Concurrent Programming Languages

Channel-based Concurrency Module

Lecture 4: Applied Research Topics

2 November 2021

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(log3/2 n) Parallel Time Population Protocol for Majority with O(logn) 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
  - Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms
  - Partially Observable Concurrent Kleene Algebra
  - Sized Types with Usages for Parallel Complexity of Pi-Calculus Processes

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.
  - A Separation Logic for Effect Handlers
  - Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic
  - Practical Smart Contract Sharding with Ownership and Commutativity Analysis
  - 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?

- This month you will see the interaction of 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` $\leq$ `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

```
stype 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

```
stype 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

```
stype 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

```
stype 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

```
stype 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

```
stype 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.

```
stype 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).

```
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]

- 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]

- 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.]

Dependent Session Types [TCP11]

- Add logical assertions on data to types:

```
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
   }
}}
```

Dependent Session Types [TCP11]

- Add logical assertions on data to types:

```
stype T = Choice{Add: ?(x:int);?(y:int);!(z:int|z=y+x);T,
                 Sym: ?(x:int);!(y:int|y=-x);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
   }
}}
```

Dependent Session Types [TCP11]

- Add logical assertions on data to types:

```
stype T = Choice{Add: ?(x:int);?(y:int);!(z:int|z=y+x);T,
                 Sym: ?(x:int);!(y:int|y=-x);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
   }
}}
```

Dependent Session Types [TCP11]

- Add logical assertions on data to types:

```
stype T = Choice{Add: ?(x:int);?(y:int);!(z:int|z=y+x);T,
                 Sym: ?(x:int);!(y:int|y=-x);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
   }
}}
```

Dependent Session Types [TCP11]

- Add logical assertions on data to types:

```
stype T = Choice{Add: ?(x:int);?(y:int);!(z:int|z=y+x);T,
                 Sym: ?(x:int);!(y:int|y=-x);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
   }
}}
```

Dependent Session Types [TCP11]

- Add logical assertions on data to types…
- More to it than that:
  - Compile-time verification requires decidable assertions
  - …or explicit proof objects
  - Implications on trust if in a distributed setting
  - Type dependency + linearity is very tricky

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:

$$
\begin{array}{l}
\mathsf{B} \to \mathsf{S} : \mathsf{ItemId}. \\
\mathsf{S} \to \mathsf{B} : \mathsf{Quote}. \\
\mathsf{B} \to \mathsf{S} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \mathsf{Address}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Sh} \to \mathsf{S} : \mathsf{Receipt}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{S} \to \mathsf{B} : \mathsf{Receipt}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{end}, \\
\qquad\qquad\qquad\qquad\qquad\quad Quit : \ldots \} \\
\qquad\quad Quit : \ldots \}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathsf{B} \to \mathsf{S} : \mathsf{ItemId}. \\
\mathsf{S} \to \mathsf{B} : \mathsf{Quote}. \\
\mathsf{B} \to \mathsf{S} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \{ Ok : \begin{array}[t]{l}
\mathsf{S} \to \mathsf{Sh} : \mathsf{Address}. \\
\mathsf{Sh} \to \mathsf{S} : \mathsf{Receipt}. \\
\mathsf{S} \to \mathsf{B} : \mathsf{Receipt}. \\
\mathbf{end},
\end{array} \\
\qquad\qquad\qquad\qquad\qquad\qquad Quit : \ldots \} \\
\qquad\quad Quit : \ldots \}
\end{array}
$$

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:

$$
\begin{aligned}
&\mathsf{B} \to \mathsf{S} : \mathsf{ItemId}.\\
&\mathsf{S} \to \mathsf{B} : \mathsf{Quote}.\\
&\mathsf{B} \to \mathsf{S} : \{Ok : \mathsf{S} \to \mathsf{Sh} : \{Ok : \mathsf{S} \to \mathsf{Sh} : \mathsf{Address}.\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Sh} \to \mathsf{S} : \mathsf{Receipt}.\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{S} \to \mathsf{B} : \mathsf{Receipt}.\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{end},\\
&\qquad\qquad\qquad\qquad\qquad\quad Quit : \ldots\}\\
&\qquad\quad Quit : \ldots\}
\end{aligned}
$$

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:

$$
\begin{array}{l}
\mathsf{B} \to \mathsf{S} : \mathsf{ItemId}. \\
\mathsf{S} \to \mathsf{B} : \mathsf{Quote}. \\
\mathsf{B} \to \mathsf{S} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \{ Ok : \begin{array}[t]{l} \mathsf{S} \to \mathsf{Sh} : \mathsf{Address}. \\ \mathsf{Sh} \to \mathsf{S} : \mathsf{Receipt}. \\ \mathsf{S} \to \mathsf{B} : \mathsf{Receipt}. \\ \mathbf{end}, \end{array} \\
\qquad\qquad\qquad\qquad\qquad Quit : \ldots \} \\
\qquad\quad Quit : \ldots \}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathsf{B} \to \mathsf{S} : \mathsf{ItemId}. \\
\mathsf{S} \to \mathsf{B} : \mathsf{Quote}. \\
\mathsf{B} \to \mathsf{S} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \{ Ok : \mathsf{S} \to \mathsf{Sh} : \mathsf{Address}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Sh} \to \mathsf{S} : \mathsf{Receipt}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{S} \to \mathsf{B} : \mathsf{Receipt}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{end}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Quit : \ldots \} \\
\qquad\qquad\quad Quit : \ldots \}
\end{array}
$$

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:

$$\begin{aligned}
G \upharpoonright \mathsf{B} \quad &= \quad \mathsf{S}!(\mathsf{ItemId}); \mathsf{S}?(\mathsf{Quote}); \mathsf{S}!\{\mathit{Ok} : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \mathit{Quit} : \ldots\}\\
G \upharpoonright \mathsf{S} \quad &= \quad \mathsf{B}?(\mathsf{ItemId}); \mathsf{B}!(\mathsf{Quote}); \mathsf{B}?\{\mathit{Ok} : \mathsf{Sh}!\{\mathit{Ok} : \mathsf{Sh}!(\mathsf{Address}); \mathsf{Sh}?(\mathsf{Receipt}); \mathsf{B}!(\mathsf{Receipt}), \mathit{Quit} : \ldots\},\\
&\qquad\qquad \mathit{Quit} : \ldots\}\\
G \upharpoonright \mathsf{Sh} \quad &= \quad \mathsf{S}?\{\mathit{Ok} : \mathsf{S}?(\mathsf{Address}); \mathsf{S}!(\mathsf{Receipt}); \mathbf{end}, \mathit{Quit} : \ldots\}
\end{aligned}$$

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:

$$
\begin{aligned}
G \upharpoonright \mathsf{B} \quad &= \quad \mathsf{S}!(\mathsf{ItemId}); \mathsf{S}?(\mathsf{Quote}); \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, Quit : \ldots\} \\
G \upharpoonright \mathsf{S} \quad &= \quad \mathsf{B}?(\mathsf{ItemId}); \mathsf{B}!(\mathsf{Quote}); \mathsf{B}?\{Ok : \mathsf{Sh}!\{Ok : \mathsf{Sh}!(\mathsf{Address}); \mathsf{Sh}?(\mathsf{Receipt}); \mathsf{B}!(\mathsf{Receipt}), Quit : \ldots\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad Quit : \ldots\} \\
G \upharpoonright \mathsf{Sh} \quad &= \quad \mathsf{S}?\{Ok : \mathsf{S}?(\mathsf{Address}); \mathsf{S}!(\mathsf{Receipt}); \mathbf{end}, Quit : \ldots\}
\end{aligned}
$$

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:

$$
\begin{aligned}
G \upharpoonright \mathsf{B} &= \mathsf{S}!(\mathsf{ItemId}); \mathsf{S}?(\mathsf{Quote}); \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, Quit : \ldots\} \\
G \upharpoonright \mathsf{S} &= \mathsf{B}?(\mathsf{ItemId}); \mathsf{B}!(\mathsf{Quote}); \mathsf{B}?\{Ok : \mathsf{Sh}!\{Ok : \mathsf{Sh}!(\mathsf{Address}); \mathsf{Sh}?(\mathsf{Receipt}); \mathsf{B}!(\mathsf{Receipt}), Quit : \ldots\}, \\
&\qquad Quit : \ldots\} \\
G \upharpoonright \mathsf{Sh} &= \mathsf{S}?\{Ok : \mathsf{S}?(\mathsf{Address}); \mathsf{S}!(\mathsf{Receipt}); \mathbf{end}, Quit : \ldots\}
\end{aligned}
$$

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:

$$\begin{array}{lcl}
G \upharpoonright \mathsf{B} & = & \mathsf{S}!(\mathsf{ItemId}); \mathsf{S}?(\mathsf{Quote}); \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, Quit : \ldots\} \\
G \upharpoonright \mathsf{S} & = & \mathsf{B}?(\mathsf{ItemId}); \mathsf{B}!(\mathsf{Quote}); \mathsf{B}?\{Ok : \mathsf{Sh}!\{Ok : \mathsf{Sh}!(\mathsf{Address}); \mathsf{Sh}?(\mathsf{Receipt}); \mathsf{B}!(\mathsf{Receipt}), Quit : \ldots\}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad Quit : \ldots\} \\
G \upharpoonright \mathsf{Sh} & = & \mathsf{S}?\{Ok : \mathsf{S}?(\mathsf{Address}); \mathsf{S}!(\mathsf{Receipt}); \mathbf{end}, Quit : \ldots\}
\end{array}$$

Multiparty Session Types [HYC08]

$$\begin{array}{lcl}
G \upharpoonright \mathsf{B} & = & \mathsf{S}!(\mathsf{ItemId}); \mathsf{S}?(\mathsf{Quote}); \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, Quit : \ldots\} \\
G \upharpoonright \mathsf{S} & = & \mathsf{B}?(\mathsf{ItemId}); \mathsf{B}!(\mathsf{Quote}); \mathsf{B}?\{Ok : \mathsf{Sh}!\{Ok : \mathsf{Sh}!(\mathsf{Address}); \mathsf{Sh}?(\mathsf{Receipt}); \mathsf{B}!(\mathsf{Receipt}), Quit : \ldots\}, \\
 & & \qquad\qquad\qquad\qquad\qquad Quit : \ldots\} \\
G \upharpoonright \mathsf{Sh} & = & \mathsf{S}?\{Ok : \mathsf{S}?(\mathsf{Address}); \mathsf{S}!(\mathsf{Receipt}); \mathbf{end}, Quit : \ldots\}
\end{array}$$

- 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 ($A \land B$, $\neg A$) +
  - “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** ¬(reactor_temp > 1000)
  - **G** ¬(crit_region1 ∧ crit_region2)
- Liveness properties (“something good will happen”)
  - **G** (sending ⇒ **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?

$$
\begin{array}{rcl}
G \upharpoonright \mathsf{B} & = & \mathsf{S}!(\mathsf{ItemId}); \\
& & \mathsf{S}?(\mathsf{Quote}); \\
& & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\
& & \quad Quit : \ldots\}
\end{array}
$$

```
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:

$$
\begin{aligned}
G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId}); \\
& \mathsf{S}?(\mathsf{Quote}); \\
& \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\
& \quad Quit : \ldots\}
\end{aligned}
$$

