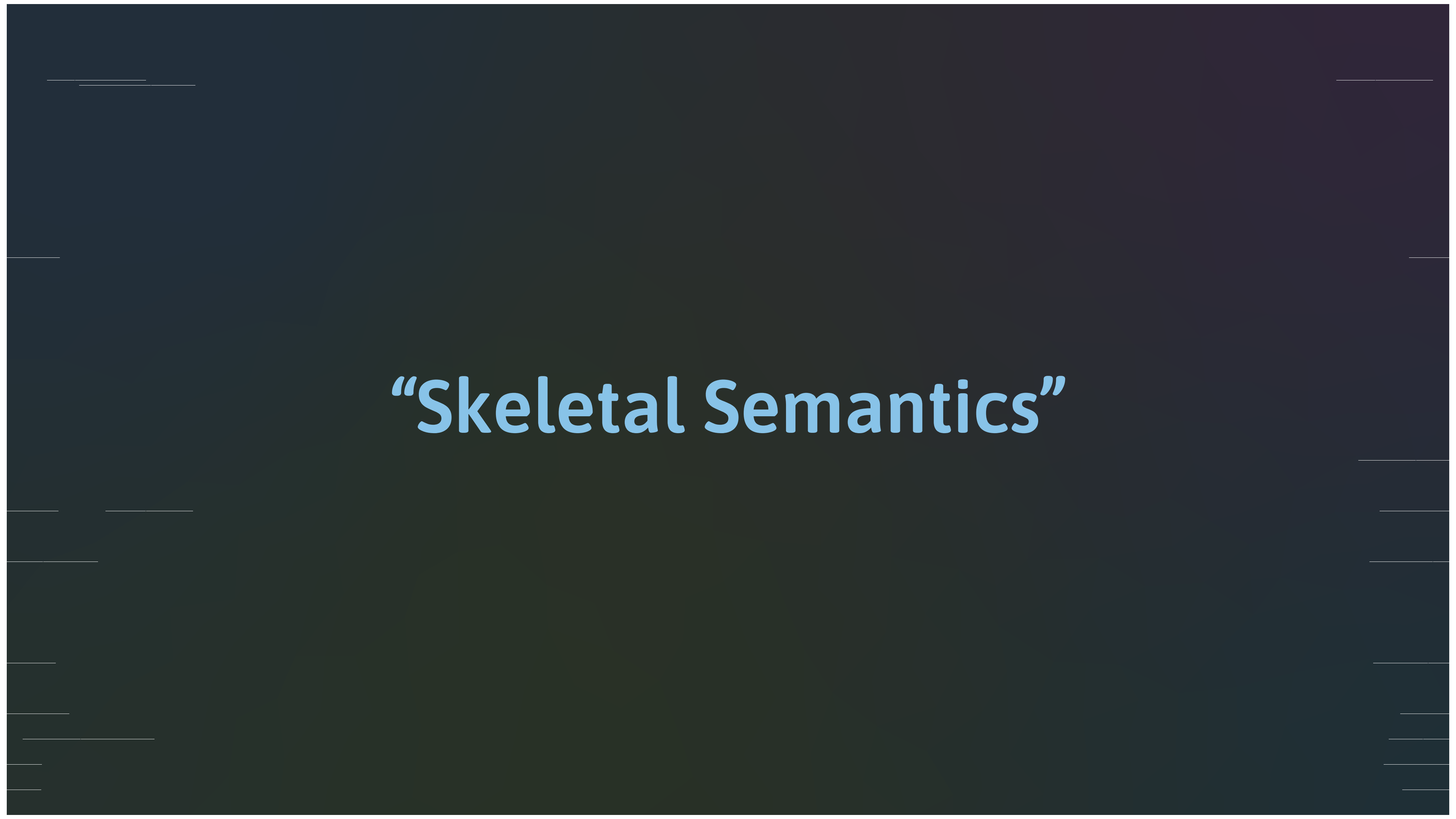
Greetings

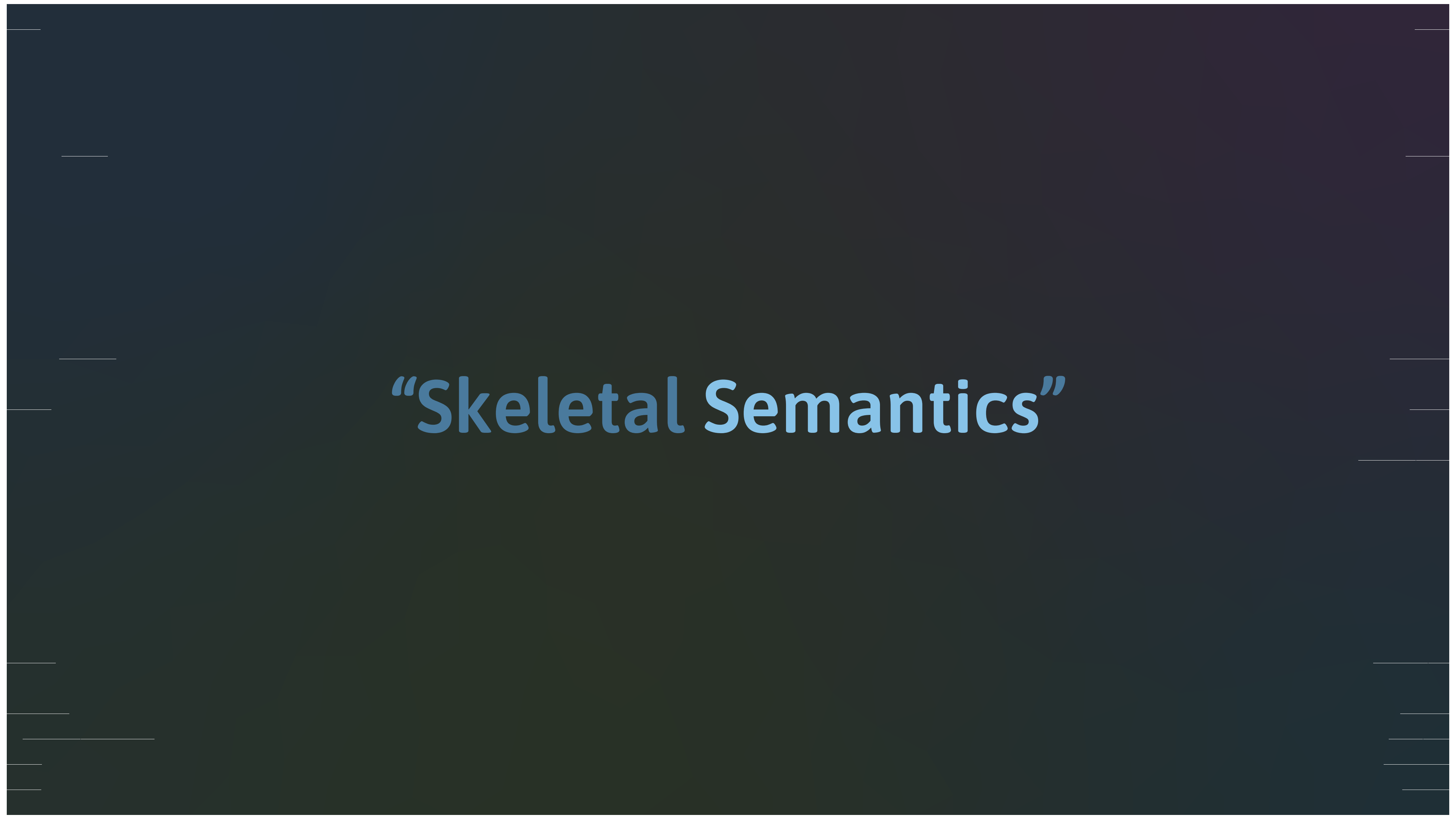
Name: Miguel Bartelsman

Project title: Skeletal semantics from actors to channels and back

Introduce theoretical backbone



First pillar: skeletal semantics



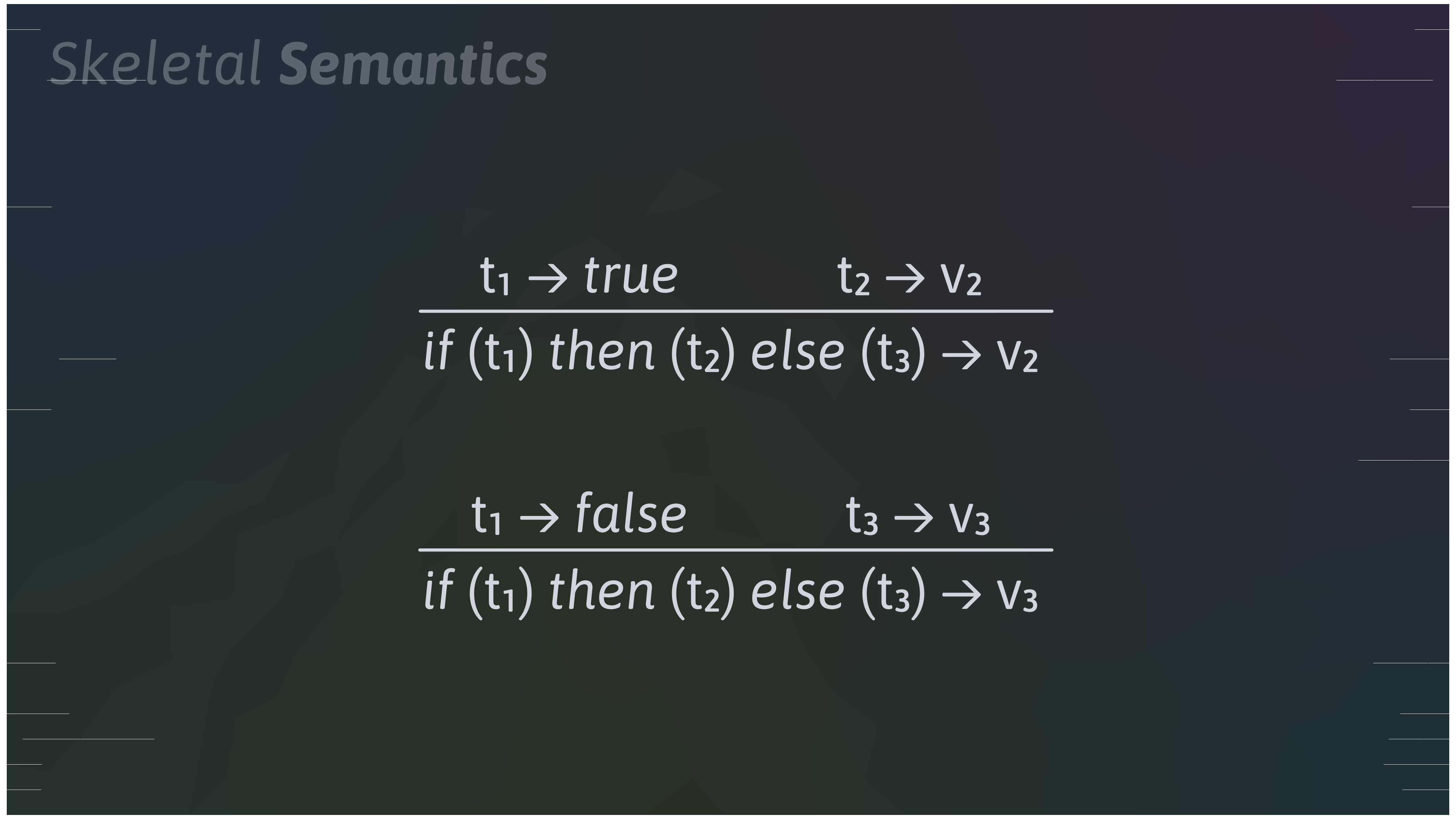
"First, let's analyze what we mean by semantics"

Study of the rigorous meaning behind programming languages.

Given a syntactic construct in a language, its semantics tell us its result upon evaluation, correctness, and other properties.

