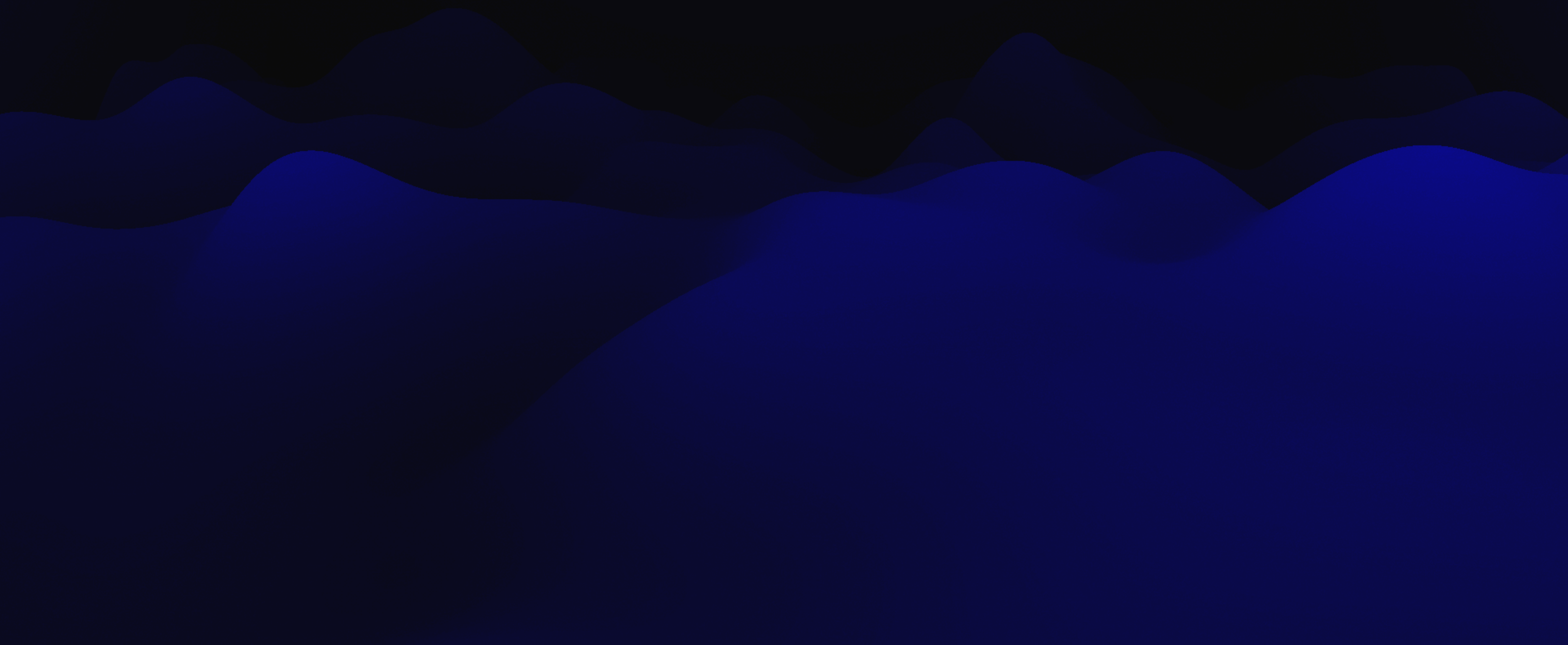
New spaces for denotational semantics

Jonathan Sterling • 2 February 2023 • MSCA Postdoctoral Fellow, Aarhus University