```
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]

$$\begin{aligned} G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId}); \\ & \mathsf{S}?(\mathsf{Quote}); \\ & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\ & \quad Quit : \ldots\} \end{aligned}$$

```
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]

$$\begin{array}{lcl} G \upharpoonright \mathsf{B} & = & \mathsf{S}!(\mathsf{ItemId}); \\ & & \mathsf{S}?(\mathsf{Quote}); \\ & & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\ & & \quad Quit : \ldots\} \end{array}$$

$$\begin{array}{lcl} G \upharpoonright \mathsf{S} & = & \mathsf{B}?(\mathsf{ItemId}); \\ & & \mathsf{B}!(\mathsf{Quote}); \\ & & \mathsf{B}?\{Ok : \mathsf{Sh}!(\mathsf{Address}); \\ & & \qquad\quad \mathsf{Sh}?(\mathsf{Receipt}); \\ & & \qquad\quad \mathsf{B}!(\mathsf{Receipt}), \\ & & \quad Quit : \ldots\} \end{array}$$

$$\begin{array}{lcl} G \upharpoonright \mathsf{Sh} & = & \mathsf{S}?\{Ok : \mathsf{S}?(\mathsf{Address}); \\ & & \qquad\quad \mathsf{S}!(\mathsf{Receipt}); \\ & & \qquad\quad \mathbf{end}, \\ & & \quad Quit : \ldots\} \end{array}$$

- We can engineer formulae that denote properties of interest:
  - $G(\langle S! \rangle \, \mathsf{T} \Rightarrow F(\langle S!? \rangle \, \mathsf{T}))$ “Eventual reception for S”
  - $G(\langle S! \rangle \, \mathsf{T} \vee \langle S? \rangle \, \mathsf{T} \vee \langle B! \rangle \, \mathsf{T} \vee \ldots) \Rightarrow \langle * \rangle \, \mathsf{T}$ ”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

$$
\begin{array}{rcl}
G \upharpoonright \mathsf{B} & = & \mathsf{S}!(\mathsf{ItemId}); \\
 & & \mathsf{S}?(\mathsf{Quote}); \\
 & & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\
 & & \quad Quit : \ldots\}
\end{array}
$$

Session Types in Practice

$$
\begin{aligned}
G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId}); \\
& \mathsf{S}?(\mathsf{Quote}); \\
& \mathsf{S}!\{\mathit{Ok} : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\
& \quad \mathit{Quit} : \ldots\}
\end{aligned}
$$

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

Session Types in Practice

$$\begin{aligned} G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId}); \\ & \mathsf{S}?(\mathsf{Quote}); \\ & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\ & \quad Quit : \ldots\} \end{aligned}$$

```
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}
}
```

$$\begin{aligned} G \upharpoonright \mathsf{B} \;=\;& \mathsf{S}!(\mathsf{ItemId});\\ & \mathsf{S}?(\mathsf{Quote});\\ & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end},\\ & \quad Quit : \ldots\} \end{aligned}$$

Session Types in Practice

$$\begin{aligned} G \upharpoonright \mathsf{B} \quad &= \quad \mathsf{S}!(\mathsf{ItemId}); \\ & \quad\quad \mathsf{S}?(\mathsf{Quote}); \\ & \quad\quad \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\ & \quad\quad\quad Quit : \ldots\} \end{aligned}$$

```go
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

$$\begin{aligned} G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId}); \\ & \mathsf{S}?(\mathsf{Quote}); \\ & \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end}, \\ & \quad Quit : \ldots\} \end{aligned}$$

```
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()
     …
```

$$
\begin{aligned}
G \upharpoonright \mathsf{B} \quad = \quad & \mathsf{S}!(\mathsf{ItemId});\\
& \mathsf{S}?(\mathsf{Quote});\\
& \mathsf{S}!\{Ok : \mathsf{S}?(\mathsf{Receipt}).\mathbf{end},\\
& \quad Quit : \ldots\}
\end{aligned}
$$

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?

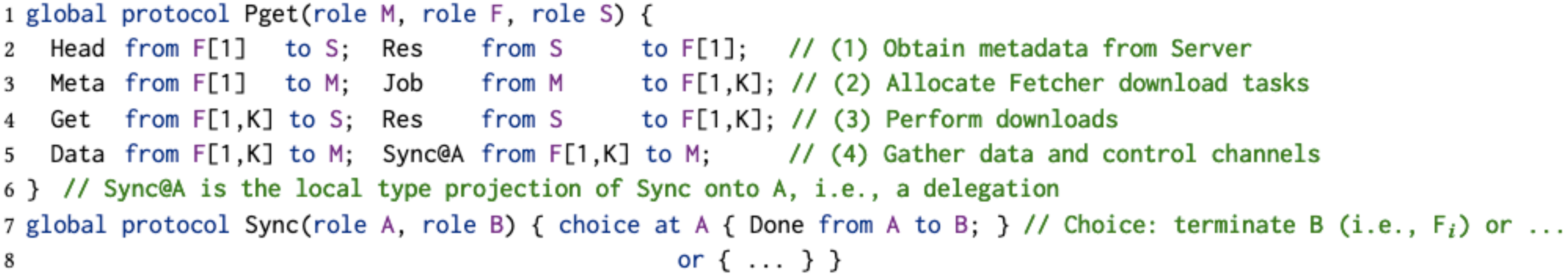
Session Types in Practice

```
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]

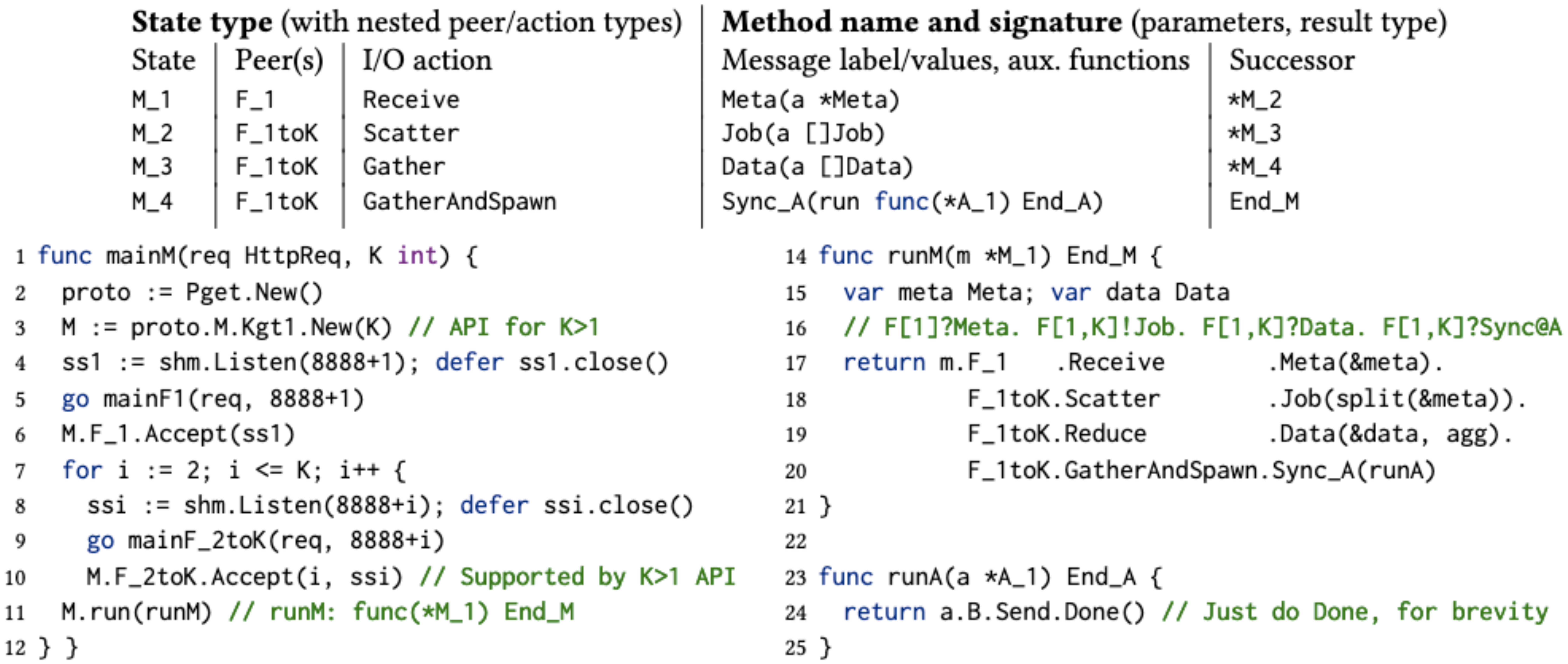


```
global protocol Pget(role M, role F, role S) {
  Head from F[1]   to S;  Res    from S      to F[1];   // (1) Obtain metadata from Server
  Meta from F[1]   to M;  Job    from M      to F[1,K]; // (2) Allocate Fetcher download tasks
  Get  from F[1,K] to S;  Res    from S      to F[1,K]; // (3) Perform downloads
  Data from F[1,K] to M;  Sync@A from F[1,K] to M;      // (4) Gather data and control channels
} // Sync@A is the local type projection of Sync onto A, i.e., a delegation
global protocol Sync(role A, role B) { choice at A { Done from A to B; } // Choice: terminate B (i.e., F_i) or ...
                                                 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)			Method name and signature (parameters, result type)	
State	Peer(s)	I/O action	Message label/values, aux. functions	Successor
`M_1`	`F_1`	`Receive`	`Meta(a *Meta)`	`*M_2`
`M_2`	`F_1toK`	`Scatter`	`Job(a []Job)`	`*M_3`
`M_3`	`F_1toK`	`Gather`	`Data(a []Data)`	`*M_4`
`M_4`	`F_1toK`	`GatherAndSpawn`	`Sync_A(run func(*A_1) End_A)`	`End_M`

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

```
func runM(m *M_1) End_M {
  var meta Meta; var data Data
  // F[1]?Meta. F[1,K]!Job. F[1,K]?Data. F[1,K]?Sync@A
  return m.F_1   .Receive       .Meta(&meta).
           F_1toK.Scatter       .Job(split(&meta)).
           F_1toK.Reduce        .Data(&data, agg).
           F_1toK.GatherAndSpawn.Sync_A(runA)
}

func runA(a *A_1) End_A {
  return a.B.Send.Done() // Just do Done, for brevity
}
```

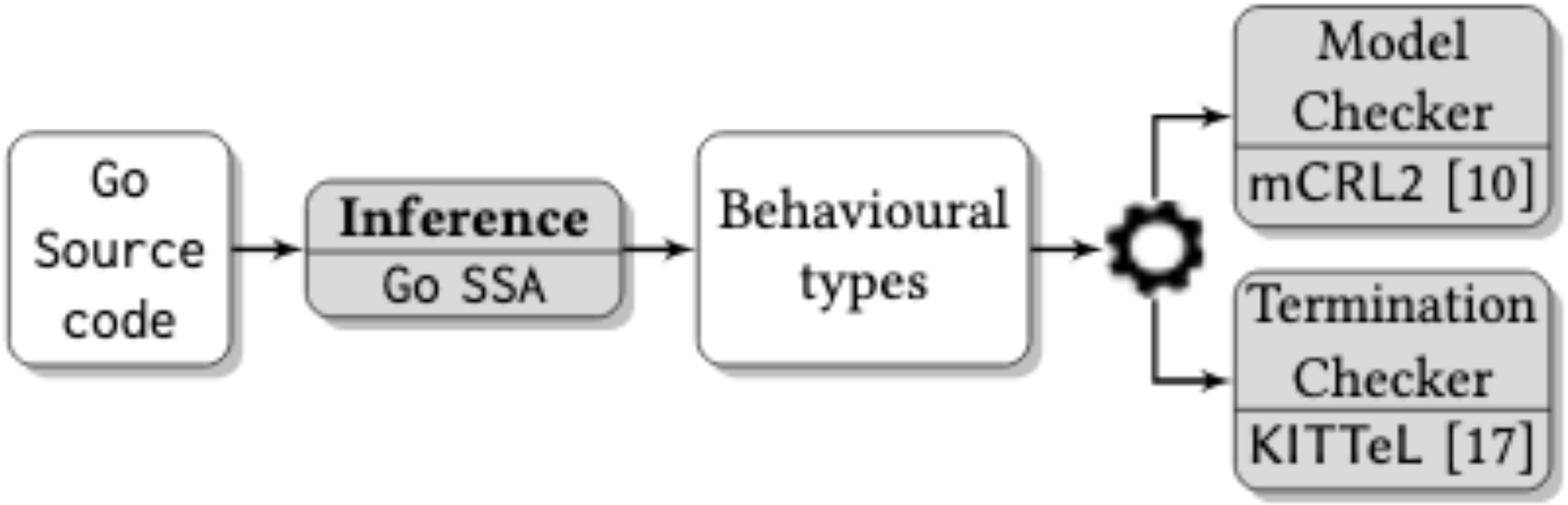
Session Types in Practice [CHJNY19]

- 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].

