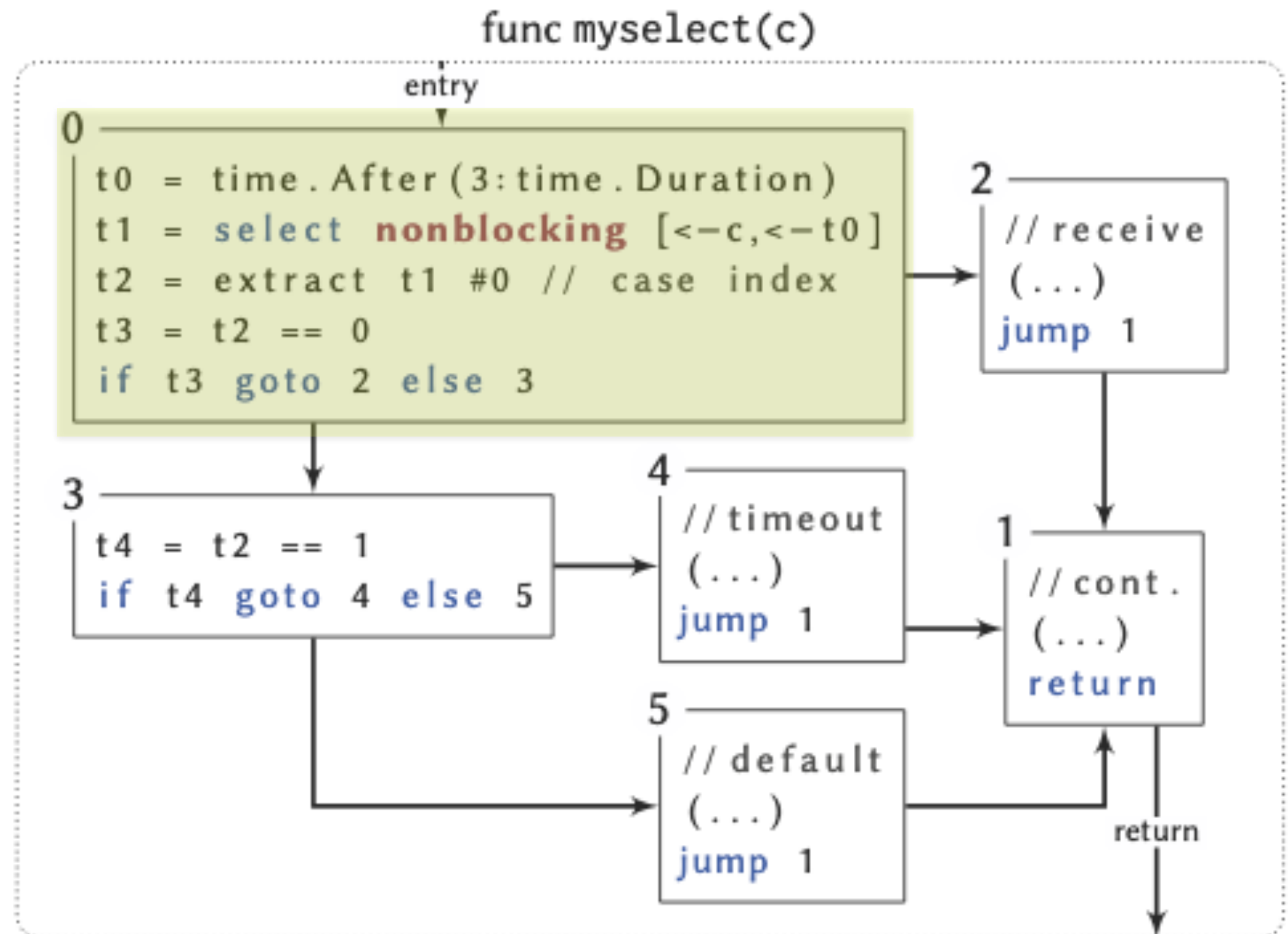
- Analyze Go programs in **SSA form**:

```
1  func myselect(c chan int) {
2      select {
3          case msg := <-c:
4              print("received: ", msg)
5          case <-time.After(time.Second):
6              print("timeout: ready in 1s")
7          default:
8              print("default: always ready")
9      }
10 }
```

Model Checking Go Types

- Analyze Go programs in **SSA form**:

