Crossing Boundaries in Program Semantics

Jonathan Sterling • 9 February 2023 • Marie Skłodowska-Curie Postdoctoral Fellow @ Aarhus University

“Everyday human activities such as building a house on a hill by a stream, laying a network of telephone conduits, navigating the solar system, require **plans that can work**. Planning any such undertaking requires the development of *thinking about space*. Each development involves many steps of thought and many related geometrical constructions on spaces. Because of the necessary multistep nature of thinking about space, **uniquely mathematical measures** must be taken to make it reliable. Only explicit principles of thinking (logic) and explicit principles of space (geometry) can guarantee reliability.”

F. William Lawvere, may he rest in peace

Logics and geometries for reliable programming

Logics and geometries for reliable programming

My work aims to facilitate reliable programming by making explicit the logical & geometrical principles governing the execution of computer programs.

Logics and geometries for reliable programming

My work aims to facilitate reliable programming by making explicit the logical & geometrical principles governing the execution of computer programs.

To this end, I develop ***mathematical models*** that approximate *different aspects* of the **physical reality** of program execution on hardware.

Logics and geometries for reliable programming

My work aims to facilitate reliable programming by making explicit the logical & geometrical principles governing the execution of computer programs.

To this end, I 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 at different scales.)

Logics and geometries for reliable programming

My work aims to facilitate reliable programming by making explicit the logical & geometrical principles governing the execution of computer programs.

To this end, I 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 at different scales.)

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

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

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

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

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

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

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

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

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

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. ***Key idea: "phase composition".***

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

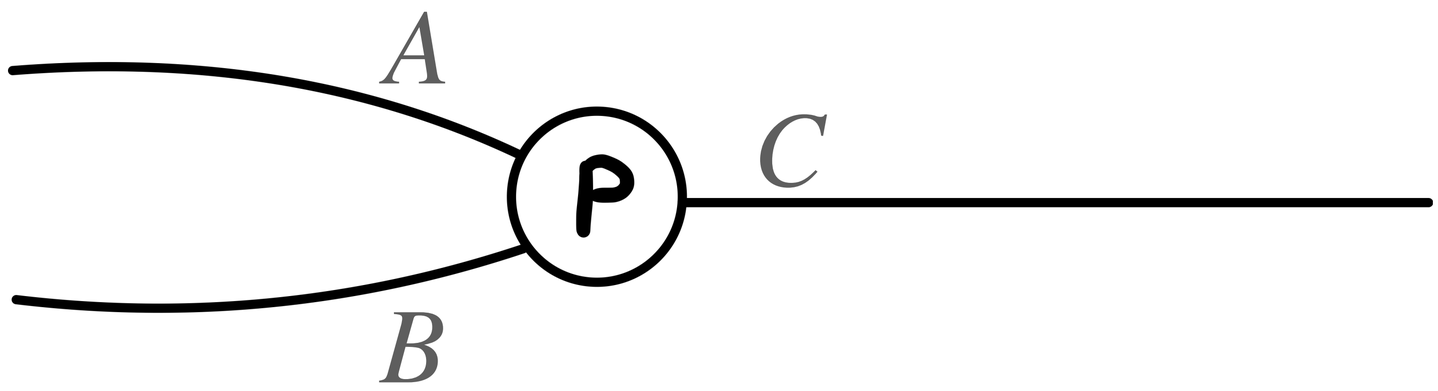
- In practice, verifications and results tend to ***cut across*** multiple facets of program execution in non-trivial ways.
  - For example, establishing a complexity bound for a sorting function may involve functional correctness (I/O behavior) lemmas for its subroutines.
  - Conversely, a complexity 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. ***Key idea: "phase composition".***
  - This research program has led to the solution of several open problems in dependent type theory.

prelude:

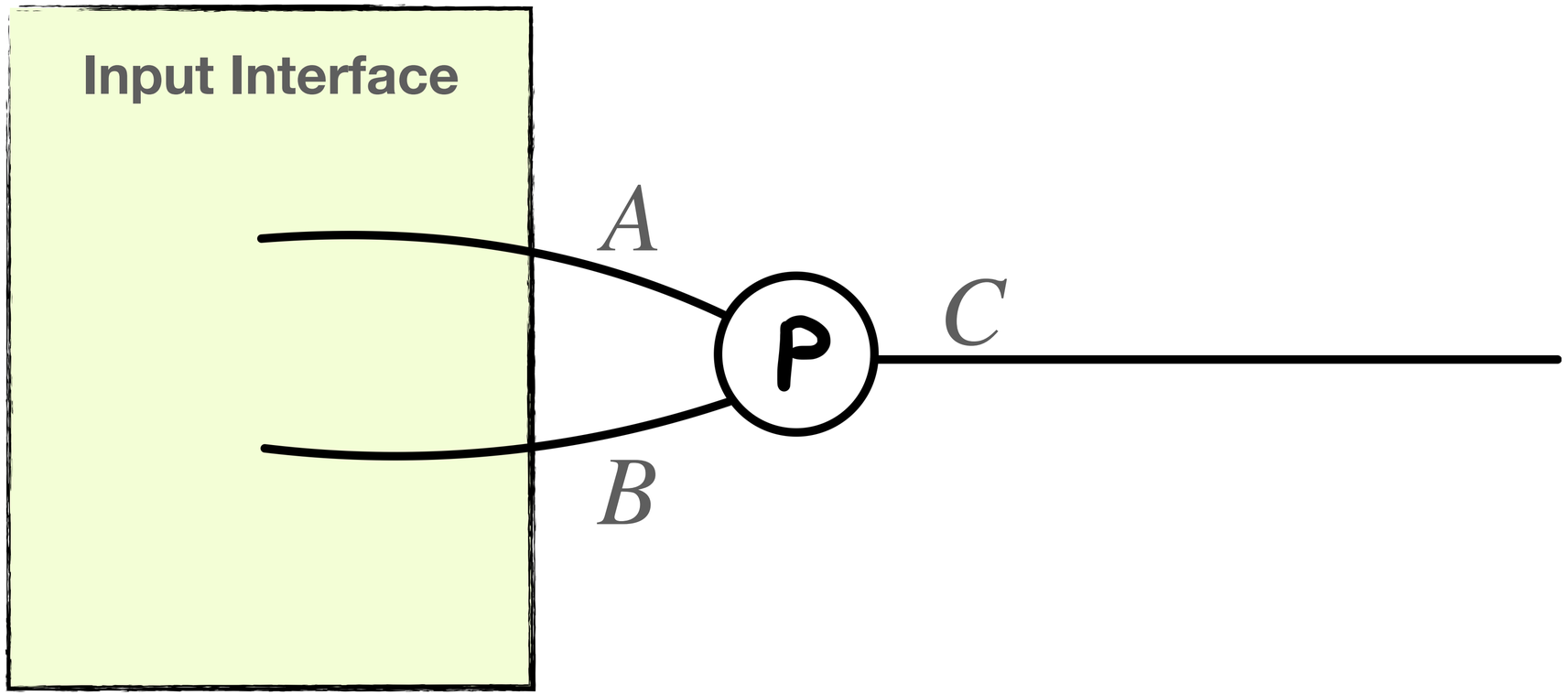
Phases, a new dimension of composition

What is a program?