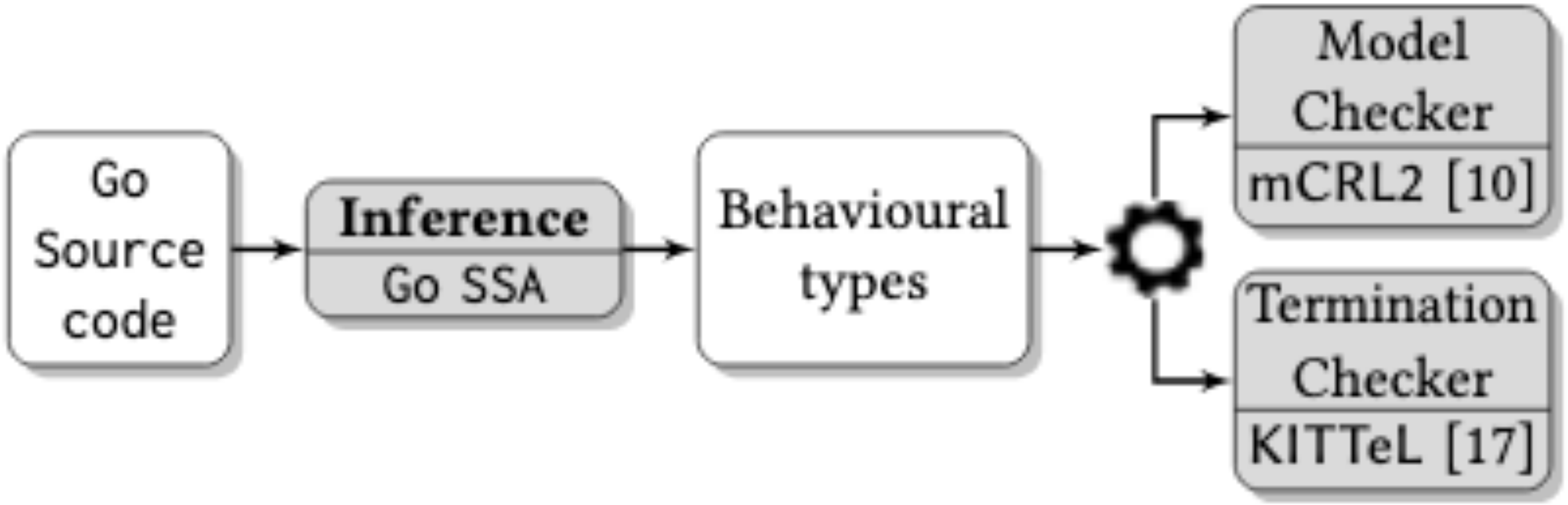
Model Checking Go Types [LNTY17,LNTY18]

- 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



Go Source code

Inference

Go SSA

Behavioural types

Model Checker

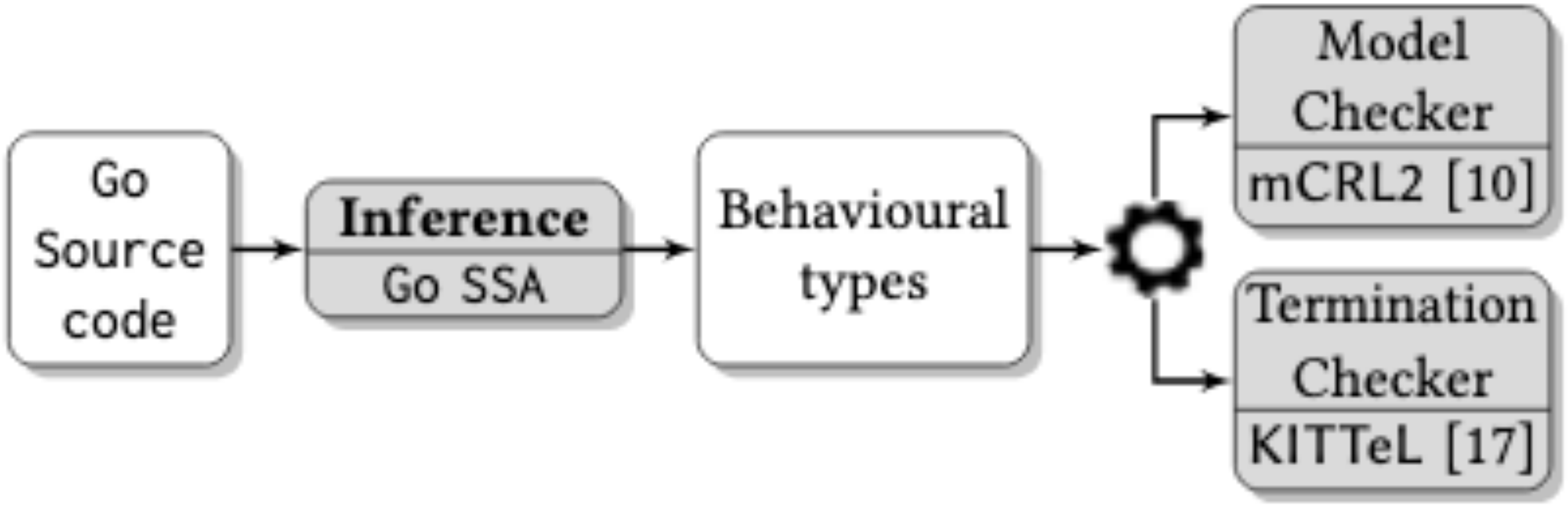
mCRL2 [10]

Termination Checker

KITTeL [17]

```go
func prod(ch chan int) {
  for i := 0; i < 5; i++ {
    ch <- i // Send i to ch
  }
  close(ch) // No further values accepted at ch
}
func cons(ch1, ch2 chan int) {
  for {
    select {
    case x := <-ch1: print(x) // Either input from ch1
    case x := <-ch2: print(x) // or input from ch2
    }
  }
}
func main() {
  ch1, ch2 := make(chan int), make(chan int)
  go prod(ch1)
  go prod(ch2)
  cons(ch1, ch1)
}
```