I study *mathematical models* of program execution



I study *mathematical models* of program execution

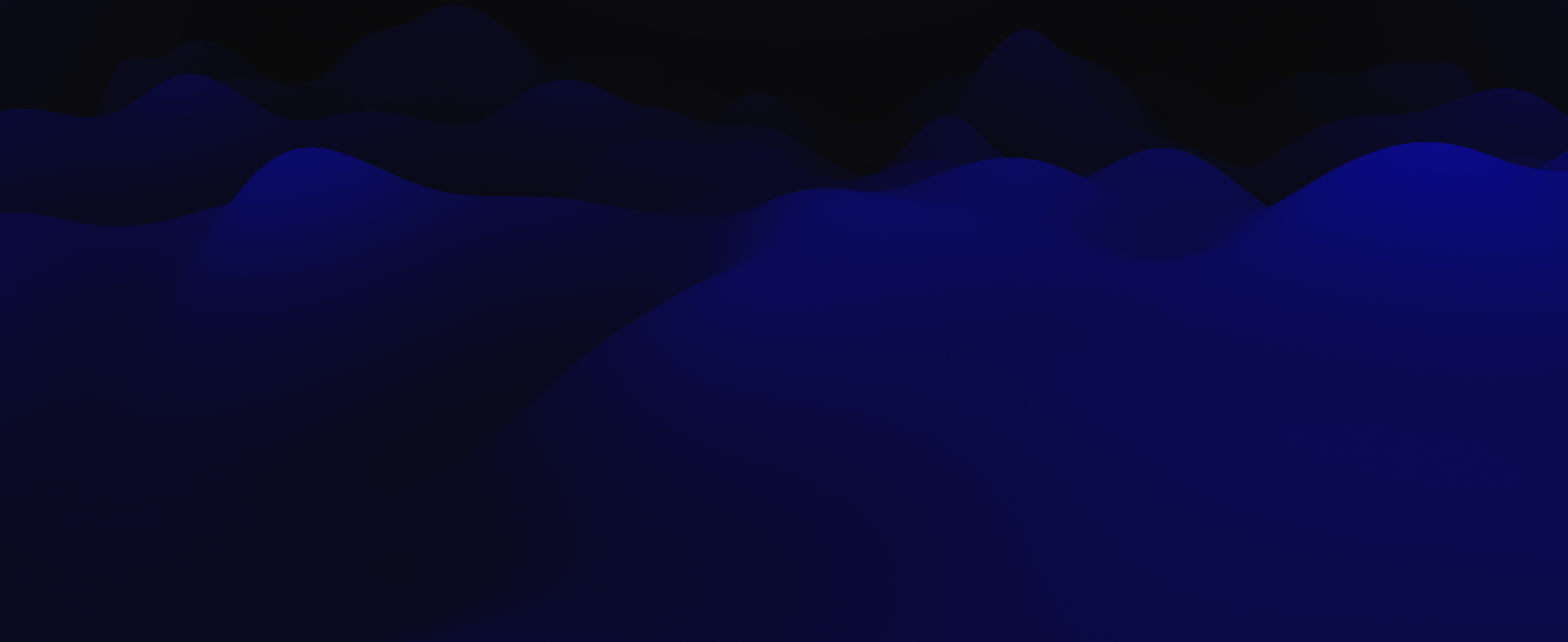
To understand and verify the correctness of computer programs, we develop ***mathematical models*** that approximate *different aspects* of the **physical reality** of program execution on hardware.

I study *mathematical models* of program execution

To understand and verify the correctness of computer programs, we develop ***mathematical models*** that approximate *different aspects* of the **physical reality** of program execution on hardware.

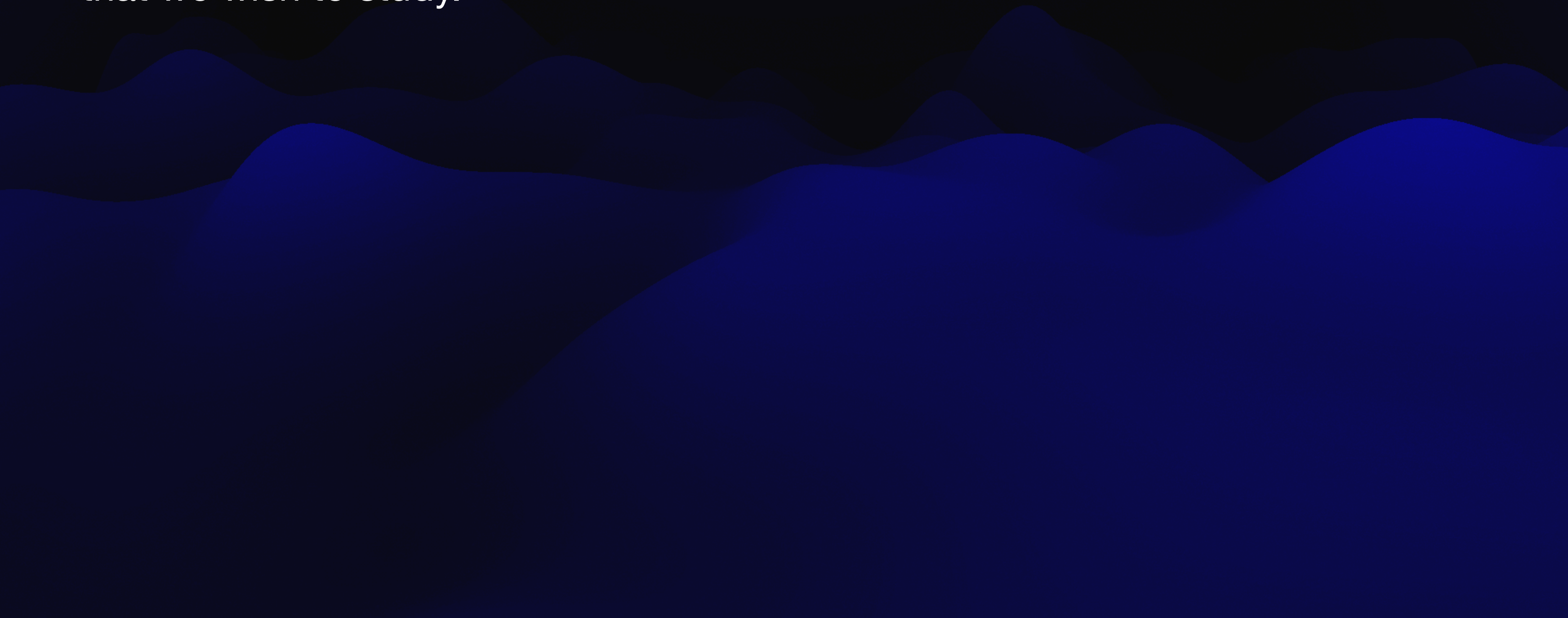
(Just like physicists create many idealized mathematical models to study different aspects of the material reality of the universe.)