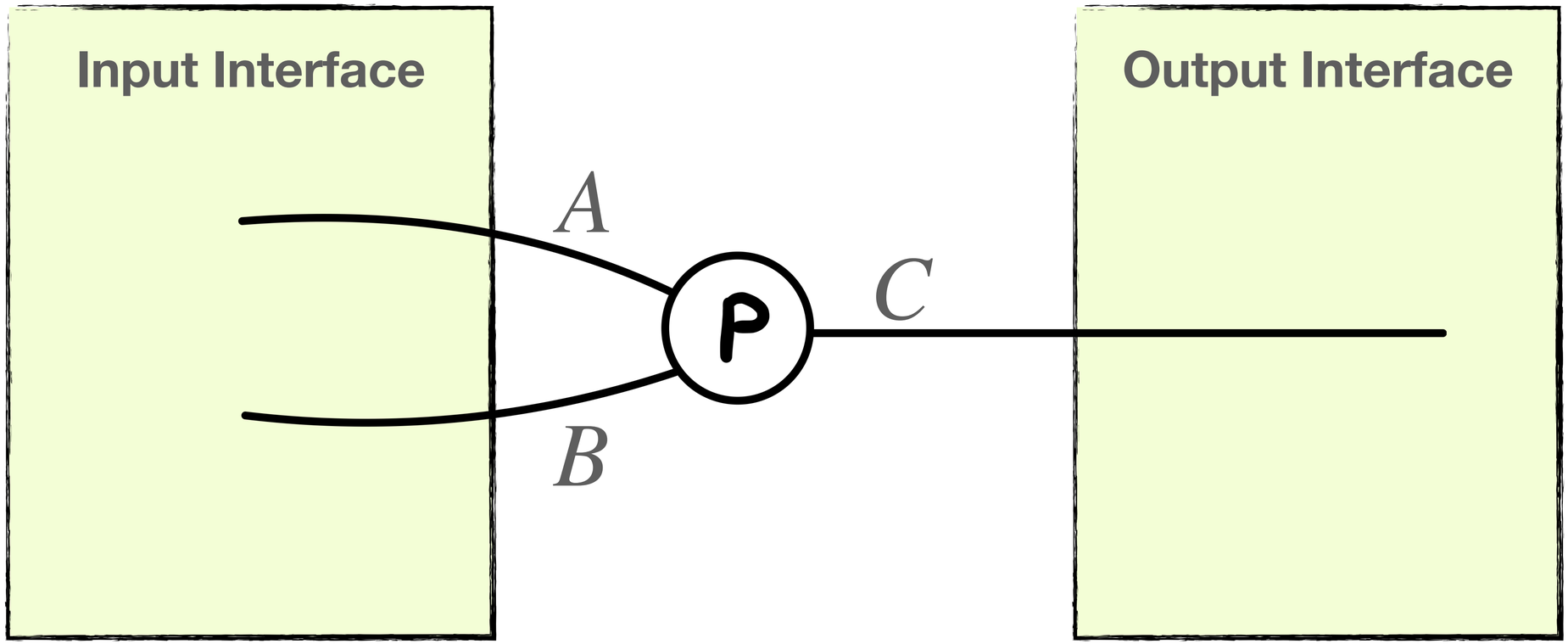
What is a program?



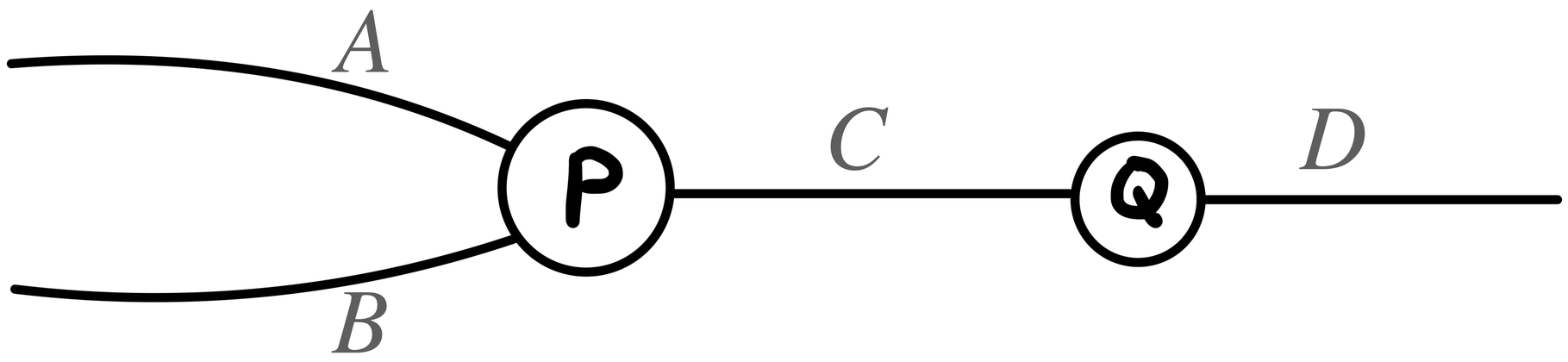
What is a program?