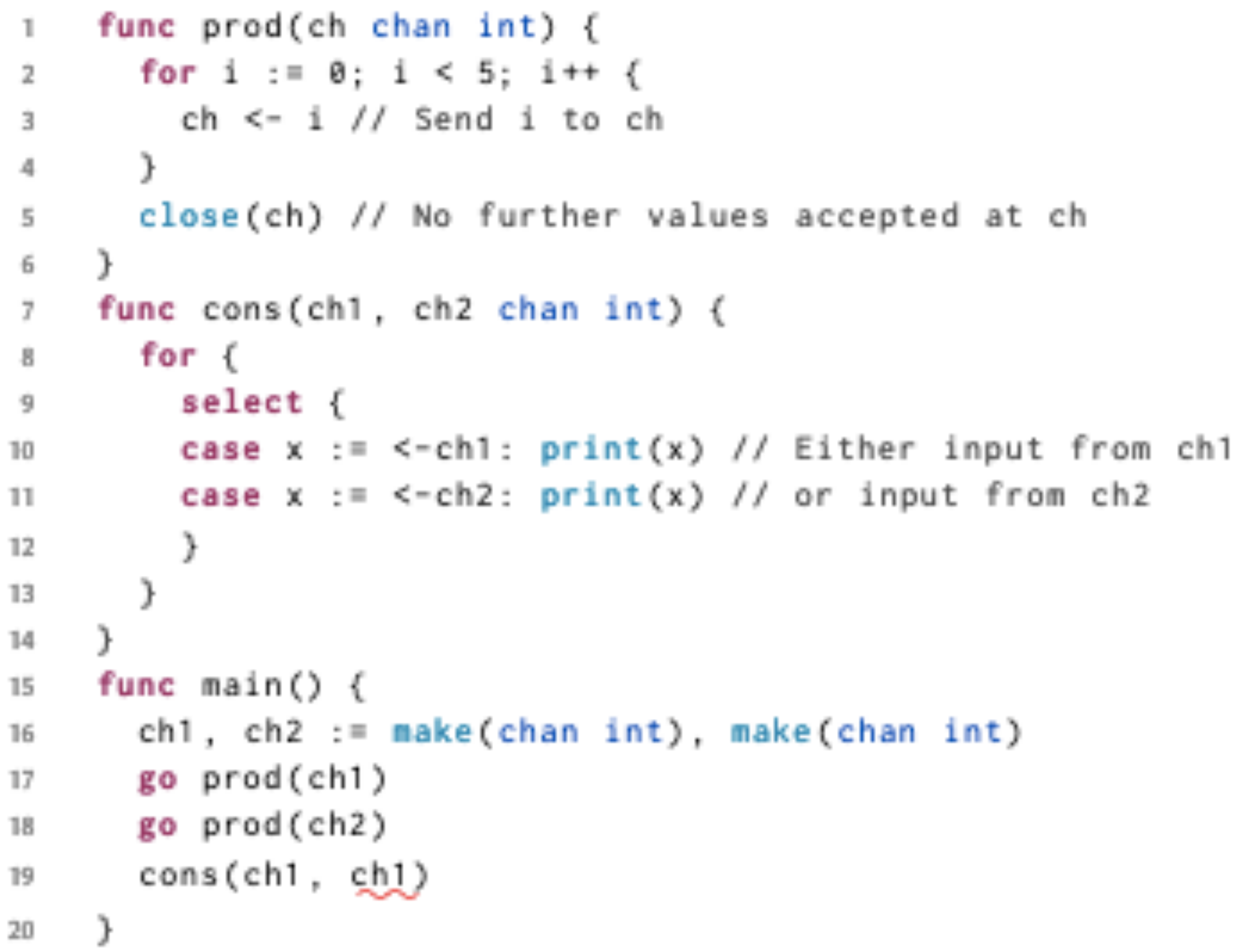
Model Checking Go Types



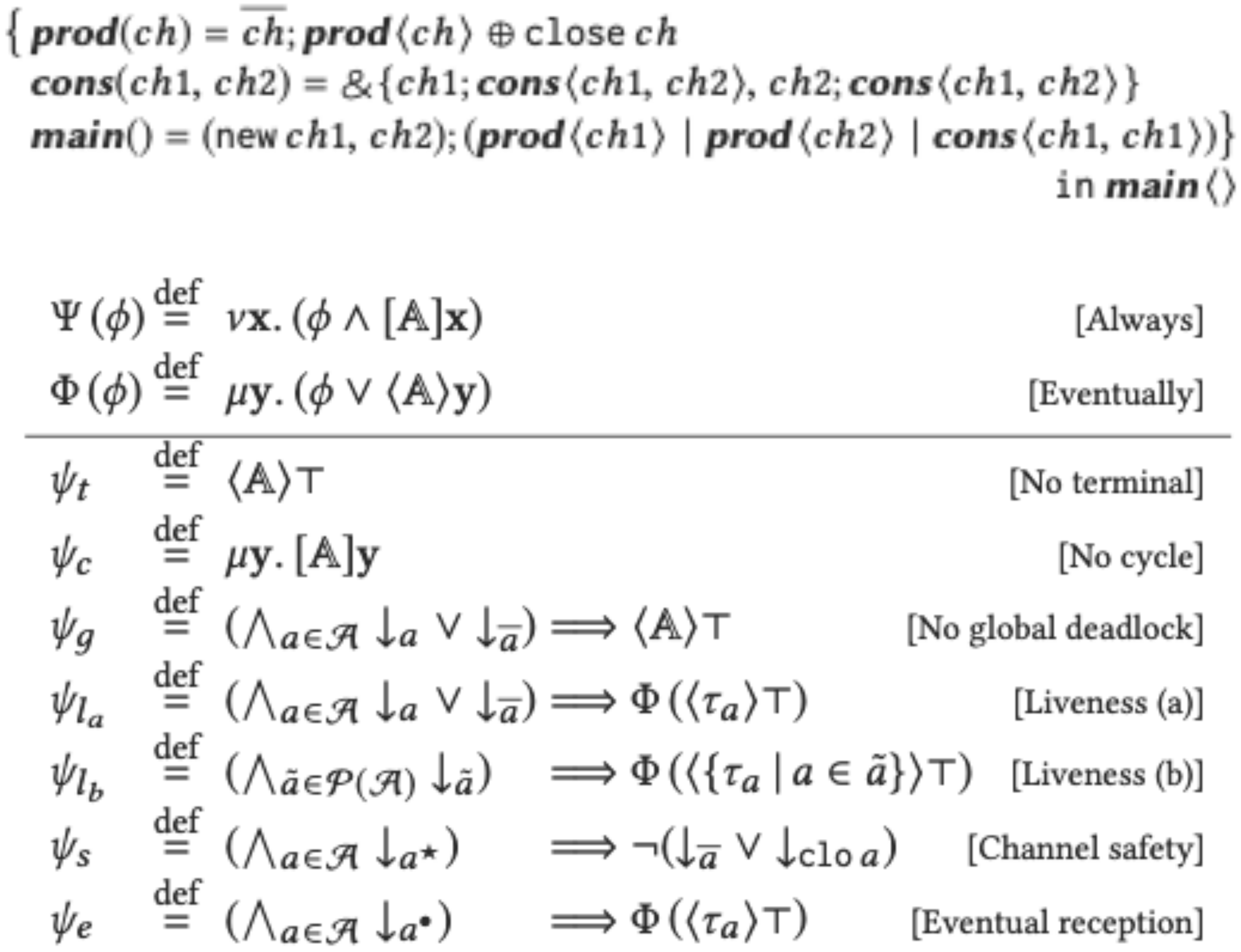
```go
func prod(ch chan int) {
  for i := 0; i < 5; i++ {
    ch <- i // Send i to ch
  }
  close(ch) // No further values accepted at ch
}
func cons(ch1, ch2 chan int) {
  for {
    select {
    case x := <-ch1: print(x) // Either input from ch1
    case x := <-ch2: print(x) // or input from ch2
    }
  }
}
func main() {
  ch1, ch2 := make(chan int), make(chan int)
  go prod(ch1)
  go prod(ch2)
  cons(ch1, ch1)
}
```

$$\left\{ \begin{array}{l} \boldsymbol{prod}(ch) = \overline{ch}; \boldsymbol{prod}\langle ch\rangle \oplus \mathsf{close}\ ch \\ \boldsymbol{cons}(ch1, ch2) = \&\{ch1; \boldsymbol{cons}\langle ch1, ch2\rangle, ch2; \boldsymbol{cons}\langle ch1, ch2\rangle\} \\ \boldsymbol{main}() = (\mathsf{new}\ ch1, ch2); (\boldsymbol{prod}\langle ch1\rangle \mid \boldsymbol{prod}\langle ch2\rangle \mid \boldsymbol{cons}\langle ch1, ch1\rangle) \end{array} \right\} \\ \mathsf{in}\ \boldsymbol{main}\langle\rangle$$

Model Checking Go Types



```go
func prod(ch chan int) {
  for i := 0; i < 5; i++ {
    ch <- i // Send i to ch
  }
  close(ch) // No further values accepted at ch
}
func cons(ch1, ch2 chan int) {
  for {
    select {
    case x := <-ch1: print(x) // Either input from ch1
    case x := <-ch2: print(x) // or input from ch2
    }
  }
}
func main() {
  ch1, ch2 := make(chan int), make(chan int)
  go prod(ch1)
  go prod(ch2)
  cons(ch1, ch1)
}
```



$$\begin{array}{l}\{\, \boldsymbol{prod}(ch) = \overline{ch}; \boldsymbol{prod}\langle ch\rangle \oplus \mathtt{close}\, ch \\ \;\; \boldsymbol{cons}(ch1, ch2) = \&\{ch1; \boldsymbol{cons}\langle ch1, ch2\rangle, ch2; \boldsymbol{cons}\langle ch1, ch2\rangle\} \\ \;\; \boldsymbol{main}() = (\mathsf{new}\, ch1, ch2); (\boldsymbol{prod}\langle ch1\rangle \mid \boldsymbol{prod}\langle ch2\rangle \mid \boldsymbol{cons}\langle ch1, ch1\rangle)\} \\ \hfill \mathtt{in}\; \boldsymbol{main}\langle\rangle \end{array}$$

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

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

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

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

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

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

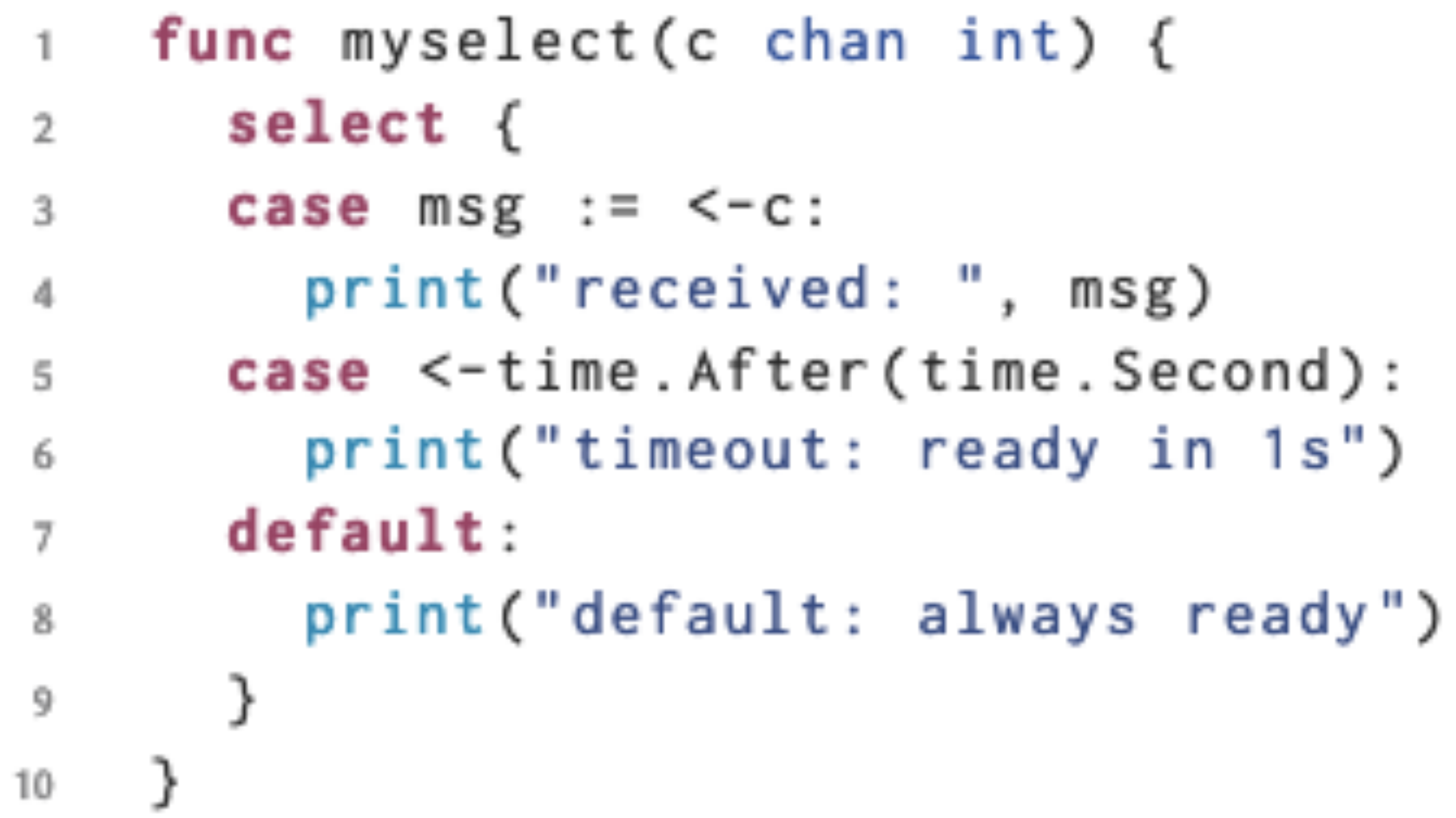
$$\psi_{l_b} \overset{\text{def}}{=} (\textstyle\bigwedge_{\tilde{a}\in\mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}) \implies \Phi(\langle\{\tau_a \mid a \in \tilde{a}\}\rangle\top) \qquad \text{[Liveness (b)]}$$

$$\psi_s \overset{\text{def}}{=} (\textstyle\bigwedge_{a\in\mathcal{A}} \downarrow_{a^\star}) \implies \neg(\downarrow_{\overline{a}} \vee \downarrow_{\mathtt{clo}\, a}) \qquad \text{[Channel safety]}$$