Three types: we'll focus on operational

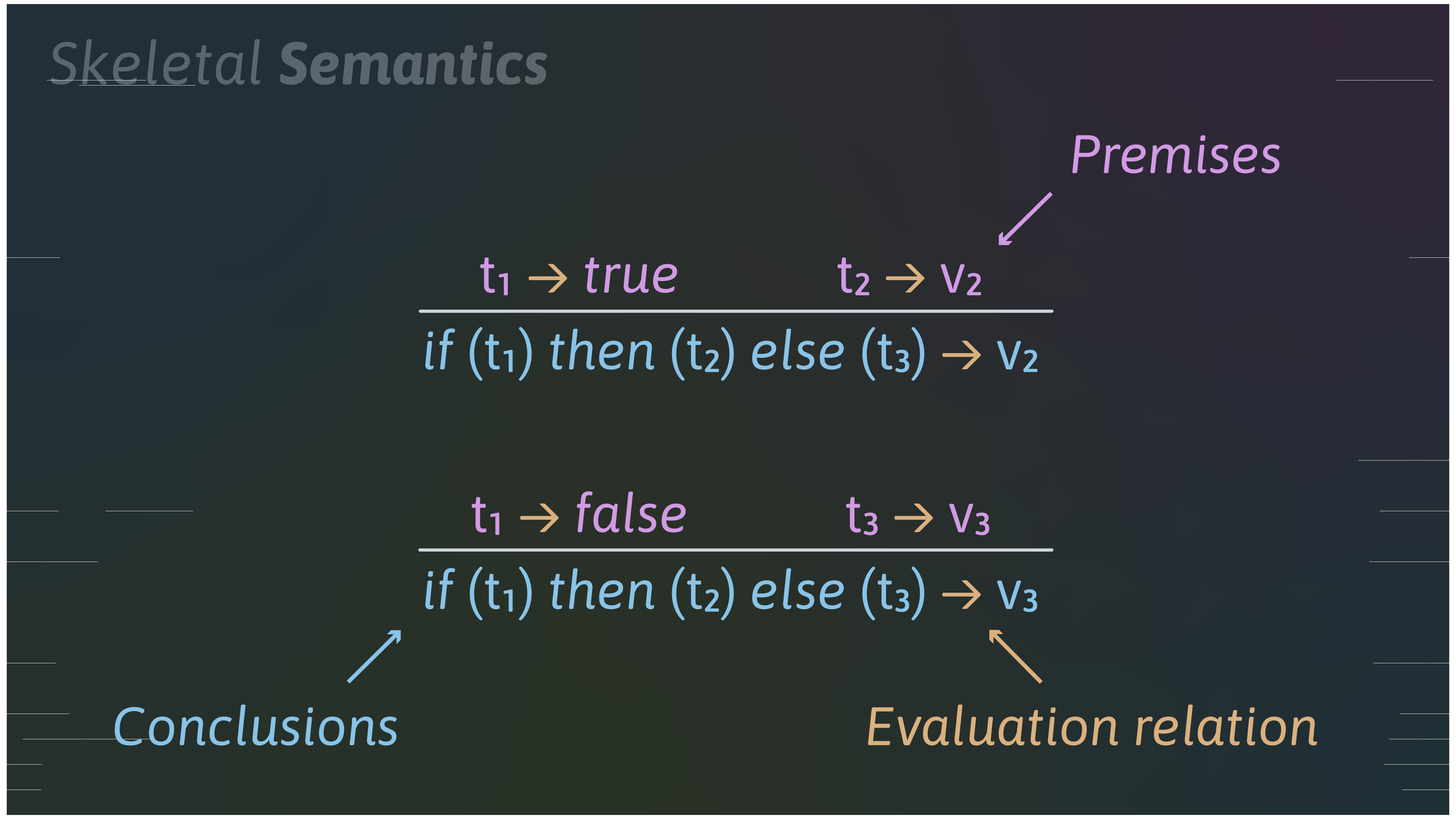
Operational: repeatedly applying a set of reduction rules to a program



Here are the operational semantics rules for a hypotethical if-statement.

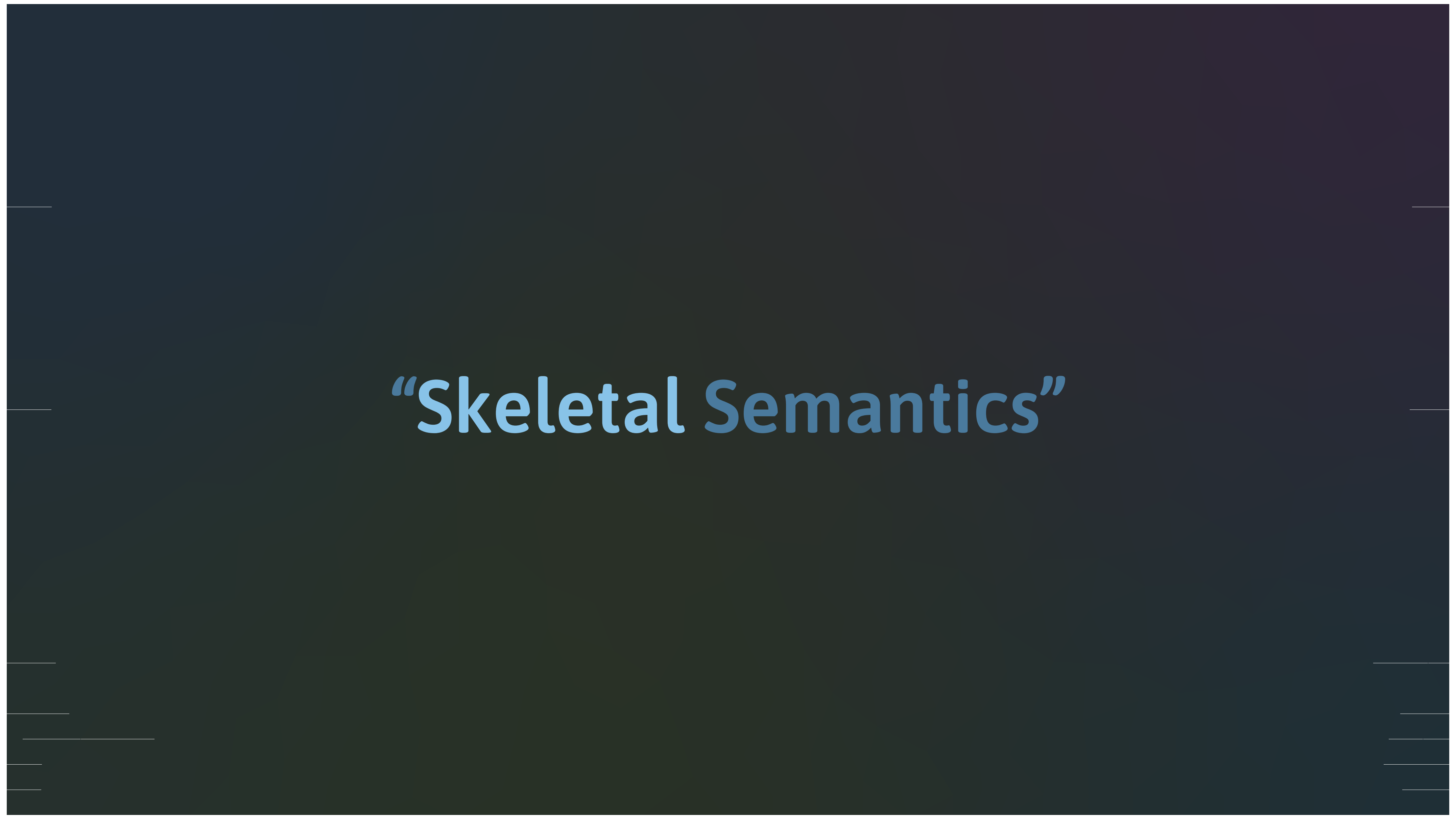
Top: guard is true

Bottom: guard is false



Explain the parts of a rule

Walk through one of them



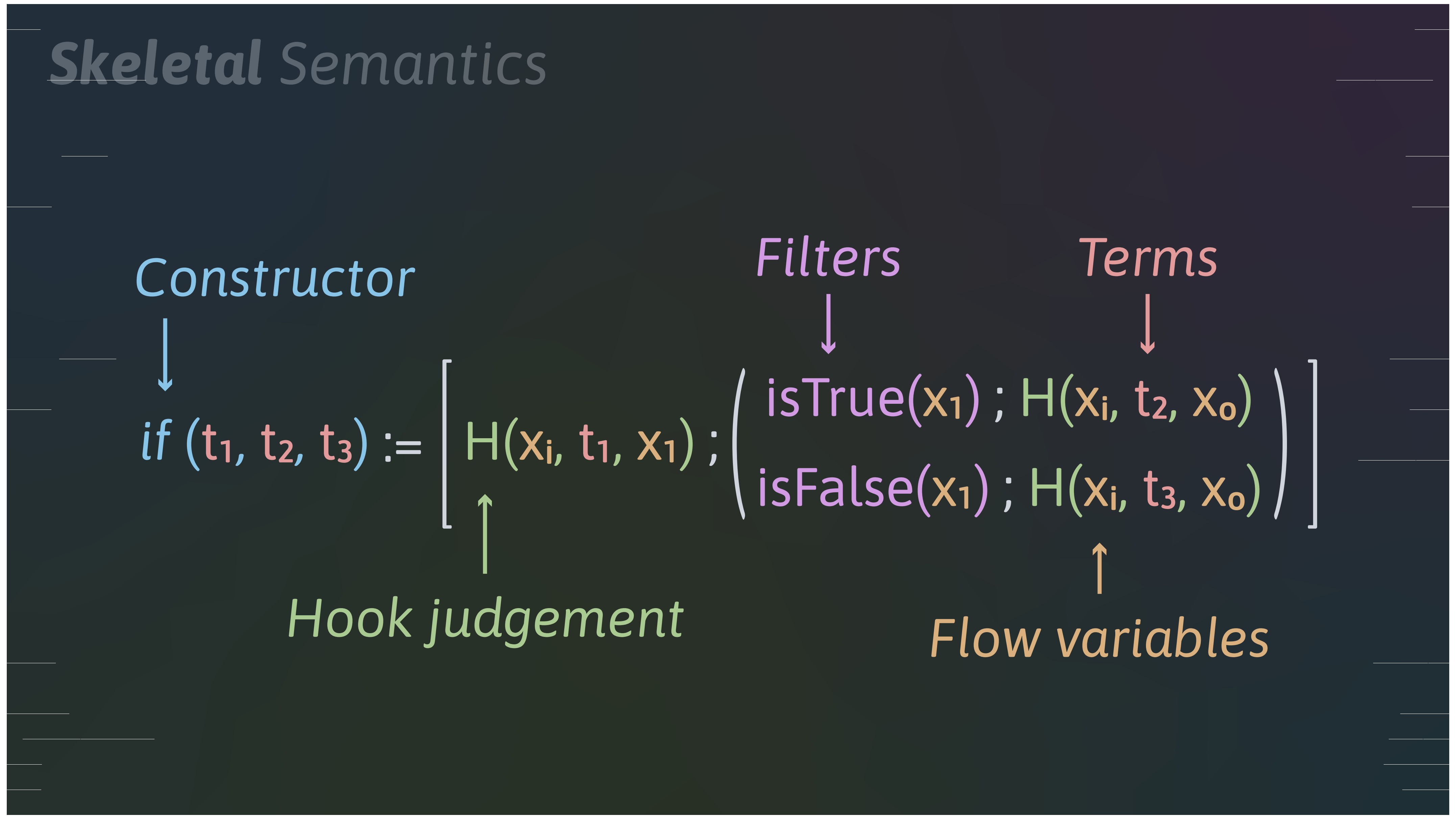
Skeletal semantics is a theoretical framework for describing the operational semantics of a programming language.

Many properties (shoutout to Laura)

Skeletal *Semantics*

$$
\mathit{if}\,(t_1, t_2, t_3) := \left[H(x_i, t_1, x_1) \,;\, \begin{pmatrix} \mathrm{isTrue}(x_1) \,;\, H(x_i, t_2, x_o) \\ \mathrm{isFalse}(x_1) \,;\, H(x_i, t_3, x_o) \end{pmatrix} \right]
$$

This rules describes an if statement.