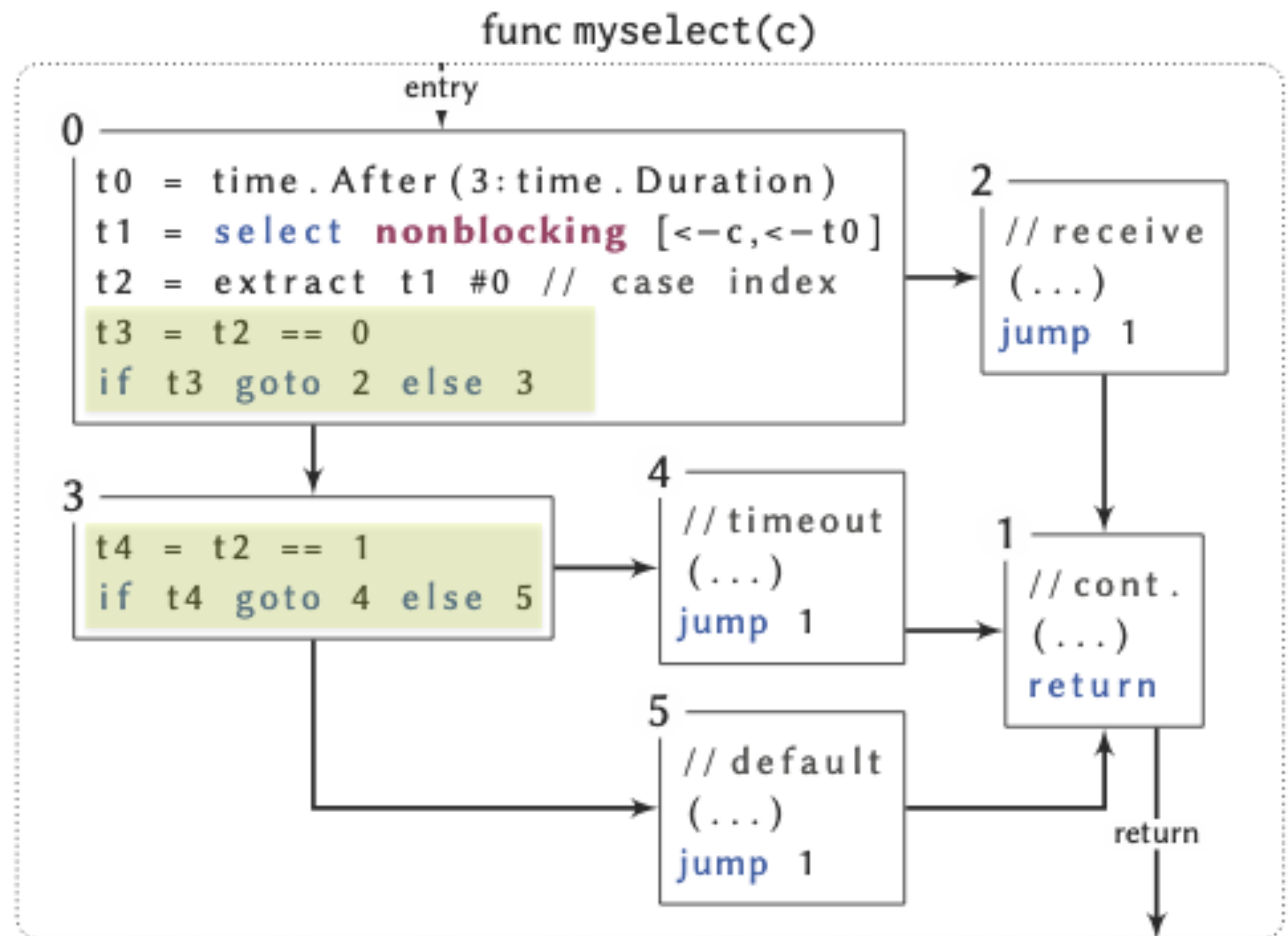
```
1 func myselect(c chan int) {  
2     select {  
3     case msg := <-c:  
4         print("received: ", msg)  
5     case <-time.After(time.Second):  
6         print("timeout: ready in 1s")  
7     default:  
8         print("default: always ready")  
9     }  
10 }
```



Model Checking Go Types

- Analyze Go programs in **SSA form**:

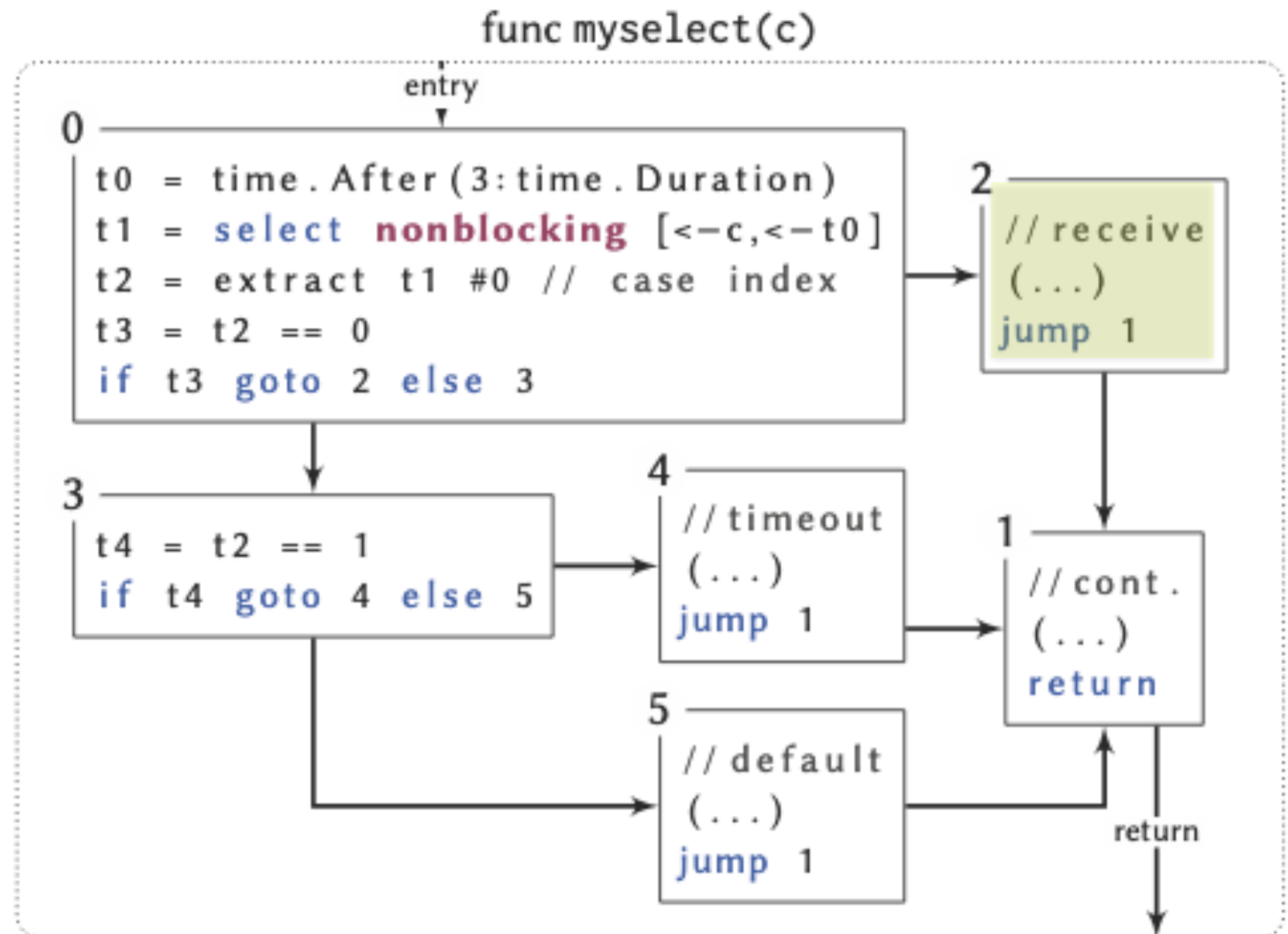
```
1 func myselect(c chan int) {  
2     select {  
3     case msg := <-c:  
4         print("received: ", msg)  
5     case <-time.After(time.Second):  
6         print("timeout: ready in 1s")  
7     default:  
8         print("default: always ready")  
9     }  
10 }
```



Model Checking Go Types

- Analyze Go programs in **SSA form**:

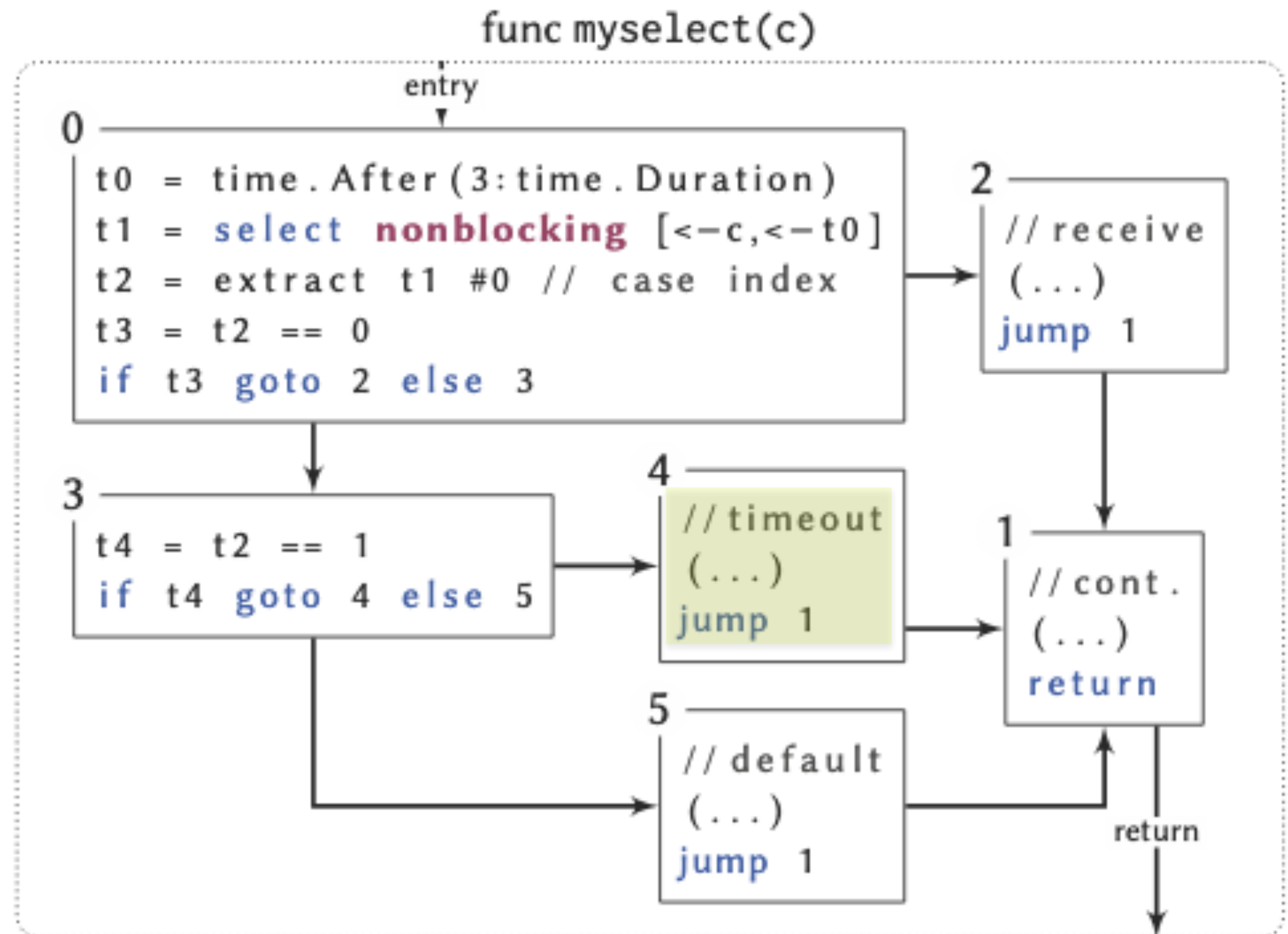
```
1 func myselect(c chan int) {  
2     select {  
3     case msg := <-c:  
4         print("received: ", msg)  
5     case <-time.After(time.Second):  
6         print("timeout: ready in 1s")  
7     default:  
8         print("default: always ready")  
9     }  
10 }
```



Model Checking Go Types

- Analyze Go programs in **SSA form**:

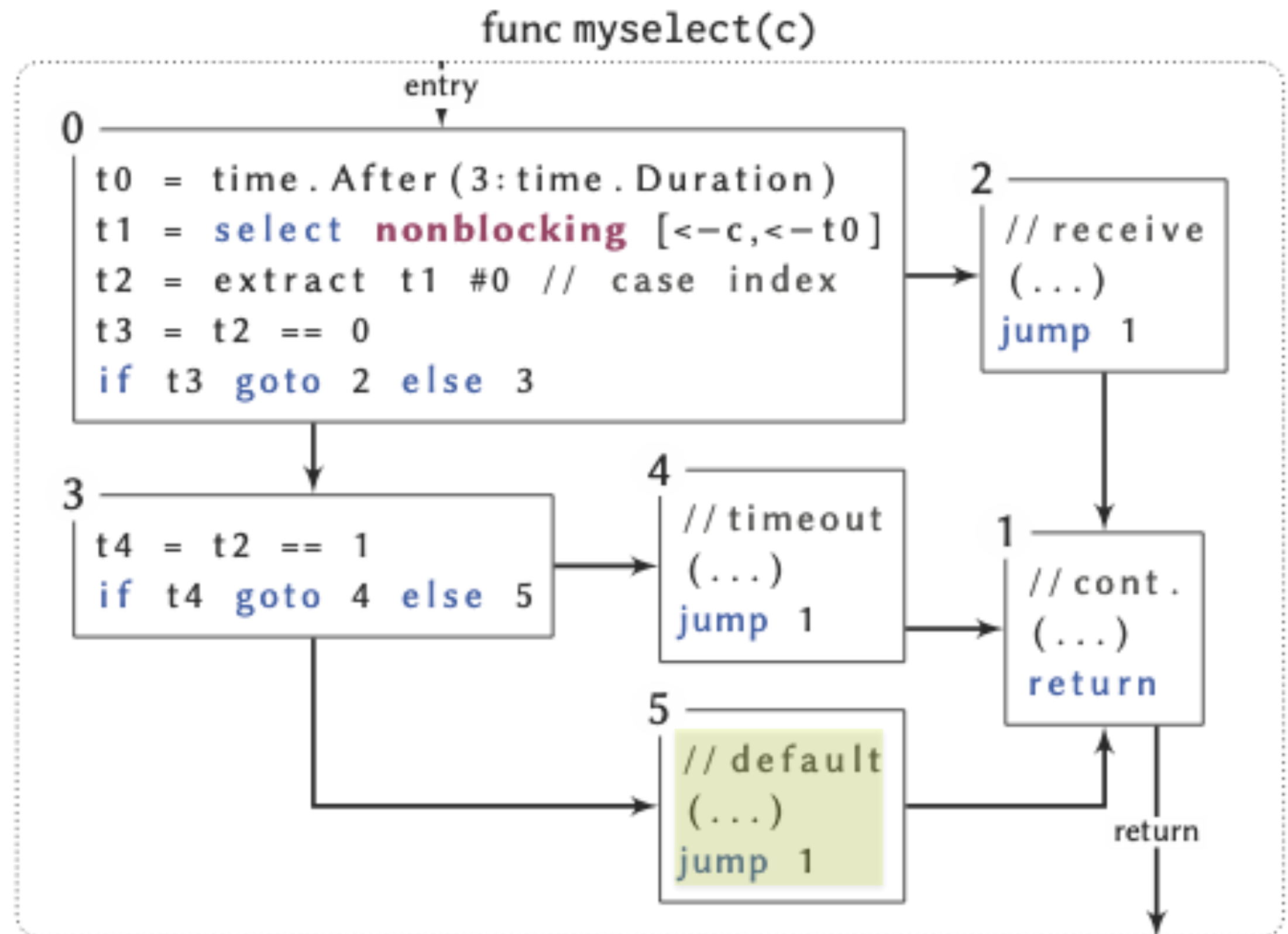
```
1 func myselect(c chan int) {  
2     select {  
3     case msg := <-c:  
4         print("received: ", msg)  
5     case <-time.After(time.Second):  
6         print("timeout: ready in 1s")  
7     default:  
8         print("default: always ready")  
9     }  
10 }
```



Model Checking Go Types

- Analyze Go programs in **SSA form**:

```
1 func myselect(c chan int) {  
2     select {  
3     case msg := <-c:  
4         print("received: ", msg)  
5     case <-time.After(time.Second):  
6         print("timeout: ready in 1s")  
7     default:  
8         print("default: always ready")  
9     }  
10 }
```

