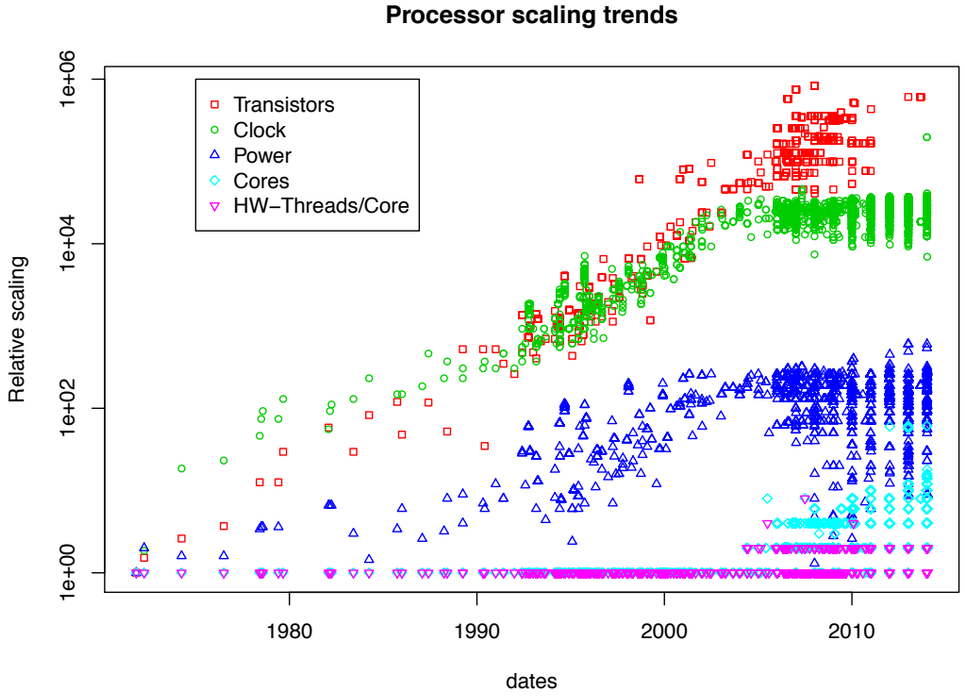
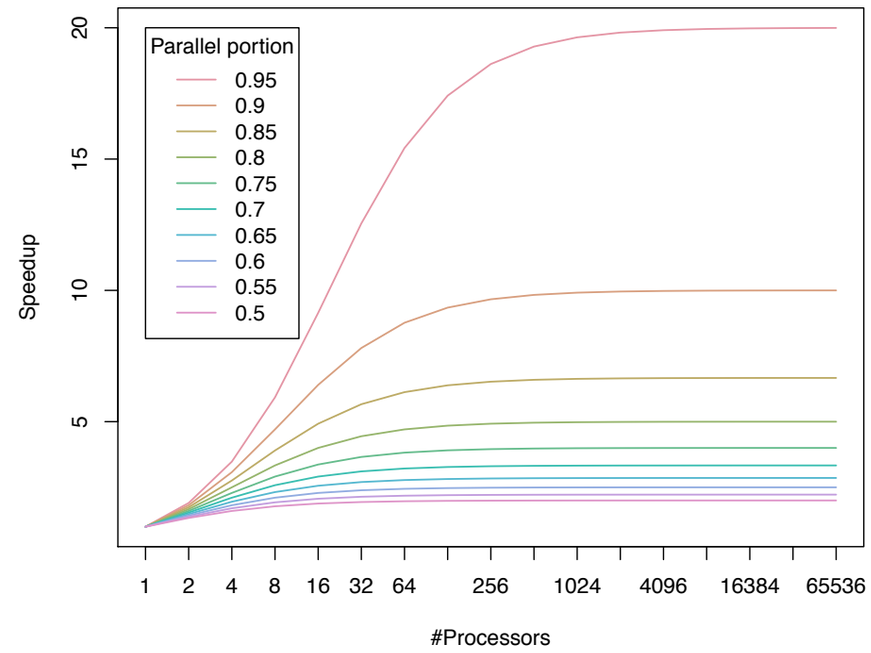
Tech Problem

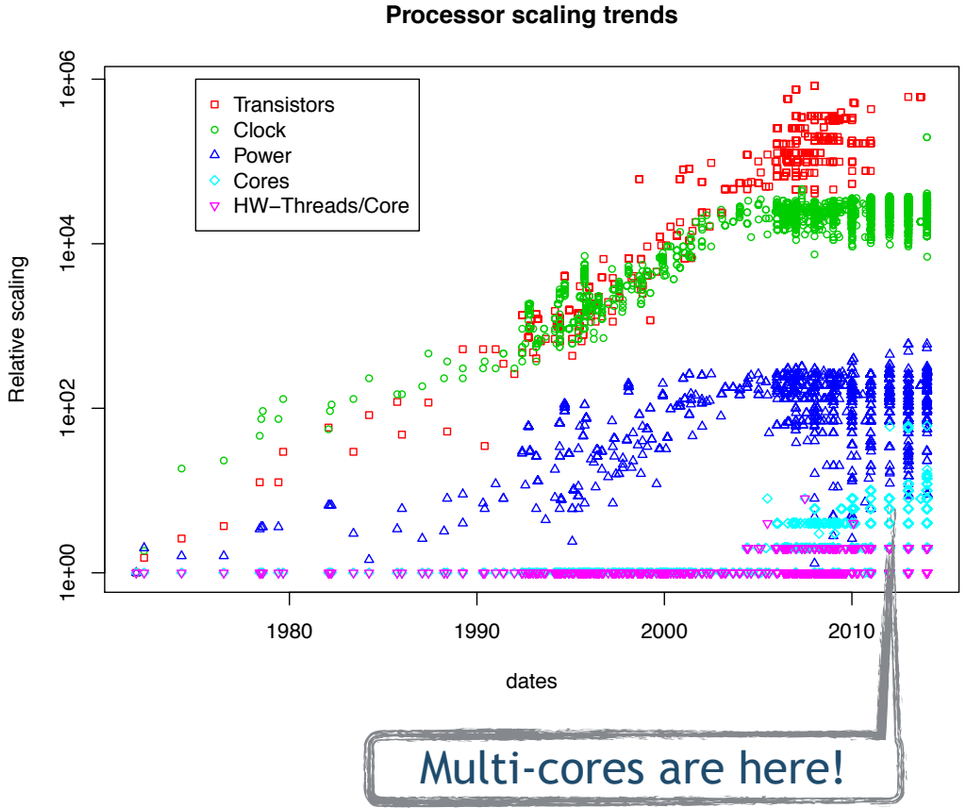
Sebastian Ertel

My Tech Problem of the last years: Implicit Parallel Programming

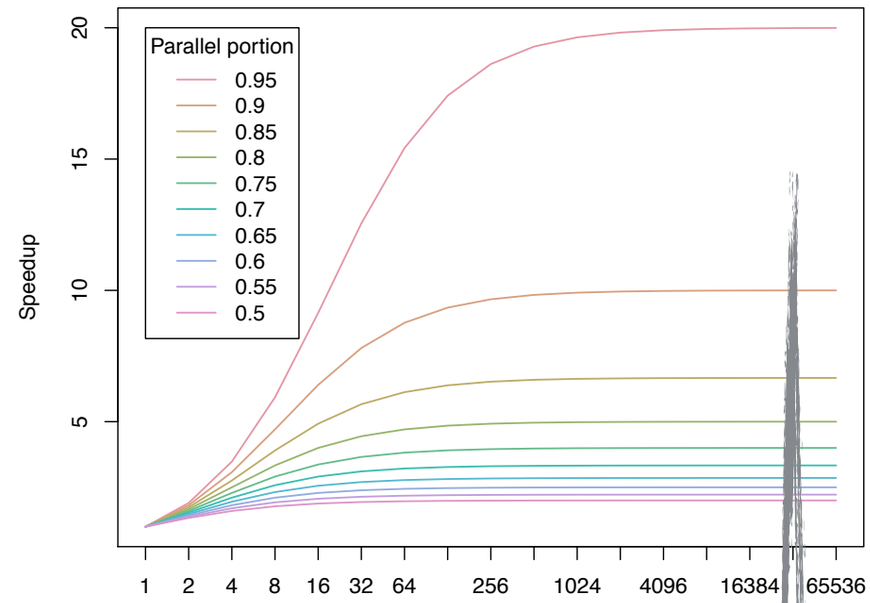




My Tech Problem of the last years: Implicit Parallel Programming

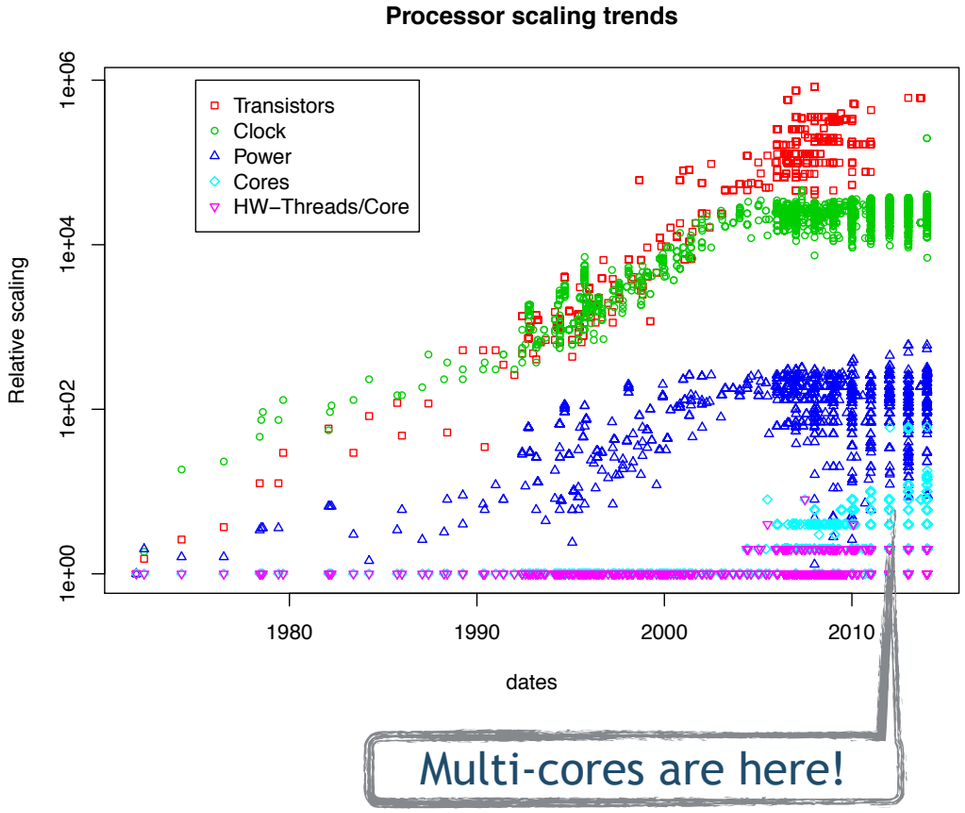


Multi-cores are here!

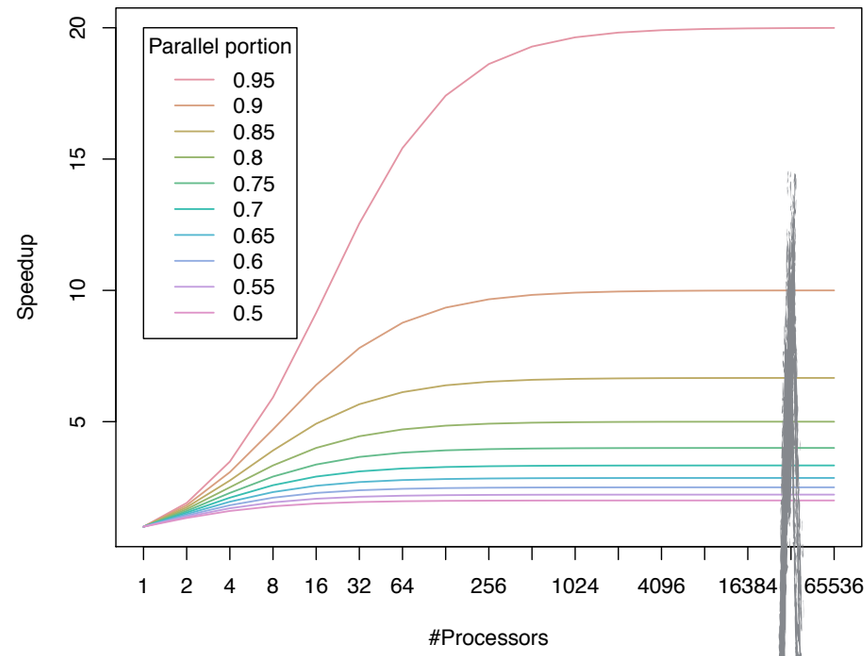


Squeeze out parallelism!