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

Model Checking Go Types

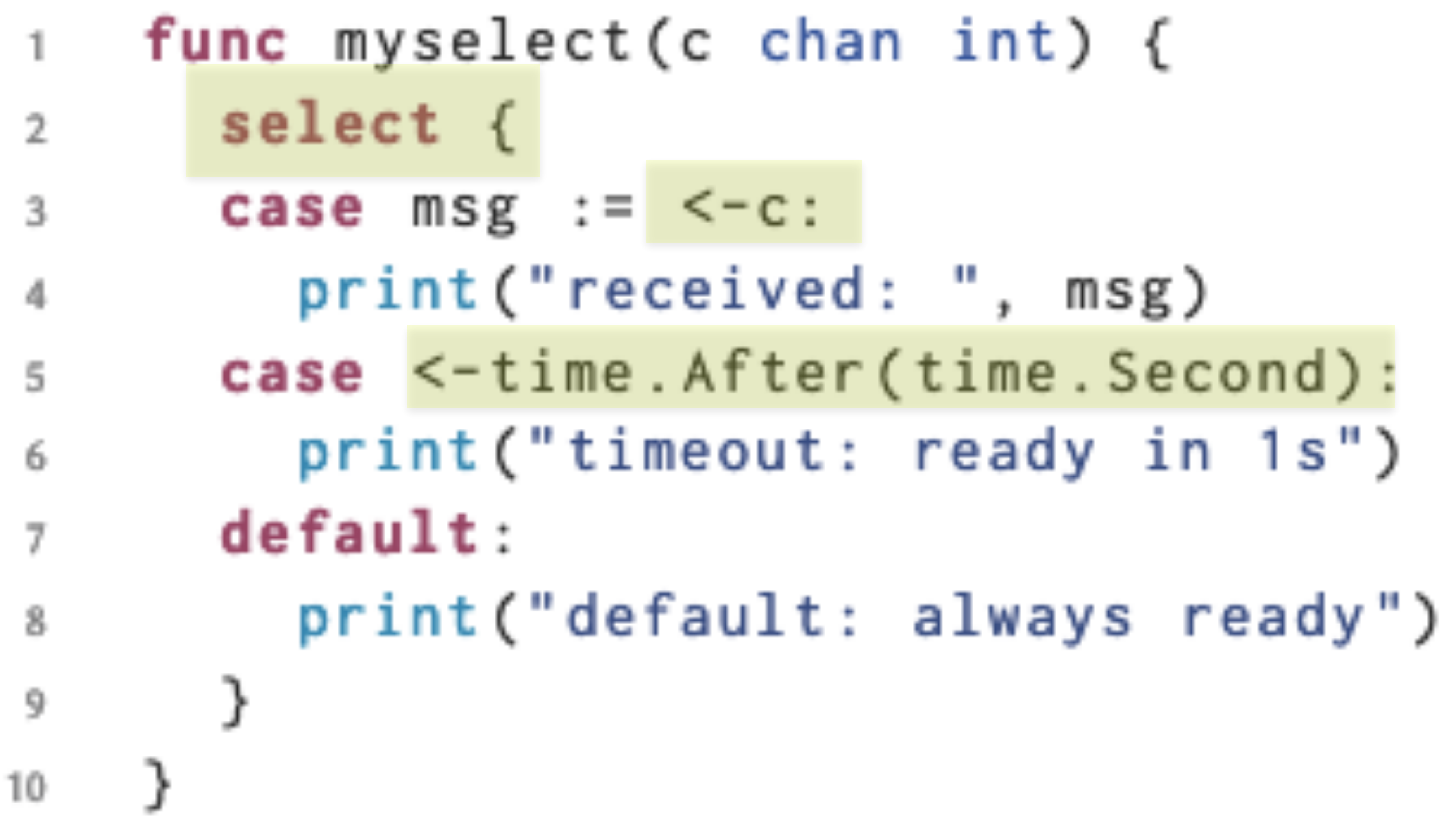
- Analyze Go programs in **SSA form:**



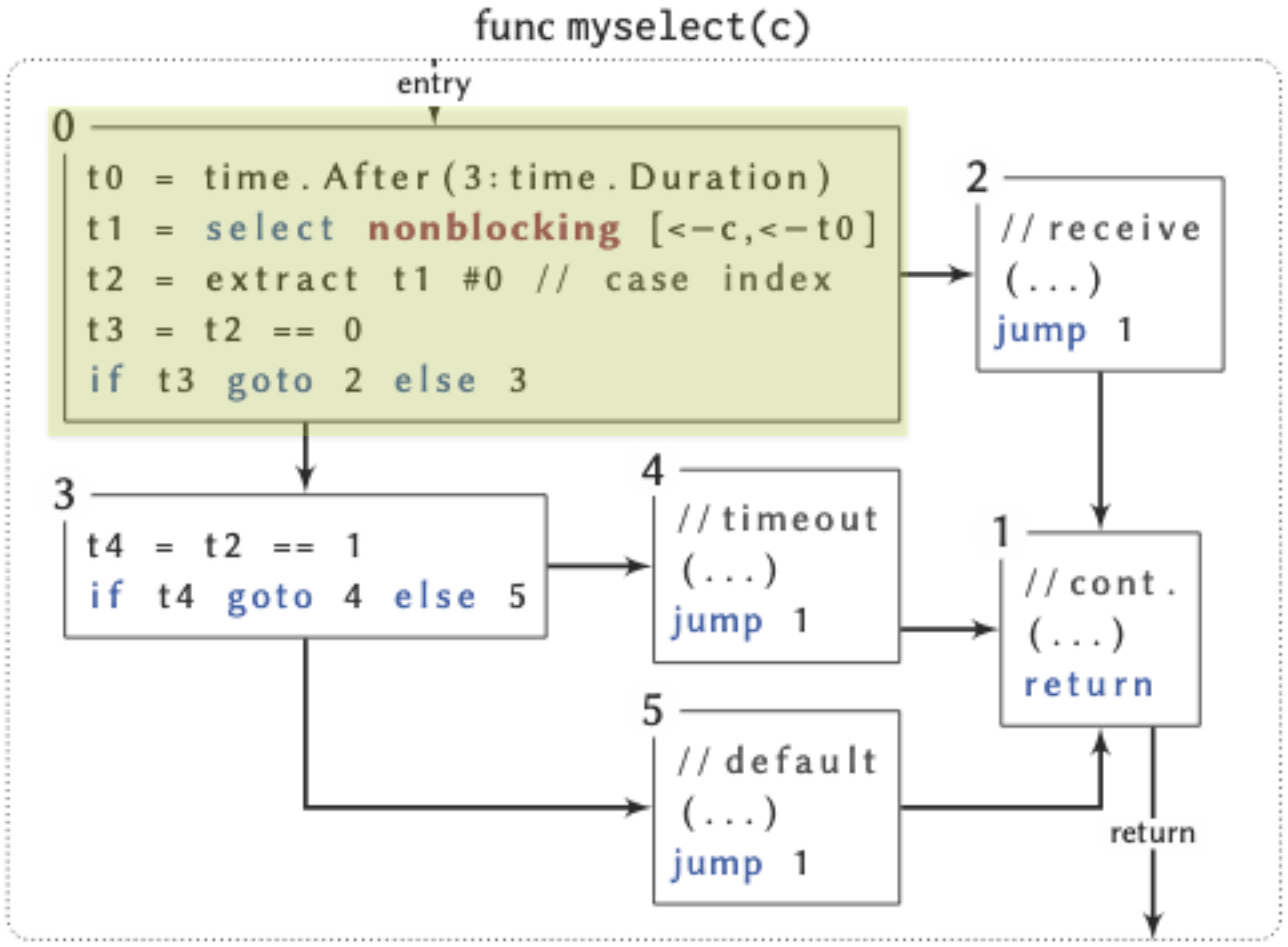
```go
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```

Model Checking Go Types

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



```
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```



```
0
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<−c,<−t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3
```

```
2
//receive
(...)
jump 1
```

```
3
t4 = t2 == 1
if t4 goto 4 else 5
```

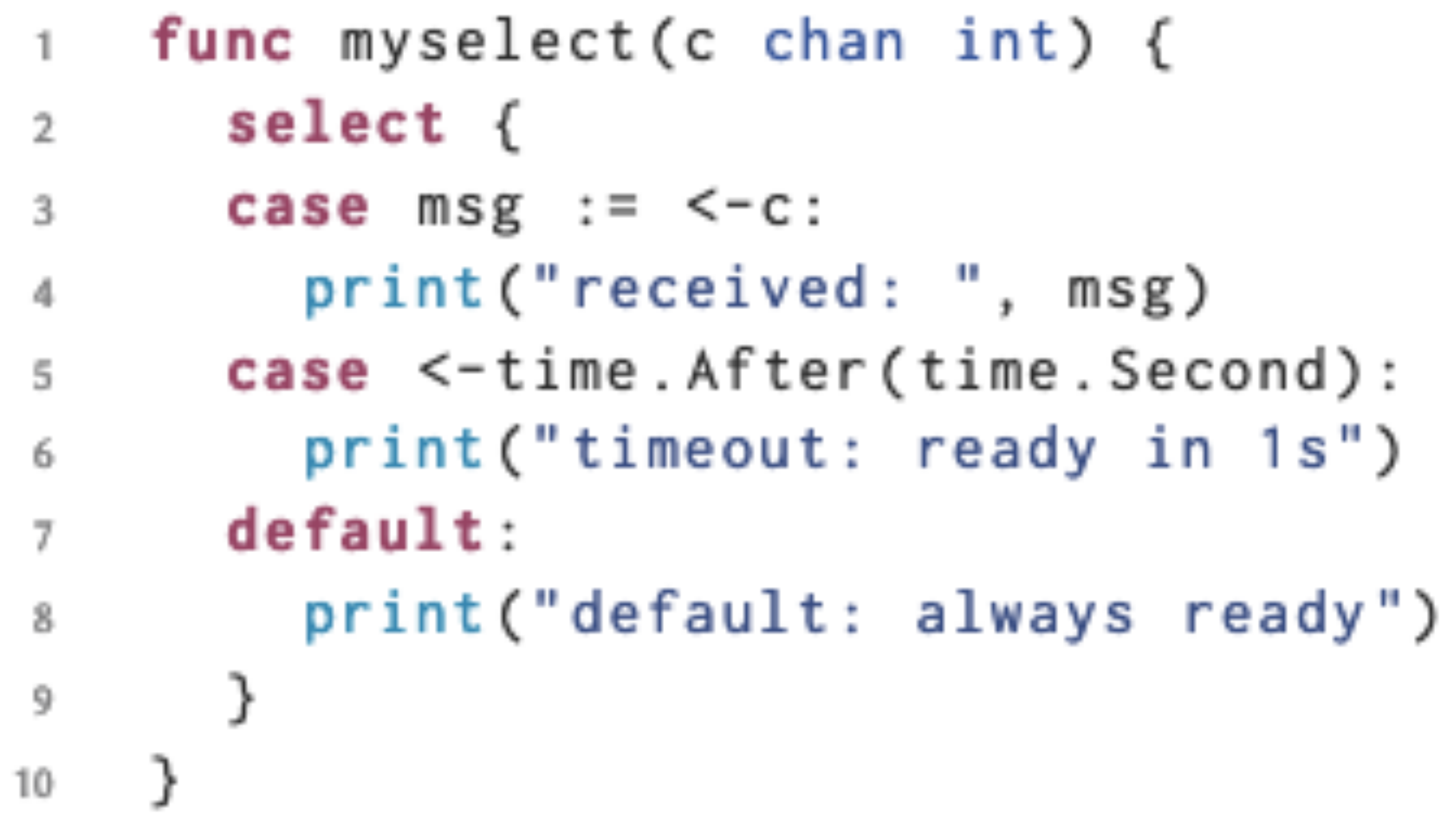
```
4
//timeout
(...)
jump 1
```

```
1
//cont.
(...)
return
```