We need more than one model...



We need more than one model...

Different models of computation surface different *facets* of program execution that we wish to study.



input-output behavior

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

information flow

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

information flow

Observer model: executions are modeled by (possibly empty) sets of clients who can *“observe”* the termination of the program.

To verify: security and noninterference

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

information flow

Observer model: executions are modeled by (possibly empty) sets of clients who can *“observe”* the termination of the program.

To verify: security and noninterference

cost & complexity

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

information flow

Observer model: executions are modeled by (possibly empty) sets of clients who can *“observe”* the termination of the program.

To verify: security and noninterference

cost & complexity

Cost model: executions are modeled “intensionally” by the (potentially infinite) *quantity* of resources it takes for them to terminate.

To verify: complexity bounds

input-output behavior

Black box model: executions are modeled “extensionally” by a *termination bit* indicating whether the program terminated.

To verify: functional correctness

information flow

Observer model: executions are modeled by (possibly empty) sets of clients who can *“observe”* the termination of the program.

To verify: security and noninterference

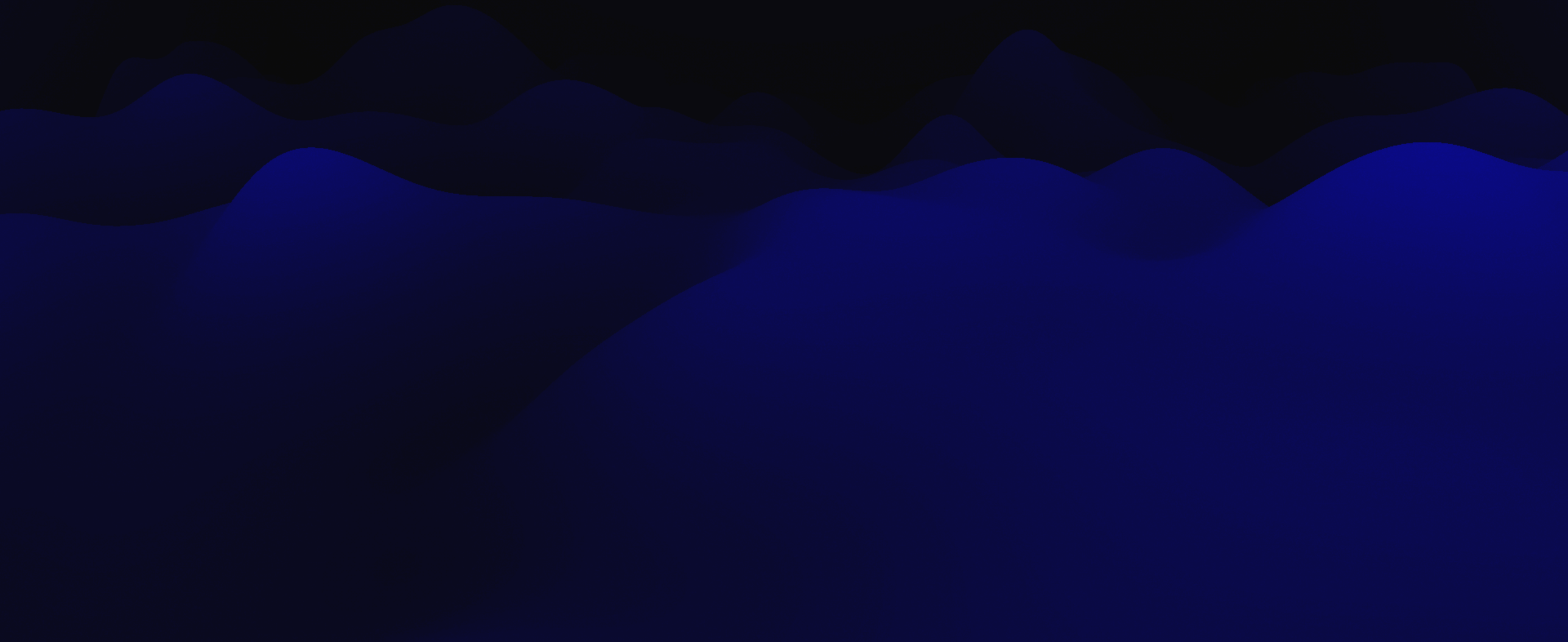
cost & complexity

Cost model: executions are modeled “intensionally” by the (potentially infinite) *quantity* of resources it takes for them to terminate.

To verify: complexity bounds

Each of these ***mathematical models*** represents a different *abstraction* of the behavior of programs on physical hardware.