My Tech Problem of the last years: Implicit Parallel Programming



Multi-cores are here!



Squeeze out parallelism!

Parallel programming is hard!

- ➪ Manually introduce concurrency.
- ➪ Shared memory: data races.
- ➪ Threads/locks: deadlocks.

Programming Model

```
fn server(port:int) {
  let parse_state = init_parse_state();

  let acceptor = init_server(port);
  for cnn in acceptor {
    let data = read(cnn);
    let req = parse_state.parse(data);
    let resp = if (is_get(req)) {
                 load(req)
               } else {
                 store(req)
               }
    reply(resp);
  };
}
```

Programming Model

Algorithms vs. Functions

```
fn server(port:int) {
  let parse_state = init_parse_state();

  let acceptor = init_server(port);
  for cnn in acceptor {
    let data = read(cnn);
    let req = parse_state.parse(data);
    let resp = if (is_get(req)) {
                 load(req)
               } else {
                 store(req)
               }
    reply(resp);
  };
}
```

Stateless vs. Stateful Functions

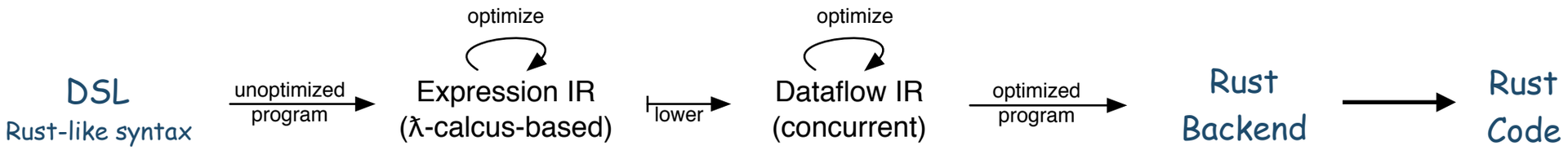
Programming Model

Algorithms vs. Functions

```
fn server(port:int) {
  let parse_state = init_parse_state();

  let acceptor = init_server(port);
  for cnn in acceptor {
    let data = read(cnn);
    let req = parse_state.parse(data);
    let resp = if (is_get(req)) {
                 load(req)
               } else {
                 store(req)
               }
    reply(resp);
  };
}
```

Stateless vs. Stateful Functions

Programming Model



Algorithms vs. Functions

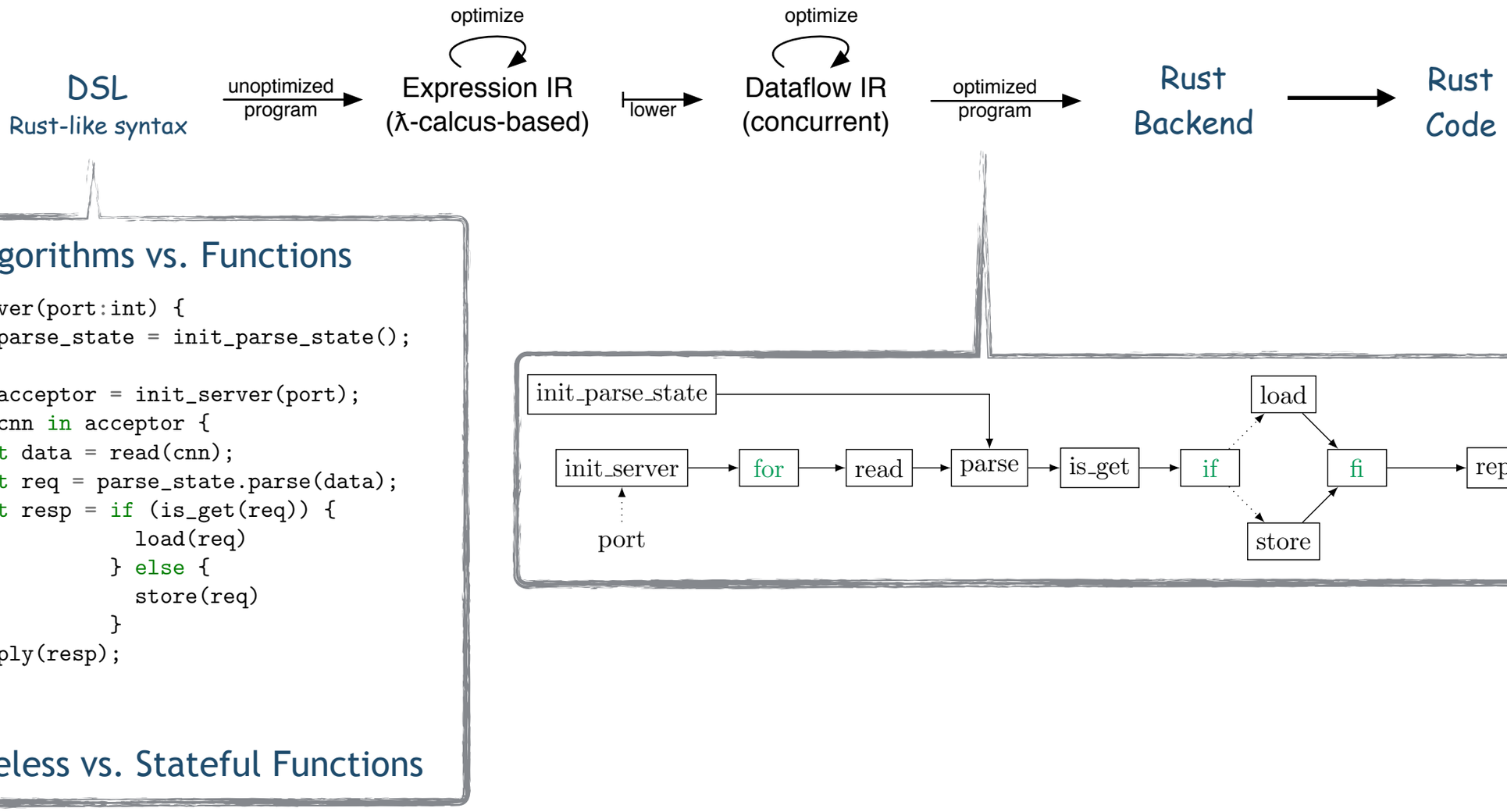
```
fn server(port:int) {
  let parse_state = init_parse_state();

  let acceptor = init_server(port);
  for cnn in acceptor {
    let data = read(cnn);
    let req = parse_state.parse(data);
    let resp = if (is_get(req)) {
                 load(req)
               } else {
                 store(req)
               }
    reply(resp);
  };
}
```

Stateless vs. Stateful Functions

➪ Broadly applicable!

➪ Algorithms as **higher-level abstractions!**

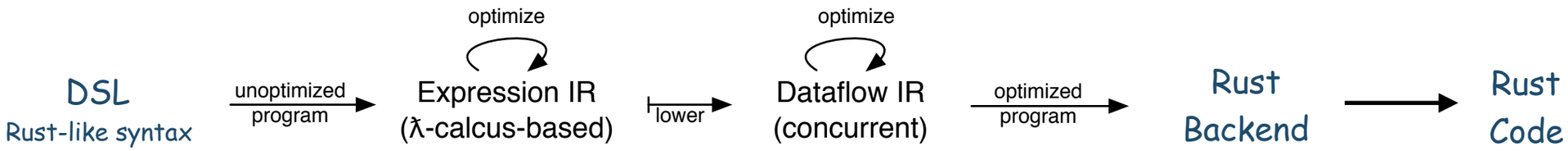
Programming Model



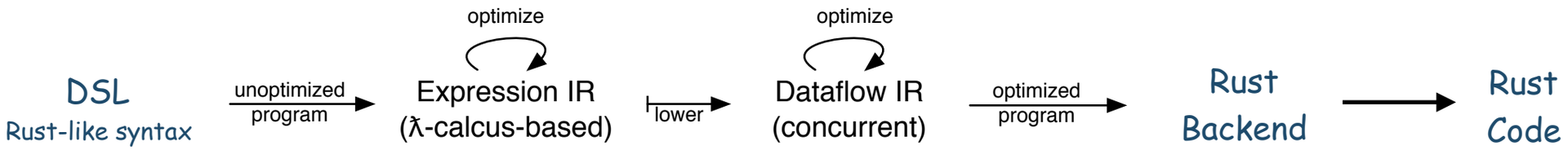
➪ Broadly applicable!

➪ Algorithms as **higher-level abstractions!**

Programming Framework

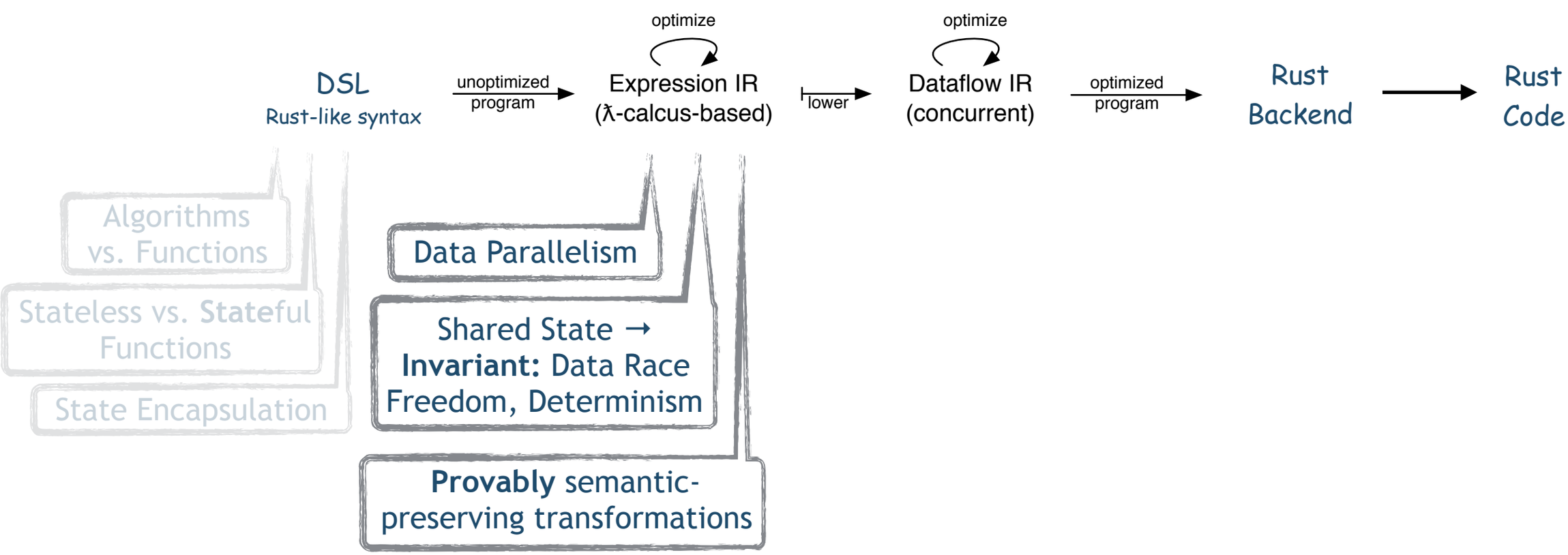


Programming Framework

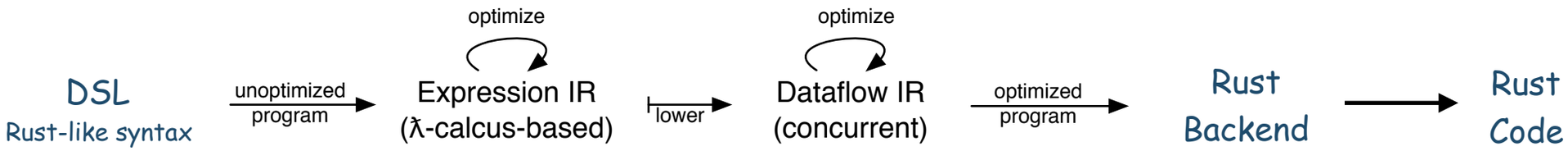


- Algorithms vs. Functions
- Stateless vs. **State**ful Functions
- State Encapsulation