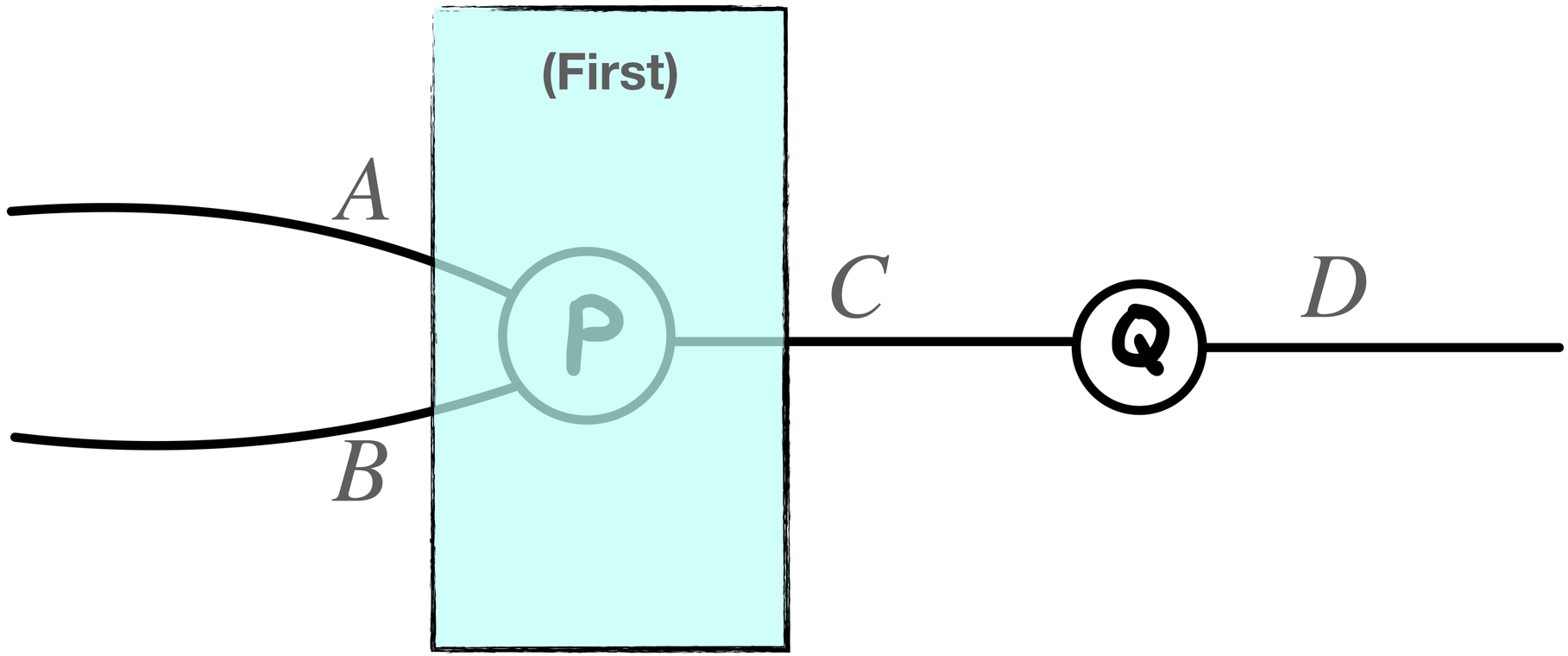
What is a program?



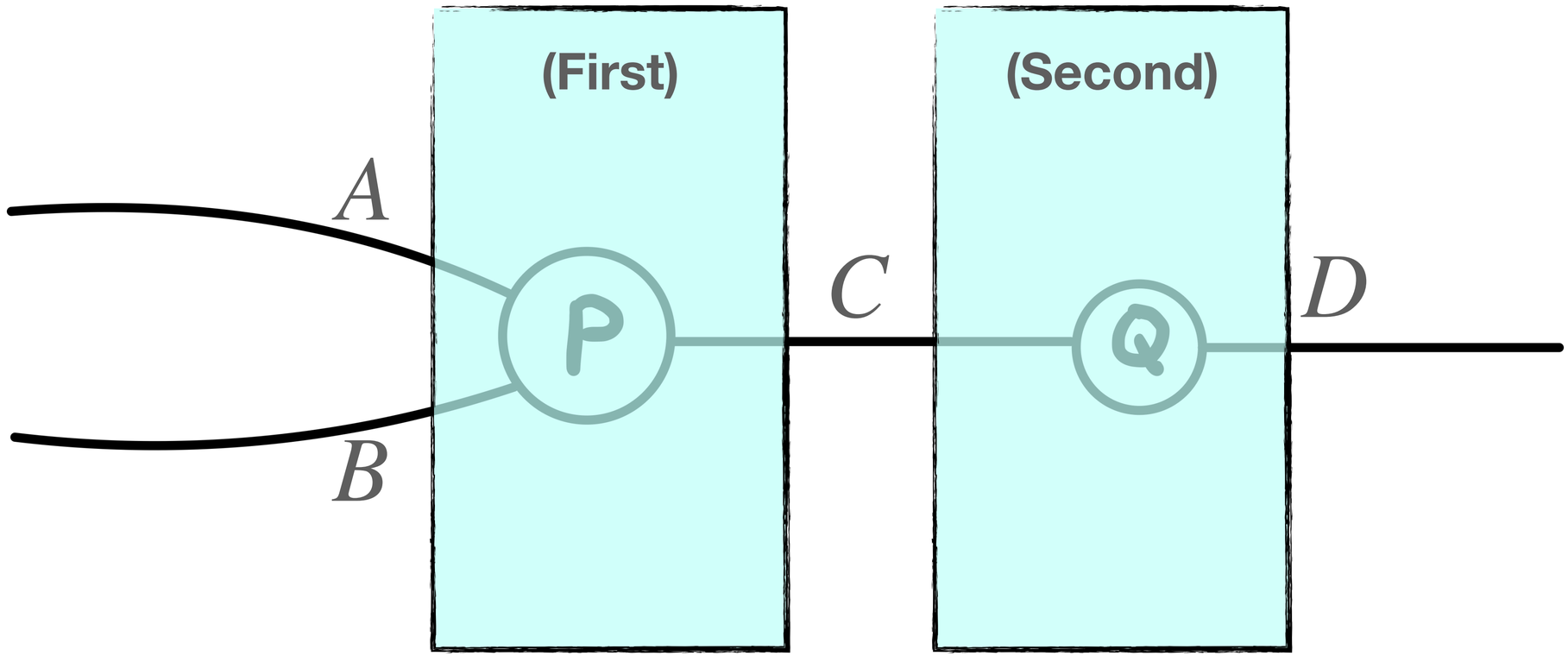
Programs can be composed sequentially.