We need more than one model ... *at once!*



We need more than one model ... *at once!*

- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.

We need more than one model ... *at once!*

- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.

We need more than one model ... *at once!*

- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.
  - Conversely, a complexity bound implies termination, which is a property of I/O behavior and could be used to establish functional correctness.

We need more than one model ... *at once!*

- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.
  - Conversely, a complexity bound implies termination, which is a property of I/O behavior and could be used to establish functional correctness.
  - Blackbox models and cost models disagree: does mergesort = insertionsort?

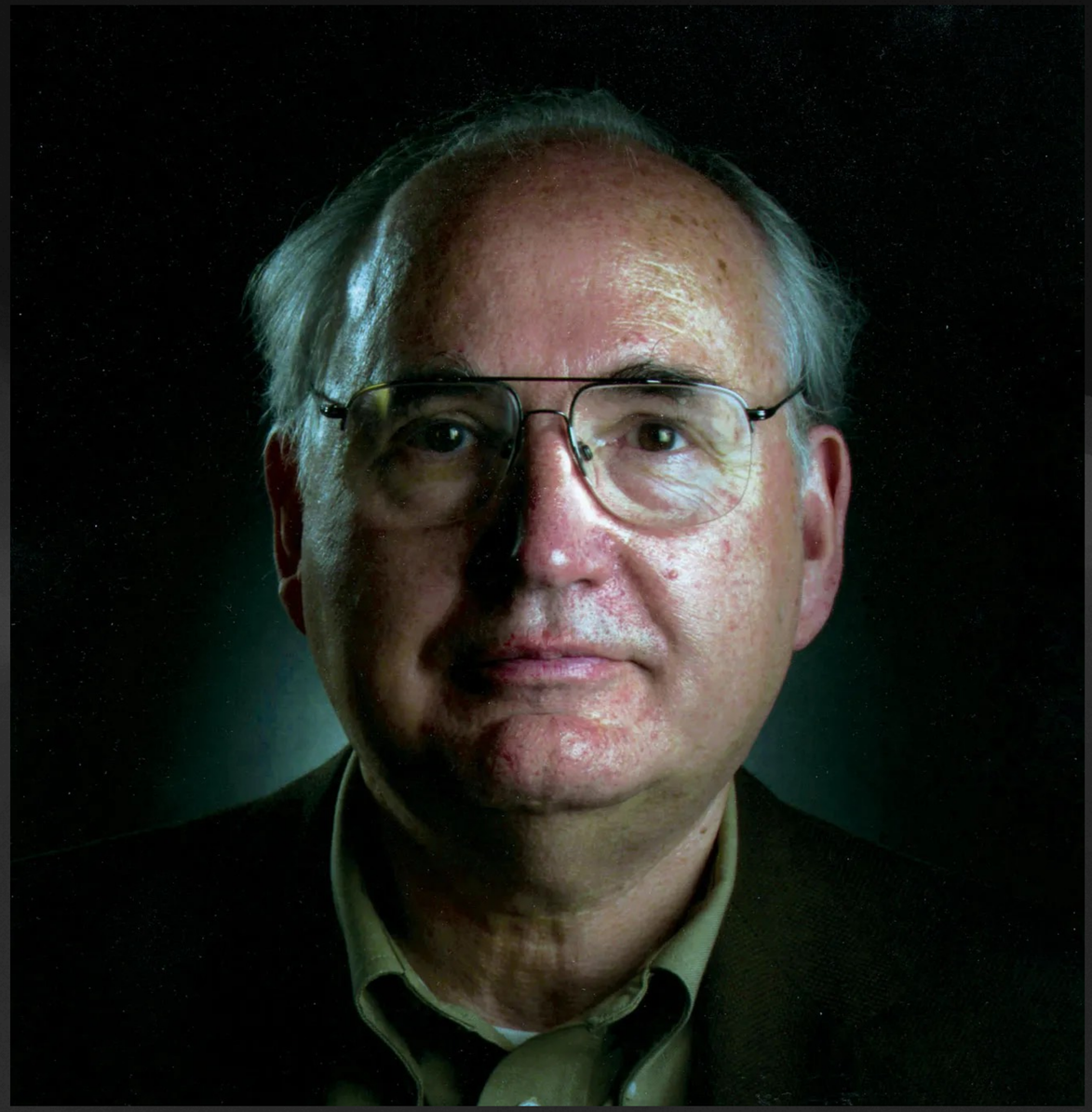
We need more than one model ... *at once!*

- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.
  - Conversely, a complexity bound implies termination, which is a property of I/O behavior and could be used to establish functional correctness.
  - Blackbox models and cost models disagree: does mergesort = insertionsort?
- **My research** for the past three years has aimed to uncover the **“laws of motion”** that govern the interaction between *all* such facets — to facilitate modular verifications that cut across different facets of program execution.