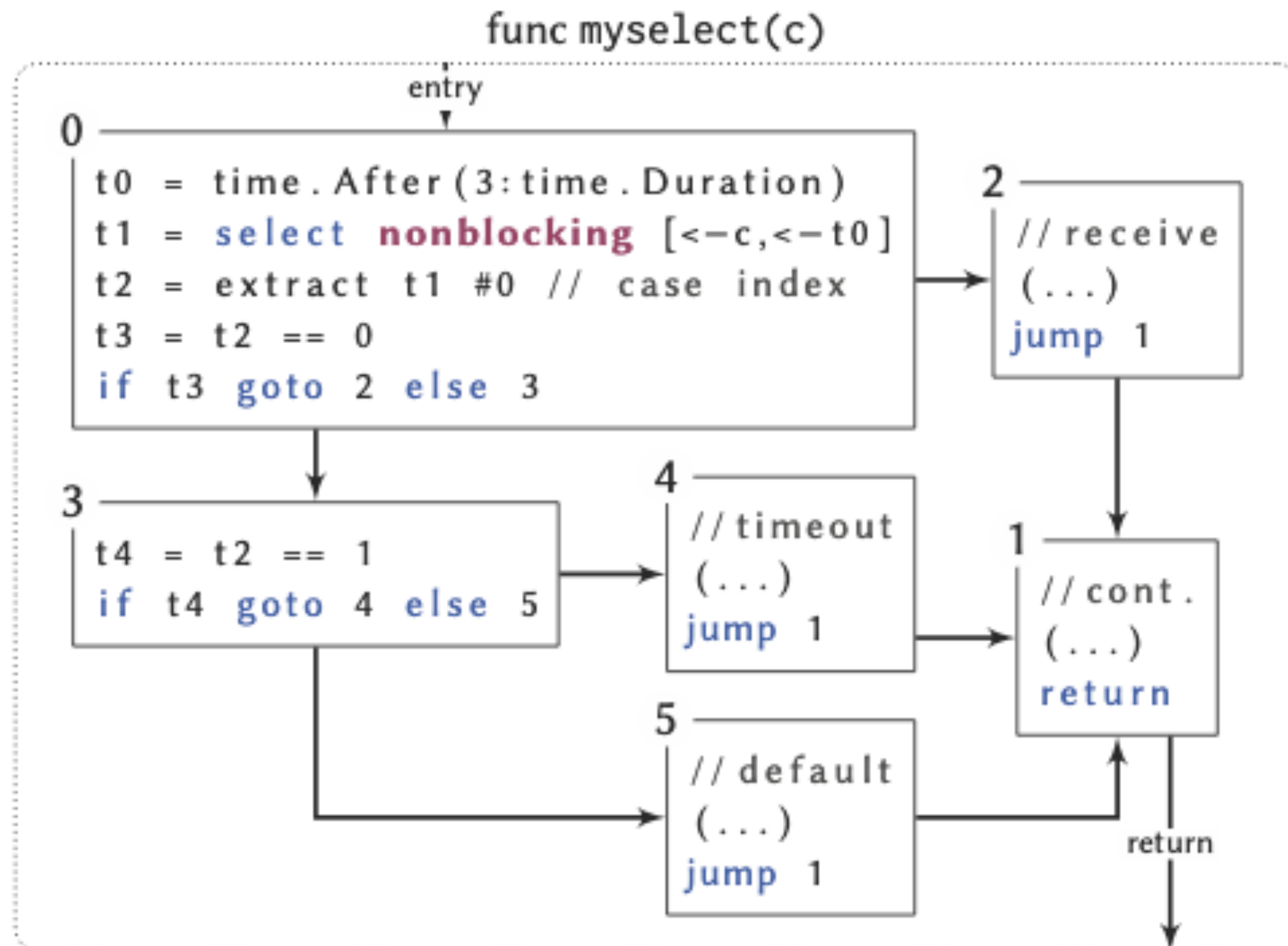


Model Checking Go Types

- Analyze Go programs in **SSA form**.
- Since Go's channel-based comm. are language primitives, they are all explicit in the SSA IR.
- Roughly, each SSA block is extracted as a separate type definition.
- Some post-processing can minimize the definitions.