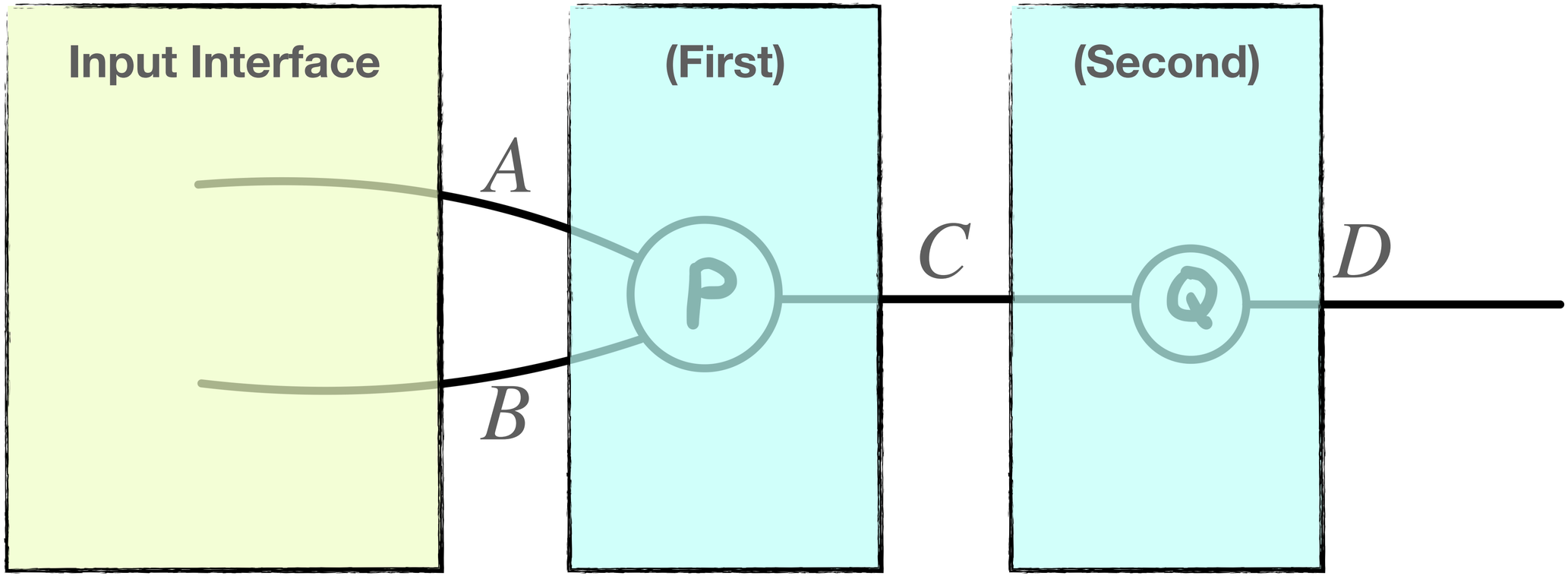
Programs can be composed sequentially.



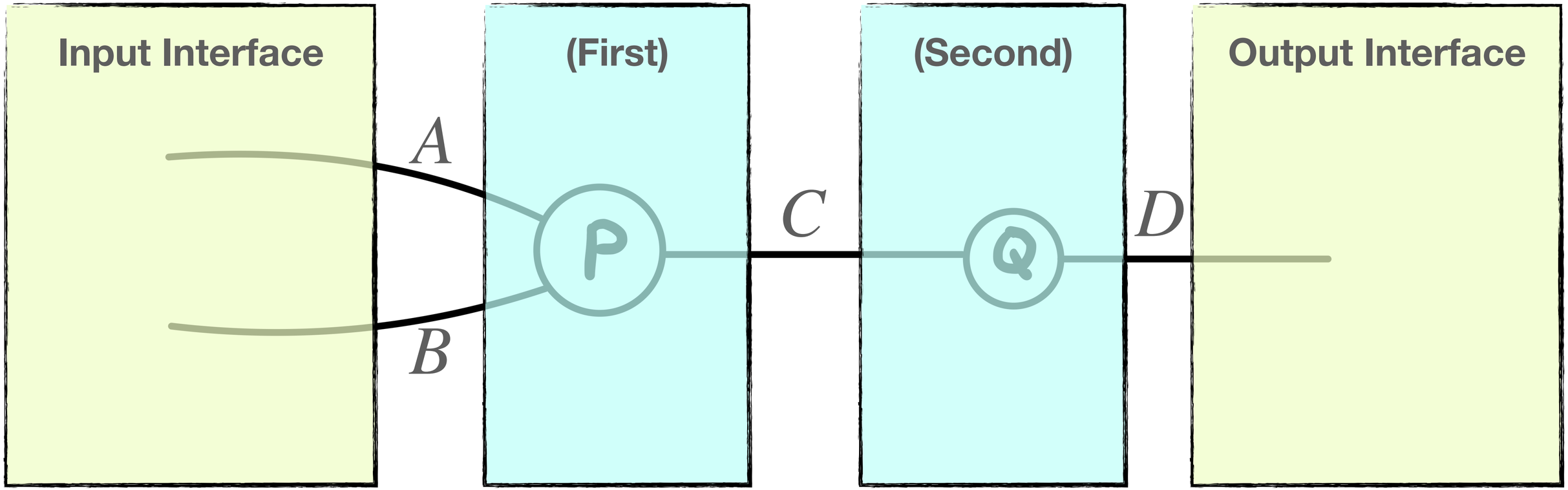
Programs can be composed sequentially.