We need more than one model ... *at once!*

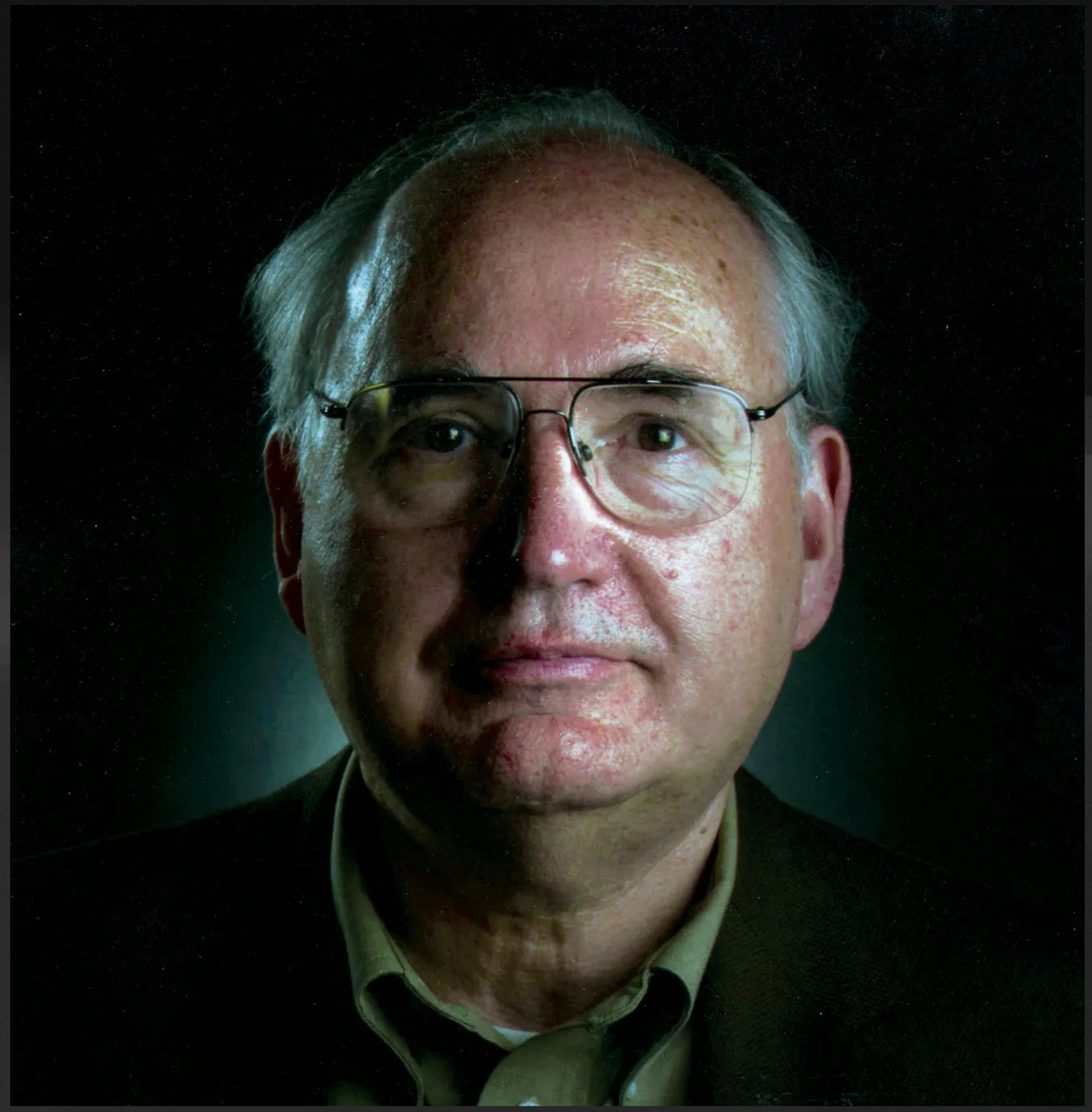
- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.
  - Conversely, a complexity bound implies termination, which is a property of I/O behavior and could be used to establish functional correctness.
  - Blackbox models and cost models disagree: does mergesort = insertionsort?
- **My research** for the past three years has aimed to uncover the **“laws of motion”** that govern the interaction between *all* such facets — to facilitate modular verifications that cut across different facets of program execution.
  - This research program has led to the solution of several open problems in dependent type theory.

Technically, these models take place in a **domain theory**, *i.e.* a given variation on the **theory of topological spaces** suitable for studying computation.