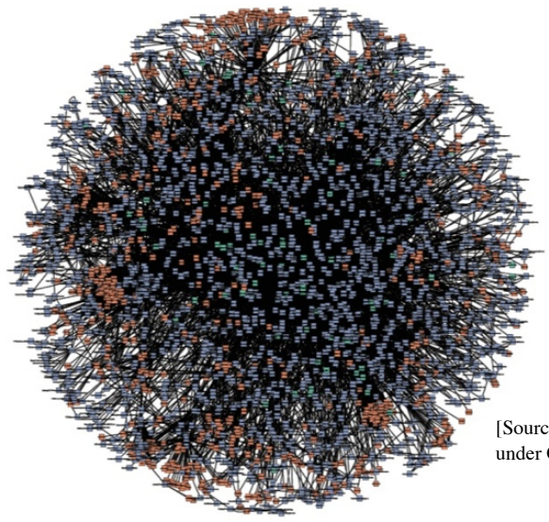
Programming Framework



Optimizing I/O of Micro-services



Monolithic server programs



amazon

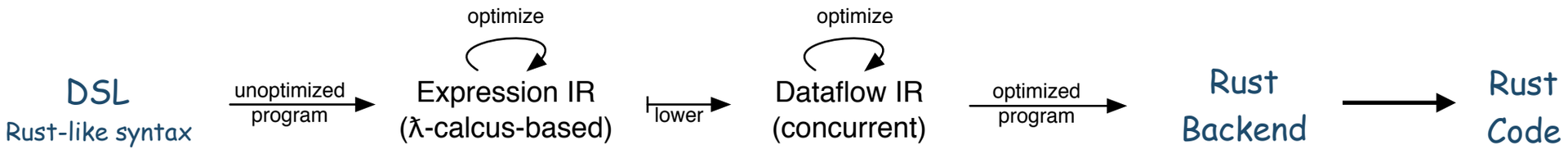
http://amazon.com

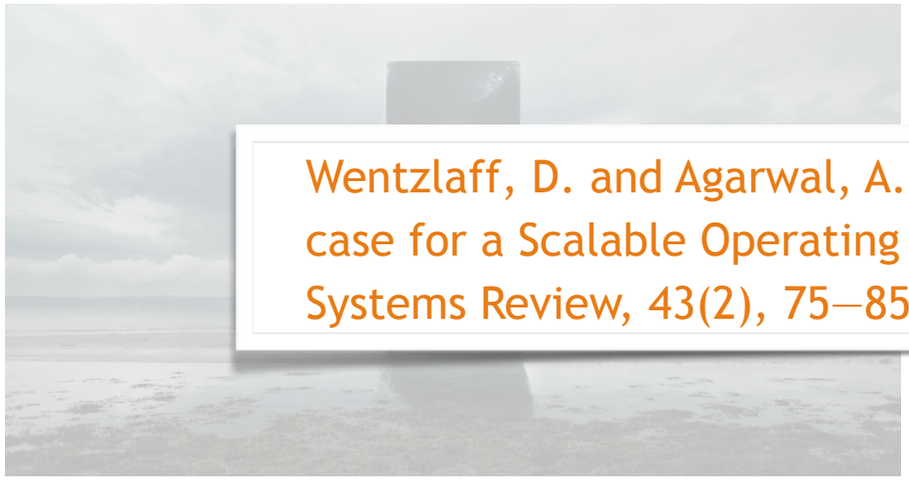
[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: h p://bit.ly/2zboHTK]

Micro-service-based systems

Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

Optimizing I/O of Micro-services





amazon

Wentzlaff, D. and Agarwal, A., (2009). Factored Operating Systems (fos): The case for a Scalable Operating System for Multicores. ACM SIGOPS Operating Systems Review, 43(2), 75—85

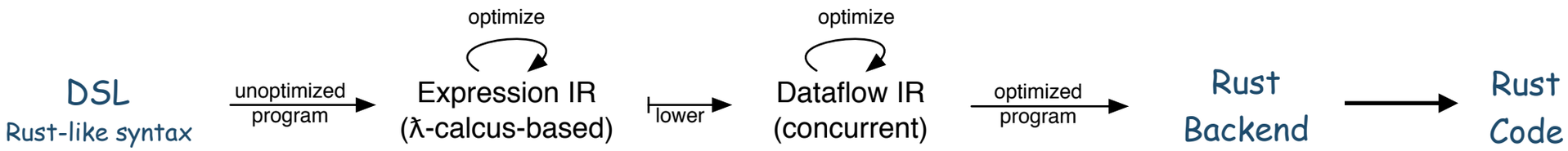
[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: h p://bit.ly/2zboHTK]

Monolithic server programs

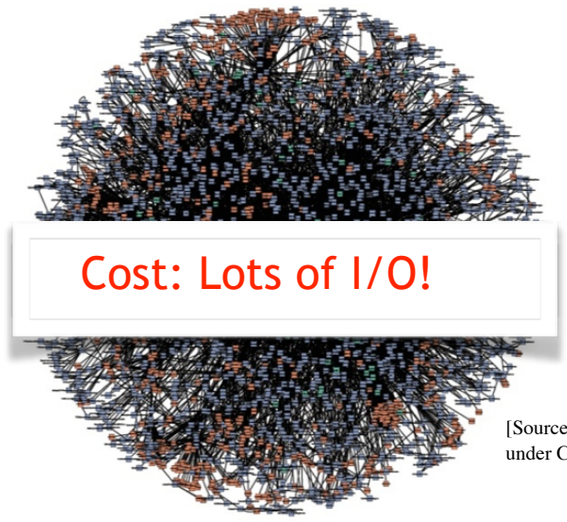
Micro-service-based systems

Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

Optimizing I/O of Micro-services



Monolithic server programs

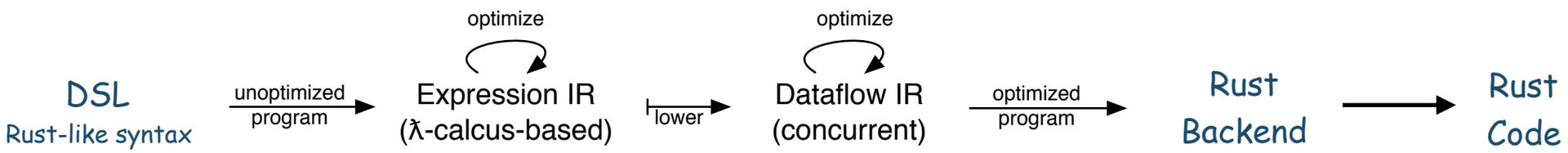


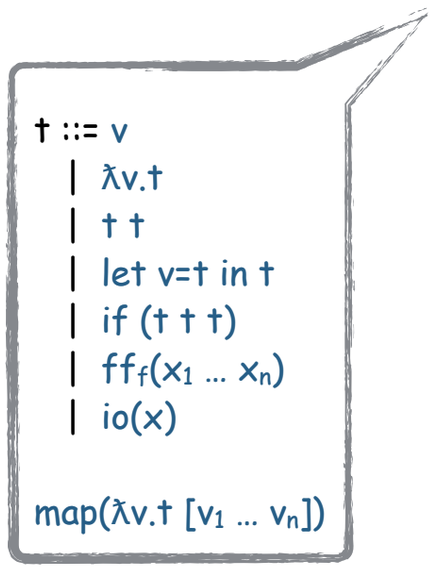
[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: h p://bit.ly/2zboHTK]

Micro-service-based systems

Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

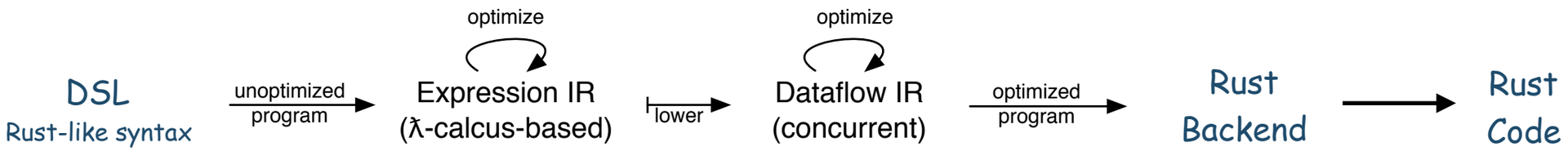
Optimizing I/O of Micro-services





Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

Optimizing I/O of Micro-services



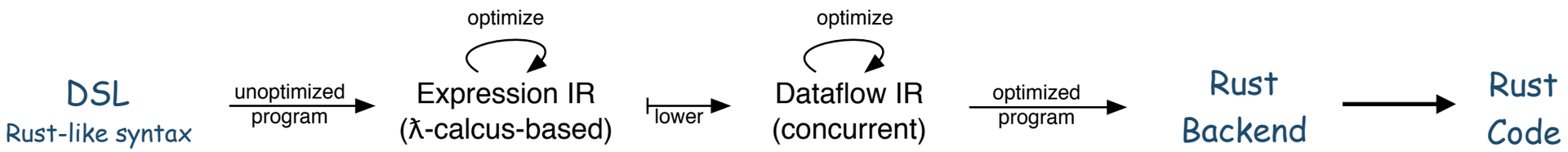
t ::= x
| $TT_f(x_1 \dots x_n)$
| io(x)

map(λv.t $[v_1 \dots v_n]$)

Disclaimer: Presentation diverges (from paper) for visualization purposes. Data flow graphs before and after the transformations instead of lambda expressions.

Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

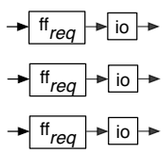
Optimizing I/O of Micro-services



t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

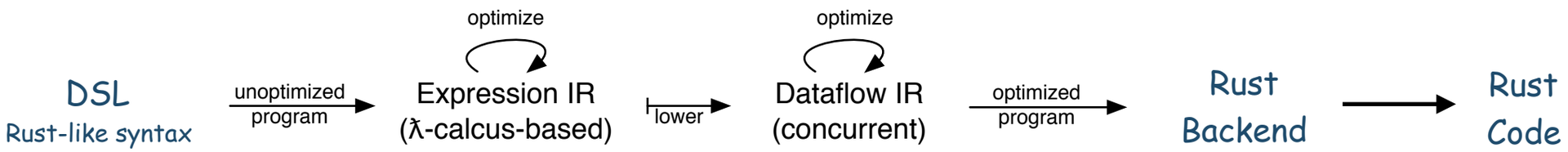
map(λv.t [$v_1 \dots v_n$])

- Grouping independent I/O calls:



Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

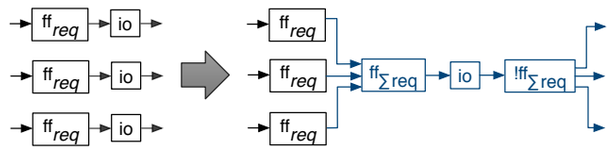
Optimizing I/O of Micro-services



t ::= v
| ƛv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

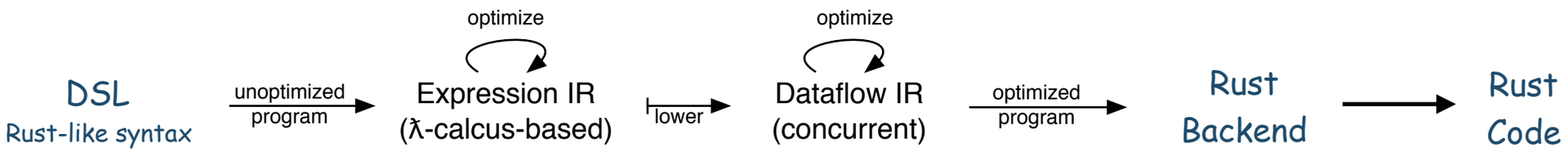
map(ƛv.t $[v_1 \dots v_n]$)

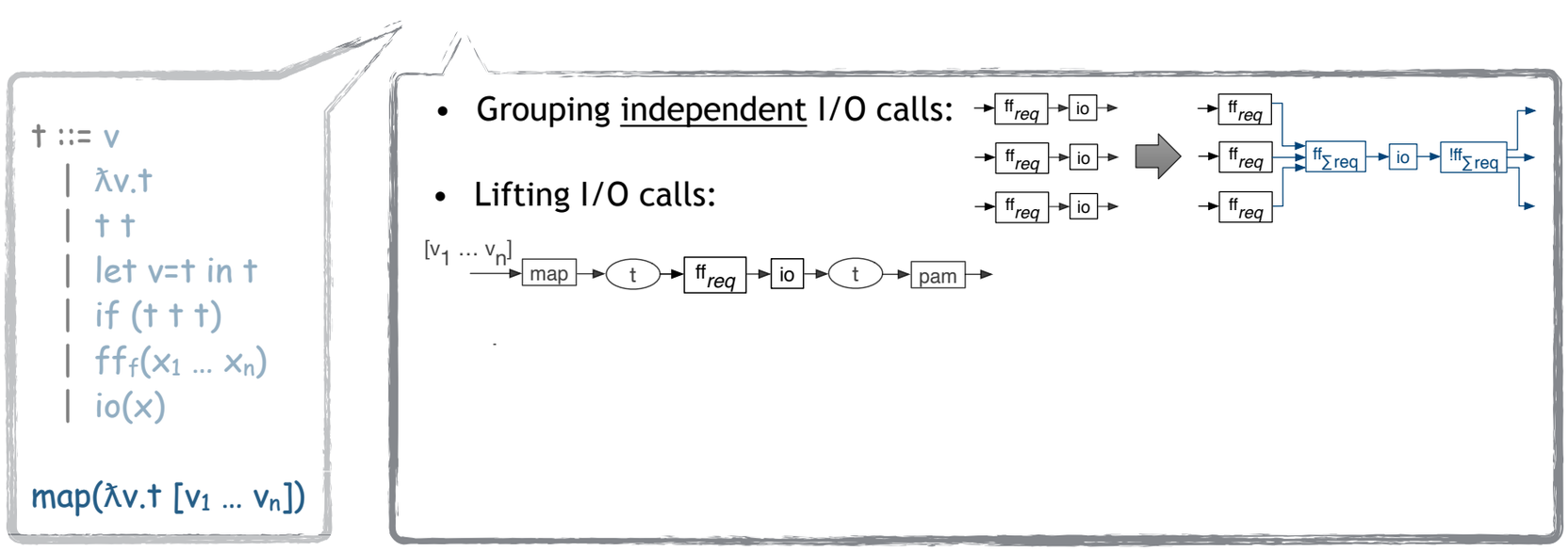
- Grouping independent I/O calls:



Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

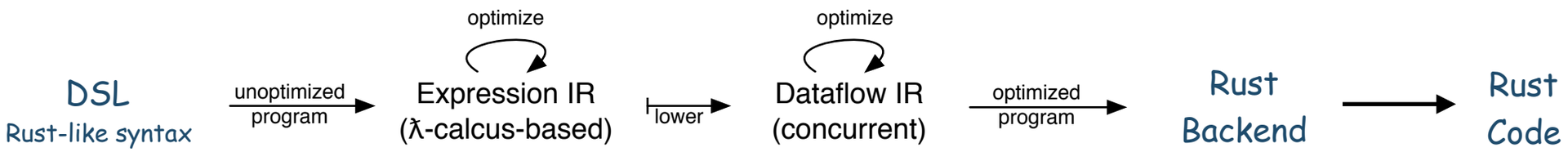
Optimizing I/O of Micro-services

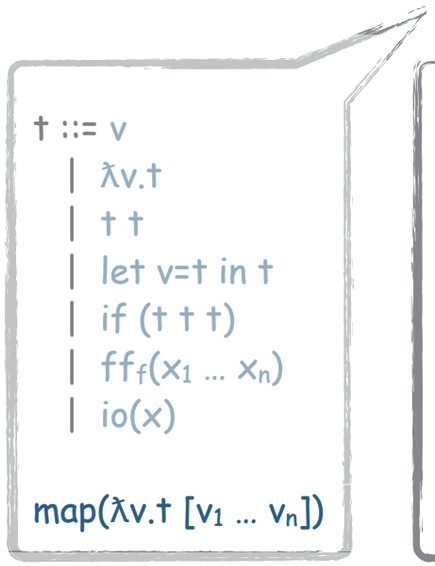


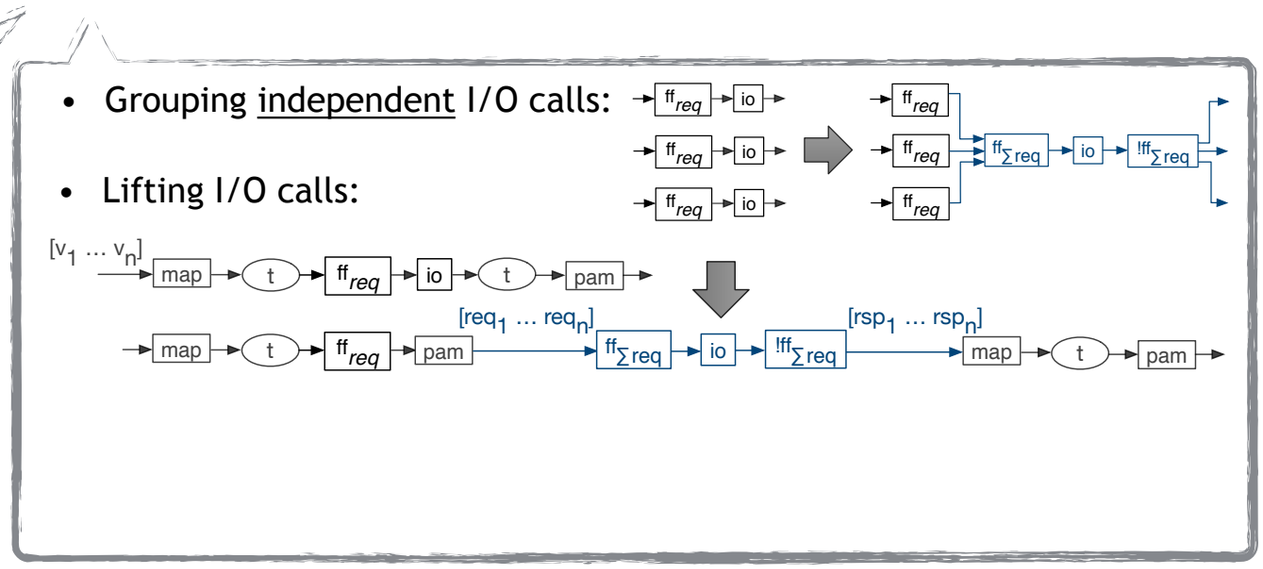


Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

Optimizing I/O of Micro-services

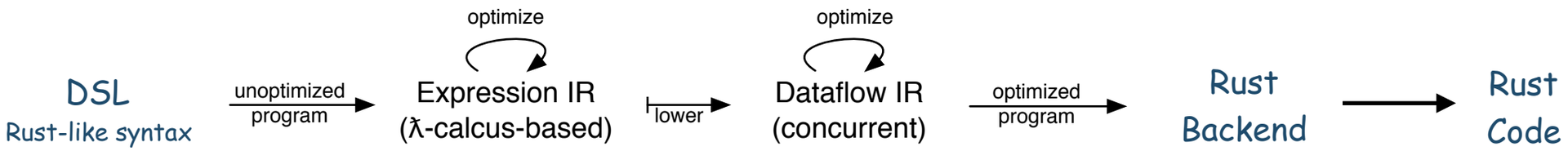


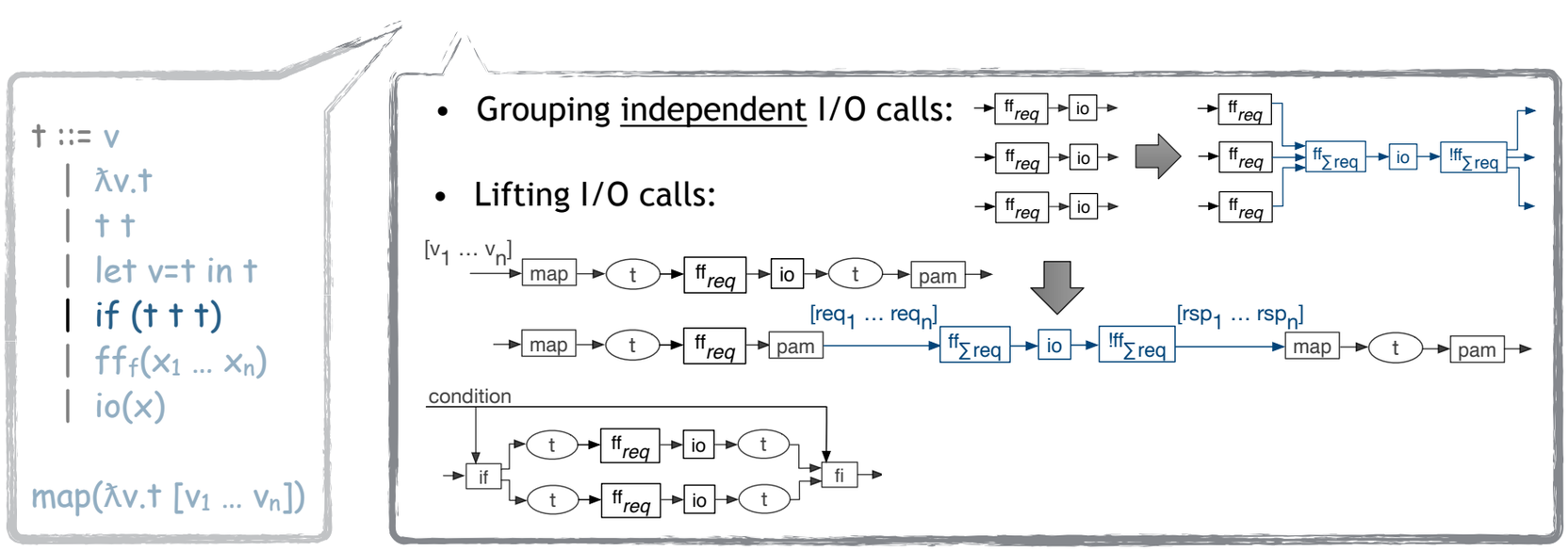




Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

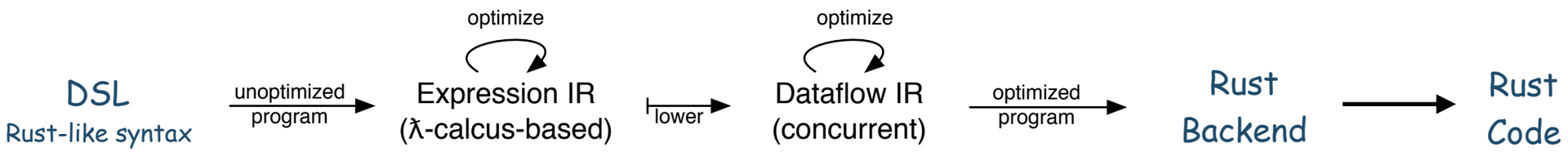
Optimizing I/O of Micro-services

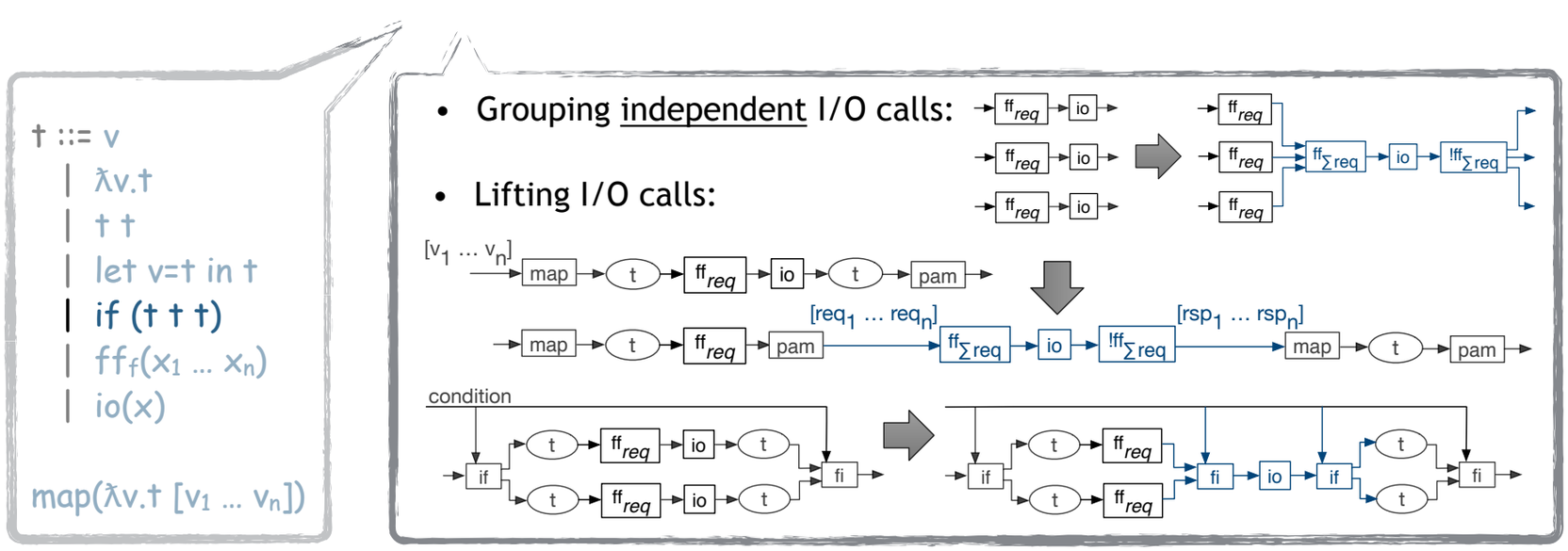




Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

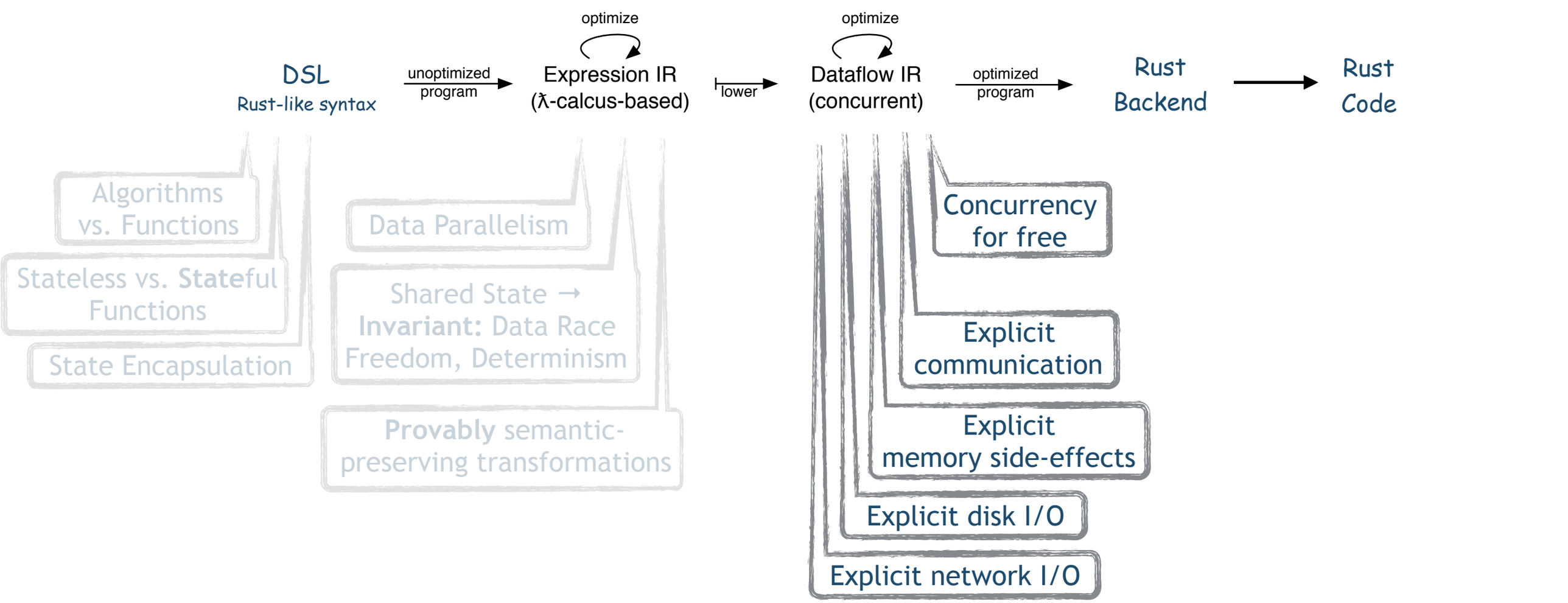
Optimizing I/O of Micro-services



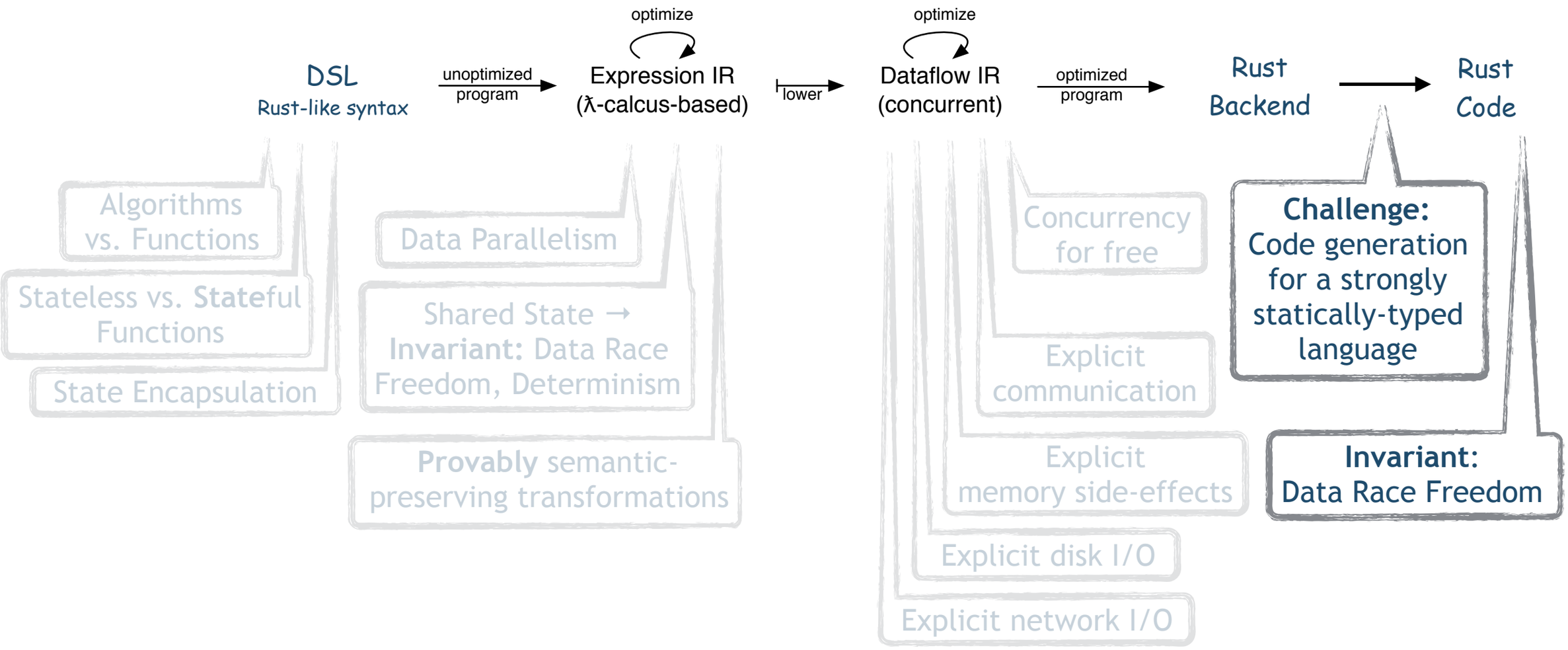


Ertel, S., Goens, A., Adam, J. and Castrillon, J., 2018, February. Compiling for concise code and efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 104-115). ACM.

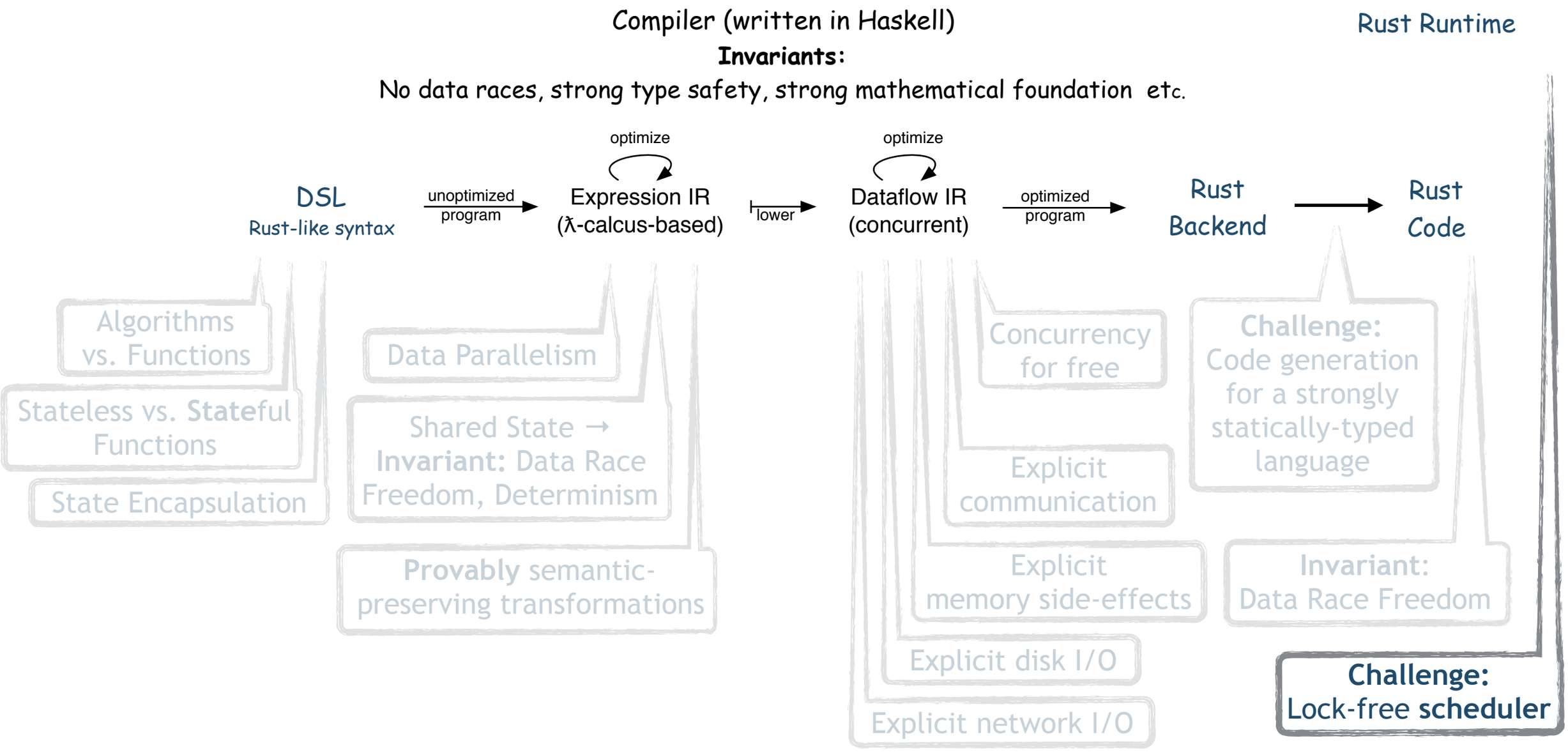
Programming Framework



Programming Framework



Programming Framework

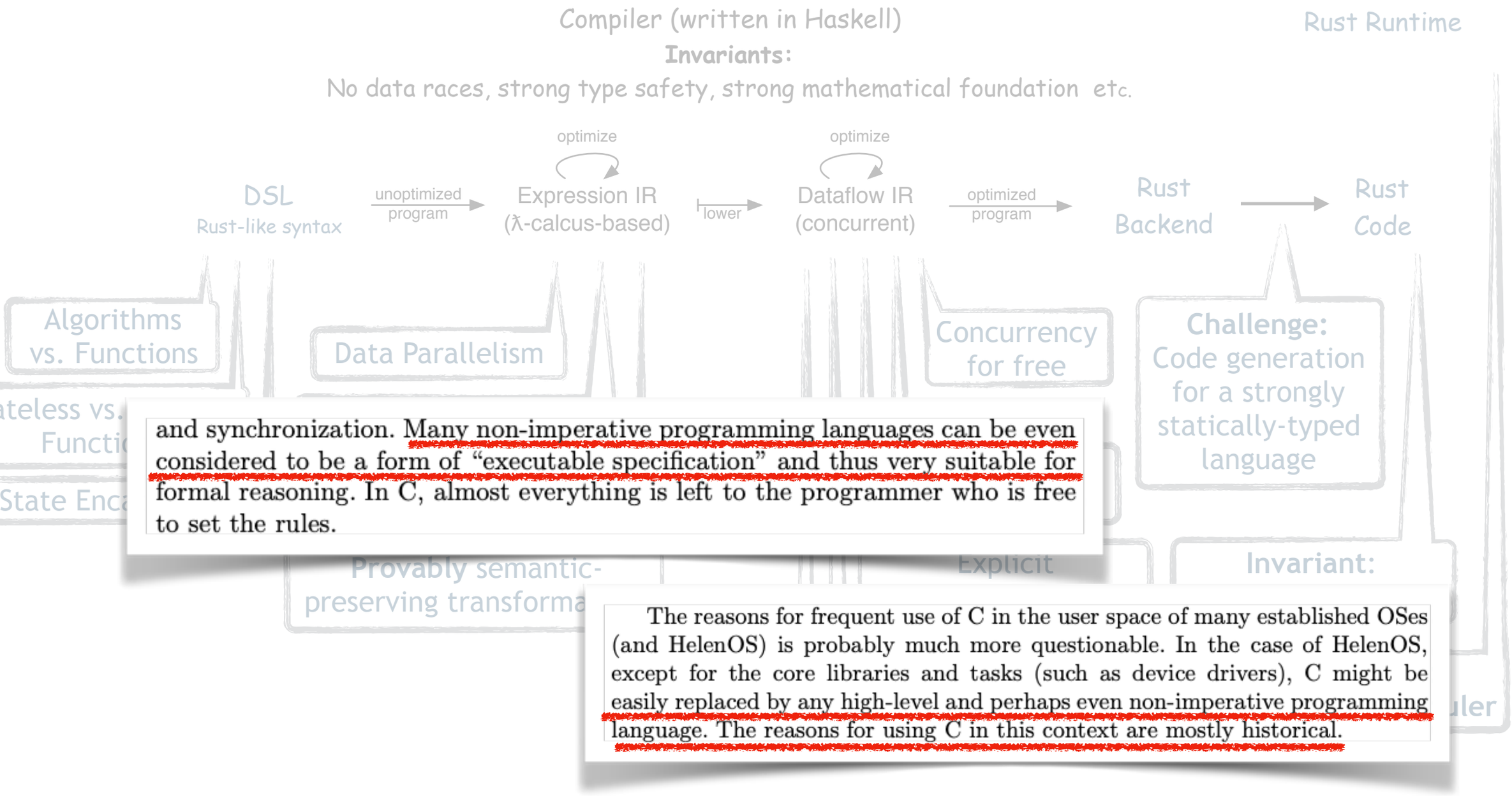


How does that relate to OS design?