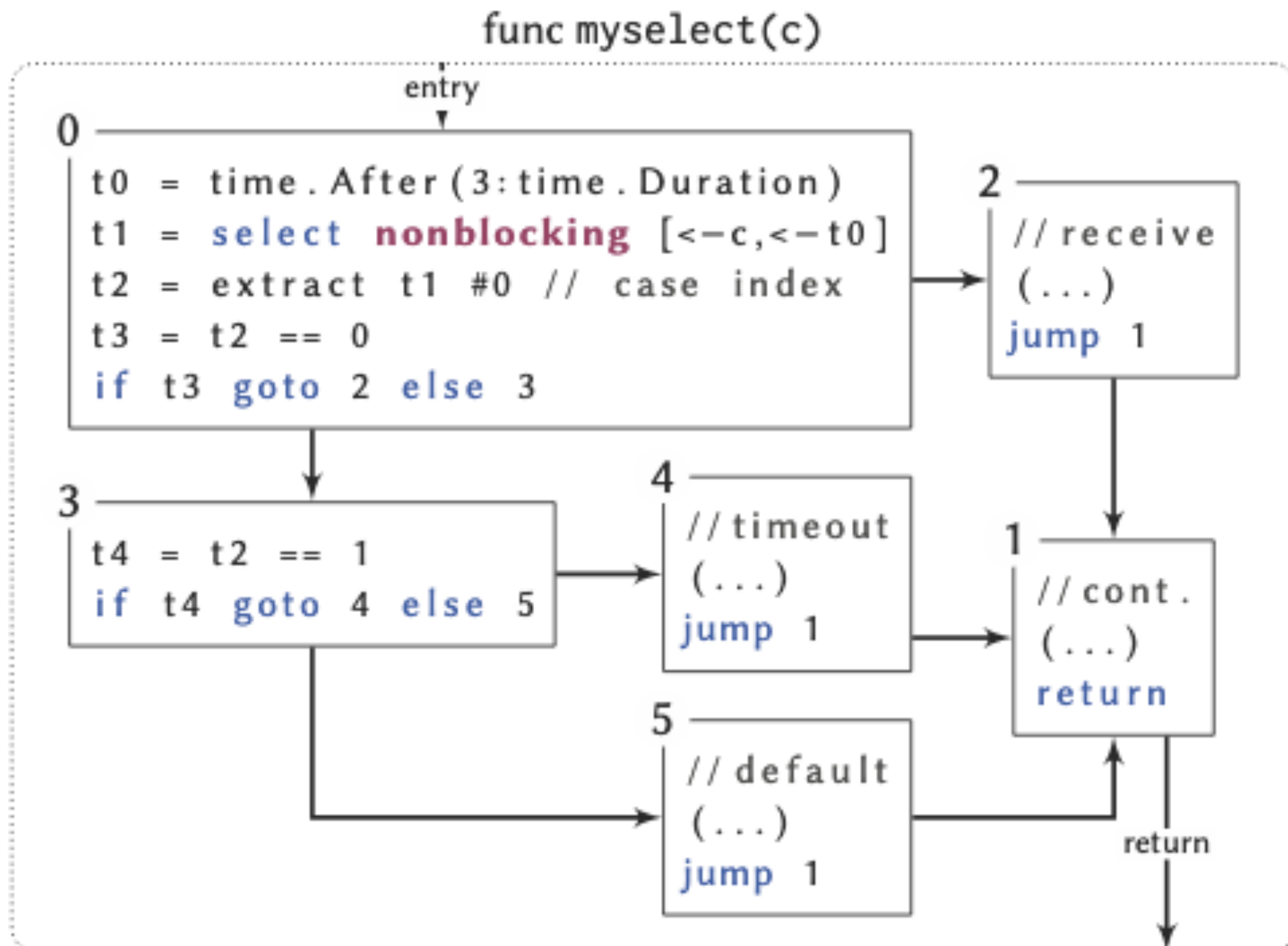
Model Checking Go Types

- Each SSA block is extracted as a separate type definition:



Model Checking Go Types

- Each SSA block is extracted as a separate type definition:



```
1 func myselect(c chan int) {
2   select {
3     case msg := <-c:
4       print("received: ", msg)
5     case <-time.After(time.Second):
6       print("timeout: ready in 1s")
7     default:
8       print("default: always ready")
9   }
10 }
```

$$\mathbf{myselect}_0(c) = \&\{c; \mathbf{myselect}_2\langle c \rangle; \mathbf{myselect}_1\langle c \rangle, \tau; \mathbf{myselect}_4\langle c \rangle; \mathbf{myselect}_1\langle c \rangle, \tau; \mathbf{myselect}_5\langle c \rangle; \mathbf{myselect}_1\langle c \rangle\}$$
$$\mathbf{myselect}_i(c) = 0 \quad \text{for } i \in \{1, 2, 4, 5\}$$
$$\mathbf{myselect}_3(c) = \mathbf{myselect}_4\langle c \rangle \oplus \mathbf{myselect}_5\langle c \rangle$$

Model Checking Go Types

- Once types are extracted, model checking for liveness and safety properties:
 - Eventual reception of messages
 - Channel safety (no send or close on closed channel)
 - Global deadlock-freedom
 - Partial deadlock-freedom
- Termination checking of loops is also employed (loop guards obtain during type extraction).