Made up of filters and hooks

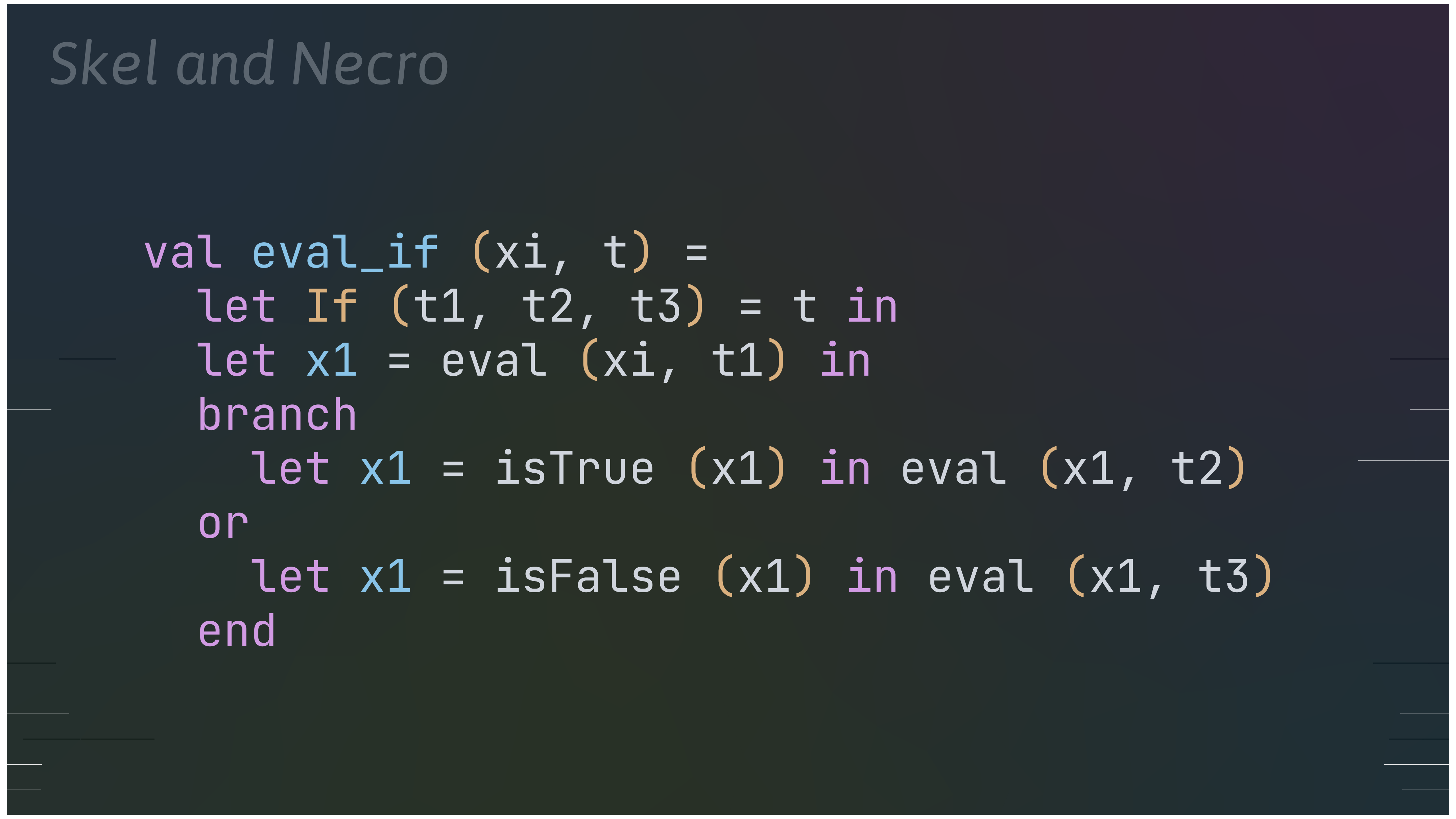
Branching

Flow variables

Constructor

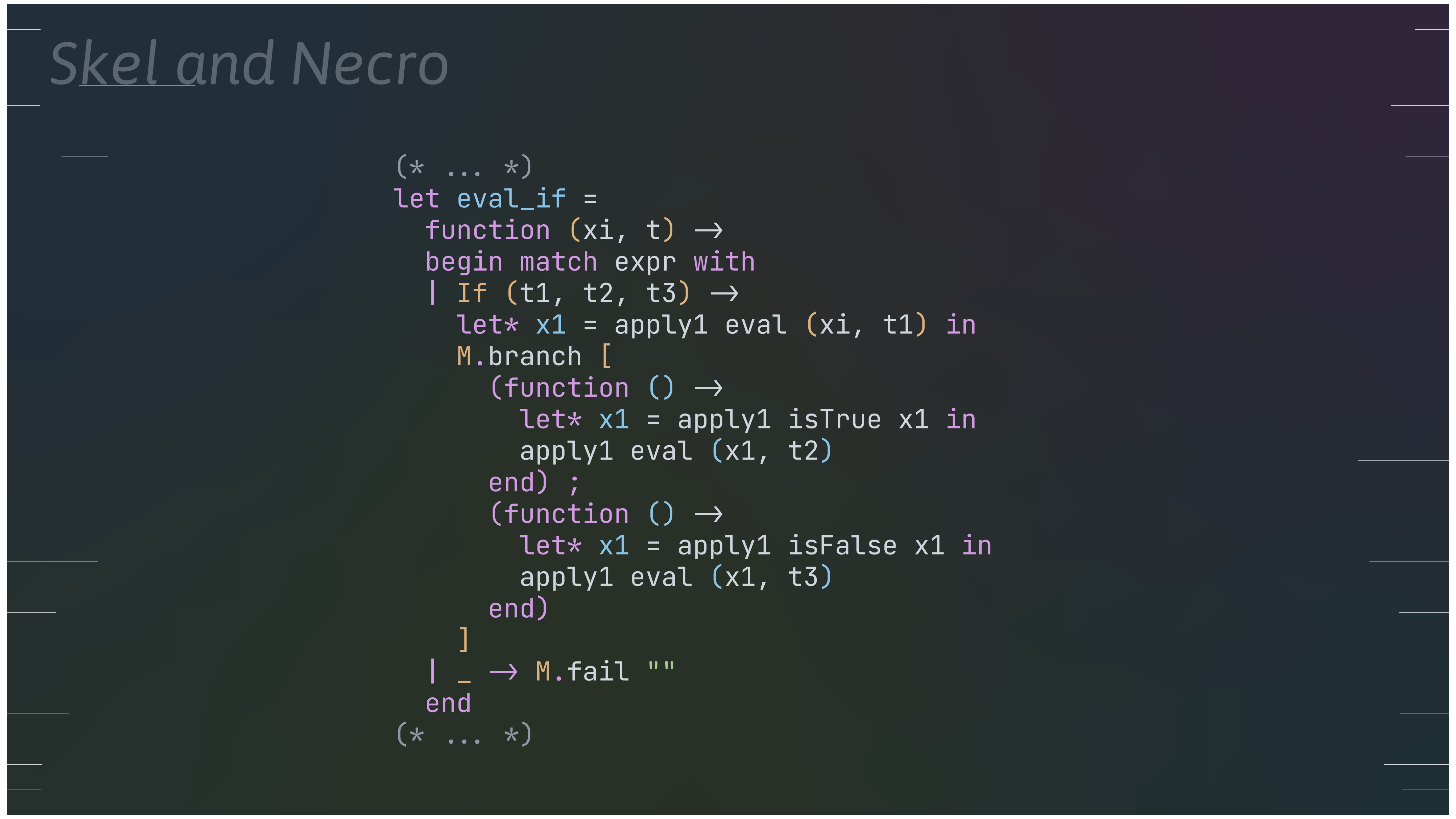
Properties of skeletal semantics: structured and sequential