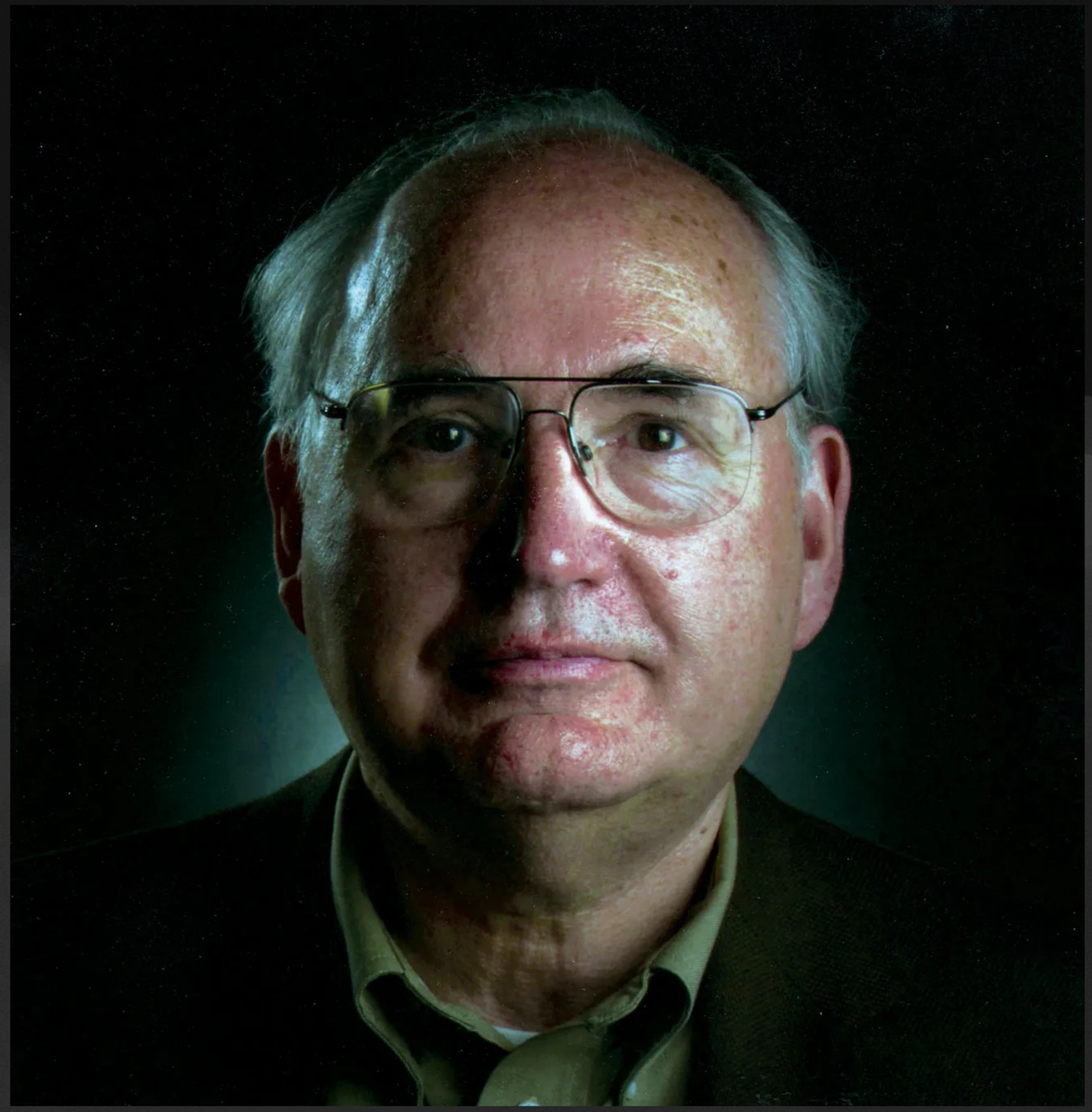
Technically, these models take place in a **domain theory**, *i.e.* a given variation on the **theory of topological spaces** suitable for studying computation.

- **Classical domain theory:** dcpos, ω-cpos, Scott domains, stable domains, coherent spaces, ultrametric spaces...



Technically, these models take place in a **domain theory**, *i.e.* a given variation on the **theory of topological spaces** suitable for studying computation.

- **Classical domain theory:** dcpos, ω-cpos, Scott domains, stable domains, coherent spaces, ultrametric spaces...
- **Synthetic domain theory:** “sets” in a topos.



Gluing together different notions of space

Gluing together different notions of space

Just as individual spaces in topology can be glued together, entire theories of spaces can also be glued together to give a “new kind of space”.

Gluing together different notions of space

Just as individual spaces in topology can be glued together, entire theories of spaces can also be glued together to give a “new kind of space”.

These gluings provide **universal** ways to surface subtle aspects of computation (see also the recent work of Matache, Moss, Staton).

Example I: secure information flow

Example I: secure information flow

- **Problem:** in a programming language with *security modalities*, how can we prove that high-security inputs do not influence low-security outputs?

Example I: secure information flow

- **Problem:** in a programming language with *security modalities*, how can we prove that high-security inputs do not influence low-security outputs?

- Given a lattice of security clearances $\mathbb{E}$, we can glue many copies of an existing domain theory together according to $\mathbb{E}$ to obtain an ***information-flow sensitive domain theory*** whose computations intrinsically carry the sets of clients who can observe their results.