Model Checking Go Types

	Programs	LoC	# states	Godel Checker							
				ψ_g	ψ_l	ψ_s	ψ_e	Infer	Live	Live+CS	Term
1	mismatch [36]	29	53	×	×	✓	✓	620.7	996.8	996.7	✓
2	fixed [36]	27	16	✓	✓	✓	✓	624.4	996.5	996.3	✓
3	fanin [36, 39]	41	39	✓	✓	✓	✓	631.1	996.2	996.2	✓
4	sieve [30, 36]	43	∞		<i>n/a</i>			-	-	-	<i>n/a</i>
5	philo [40]	41	65	×	×	✓	✓	6.1	996.5	996.6	✓
6	dinephil3 [13, 33]	55	3838	✓	✓	✓	✓	645.2	996.4	996.3	✓
7	starvephil3	47	3151	×	×	✓	✓	628.2	996.5	996.5	✓
8	sel [40]	22	103	×	×	✓	✓	4.2	996.7	996.6	✓
9	selFixed [40]	22	20	✓	✓	✓	✓	4.0	996.3	996.4	✓
10	jobsched [30]	43	43	✓	✓	✓	✓	632.7	996.7	1996.1	✓
11	forselect [30]	42	26	✓	✓	✓	✓	623.3	996.4	996.3	✓
12	cond-recur [30]	37	12	✓	✓	✓	✓	4.0	996.2	996.2	✓
13	concsys [42]	118	15	×	×	✓	✓	549.7	996.5	996.4	✓
14	alt-bit [30, 35]	70	112	✓	✓	✓	✓	634.4	996.3	996.3	✓
15	prod-cons	28	106	✓	×	✓	✓	4.1	996.4	1996.2	✓
16	nonlive	16	8	✓	✓	✓	✓	630.1	996.6	996.5	timeout
17	double-close	15	17	✓	✓	×	✓	3.5	996.6	1996.6	✓
18	stuckmsg	8	4	✓	✓	✓	×	3.5	996.6	996.6	✓
19	dinephil5	61	~1M	✓	✓	✓	✓	626.5	41.2 sec	41.4 sec	✓
20	prod3-cons3	40	57493	✓	✓	✓	✓	465.1	40.9 sec	40.9 sec	✓
21	async-prod-cons	33	164897	✓	✓	✓	✓	4.3	47.7 sec	89.4 sec	✓
22	astranet [26]	~18k	1160	✓	✓	✓	✓	2512.5	70.4 sec	75.0 sec	✓
	Column		4	5	6	7	8	9	10	11	12

Summary

- A brief and narrow overview of (message-passing) concurrency research in PL.
- Two particular instances applied to Go.
- Many (hot) topics were not covered:
 - Higher-order concurrent separation logic (hot!)
 - Logical (and richer) session types (hot!)
 - Infinite state systems?
 - Interplay of shared memory + channel-based concurrency.

References

[CPN98] David G. Clarke, John Potter, James Noble:
Ownership Types for Flexible Alias Protection. OOPSLA 1998

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, David N. Turner: Linearity and the pi-calculus. ACM TOPLAS 99 / POPL 96

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, Makoto Kubo:
Language Primitives and Type Discipline for Structured Communication-Based Programming. ESOP 1998

[HYC08] Kohei Honda, Nobuko Yoshida, Marco Carbone:
Multiparty asynchronous session types. POPL 2008

[CRR02] Sagar Chaki, Sriram K. Rajamani, Jakob Rehof:
Types as models: model checking message-passing programs. POPL 2002

[Pnueli77] Amir Pnueli:
The Temporal Logic of Programs. FOCS 1977

References

[PHJNY19] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, Nobuko Yoshida: Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures POPL 2019

[SDHY17] Alceste Scalas, Ornela Dardha, Raymond Hu, Nobuko Yoshida: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. ECOOP 2017

[DHHNY15] Romain Demangeon, Kohei Honda, Raymond Hu, Romyana Neykova, Nobuko Yoshida: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. FMDS 2015

[NHYA18] Romyana Neykova, Raymond Hu, Nobuko Yoshida, Fahd Abdeljallal: A session type provider: compile-time API generation of distributed protocols with refinements in F#. CC 2018

[LNTY17] Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida: Fencing off go: liveness and safety for channel-based programming. POPL 2017

[LNTY18] Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida: A static verification framework for message passing in Go using behavioural types. ICSE 2018

References

[CP10] Luís Caires, Frank Pfenning: Session Types as Intuitionistic Linear Propositions. CONCUR 2010

[TCP11] Bernardo Toninho, Luís Caires, Frank Pfenning: Dependent session types via intuitionistic linear type theory. PPDP 2011

[Wadler12] Philip Wadler: Propositions as sessions. ICFP 2012

[TCP13] Bernardo Toninho, Luís Caires, Frank Pfenning: Higher-Order Processes, Functions, and Sessions: A Monadic Integration. ESOP 2013

[LM16] Sam Lindley, J. Garrett Morris: Talking bananas: structural recursion for session types. ICFP 2016

[ITVW17] Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, Philip Wadler: Gradual session types. ICFP 2017

[TY18] Bernardo Toninho, Nobuko Yoshida: Depending on Session-Typed Processes. FoSSaCS 2018

[BTP19] Stephanie Balzer, Bernardo Toninho, Frank Pfenning: Manifest Deadlock-Freedom for Shared Session Types. ESOP 2019