Introduce necro

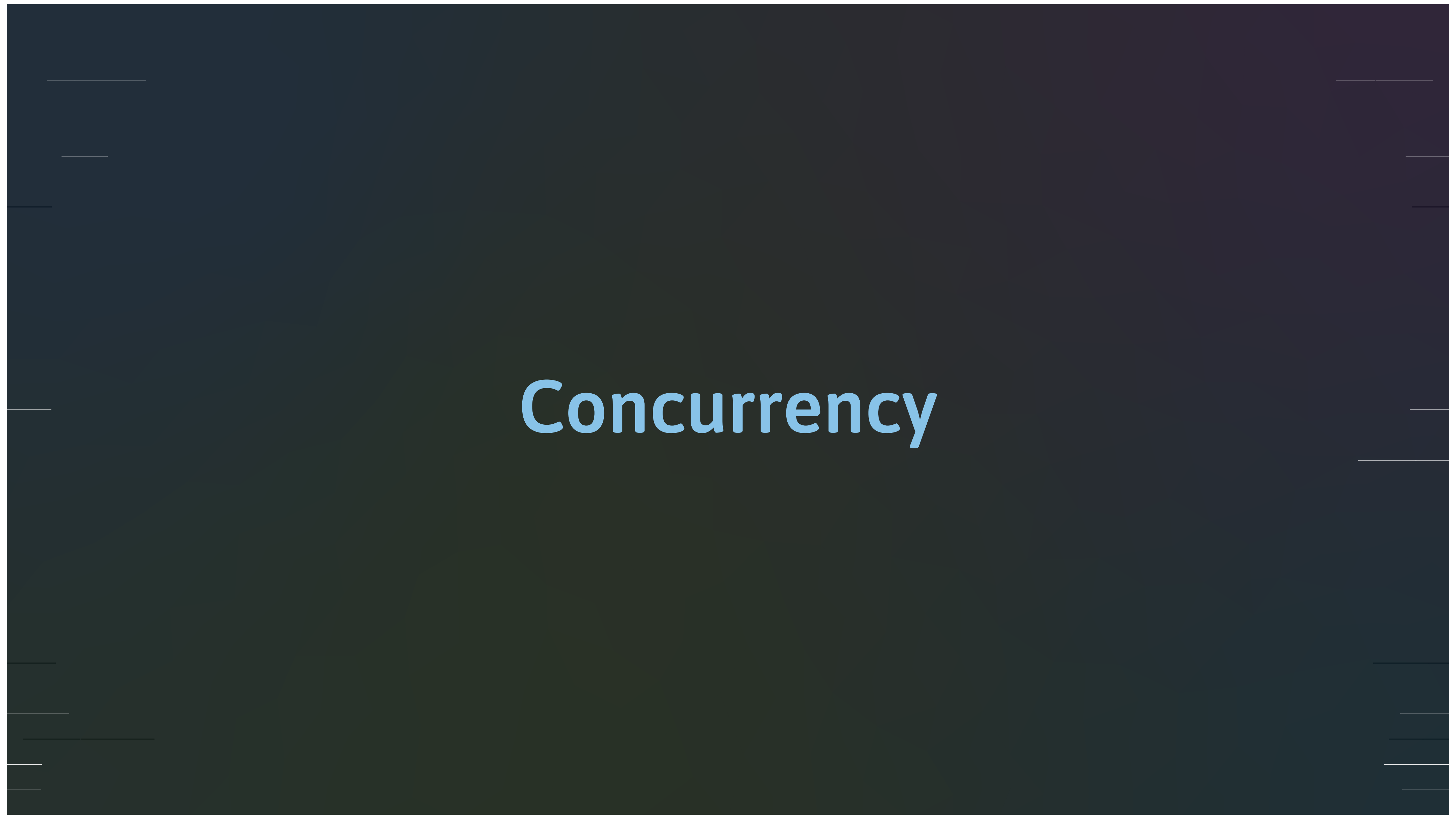


Here is the same rule, written in the DSL Skel

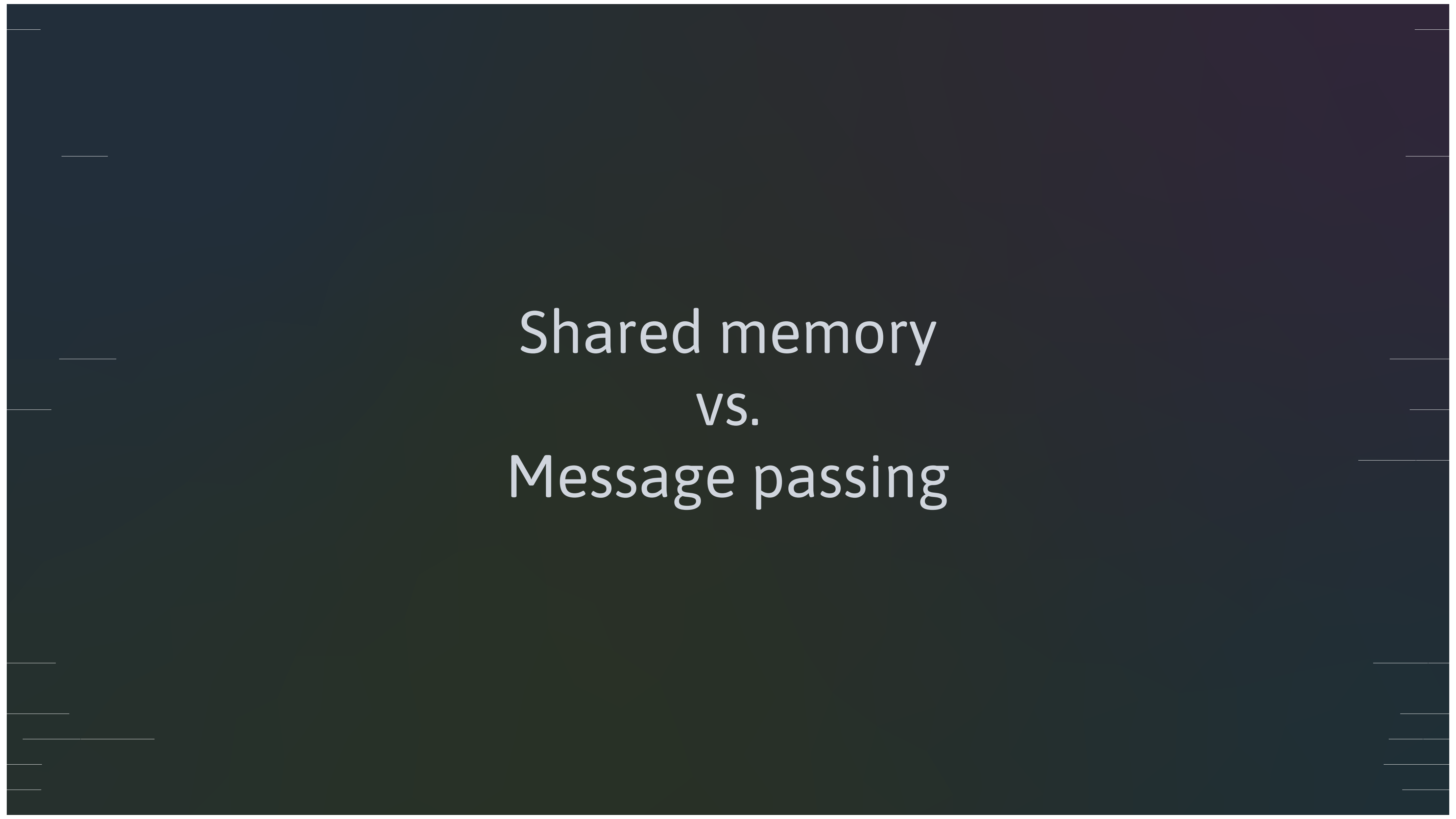
Walk through the parts



And here we have *a tiny part* of the result of compiling the rule into OCaml.



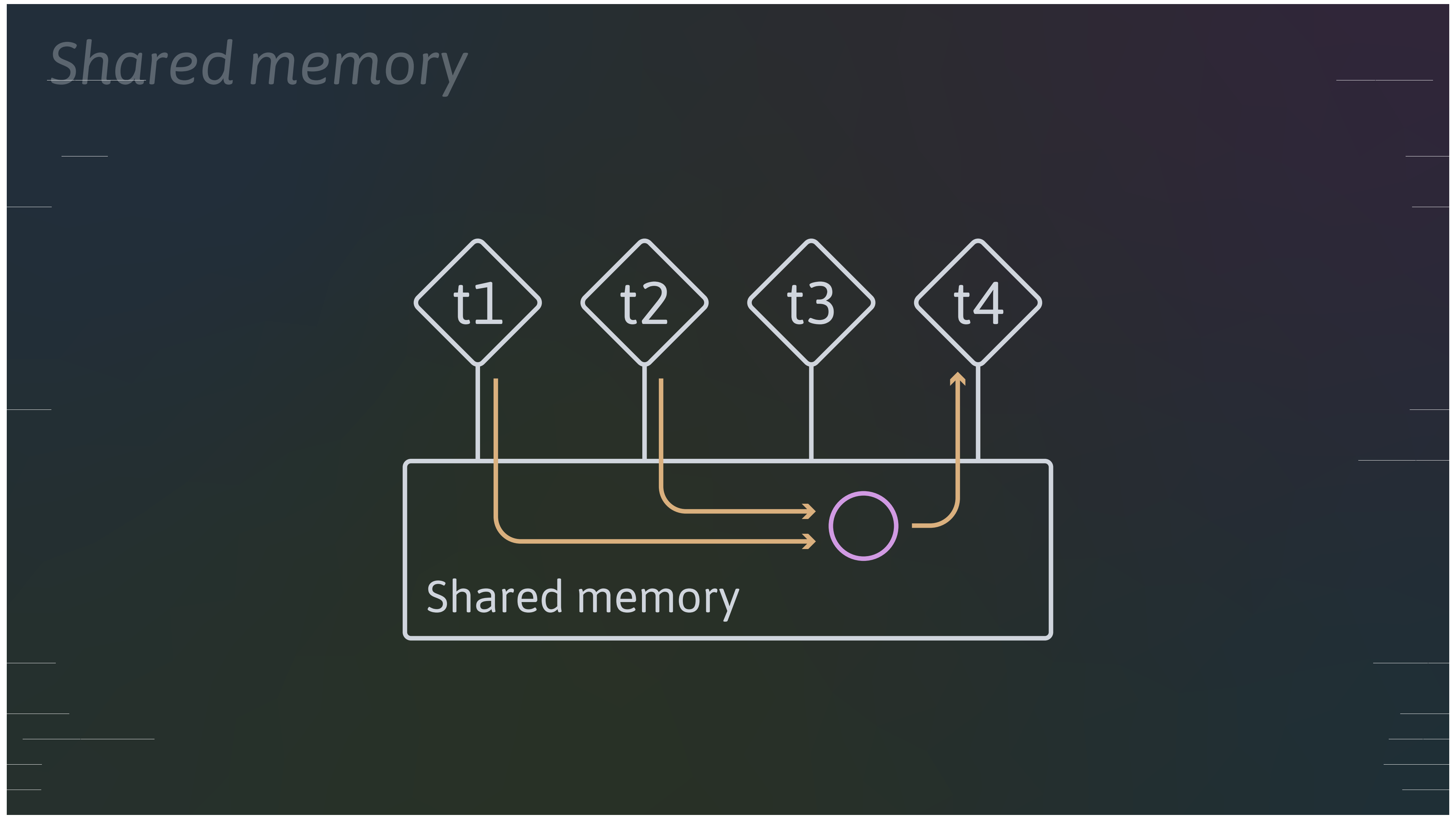
The second pillar to the project is concurrency



There are two big approaches to concurrency

Shared memory, which is what most (all?) OSes use for concurrent execution of programs.

Message passing, which sees greater use in distributed computing.

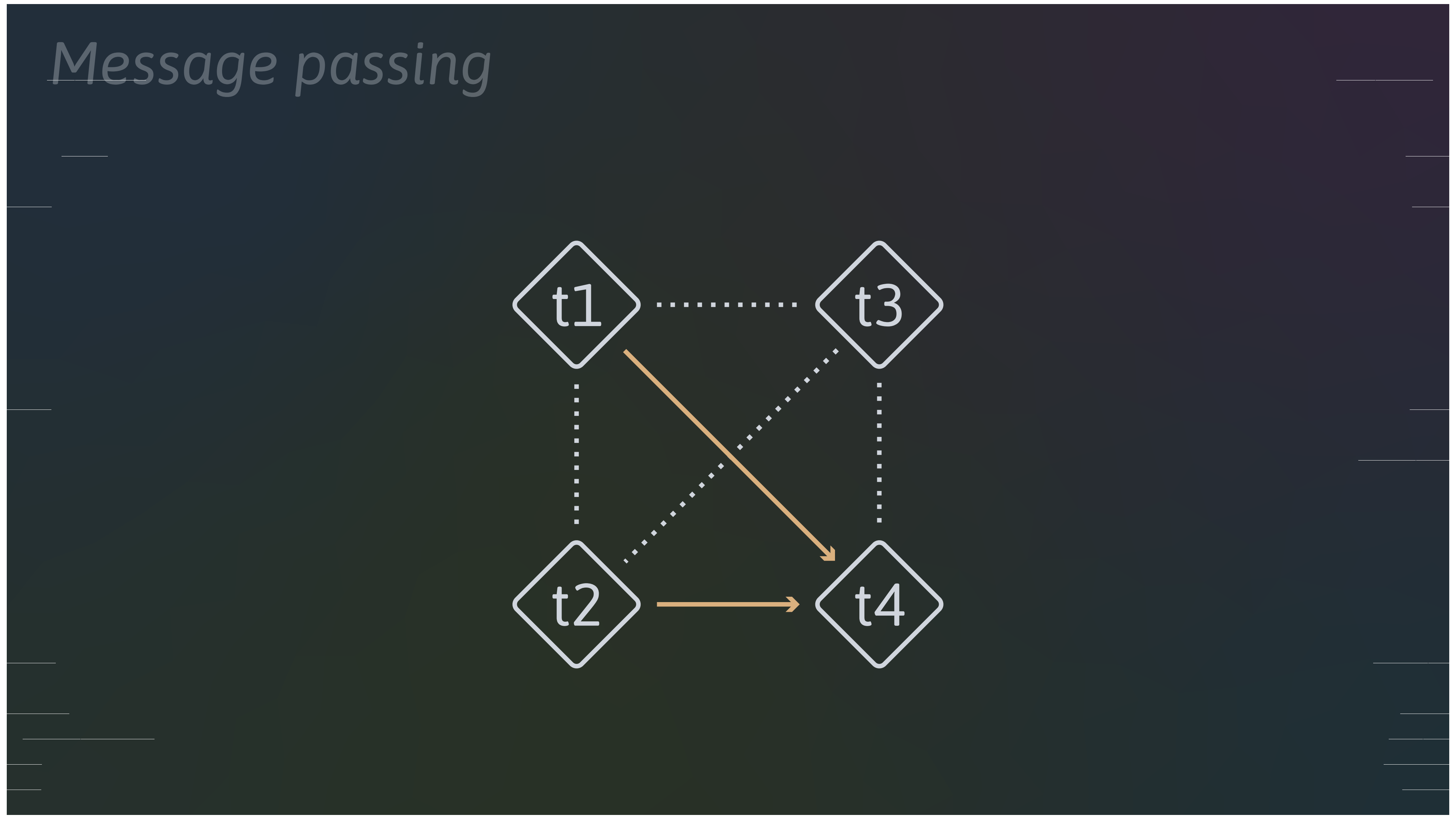


Shared memory as the name implies

Processes or threads share one common memory heap where they store data.