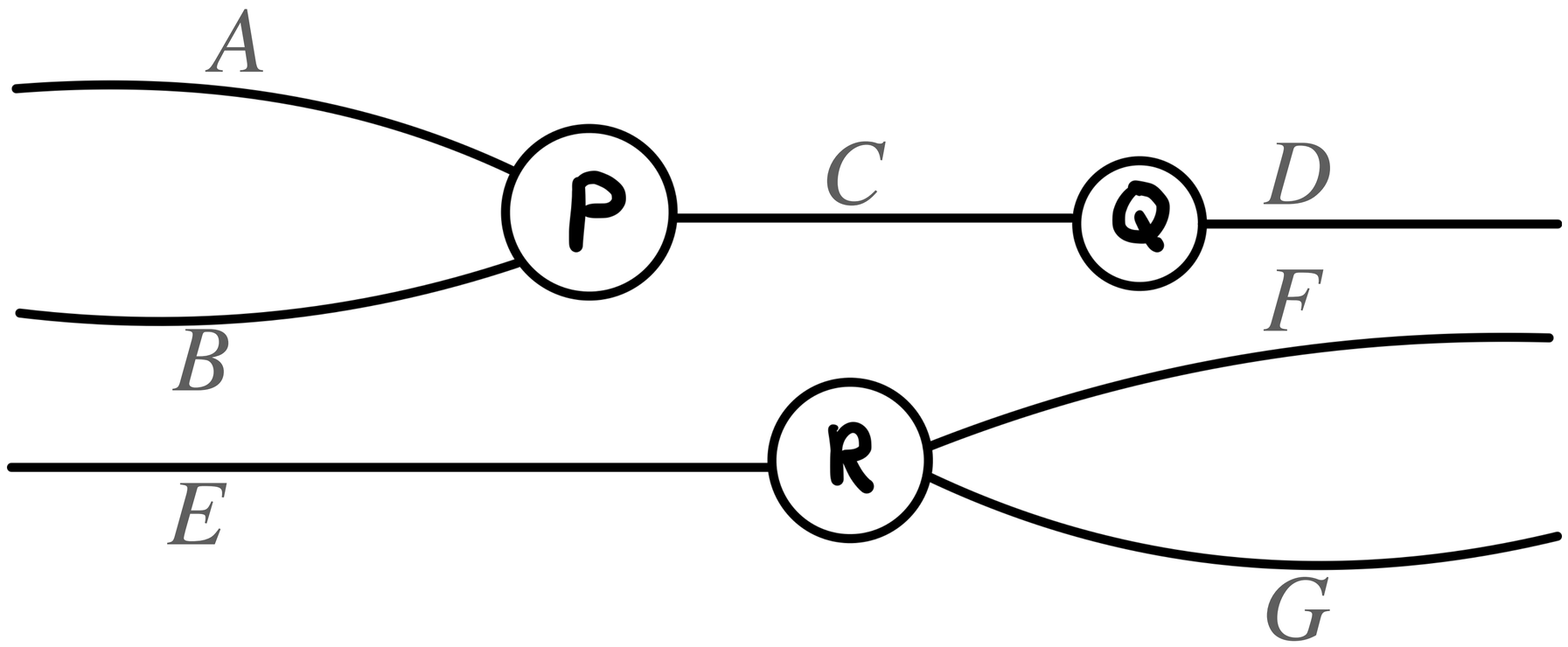
Programs can be composed sequentially.



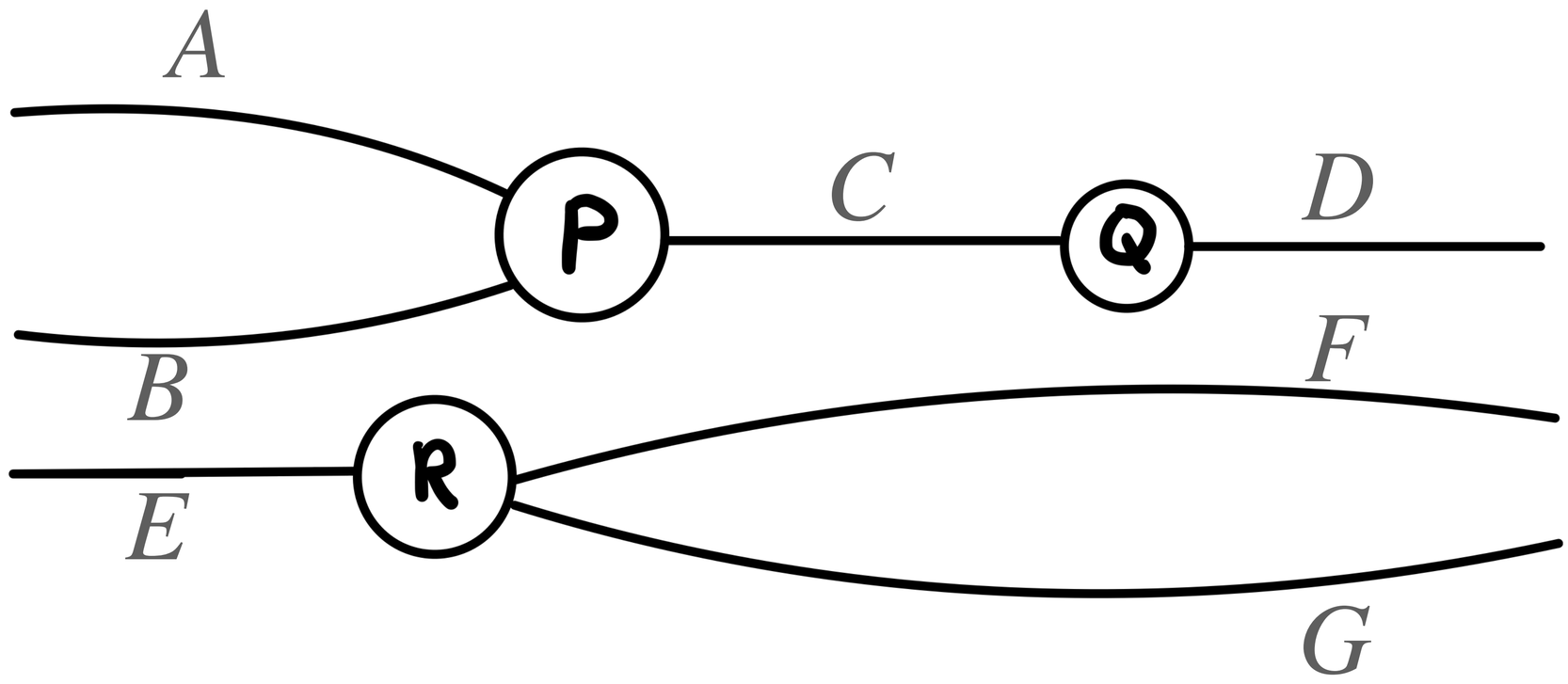
Programs can be composed sequentially.



Programs can be composed in parallel.