How does that relate to OS design?

(I think so but better you tell me!)

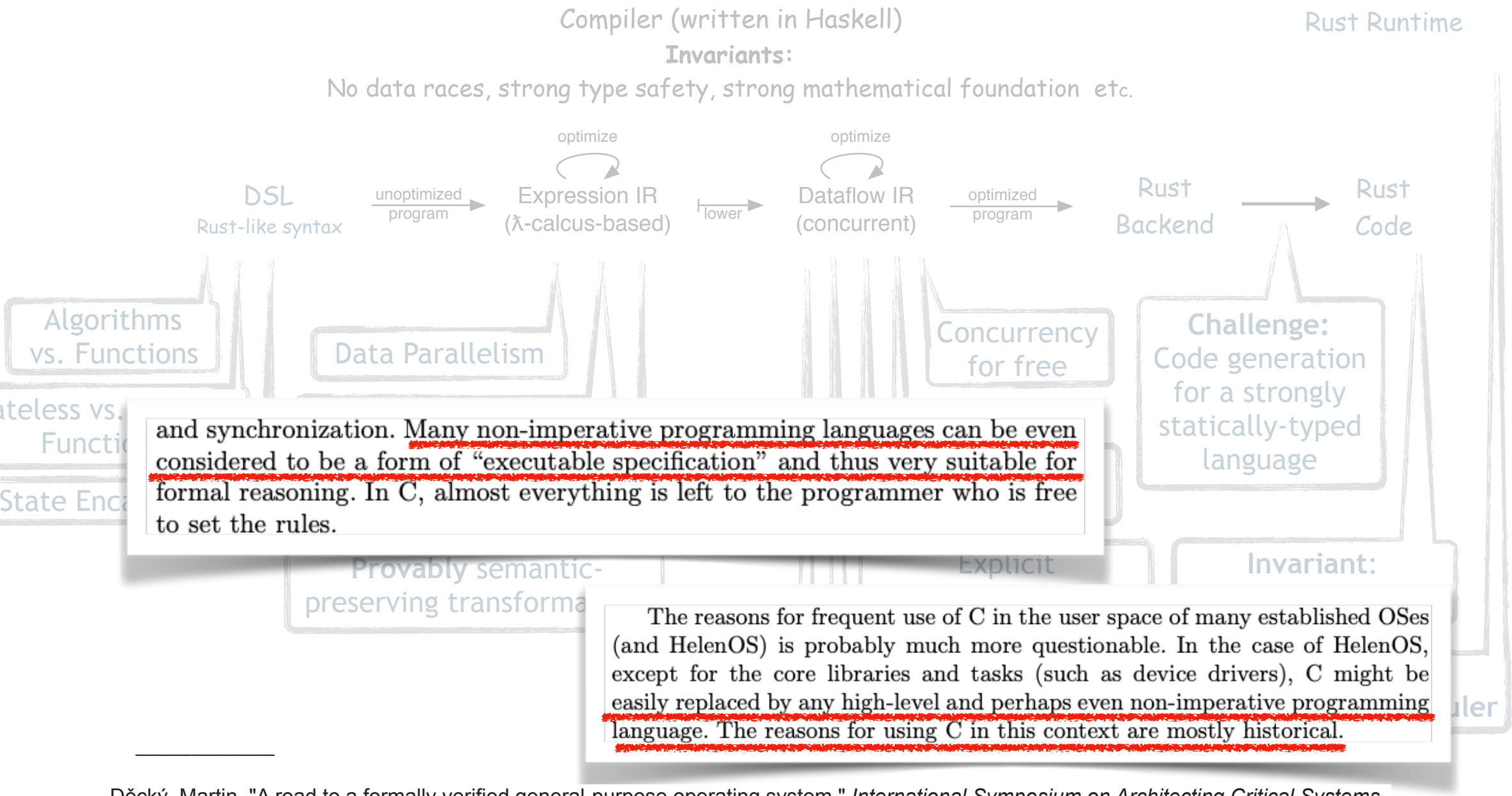
Go Functional!



and synchronization. Many non-imperative programming languages can be even considered to be a form of "executable specification" and thus very suitable for formal reasoning. In C, almost everything is left to the programmer who is free to set the rules.

The reasons for frequent use of C in the user space of many established OSes (and HelenOS) is probably much more questionable. In the case of HelenOS, except for the core libraries and tasks (such as device drivers), C might be easily replaced by any high-level and perhaps even non-imperative programming language. The reasons for using C in this context are mostly historical.

Go Functional!



and synchronization. Many non-imperative programming languages can be even considered to be a form of "executable specification" and thus very suitable for formal reasoning. In C, almost everything is left to the programmer who is free to set the rules.

The reasons for frequent use of C in the user space of many established OSes (and HelenOS) is probably much more questionable. In the case of HelenOS, except for the core libraries and tasks (such as device drivers), C might be easily replaced by any high-level and perhaps even non-imperative programming language. The reasons for using C in this context are mostly historical.

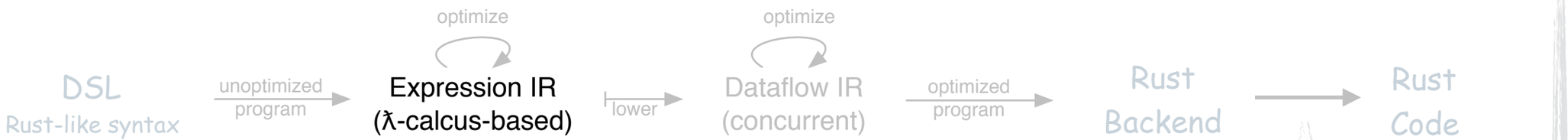
Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

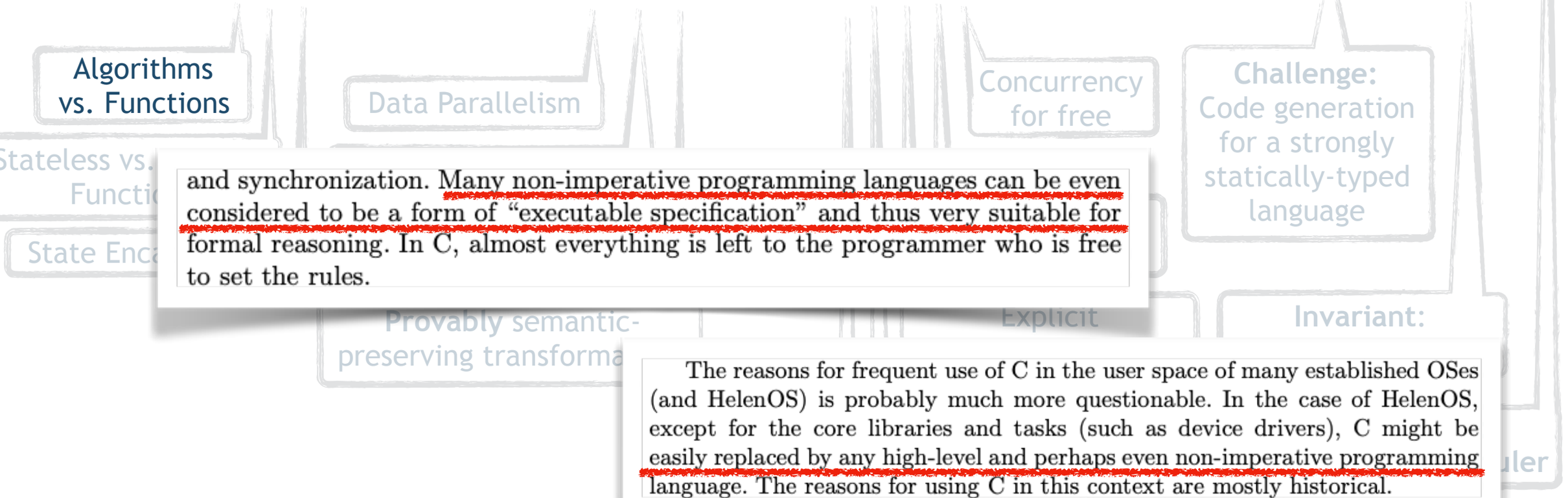
Go Functional!

Compiler (written in Haskell)

Invariants:

No data races, strong type safety, strong mathematical foundation etc.





and synchronization. Many non-imperative programming languages can be even considered to be a form of "executable specification" and thus very suitable for formal reasoning. In C, almost everything is left to the programmer who is free to set the rules.

The reasons for frequent use of C in the user space of many established OSes (and HelenOS) is probably much more questionable. In the case of HelenOS, except for the core libraries and tasks (such as device drivers), C might be easily replaced by any high-level and perhaps even non-imperative programming language. The reasons for using C in this context are mostly historical.

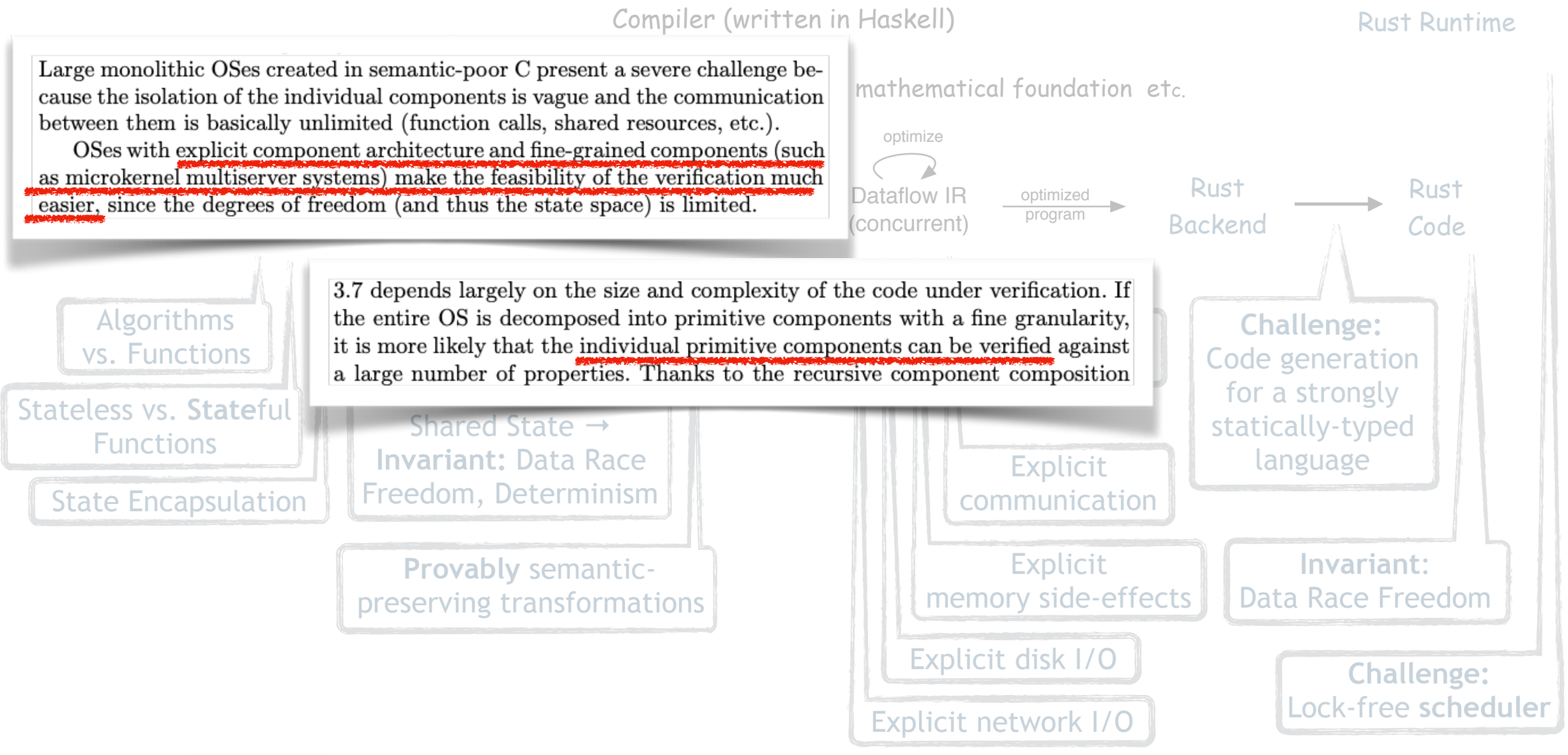
Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

Decomposition for Verification!