Communication by writing to and reading from heap

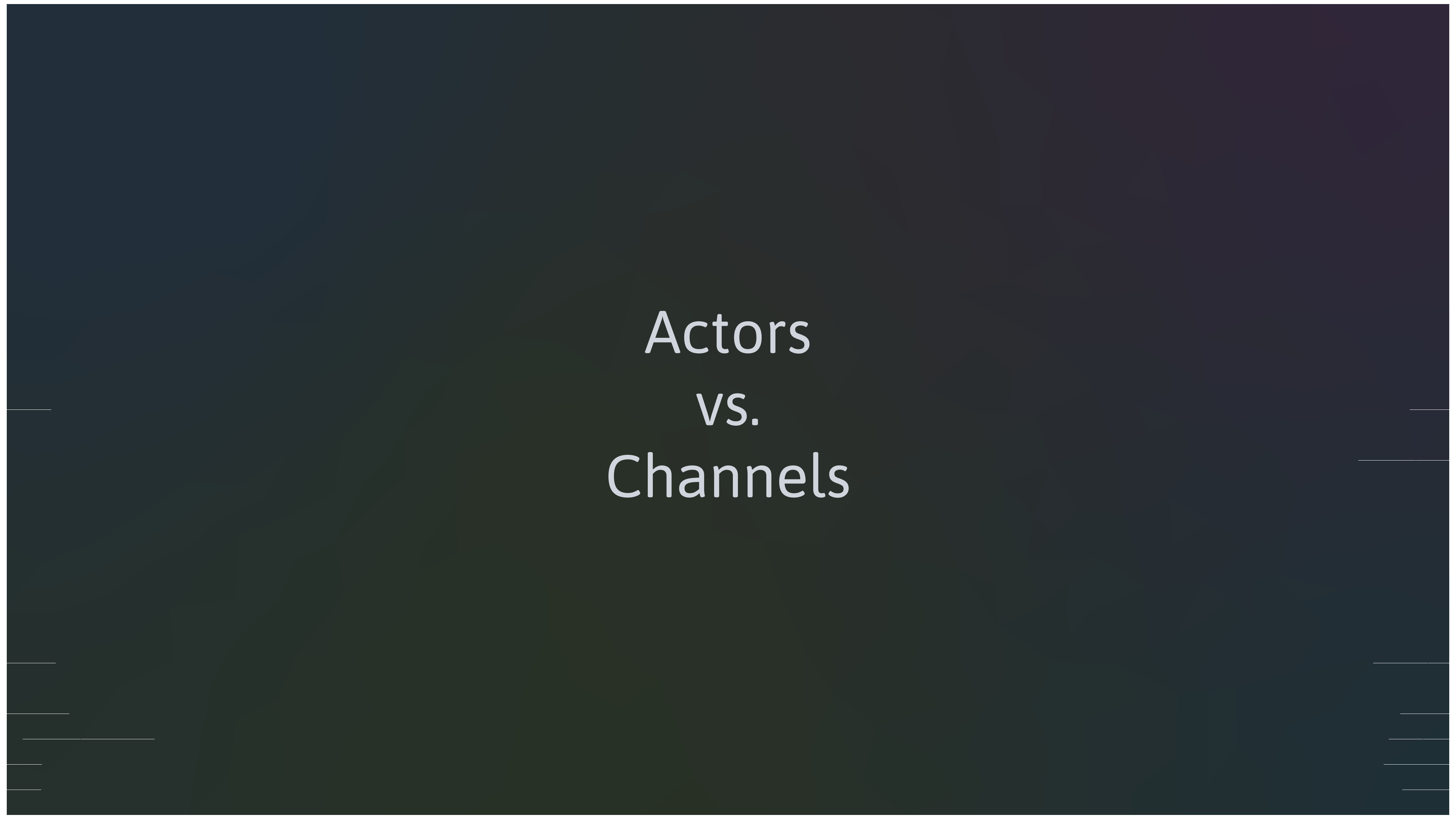
Clashes, data races, lost data, poor scalability.



Message passing addresses some of these issues.

Processes share data by sending messages. No memory is shared.

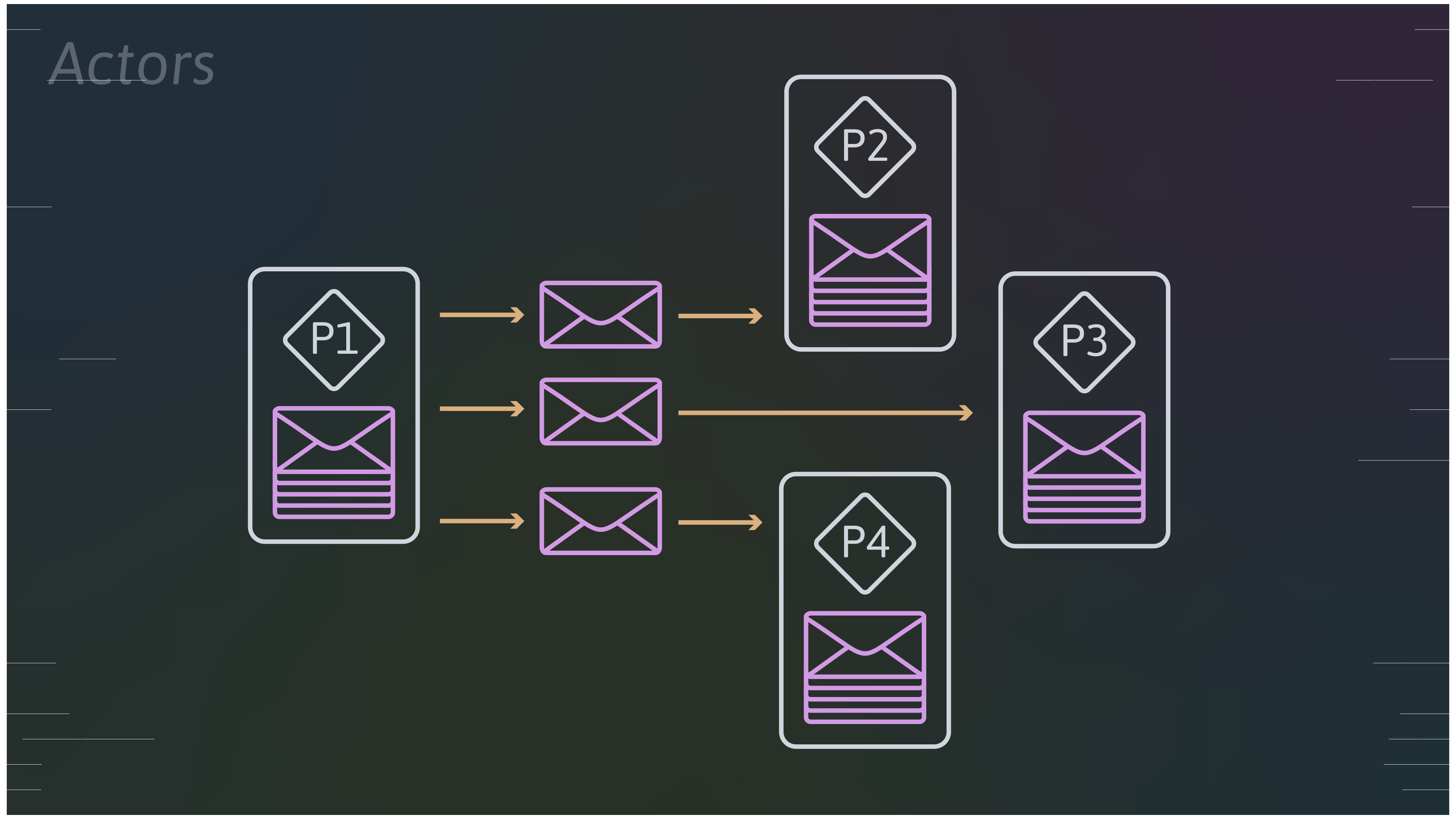
No lost data, no clashes, communication channels are abstractions that allow for extreme changes in scale for programs.