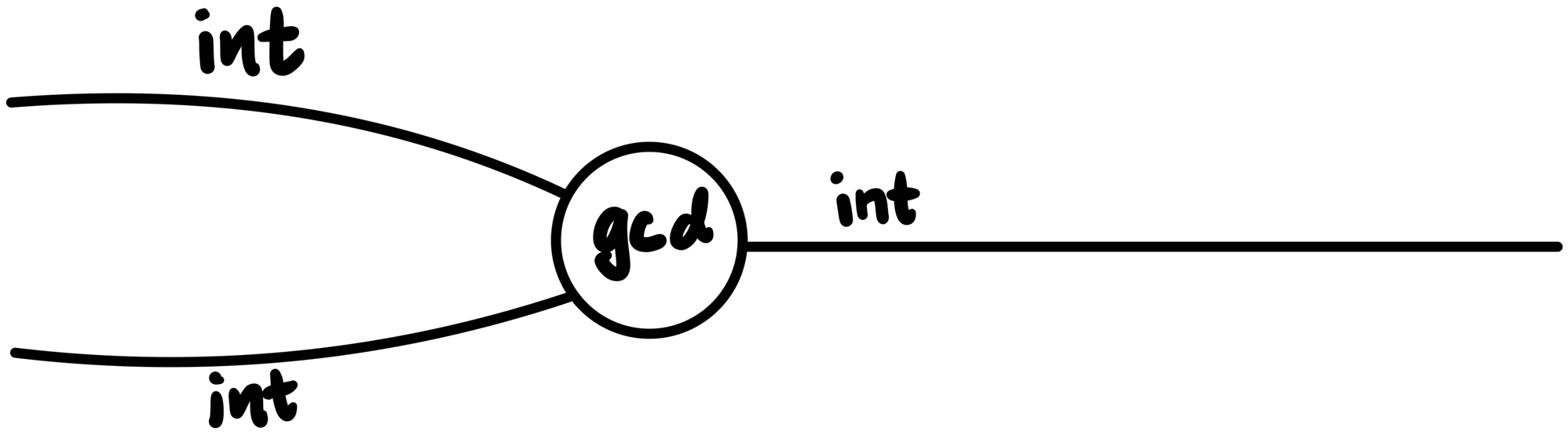


Programs can be composed in parallel.



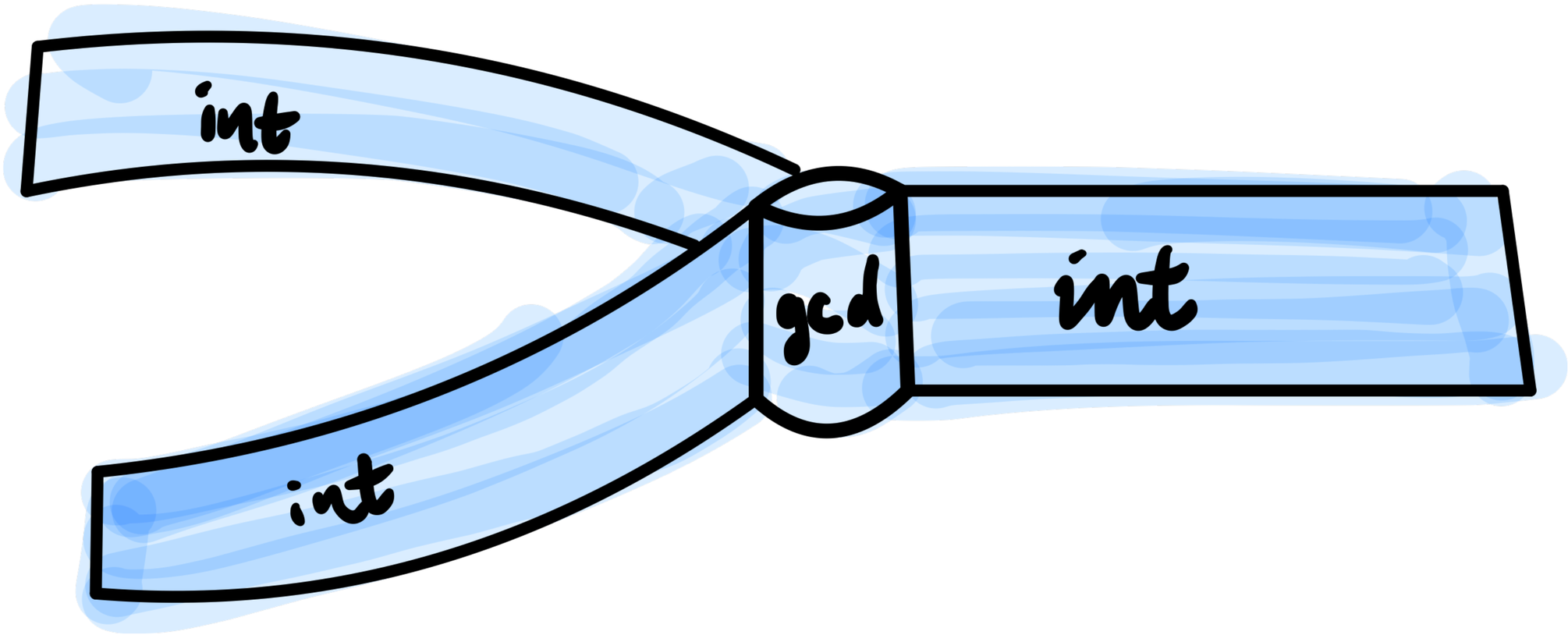
A third compositionality: “phase”?

A third compositionality: “phase”?



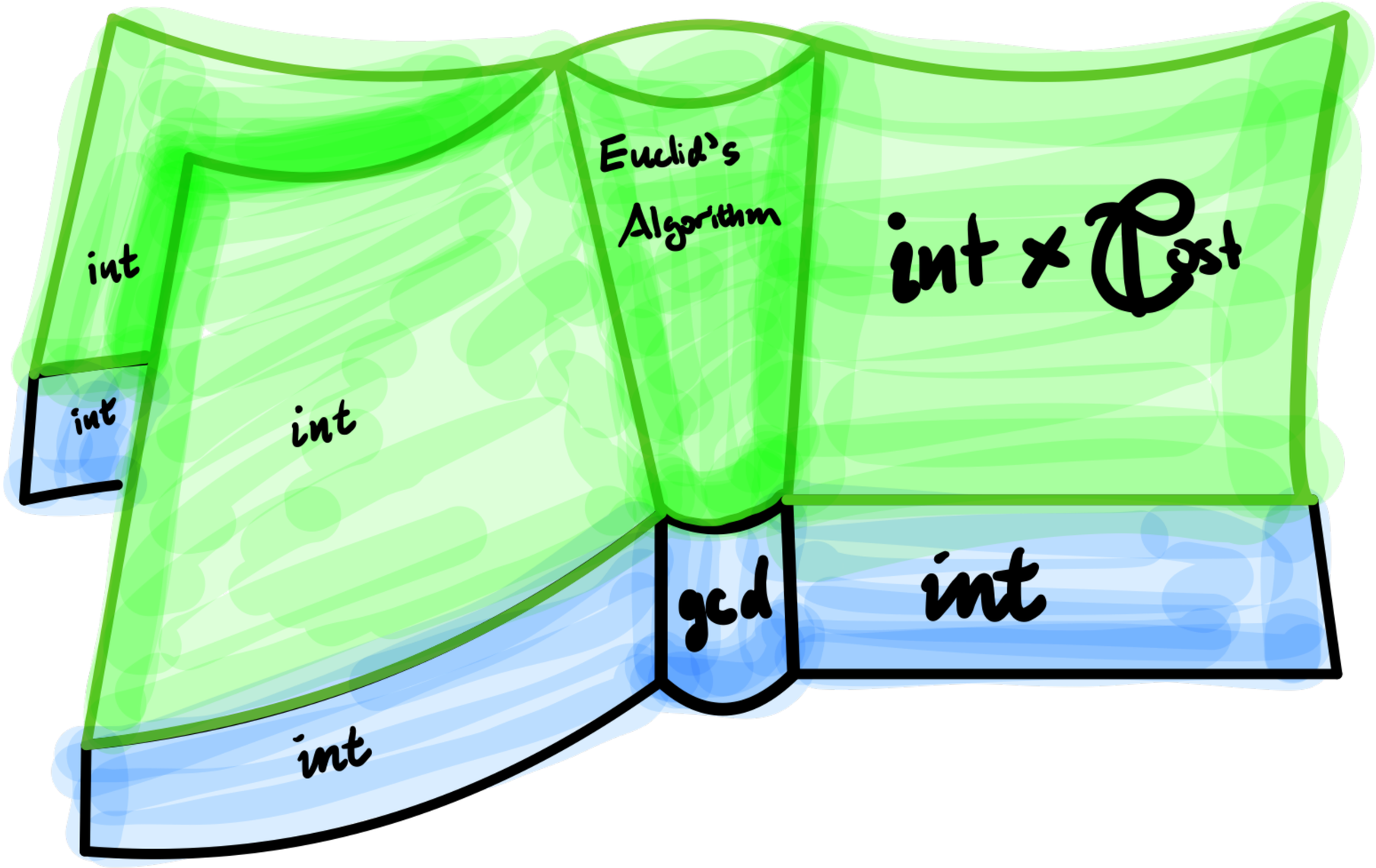
A third compositionality: “phase”?