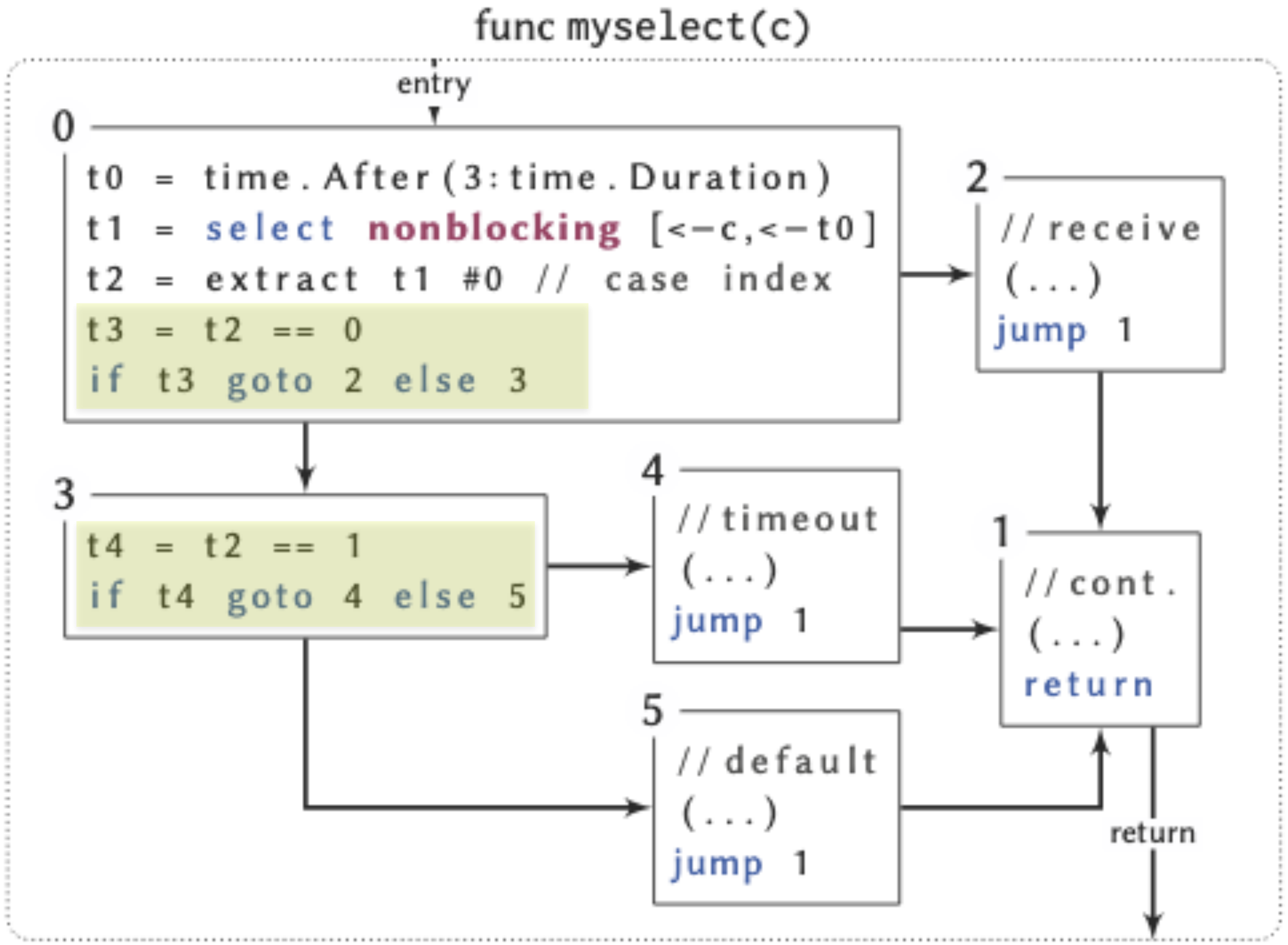
```
5
//default
(...)
jump 1
```

Model Checking Go Types

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



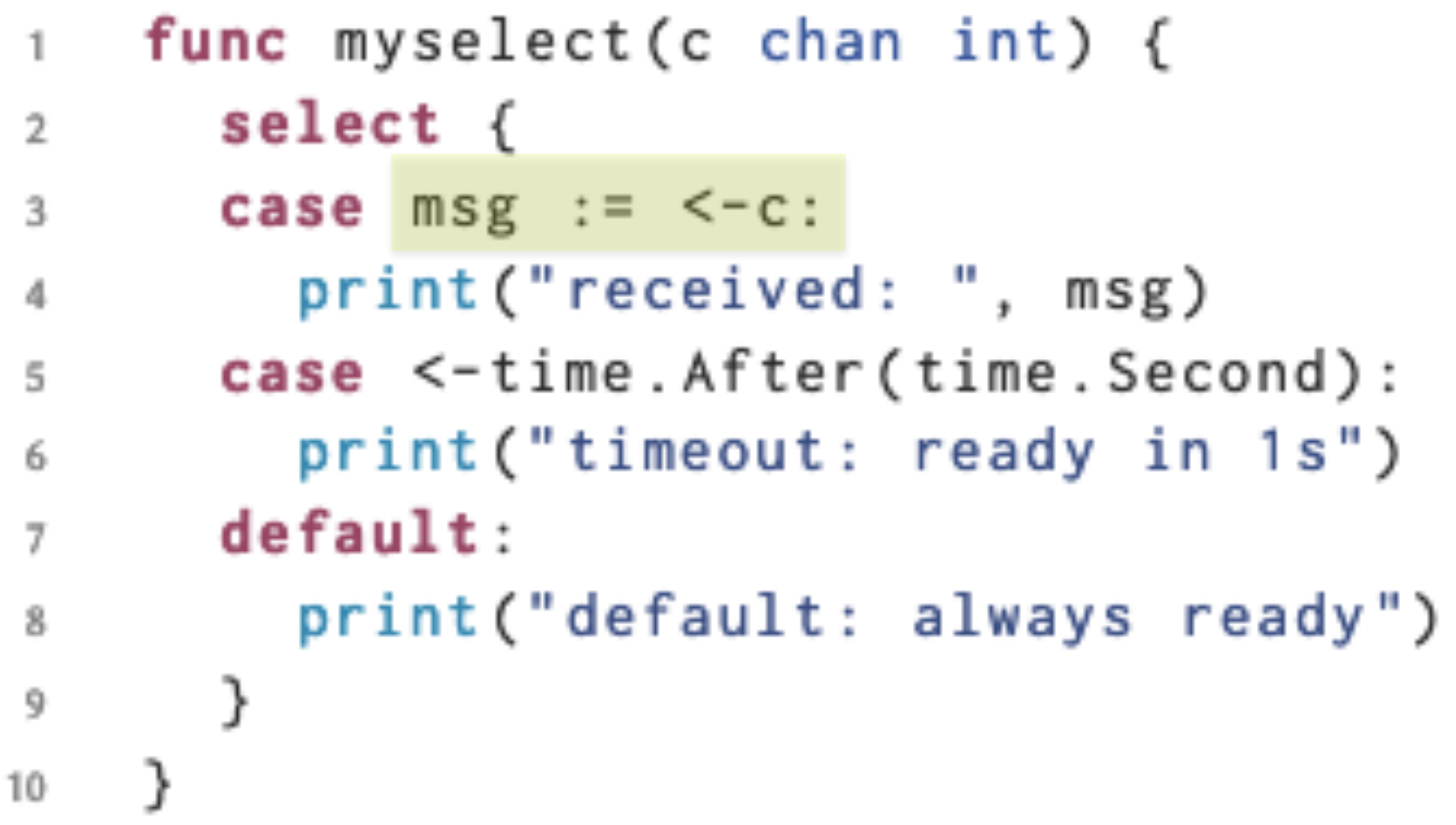
```
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```



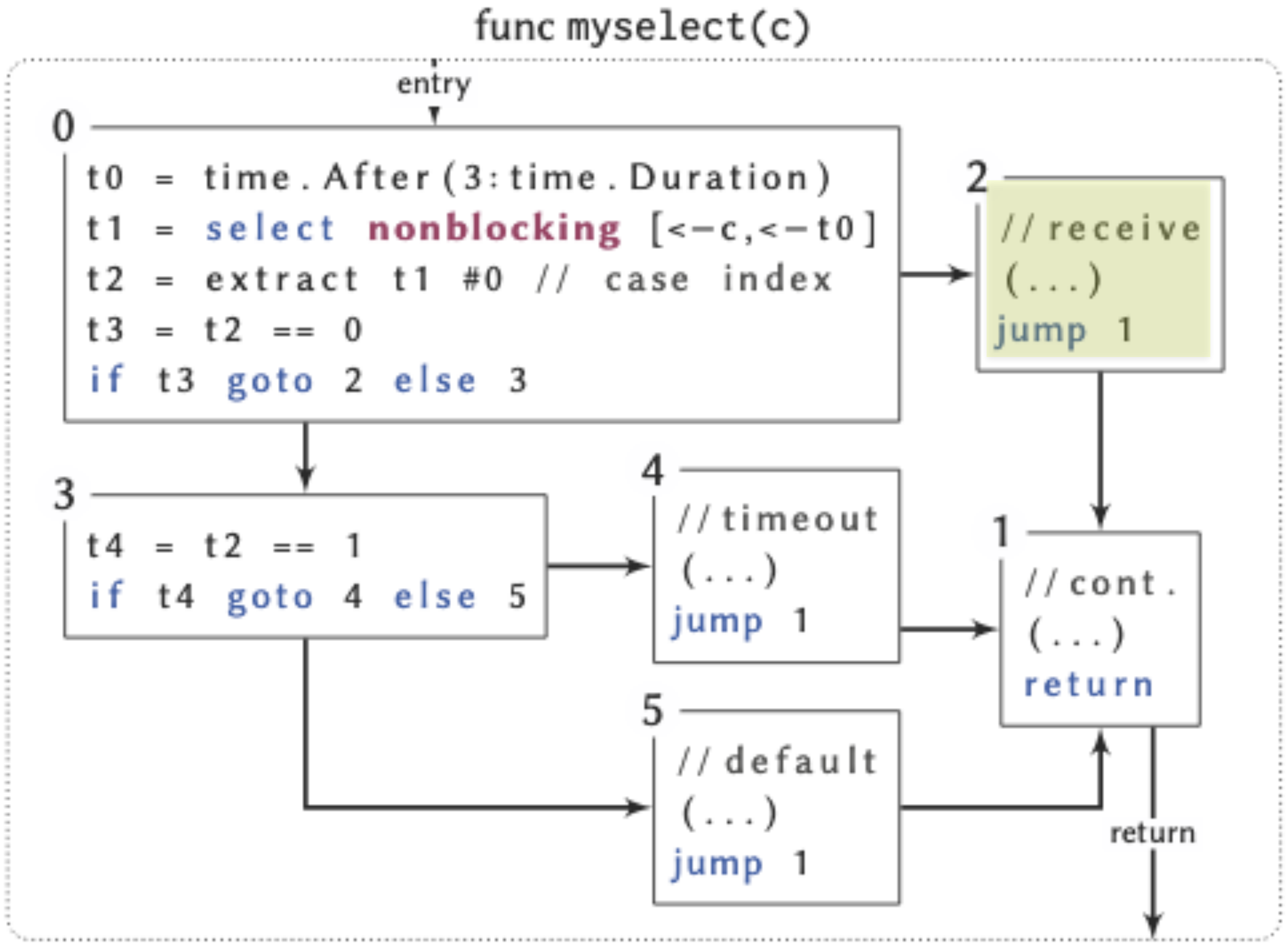
```
0
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<−c,<−t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3

2
//receive
(...)
jump 1

3
t4 = t2 == 1
if t4 goto 4 else 5

4
//timeout
(...)
jump 1

1
//cont.
(...)
return

5
//default
(...)
jump 1
```

Model Checking Go Types

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



```go
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```



```
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<−c,<−t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3
```

```
// receive
(...)
jump 1
```

```
t4 = t2 == 1
if t4 goto 4 else 5
```

```
// timeout
(...)
jump 1
```

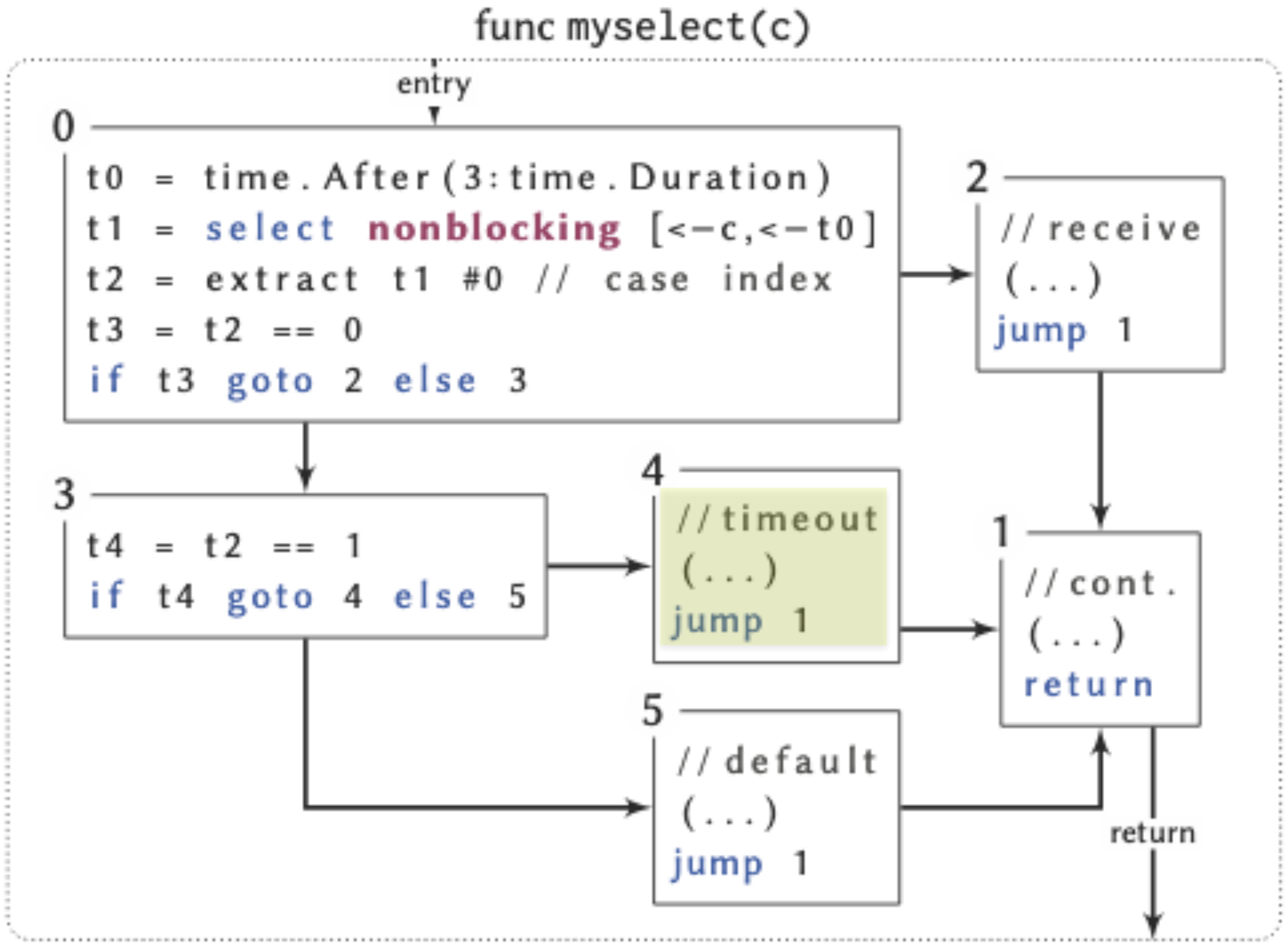
```
// cont.
(...)
return
```

```
// default
(...)
jump 1
```