Example I: secure information flow

- **Problem:** in a programming language with *security modalities*, how can we prove that high-security inputs do not influence low-security outputs?
- Given a lattice of security clearances $\mathbb{E}$, we can glue many copies of an existing domain theory together according to $\mathbb{E}$ to obtain an ***information-flow sensitive domain theory*** whose computations intrinsically carry the sets of clients who can observe their results.
- Noninterference in the model follows immediately; it is lifted from the model to the programming language by means of a further gluing.

Example I: secure information flow

- **Problem:** in a programming language with *security modalities*, how can we prove that high-security inputs do not influence low-security outputs?
- Given a lattice of security clearances $\mathbb{E}$, we can glue many copies of an existing domain theory together according to $\mathbb{E}$ to obtain an ***information-flow sensitive domain theory*** whose computations intrinsically carry the sets of clients who can observe their results.
- Noninterference in the model follows immediately; it is lifted from the model to the programming language by means of a further gluing.

S. & Harper. *"Sheaf semantics of termination-insensitive noninterference."* **FSCD '22.**

Example II: reasoning about cost and behavior

Example II: reasoning about cost and behavior

- **Problem:** to reason about the complexity of programs, we often need to prove functional correctness (which ignores cost and pertains only to I/O behavior).

Example II: reasoning about cost and behavior

- **Problem:** to reason about the complexity of programs, we often need to prove functional correctness (which ignores cost and pertains only to I/O behavior).
  - How can we reason equationally about **both** cost and behavior in the same language without painful quotienting everywhere?

Example II: reasoning about cost and behavior

- **Problem:** to reason about the complexity of programs, we often need to prove functional correctness (which ignores cost and pertains only to I/O behavior).
  - How can we reason equationally about **both** cost and behavior in the same language without painful quotienting everywhere?
- **Solution:** glue together the two models, yielding a *modal logic* for complexity analysis in which the modality forgets all cost information.

Example II: reasoning about cost and behavior

- **Problem:** to reason about the complexity of programs, we often need to prove functional correctness (which ignores cost and pertains only to I/O behavior).
  - How can we reason equationally about **both** cost and behavior in the same language without painful quotienting everywhere?
- **Solution:** glue together the two models, yielding a *modal logic* for complexity analysis in which the modality forgets all cost information.

Niu, S., Grodin, Harper. *"A Cost-Aware Logical Framework."* **POPL '22.**

Example III: representation independence and computational effects

Example III: representation independence and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?

Example III: representation independence and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?
  - Intuitively, yes! So long as the two implementations agree on their **observable interface** (Reynolds ’83). Technically, not so easy to accommodate realistic languages with state and control effects.

Example III: representation independence and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?
  - Intuitively, yes! So long as the two implementations agree on their **observable interface** (Reynolds ’83). Technically, not so easy to accommodate realistic languages with state and control effects.
- **Solution:** glue together three copies (left, right, and middle) of your computational model. The left and right sides represent two implementations of a data structure, and the “middle” carries a representation invariant between the two.

MIDDLE: Representation Invariant

$$l, r_{front}, r_{back} : \texttt{List} \mid l = \text{append}(r_{front}, \text{reverse}(r_{back}))$$

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

$$r_{front} : \text{List}$$
$$r_{back} : \text{List}$$

MIDDLE: Representation Invariant

$$l, r_{front}, r_{back} : \texttt{List} \mid l = \text{append}(r_{front}, \text{reverse}(r_{back}))$$

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

$$r_{front} : \text{List}$$
$$r_{back} : \text{List}$$

MIDDLE: Representation Invariant

$$l, r_{front}, r_{back} : \texttt{List} \mid l = \text{append}(r_{front}, \text{reverse}(r_{back}))$$

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

$$r_{front} : \text{List}$$
$$r_{back} : \text{List}$$

MIDDLE: Representation Invariant

$$l, r_{front}, r_{back} : \texttt{List} \mid l = \text{append}(r_{front}, \text{reverse}(r_{back}))$$

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

$$r_{front} : \text{List}$$
$$r_{back} : \text{List}$$

Example III: data abstraction and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?
  - Intuitively, yes! So long as the **observable interface** of the two implementations agrees (Reynolds ’83). Technically, not so easy to accommodate realistic languages with state and control effects.
- **Solution:** glue together three copies (left, right, and middle) of your computational model. The left and right sides represent two implementations of a data structure, and the “middle” carries a representation invariant between the two.

Example III: data abstraction and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?
  - Intuitively, yes! So long as the **observable interface** of the two implementations agrees (Reynolds ’83). Technically, not so easy to accommodate realistic languages with state and control effects.
- **Solution:** glue together three copies (left, right, and middle) of your computational model. The left and right sides represent two implementations of a data structure, and the “middle” carries a representation invariant between the two.

S., Harper. *“Logical Relations As Types: Proof-Relevant Parametricity for Program Modules.”* **J.ACM.**

Example III: data abstraction and computational effects

- **Problem:** if you replace your implementation of an **abstract data type** (*e.g.* a queue or a hash table), is your program still correct?
  - Intuitively, yes! So long as the **observable interface** of the two implementations agrees (Reynolds ’83). Technically, not so easy to accommodate realistic languages with state and control effects.
- **Solution:** glue together three copies (left, right, and middle) of your computational model. The left and right sides represent two implementations of a data structure, and the “middle” carries a representation invariant between the two.