Large monolithic OSes created in semantic-poor C present a severe challenge because the isolation of the individual components is vague and the communication between them is basically unlimited (function calls, shared resources, etc.).

OSes with explicit component architecture and fine-grained components (such as microkernel multiserver systems) make the feasibility of the verification much easier, since the degrees of freedom (and thus the state space) is limited.

3.7 depends largely on the size and complexity of the code under verification. If the entire OS is decomposed into primitive components with a fine granularity, it is more likely that the individual primitive components can be verified against a large number of properties. Thanks to the recursive component composition



Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

Decomposition for Verification!

Compiler (written in Haskell)

Rust Runtime

mathematical foundation etc.

> Large monolithic OSes created in semantic-poor C present a severe challenge because the isolation of the individual components is vague and the communication between them is basically unlimited (function calls, shared resources, etc.).
>
> OSes with explicit component architecture and fine-grained components (such as microkernel multiserver systems) make the feasibility of the verification much easier, since the degrees of freedom (and thus the state space) is limited.

> 3.7 depends largely on the size and complexity of the code under verification. If the entire OS is decomposed into primitive components with a fine granularity, it is more likely that the individual primitive components can be verified against a large number of properties. Thanks to the recursive component composition

optimize

Dataflow IR (concurrent)

optimized program

Rust Backend

Rust Code

Challenge: Code generation for a strongly statically-typed language

Algorithms vs. Functions

Stateless vs. Stateful Functions

State Encapsulation

Shared State → **Invariant:** Data Race Freedom, Determinism

Provably semantic-preserving transformations

Explicit communication

Explicit memory side-effects

Explicit disk I/O

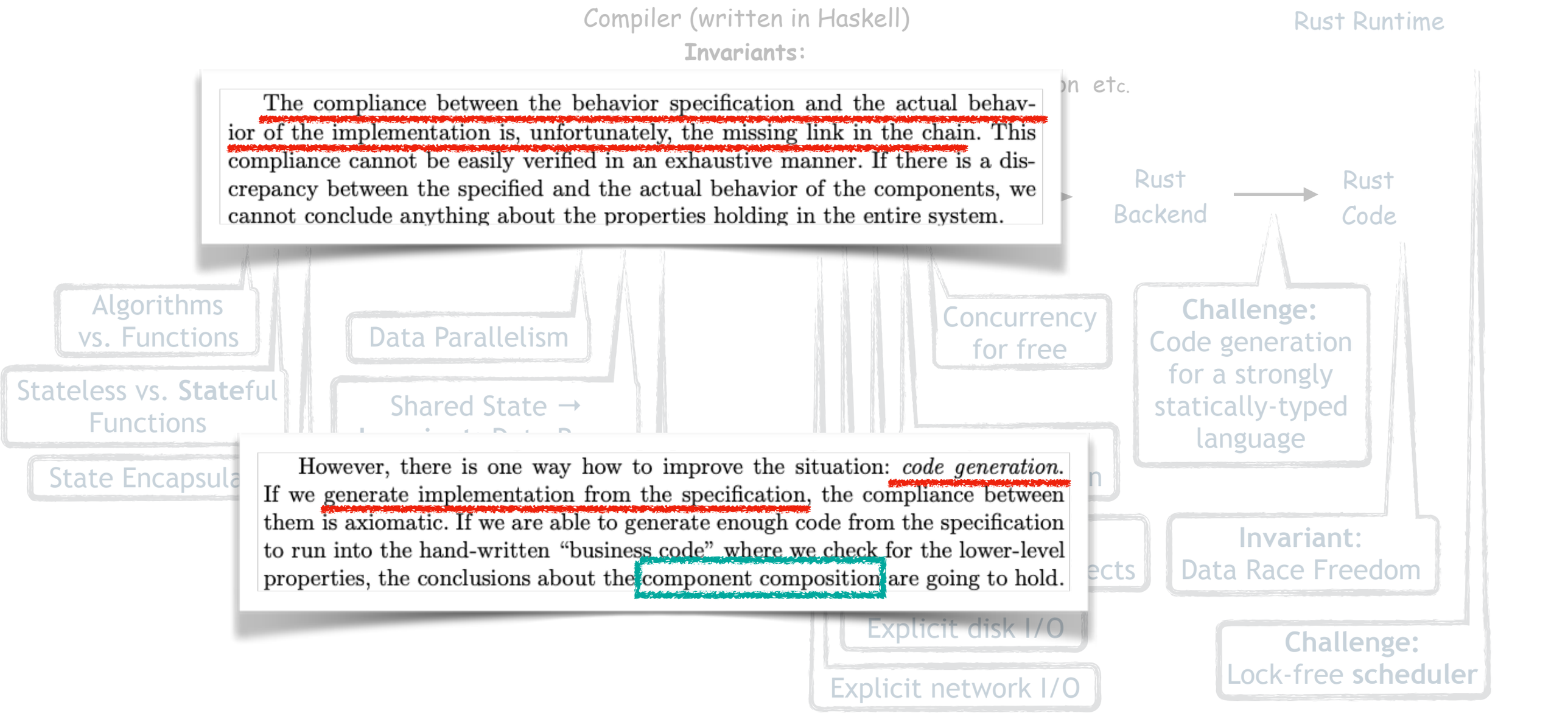
Explicit network I/O

Invariant: Data Race Freedom

Challenge: Lock-free scheduler

Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

Code Generation needed!



> The compliance between the behavior specification and the actual behavior of the implementation is, unfortunately, the missing link in the chain. This compliance cannot be easily verified in an exhaustive manner. If there is a discrepancy between the specified and the actual behavior of the components, we cannot conclude anything about the properties holding in the entire system.

> However, there is one way how to improve the situation: *code generation*. If we generate implementation from the specification, the compliance between them is axiomatic. If we are able to generate enough code from the specification to run into the hand-written "business code", where we check for the lower-level properties, the conclusions about the component composition are going to hold.

Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

Code Generation needed!

Compiler (written in Haskell)

Invariants:

> The compliance between the behavior specification and the actual behavior of the implementation is, unfortunately, the missing link in the chain. This compliance cannot be easily verified in an exhaustive manner. If there is a discrepancy between the specified and the actual behavior of the components, we cannot conclude anything about the properties holding in the entire system.

Rust Runtime

etc.

Rust Backend → Rust Code

Algorithms vs. Functions

Stateless vs. Stateful Functions

State Encapsula

Data Parallelism

Shared State →

Concurrency for free

Challenge: Code generation for a strongly statically-typed language

Invariant: Data Race Freedom

Challenge: Lock-free scheduler

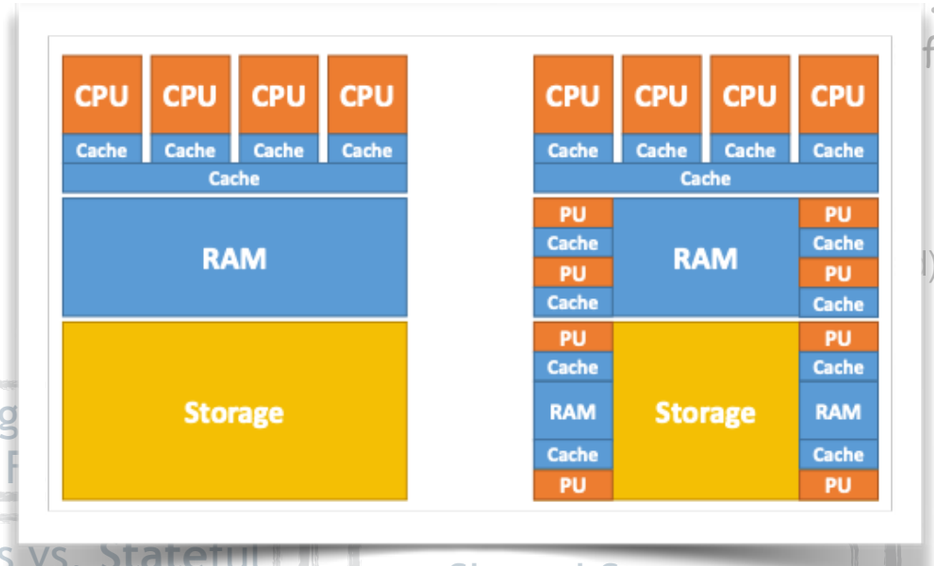
Explicit disk I/O

Explicit network I/O

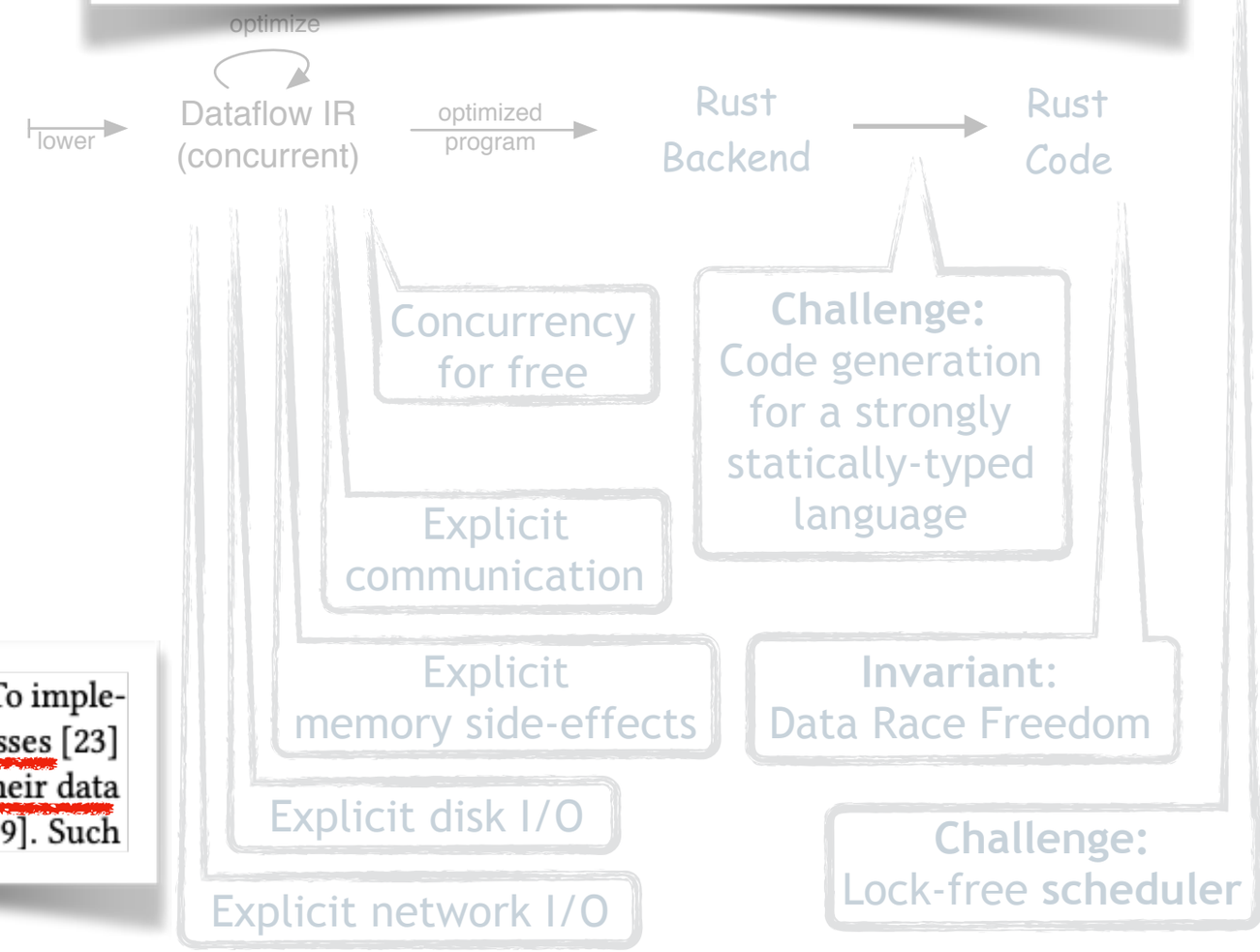
> However, there is one way how to improve the situation: *code generation*. If we generate implementation from the specification, the compliance between them is axiomatic. If we are able to generate enough code from the specification to run into the hand-written "business code" where we check for the lower-level properties, the conclusions about the component composition are going to hold.