Withing MP concurrency there are two leading models

Actors : Erlang, Elixir, MPI

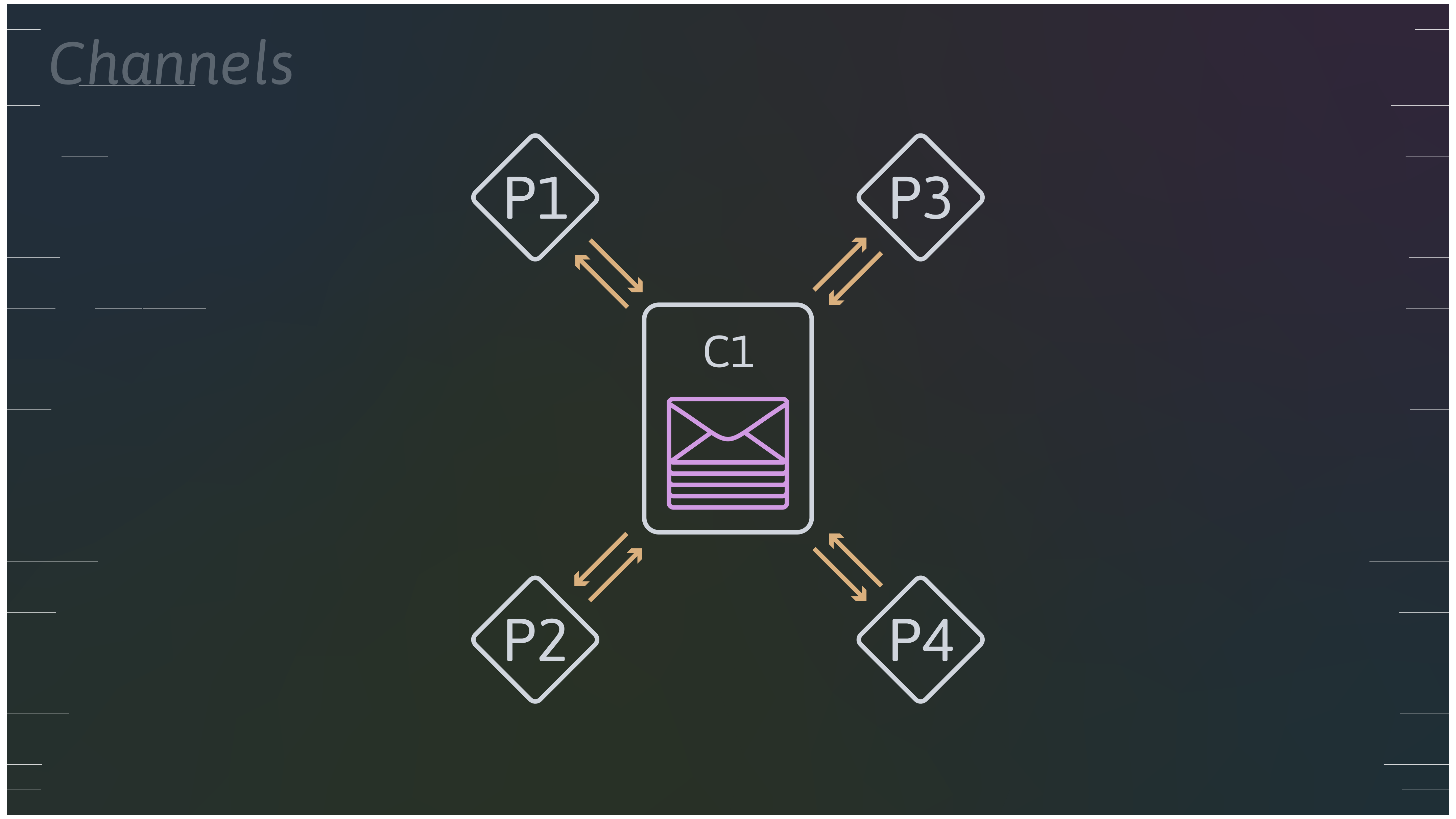
Channels : Go



Actors

Each process has an associated mailbox

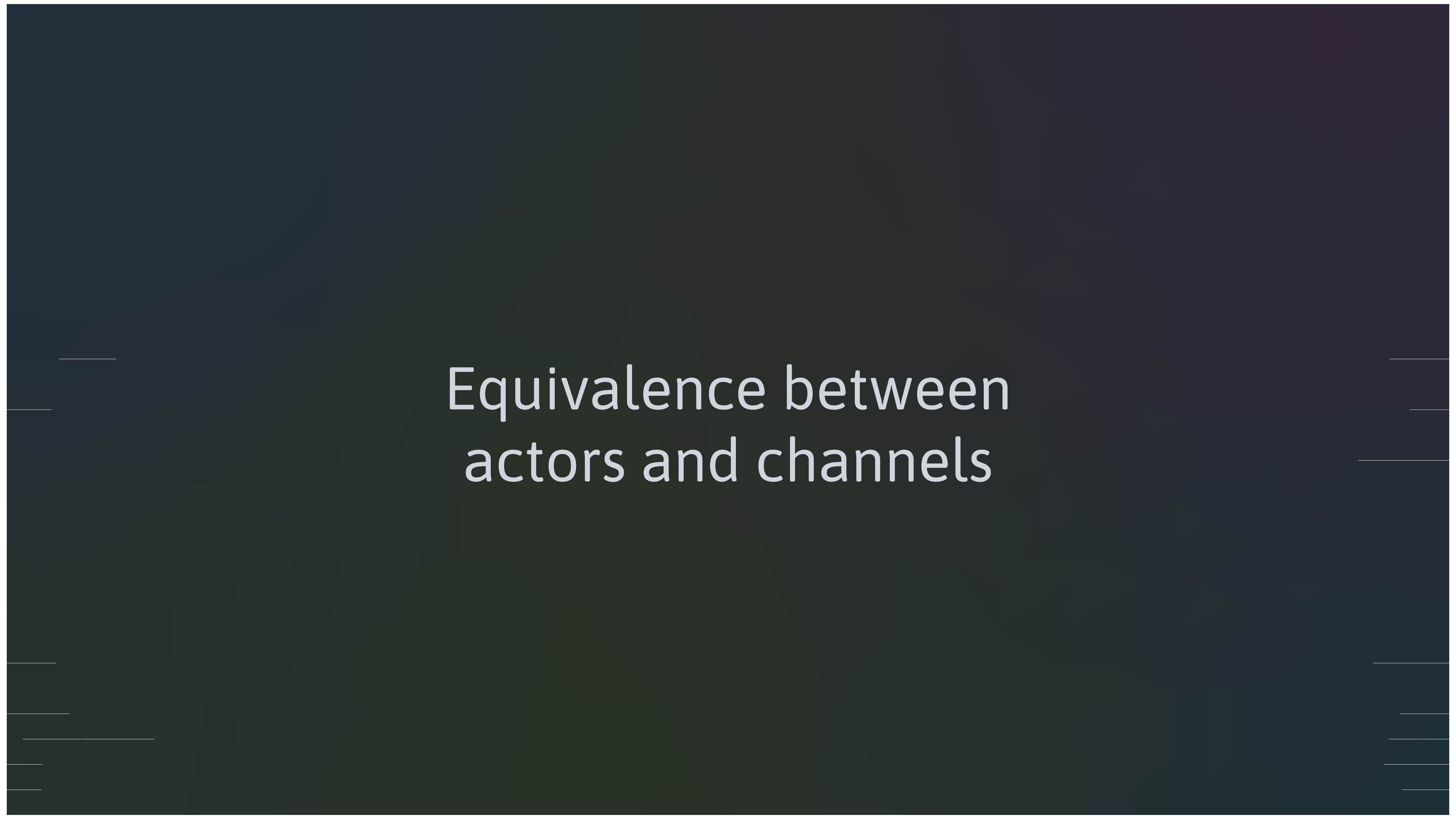
Messages are sent directly to other processes and queues in their maiboxes



Channels

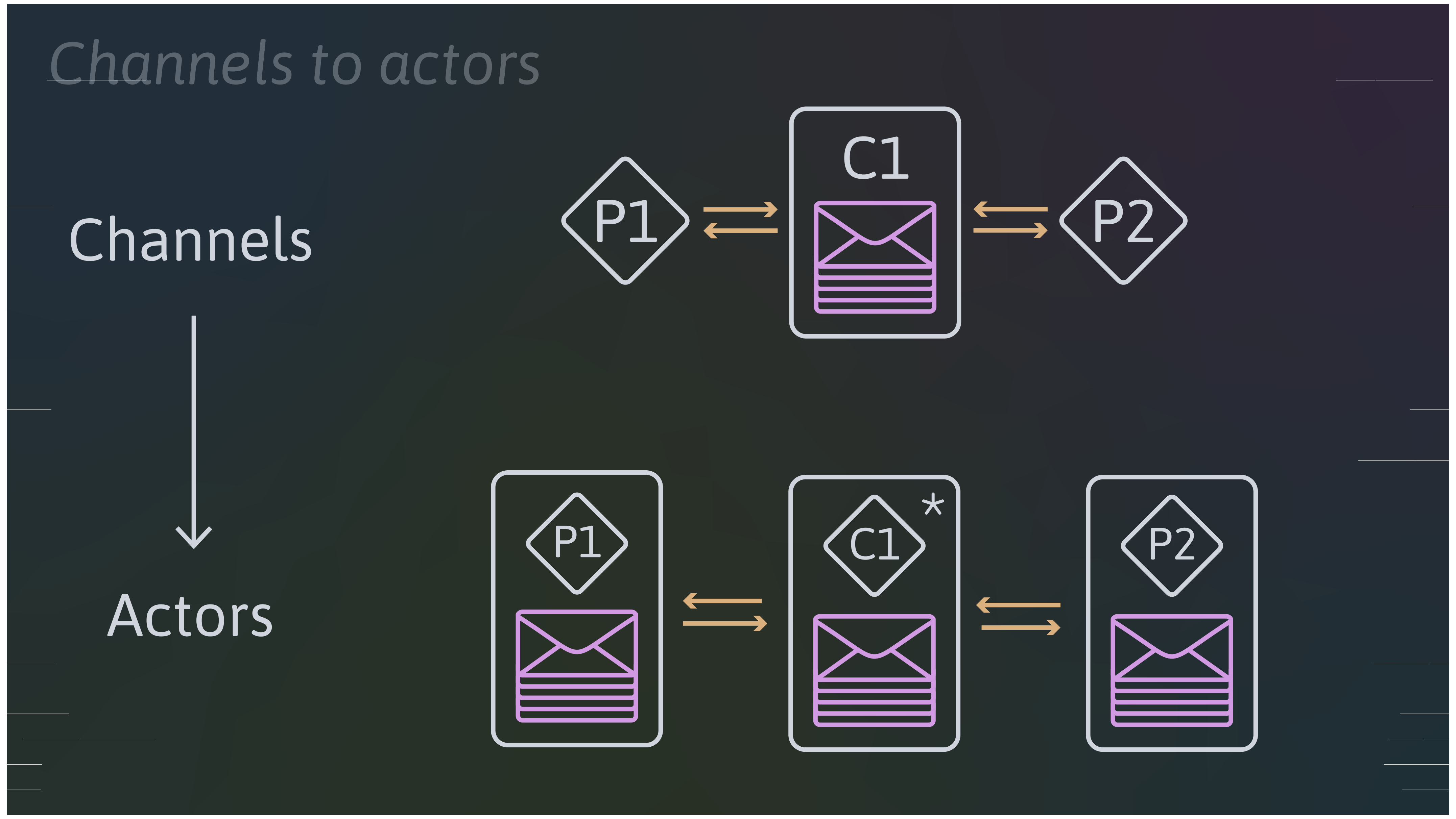
There are processes and channels

Messages can only be sent to channels and retrieved from channels. Making channels the interface for inter-process communications.



These two models, asynchronous, are equivalent.

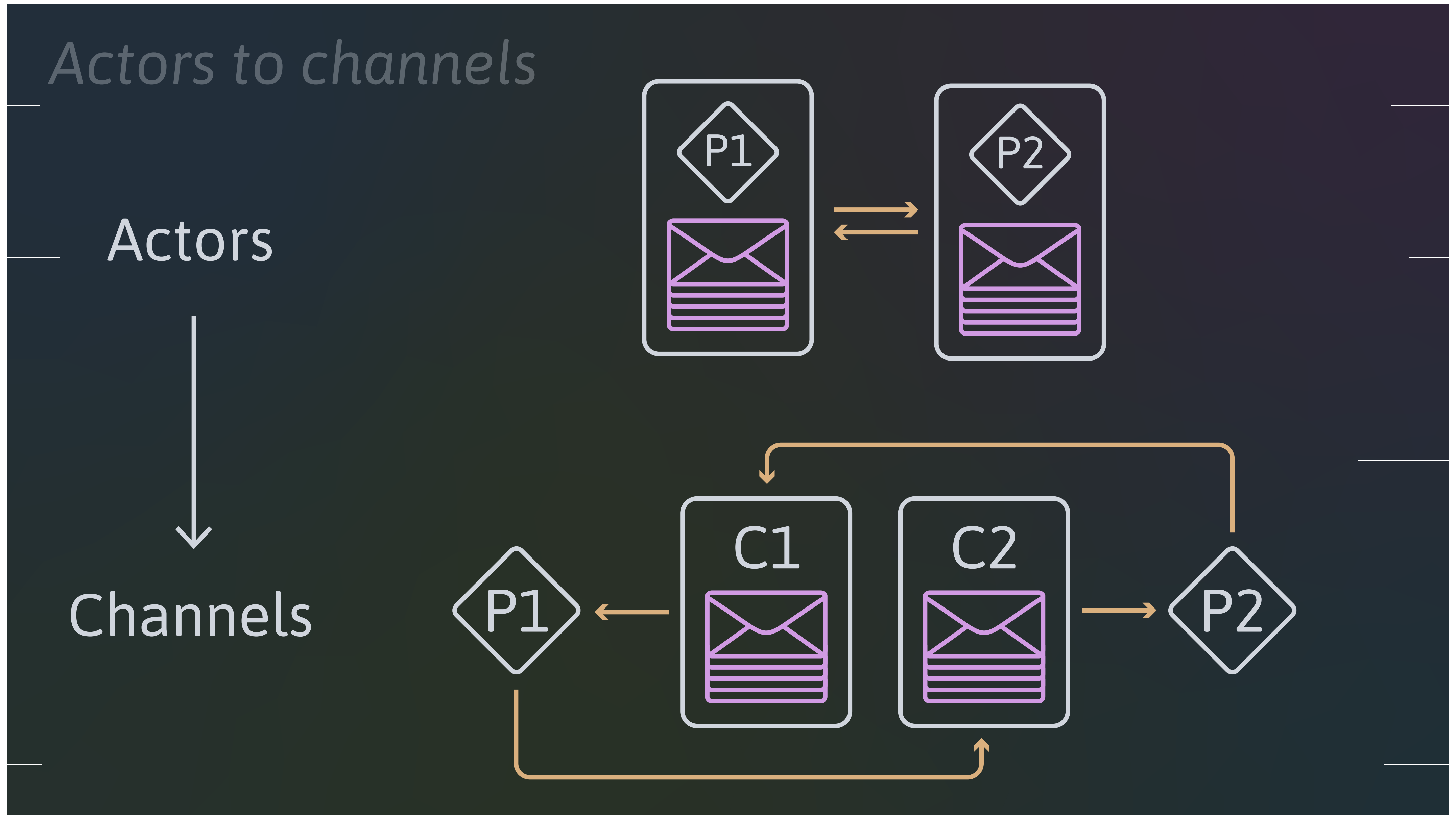
To show this, two translation schemes have been produced that show that each model can be represented in terms of the other.