S., Harper. *“Logical Relations As Types: Proof-Relevant Parametricity for Program Modules.”* **J.ACM.**

S., Gratzer, Birkedal. *“Denotational semantics of general store and polymorphism.”* Under review.

Example IV: normalization, decidability, and coherence

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).
- By gluing together a *syntactic model* of the type theory and a *model of symbolic computation*, we obtain a third model from which the normalization and decidability results are easily deduced.

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).
- By gluing together a *syntactic model* of the type theory and a *model of symbolic computation*, we obtain a third model from which the normalization and decidability results are easily deduced.
- **My methods have been instrumental to resolve three major conjectures in type theory.**

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).
- By gluing together a *syntactic model* of the type theory and a *model of symbolic computation*, we obtain a third model from which the normalization and decidability results are easily deduced.
- **My methods have been instrumental to resolve three major conjectures in type theory.**

S. & Angiuli. *"Normalization for cubical type theory."* **LICS '21.**

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).
- By gluing together a *syntactic model* of the type theory and a *model of symbolic computation*, we obtain a third model from which the normalization and decidability results are easily deduced.
- **My methods have been instrumental to resolve three major conjectures in type theory.**

S. & Angiuli. *"Normalization for cubical type theory."* **LICS '21.**

Gratzer. *"Normalization for multimodal type theory."* **LICS '22.**

Example IV: normalization, decidability, and coherence

- **Problem**: to implement a proof assistant or compiler based on a given type theory, we have to prove that the type theory is **decidable**, which follows from the extremely non-trivial **normalization lemma** (a form of symbolic computation with free variables).
- By gluing together a *syntactic model* of the type theory and a *model of symbolic computation*, we obtain a third model from which the normalization and decidability results are easily deduced.
- **My methods have been instrumental to resolve three major conjectures in type theory.**

S. & Angiuli. *"Normalization for cubical type theory."* **LICS '21.**

Gratzer. *"Normalization for multimodal type theory."* **LICS '22.**

Uemura. *"Normalization and coherence for ∞-type theories."* Unpublished manuscript.

Synthesizing a formal language for glued spaces

Synthesizing a formal language for glued spaces

- The engine behind most of the applications discussed is ***synthetic Tait computability*** / **STC** — a form of modal type theory that captures the “laws of motion” of glued notions of space.

Synthesizing a formal language for glued spaces

- The engine behind most of the applications discussed is ***synthetic Tait computability*** / **STC** — a form of modal type theory that captures the "laws of motion" of glued notions of space.
  - For experts: **STC** is the internal language of *Artin-Wraith gluings of toposes*. **STC** is modularly combined with several forms of *synthetic domain theory*.

Synthesizing a formal language for glued spaces

- The engine behind most of the applications discussed is ***synthetic Tait computability*** / **STC** — a form of modal type theory that captures the "laws of motion" of glued notions of space.
  - For experts: **STC** is the internal language of *Artin-Wraith gluings of toposes*. **STC** is modularly combined with several forms of *synthetic domain theory*.
- **STC** is a *synthetic* reformulation of ***logical relations***, the fundamental proof technique of programming languages.

Synthesizing a formal language for glued spaces

- The engine behind most of the applications discussed is ***synthetic Tait computability*** / **STC** — a form of modal type theory that captures the "laws of motion" of glued notions of space.
  - For experts: **STC** is the internal language of *Artin-Wraith gluings of toposes*. **STC** is modularly combined with several forms of *synthetic domain theory*.
- **STC** is a *synthetic* reformulation of ***logical relations***, the fundamental proof technique of programming languages.

S. *"First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory."*
Ph.D. Thesis, Carnegie Mellon University.

Future and ongoing research directions

Future and ongoing research directions

- Towards ***higher-dimensional domain theory***: applications to concurrency, moduli spaces of computational domains.

Future and ongoing research directions

- Towards ***higher-dimensional domain theory***: applications to concurrency, moduli spaces of computational domains.
- ***Relative domain theory:*** several classes of models of synthetic & axiomatic domain theory are well-studied, but the general theory of domain theories as a whole is still lacking and quite *ad hoc*. (Preliminary work in guarded setting with Palombi in MFPS ’22.)

Future and ongoing research directions

- Towards ***higher-dimensional domain theory***: applications to concurrency, moduli spaces of computational domains.
- ***Relative domain theory:*** several classes of models of synthetic & axiomatic domain theory are well-studied, but the general theory of domain theories as a whole is still lacking and quite *ad hoc*. (Preliminary work in guarded setting with Palombi in MFPS ’22.)
- ***Denotational semantics of higher-order reference types*** (with Gratzer, Birkedal) in synthetic guarded domain theory; ***higher-order separation logic over denotational semantics*** (with Aagaard, Birkedal).

Thanks!

Funded by the European Union