Děcký, Martin. "A road to a formally verified general-purpose operating system." *International Symposium on Architecting Critical Systems*. Springer, Berlin, Heidelberg, 2010.

Knowledge is power!



balance or minimize energy consumption. At the same time, a key objective of scheduling is to minimize data movements, favoring code migration instead. This goes behind the NUMA functionalities

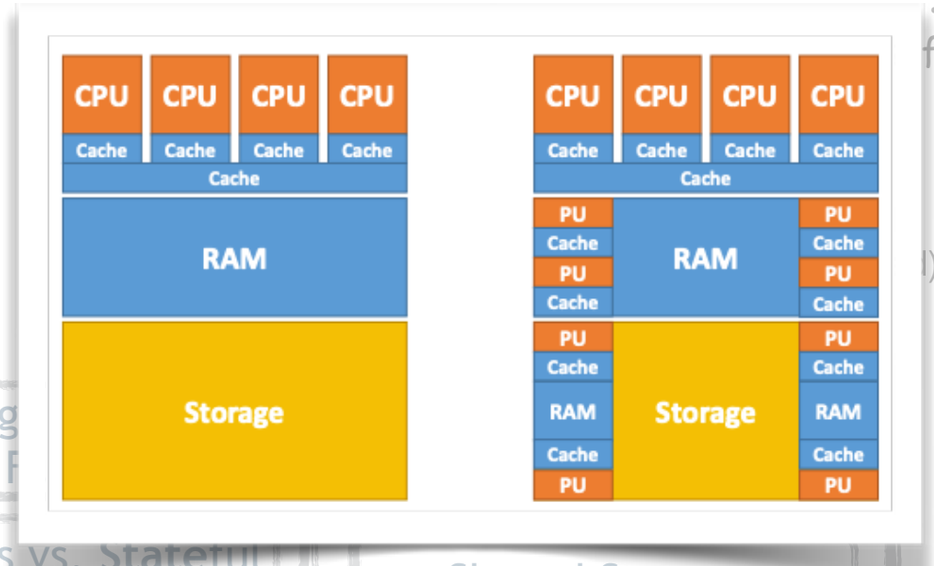


NUMA zone to which their data have been allocated [21]. To implement data-driven scheduling we propose to track data accesses [23] as well as augment applications with information about their data access pattern, for example in the binary executable itself [9]. Such

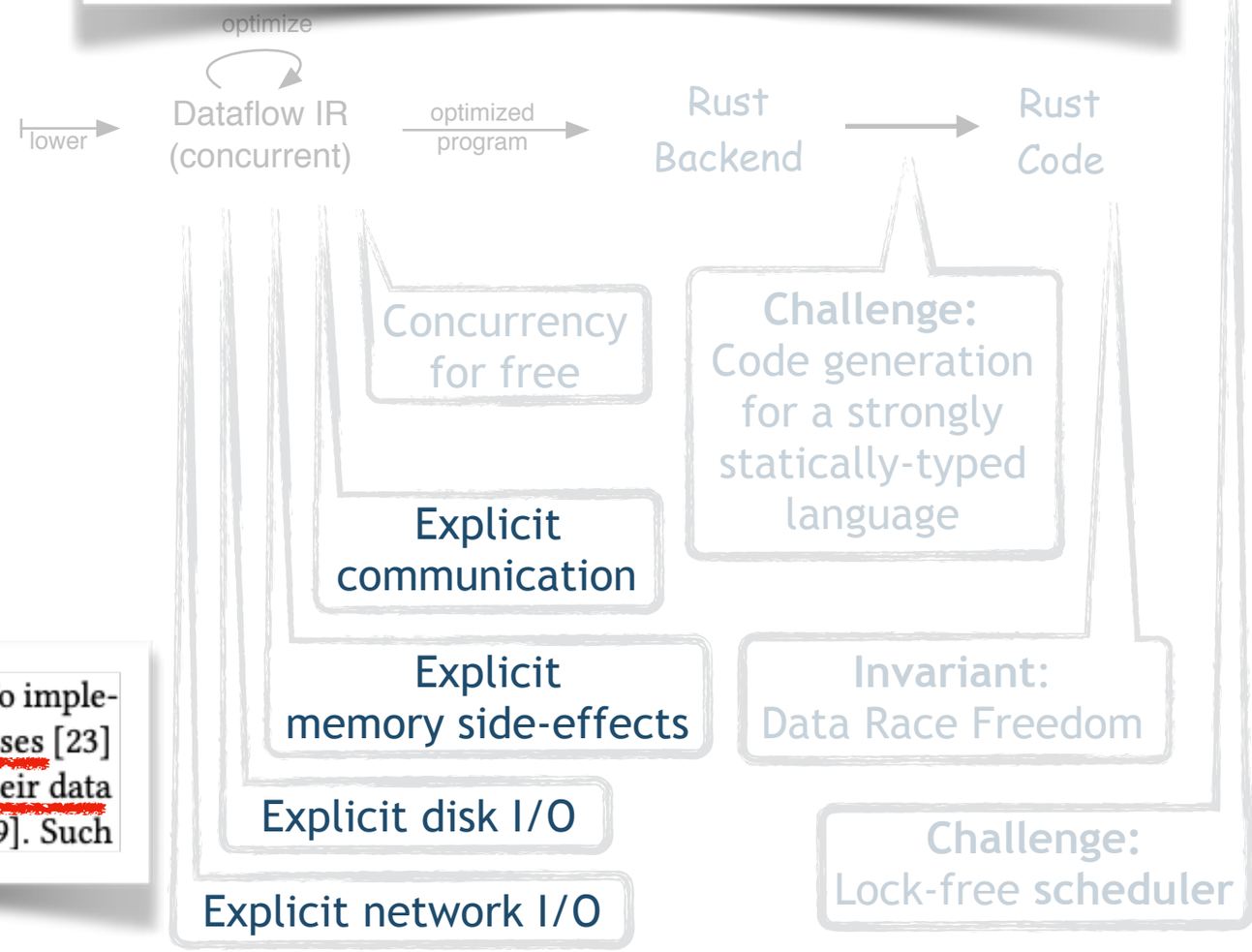
---

Barbalace, A., Iliopoulos, A., Rauchfuss, H., & Brasche, G. (2017, May). It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 56-61). ACM.

Knowledge is power!



balance or minimize energy consumption. At the same time, a key objective of scheduling is to minimize data movements, favoring code migration instead. This goes behind the NUMA functionalities

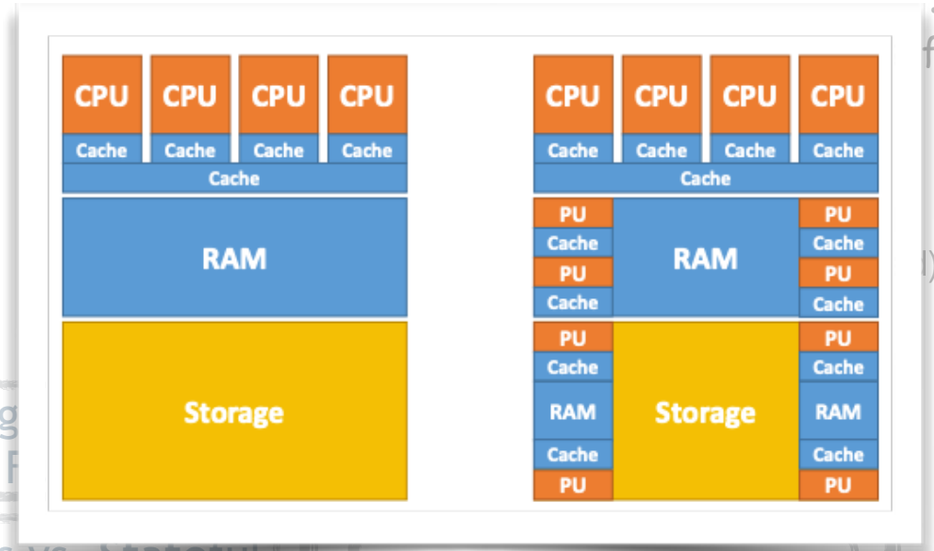


NUMA zone to which their data have been allocated [21]. To implement data-driven scheduling we propose to track data accesses [23] as well as augment applications with information about their data access pattern, for example in the binary executable itself [9]. Such

---

Barbalace, A., Iliopoulos, A., Rauchfuss, H., & Brasche, G. (2017, May). It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 56-61). ACM.

Knowledge is power!



balance or minimize energy consumption. At the same time, a key objective of scheduling is to minimize data movements, favoring code migration instead. This goes behind the NUMA functionalities

optimize

lower

Dataflow IR (concurrent)

optimized program

Rust Backend

Rust Code

similarly to [38, 57, 59]. Finally, to support low-latency IO the OS itself must be "fast" and deterministic, including the communication between OS services spanning multiple processing units.

Explicit communication

Explicit memory side-effects

Explicit disk I/O

Explicit network I/O

language

Invariant: Data Race Freedom

Challenge: Lock-free scheduler

Shared State →
Invariant: Data Race Freedom, Determinism

Stateless vs. Stateful Functions

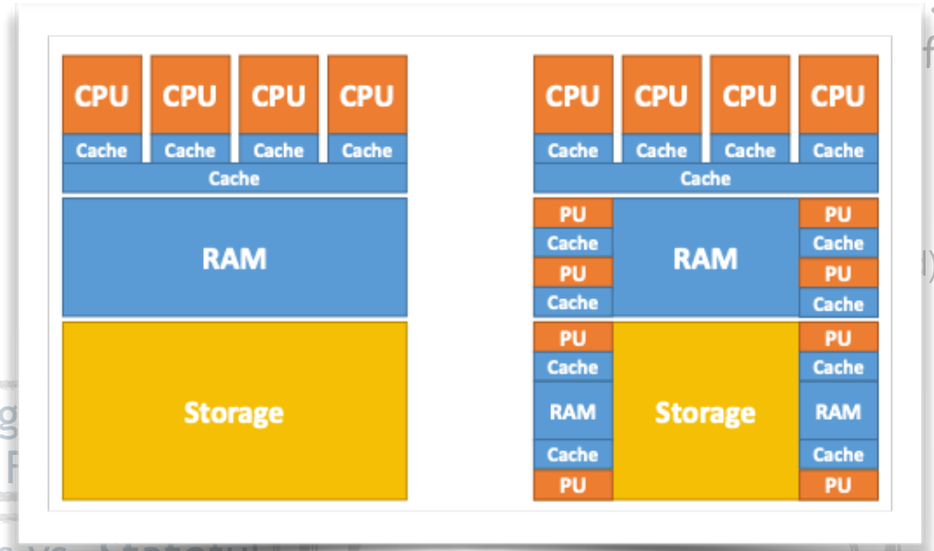
State Encapsulation

NUMA zone to which their data have been allocated [21]. To implement data-driven scheduling we propose to track data accesses [23] as well as augment applications with information about their data access pattern, for example in the binary executable itself [9]. Such

---

Barbalace, A., Iliopoulos, A., Rauchfuss, H., & Brasche, G. (2017, May). It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 56-61). ACM.

Knowledge is power!



Shared State →
Invariant: Data Race Freedom, Determinism

balance or minimize energy consumption. At the same time, a key objective of scheduling is to minimize data movements, favoring code migration instead. This goes behind the NUMA functionalities

similarly to [38, 57, 59]. Finally, to support low-latency IO the OS itself must be "fast" and deterministic, including the communication between OS services spanning multiple processing units.

Explicit communication

Explicit memory side-effects

Explicit disk I/O

Explicit network I/O

Invariant:
Data Race Freedom

Challenge:
Lock-free scheduler

NUMA zone to which their data have been allocated [21]. To implement data-driven scheduling we propose to track data accesses [23] as well as augment applications with information about their data access pattern, for example in the binary executable itself [9]. Such

---

Barbalace, A., Iliopoulos, A., Rauchfuss, H., & Brasche, G. (2017, May). It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 56-61). ACM.

Final Take-away Points

Final Take-away Points

We all are actually trying to solve the very same problem and have similar ideas.
Let’s join forces!

Final Take-away Points

We all are actually trying to solve the very same problem and have similar ideas.
Let’s join forces!

Hands down:
I don’t have the traditional kernel construction knowledge.

Final Take-away Points

We all are actually trying to solve the very same problem and have similar ideas.
Let’s join forces!

Hands down:
I don’t have the traditional kernel construction knowledge.

But:
Are you really looking for that in order to rethink kernel design?!