A third compositionality: “phase”?

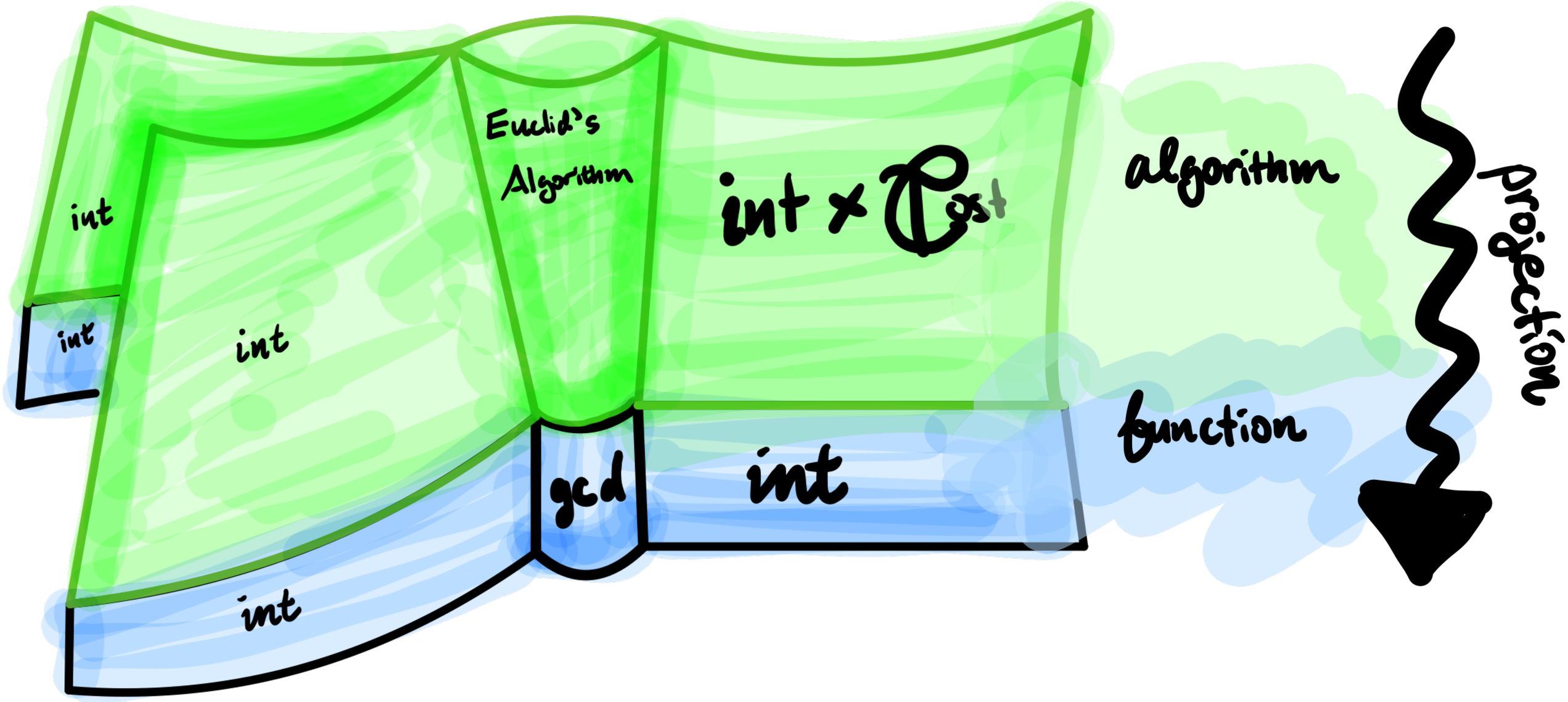


A third compositionality: “phase”?