When translating channels to actors

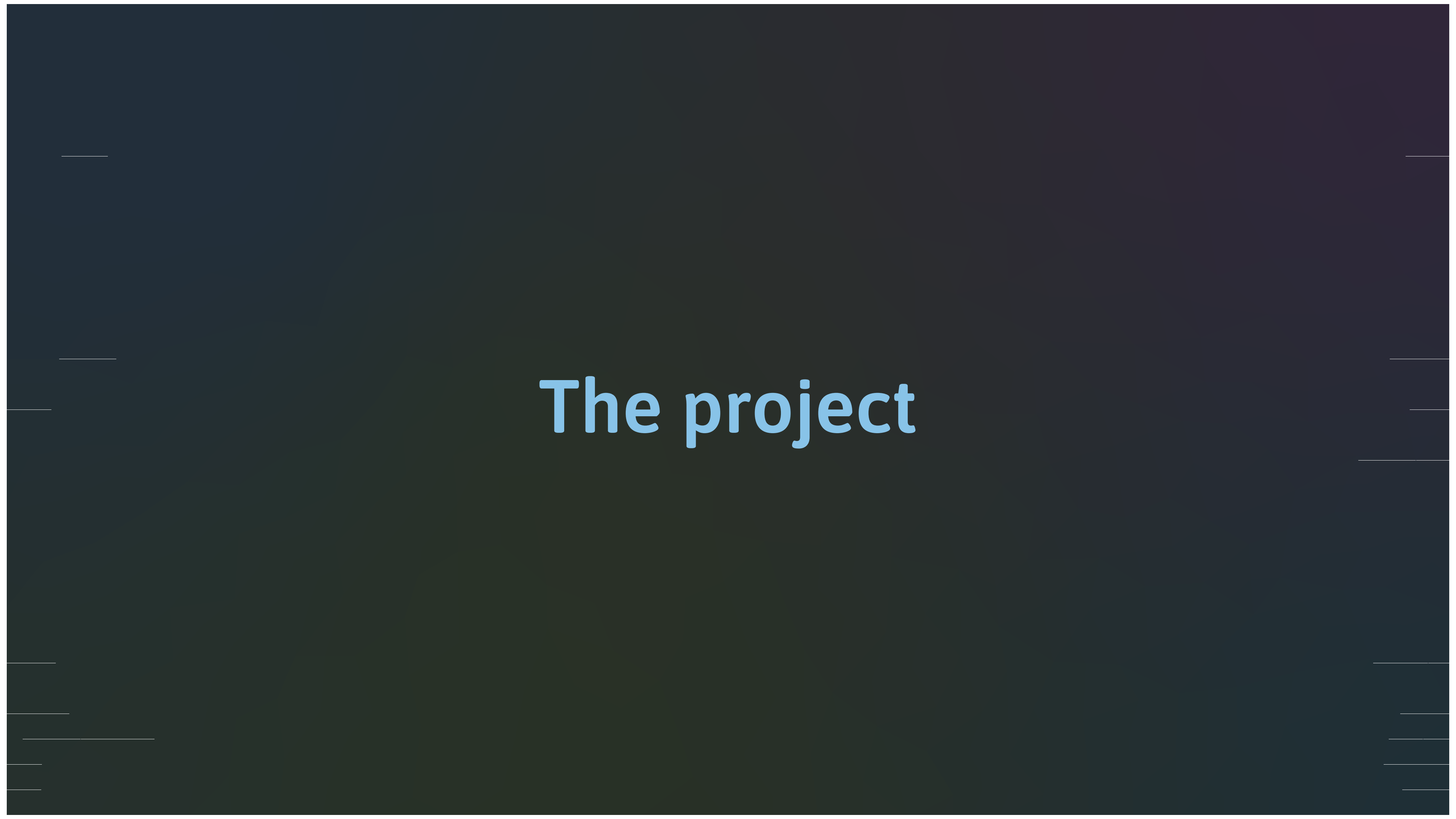
Channels become actors executing a special program.

Processes become actors, instead of direct interfacing, they use messages to retrieve data from the channel



Actors to channels,

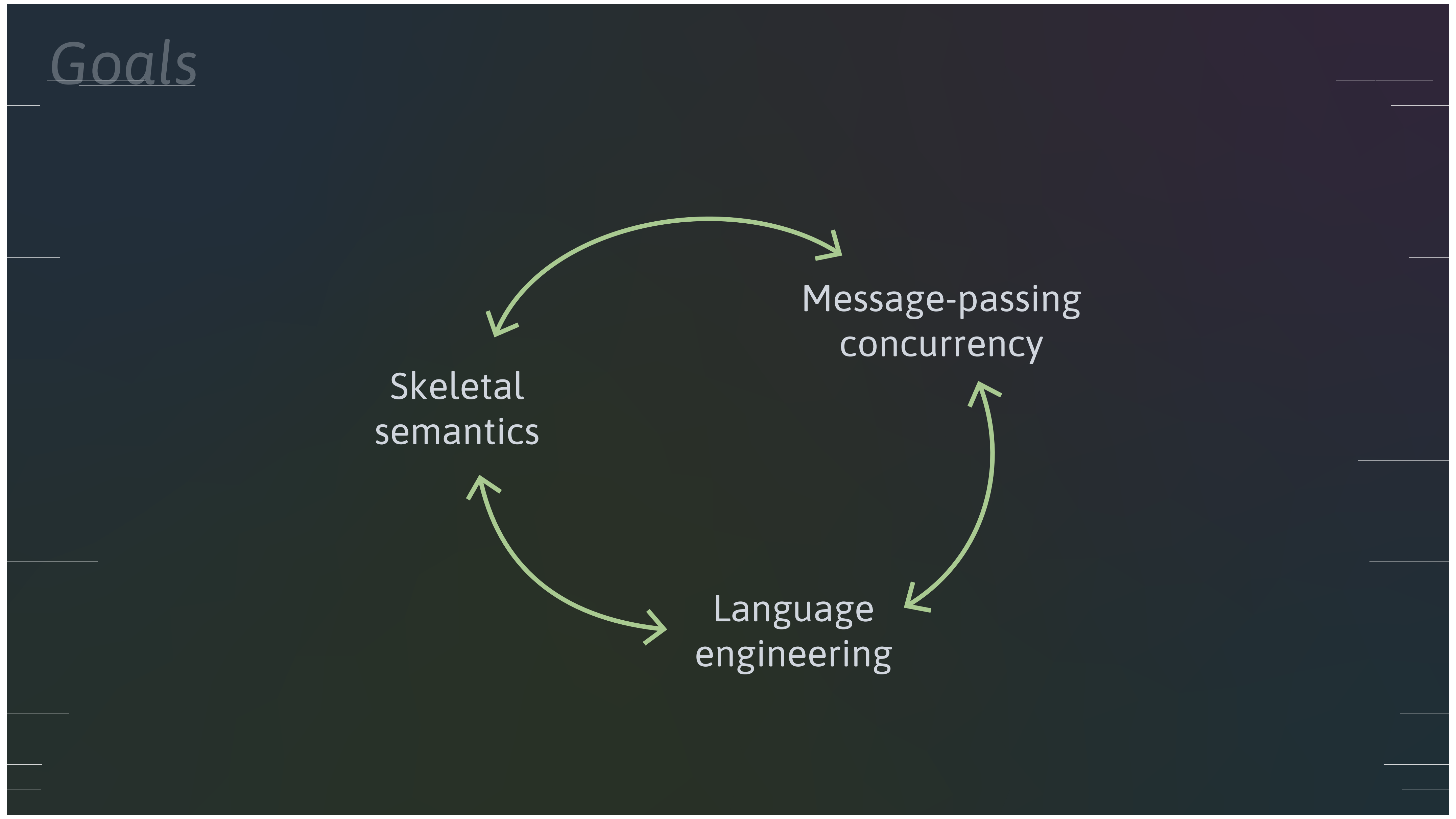
Since processes have no mailbox, each actor is converted into a process-channel pair, where the channel acts as the associated mailbox of the process.



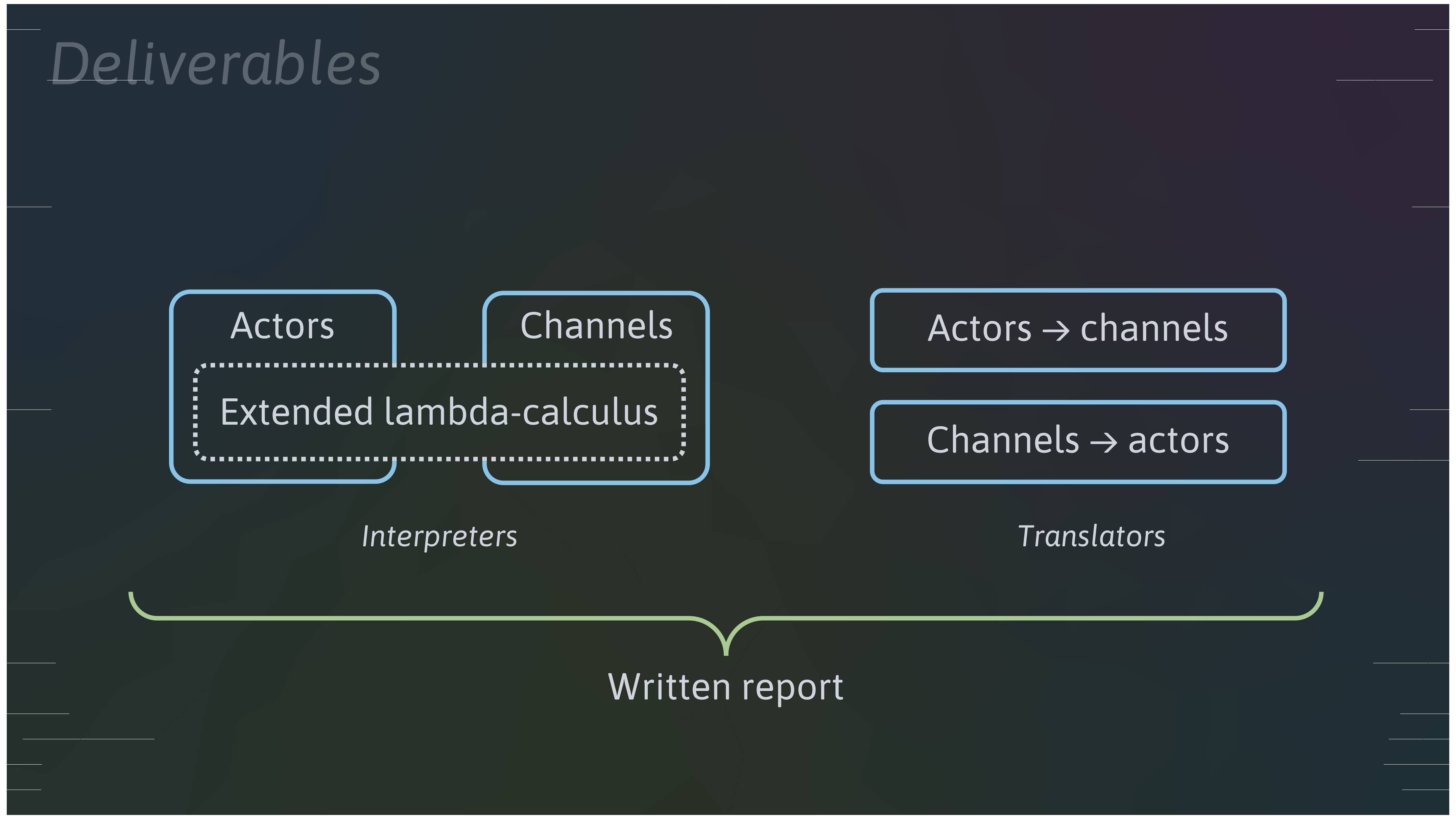
Creation of a set of tools

viability of skeletal semantics and their associated tools

correct description and implementation of concurrent programming languages.



The project explores the space between skeletal semantics, mp concurrency, and language engineering.