Model Checking Go Types

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

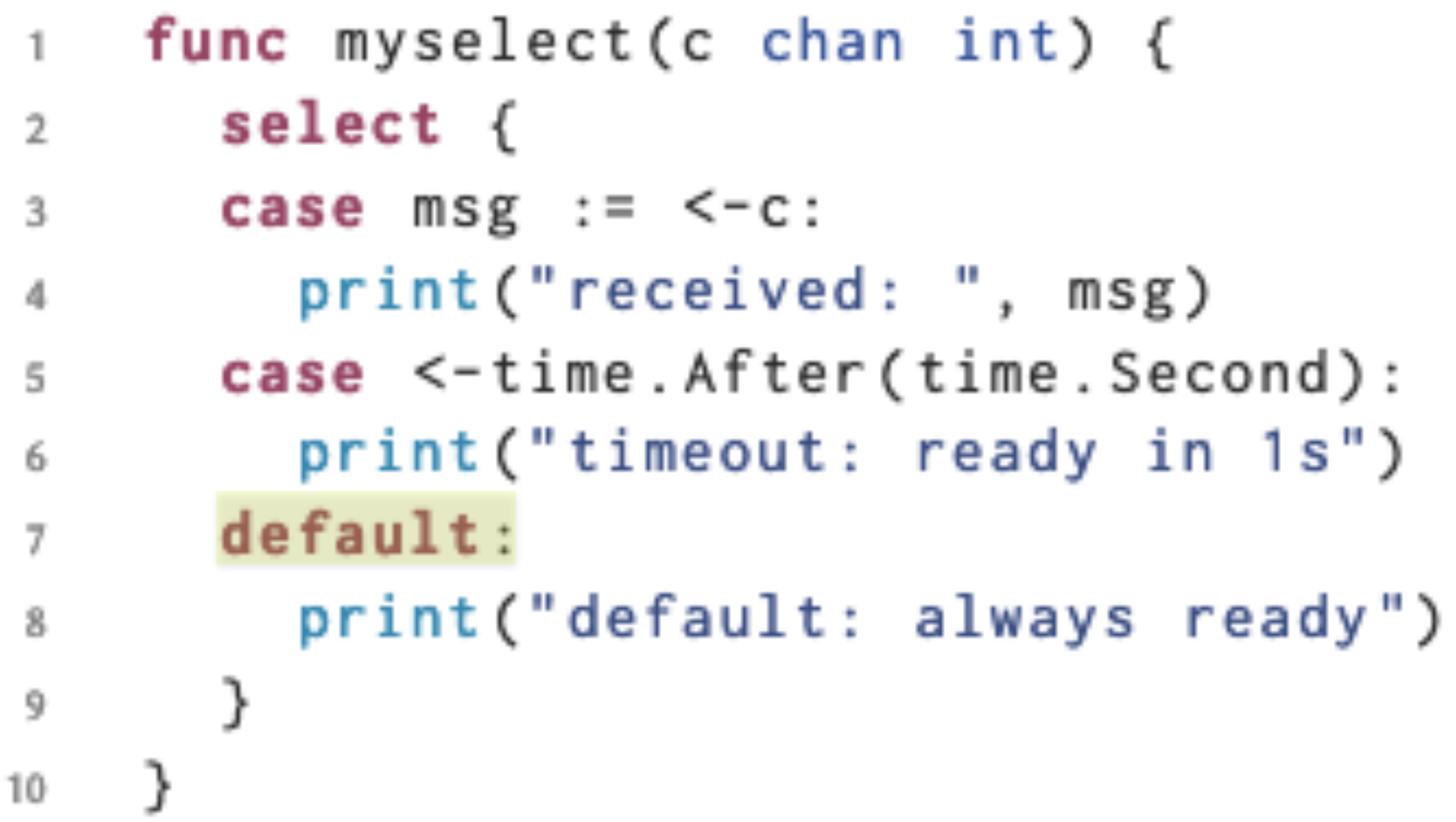
```go
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```



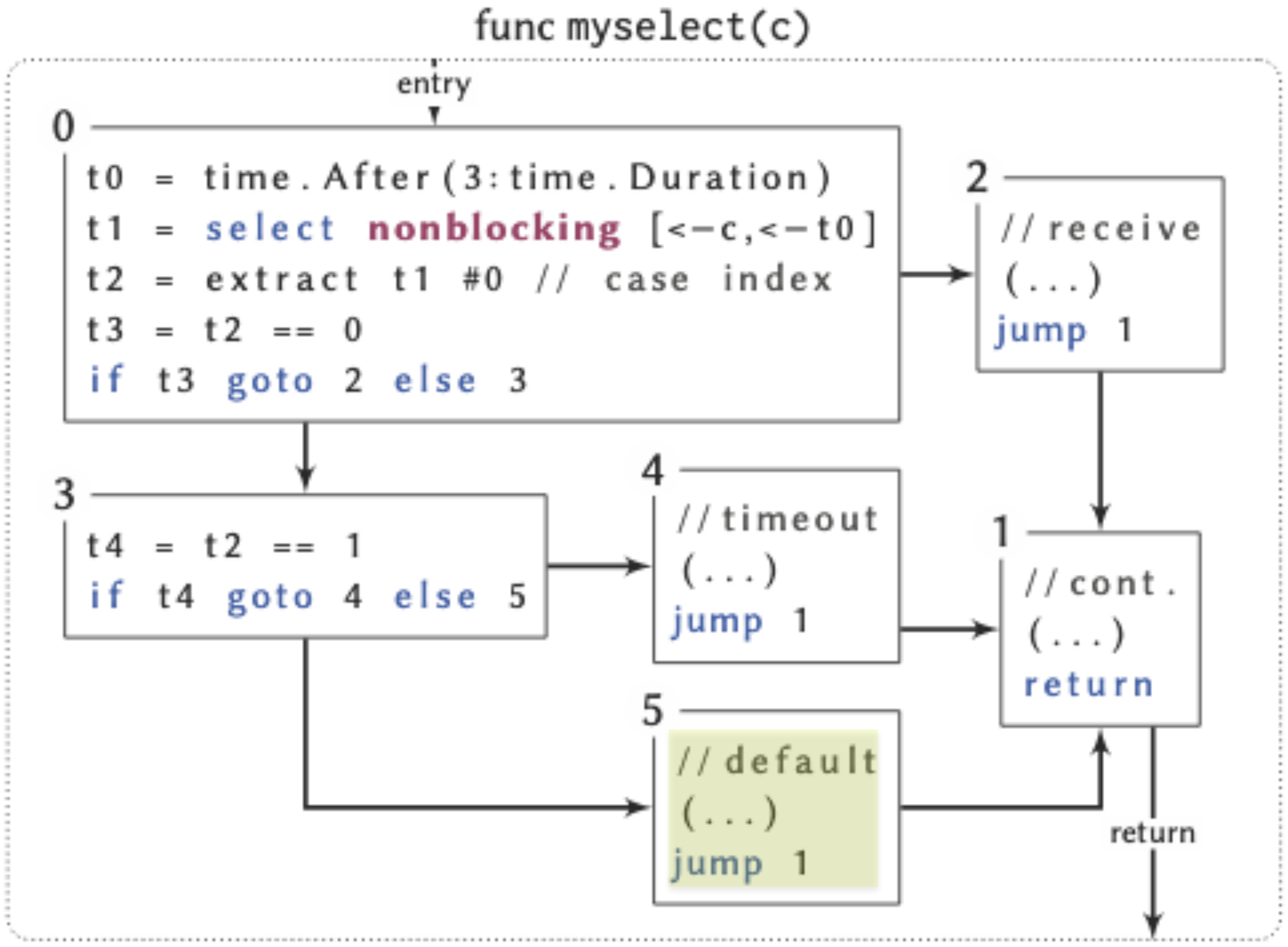
```
0
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<−c,<−t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3

2
//receive
(...)
jump 1

3
t4 = t2 == 1
if t4 goto 4 else 5

4
//timeout
(...)
jump 1

1
//cont.
(...)
return

5
//default
(...)
jump 1
```

Model Checking Go Types

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