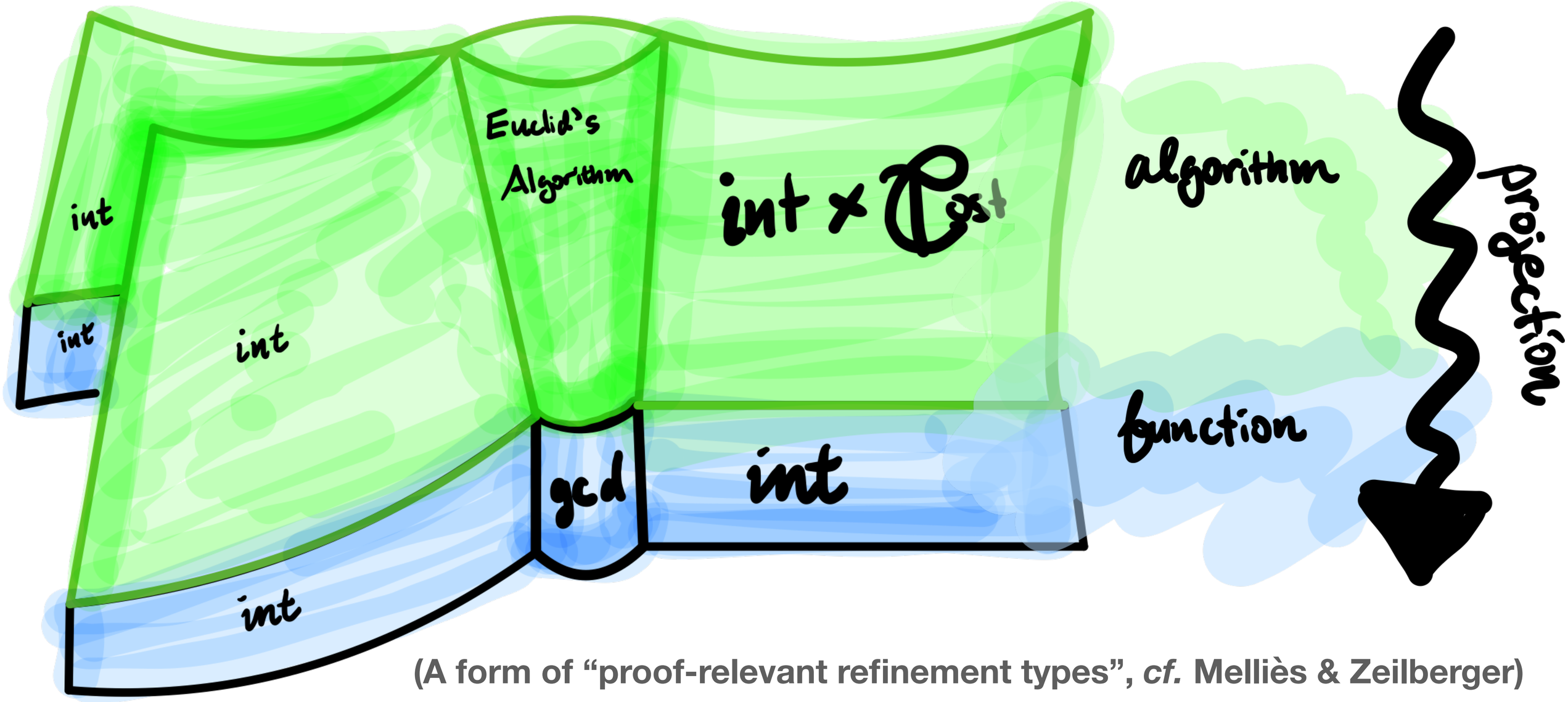
A third compositionality: “phase”?



A third compositionality: “phase”?



A third compositionality: “phase”?



(A form of “proof-relevant refinement types”, *cf.* Melliès & Zeilberger)

New Spaces for Program Semantics:

gluing phases together in computational models

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

Technically, models of program execution 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, models of program execution 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. *Amenable to* ***gluing constructions*** *that correspond to* ***phase composition****.*

Phase composition by gluing spaces

Phase composition by gluing spaces

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”.

Phase composition by gluing spaces

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 facets of computation (see also the recent work of Matache, Moss, Staton).

Example I: reasoning about cost and behavior

Example I: 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 I: 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 I: 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, with one modality that ***imposes abstraction*** (= forgets all cost information) and another complementary modality that ***breaks abstraction*** (= makes available all cost information). **These modalities are arise by treating I/O behavior as a *phase* to which a program can be truncated.**

Example I: 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, with one modality that ***imposes abstraction*** (= forgets all cost information) and another complementary modality that ***breaks abstraction*** (= makes available all cost information). **These modalities are arise by treating I/O behavior as a *phase* to which a program can be truncated.**

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

Example II: secure information flow

Example II: 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 II: 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 together many copies of an existing domain theory 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. **Each security clearance corresponds to a *phase* to which a program may be truncated.**

Example II: 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 together many copies of an existing domain theory 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. **Each security clearance corresponds to a *phase* to which a program may be truncated.**

- Noninterference in the model follows immediately; it is lifted from the model to the programming language by means of a further gluing.

Example II: 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 together many copies of an existing domain theory 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. **Each security clearance corresponds to a *phase* to which a program may be truncated.**

- 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 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 top) of your computational model. The left and right sides represent two implementations of a data structure, and the “top” carries a representation invariant between the two.

TOP: Representation Invariant

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

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

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

TOP: Representation Invariant

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

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

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

TOP: Representation Invariant

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

LEFT: Naïve Queue

$$l : \text{List}$$

RIGHT: Double-Ended Queue

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

TOP: Representation Invariant

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

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 top) of your computational model. The left and right sides represent two implementations of a data structure, and the “top” 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 top) of your computational model. The left and right sides represent two implementations of a data structure, and the “top” 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 top) of your computational model. The left and right sides represent two implementations of a data structure, and the “top” 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 make explicit 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 make explicit the "laws of motion" of glued notions of space.
    - For specialists: **STC** is the internal language of *Artin–Wraith gluings of toposes*. **STC** combines modularly 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 make explicit the "laws of motion" of glued notions of space.
    - For specialists: **STC** is the internal language of *Artin–Wraith gluings of toposes*. **STC** combines modularly 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 make explicit the "laws of motion" of glued notions of space.
  - For specialists: **STC** is the internal language of *Artin–Wraith gluings of toposes*. **STC** combines modularly 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. **Synthesis with recent developments in homotopy type theory, ∞-category theory.**

Future and ongoing research directions

- Towards ***higher-dimensional domain theory***: applications to concurrency, moduli spaces of computational domains. **Synthesis with recent developments in homotopy type theory, ∞-category theory.**

- ***General domain theory:*** several classes of models of synthetic & axiomatic domain theory are well-studied, but much remains to do in the general theory of composing domain theories. (See Fiore's thesis; on guarded side, Palombi & S. in MFPS '22.) **Needed in order to scale up my methods to more sophisticated programming constructs.**

Future and ongoing research directions

- Towards ***higher-dimensional domain theory***: applications to concurrency, moduli spaces of computational domains. **Synthesis with recent developments in homotopy type theory, ∞-category theory.**

- ***General domain theory:*** several classes of models of synthetic & axiomatic domain theory are well-studied, but much remains to do in the general theory of composing domain theories. (See Fiore's thesis; on guarded side, Palombi & S. in MFPS '22.) **Needed in order to scale up my methods to more sophisticated programming constructs.**

- ***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). **Applying semantic methods to "realistic" programming languages rather than toy languages.**

Thanks!

Funded by
the European Union