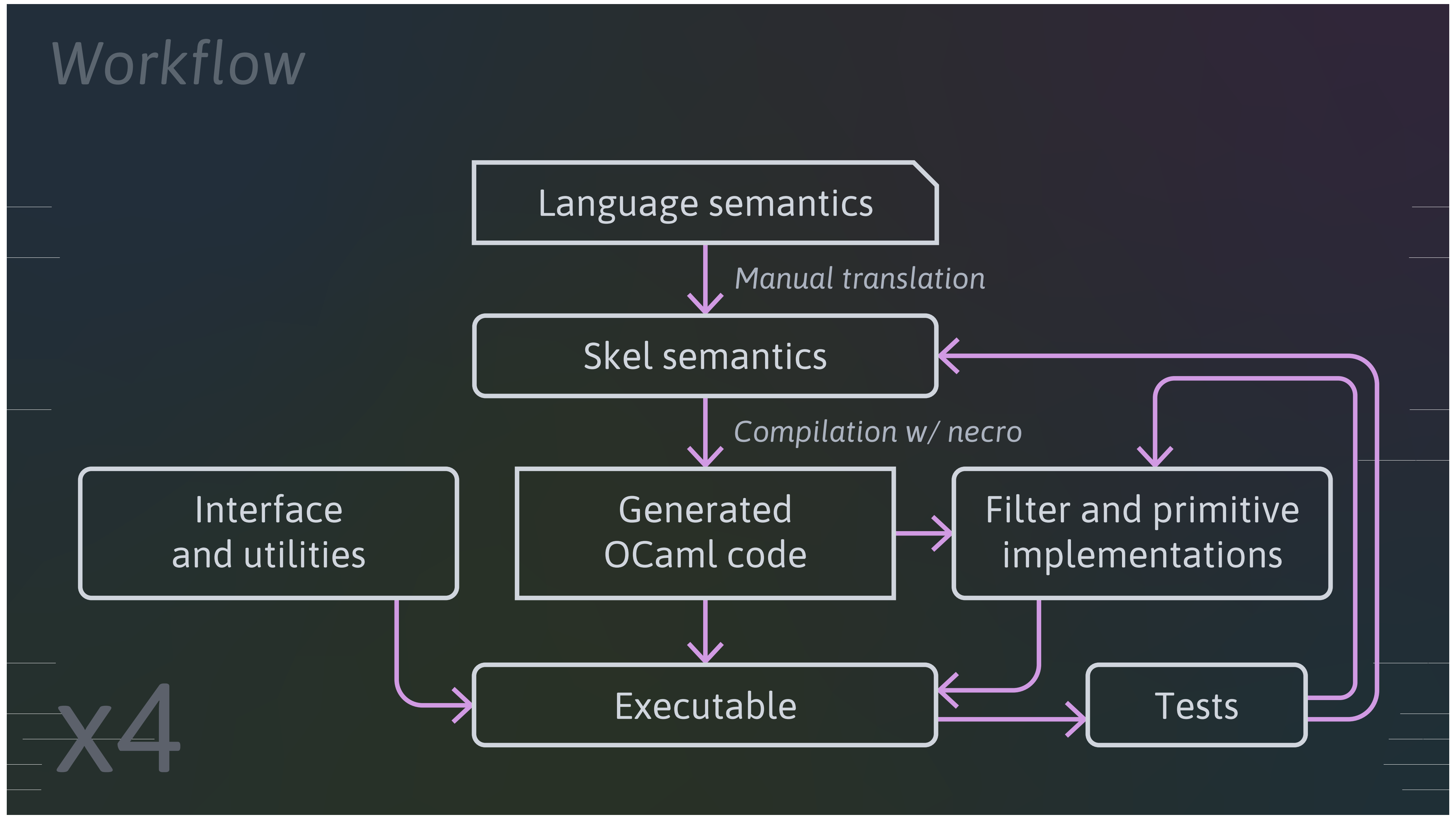
To this end

produce 4 software tools

Two interpreters for two concurrent languages built on top of an extended lambda calculus. One for each of the two models of concurrency describes.

Two translators, to go from one model to the other and back

And a written report that encompasses the entirety of the endeavour.



The workflow for the creation of the tools is as follows.

Existing semantics

Manually translated to skeletal sem.

Compiled to OCaml

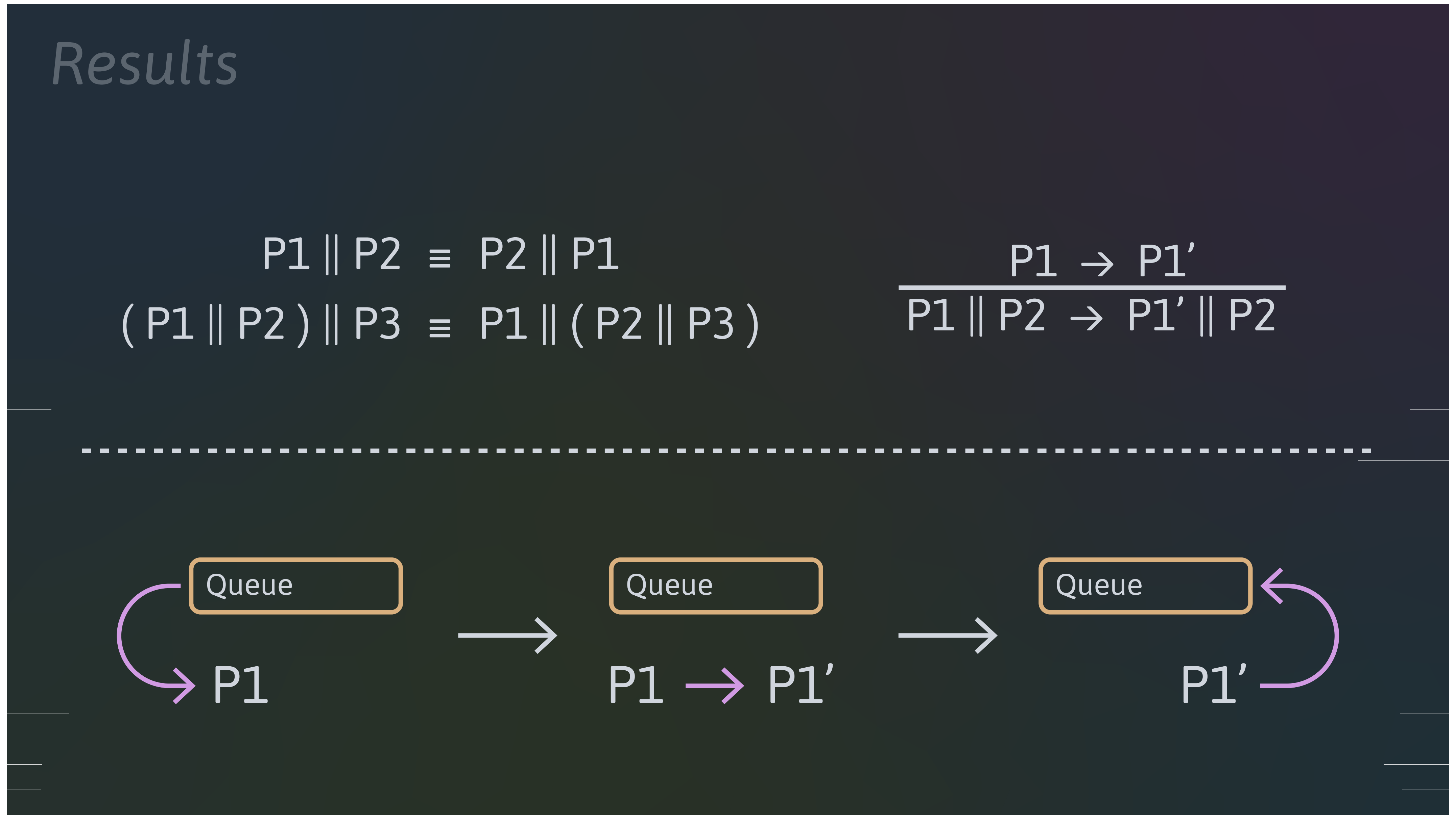
Filters are treated as abstractions of the host language and implemented in OCaml, same for primitive types.

Interface functions are made

Wrapped together and tested

Skeletal semantics are certainly powerful enough

But there's also plenty of issues to be considered.



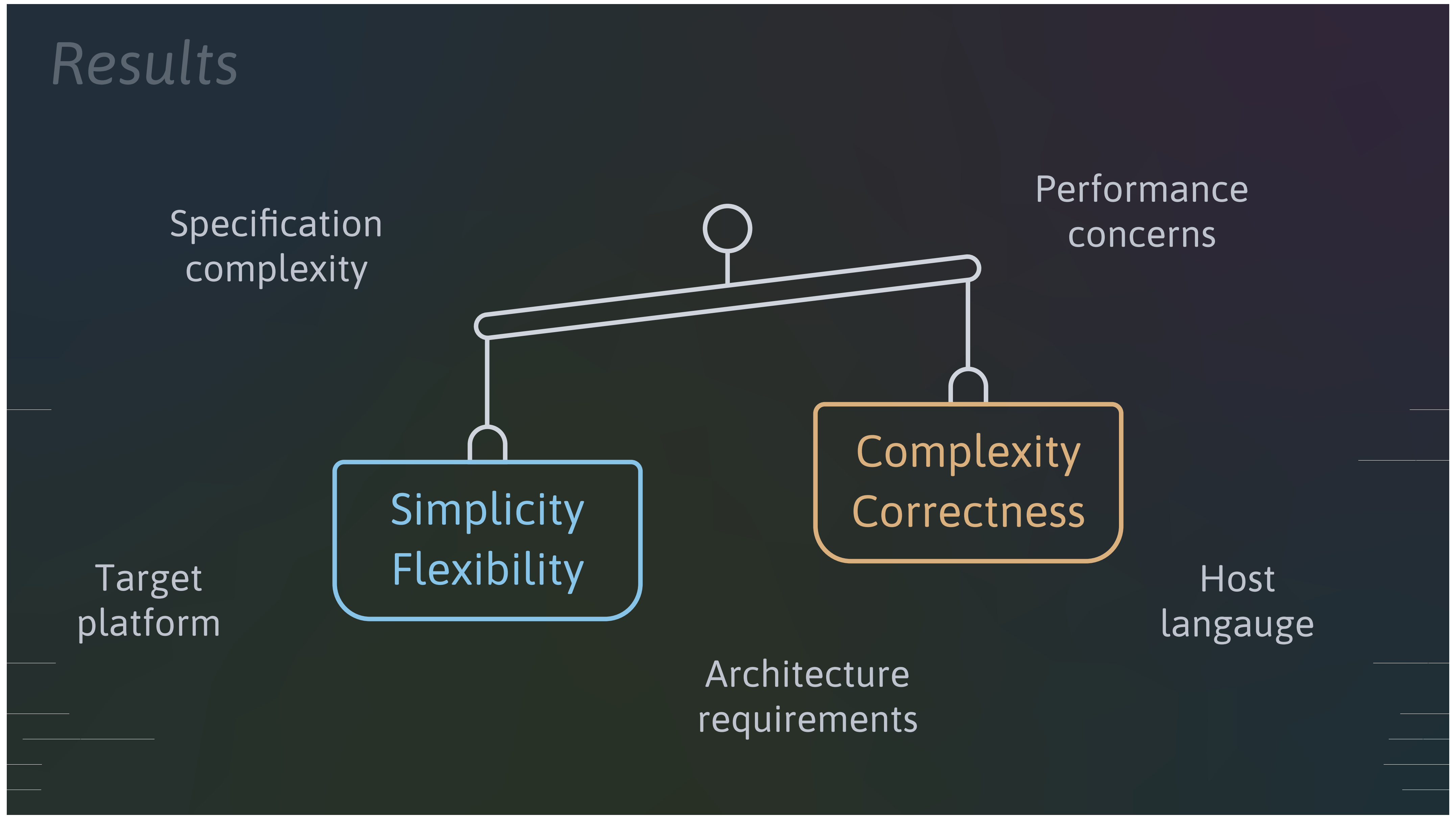
For example, take parallel configurations.

Commutative and distributive, that is, order and nesting do not matter when applying reduction rules

That is not the case when programming the semantics since computers do care about order and nesting.

The right abstractions and structures need to be chosen.

Such issues permeated the project as a whole



Additionally, the use of skel sem is a balancing act.

Since filters allow for abstractions in the host language, it becomes a balancing act.

How much to abstract depends on many factors...