```go
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```



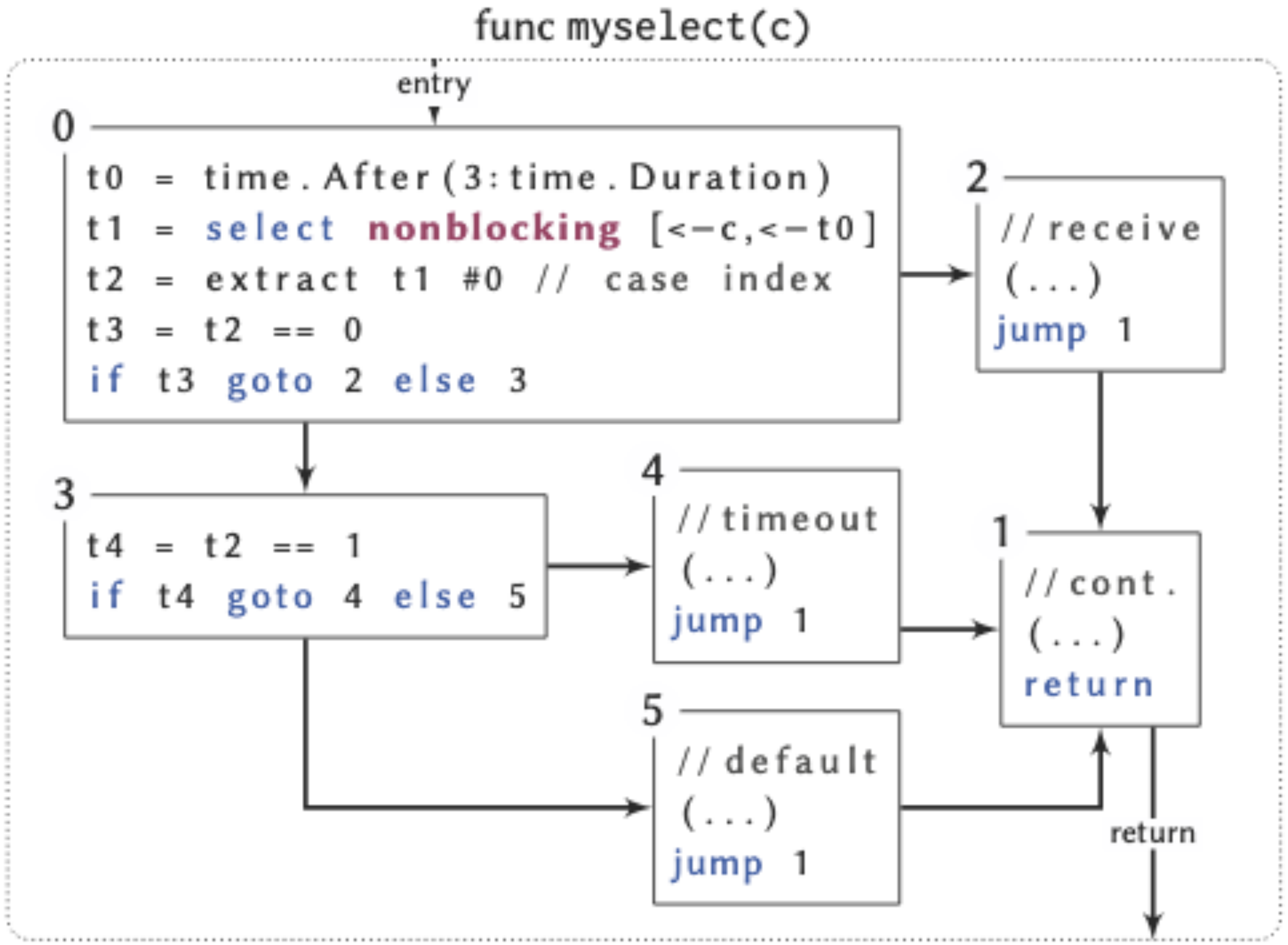
```
0:
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<-c,<-t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3

2:
//receive
(...)
jump 1

3:
t4 = t2 == 1
if t4 goto 4 else 5

4:
//timeout
(...)
jump 1

1:
//cont.
(...)
return

5:
//default
(...)
jump 1
```

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:



func myselect(c)

```
0
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<-c,<-t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3
```

```
2
//receive
(...)
jump 1
```

```
3
t4 = t2 == 1
if t4 goto 4 else 5
```

```
4
//timeout
(...)
jump 1
```

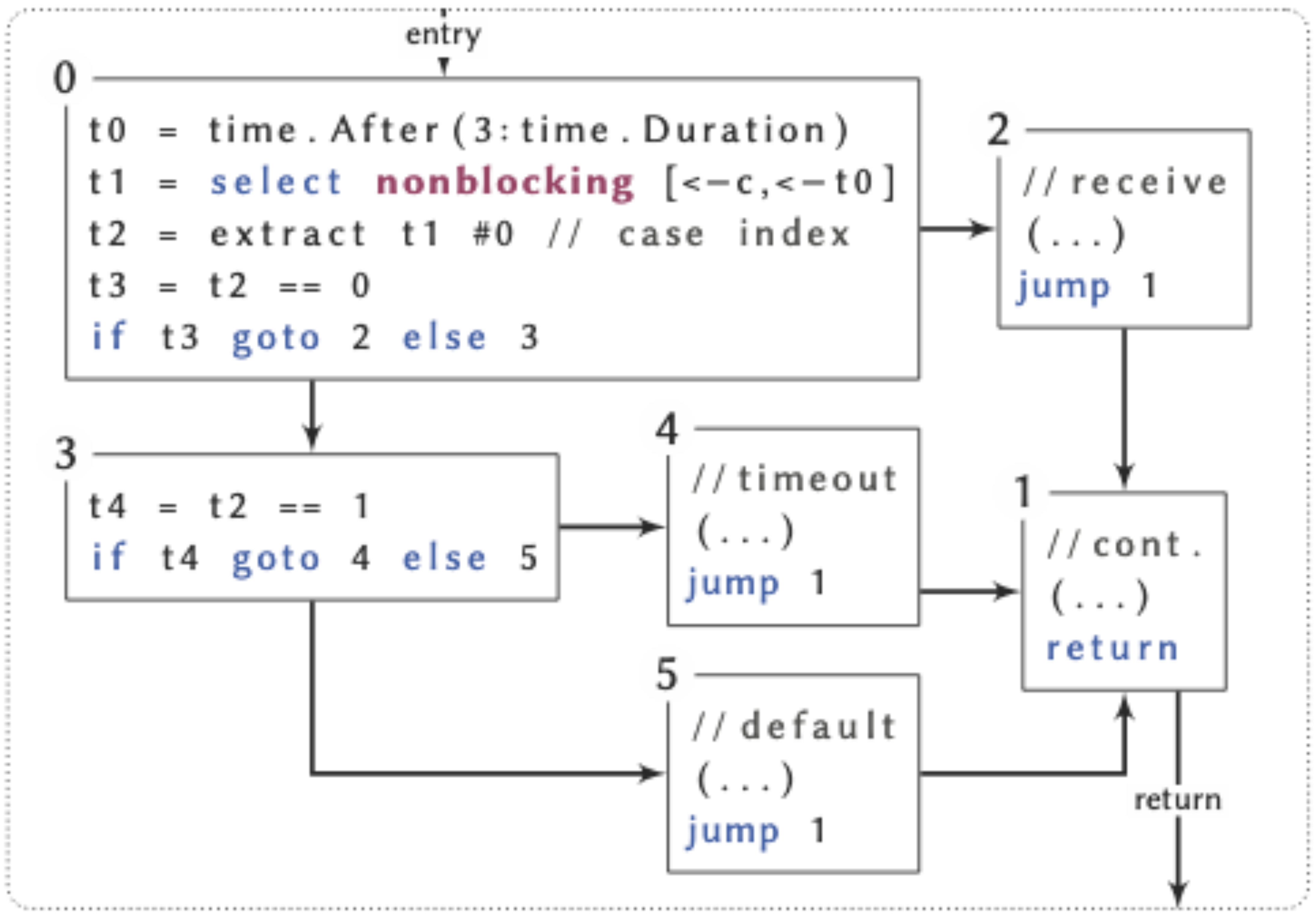
```
1
//cont.
(...)
return
```

```
5
//default
(...)
jump 1
```

Model Checking Go Types

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

func myselect(c)



```
0
t0 = time.After(3:time.Duration)
t1 = select nonblocking [<-c,<-t0]
t2 = extract t1 #0 // case index
t3 = t2 == 0
if t3 goto 2 else 3
```

```
2
//receive
(...)
jump 1
```

```
3
t4 = t2 == 1
if t4 goto 4 else 5
```

```
4
//timeout
(...)
jump 1
```

```
1
//cont.
(...)
return
```

```
5
//default
(...)
jump 1
```

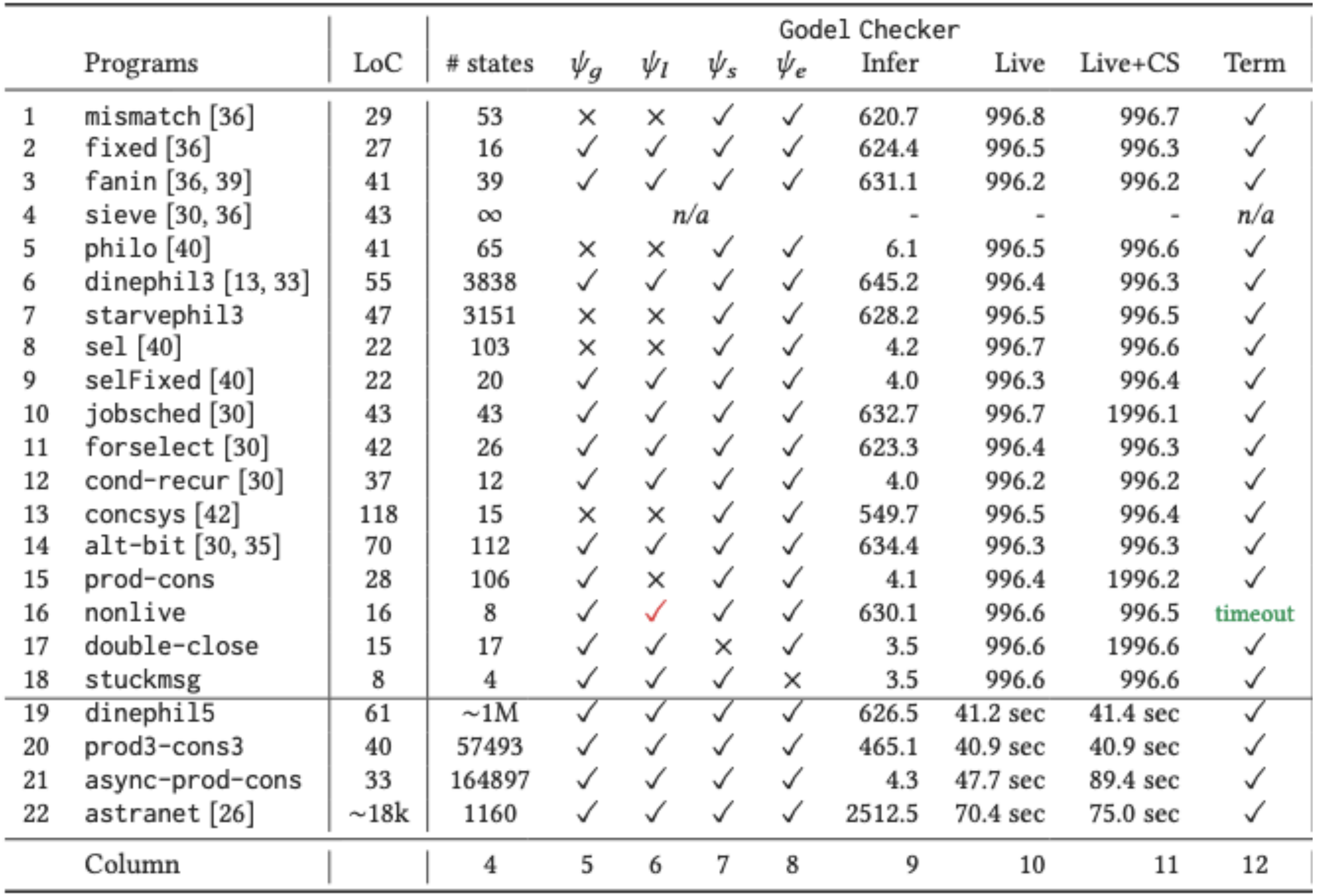
```go
func myselect(c chan int) {
  select {
  case msg := <-c:
    print("received: ", msg)
  case <-time.After(time.Second):
    print("timeout: ready in 1s")
  default:
    print("default: always ready")
  }
}
```

$$
\begin{aligned}
\boldsymbol{myselect_0}(c) &= \&\big\{c; \boldsymbol{myselect_2}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle,\\
&\qquad \tau; \boldsymbol{myselect_4}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle,\\
&\qquad \tau; \boldsymbol{myselect_5}\langle c\rangle; \boldsymbol{myselect_1}\langle c\rangle\big\}\\
\boldsymbol{myselect_i}(c) &= \ \mathbf{0} \qquad\qquad \text{for } \boldsymbol{i} \in \{1,2,4,5\}\\
\boldsymbol{myselect_3}(c) &= \ \boldsymbol{myselect_4}\langle c\rangle \oplus \boldsymbol{myselect_5}\langle c\rangle
\end{aligned}
$$

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	ψ_g	ψ_l	ψ_s	ψ_e	Godel Checker			
								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	∞	*n/a*				-	-	-	*n/a*
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, Rumyana Neykova, Nobuko Yoshida:
Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. FMSD 2015

[NHYA18] Rumyana 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