Towards Implicit Parallel Programming for Systems

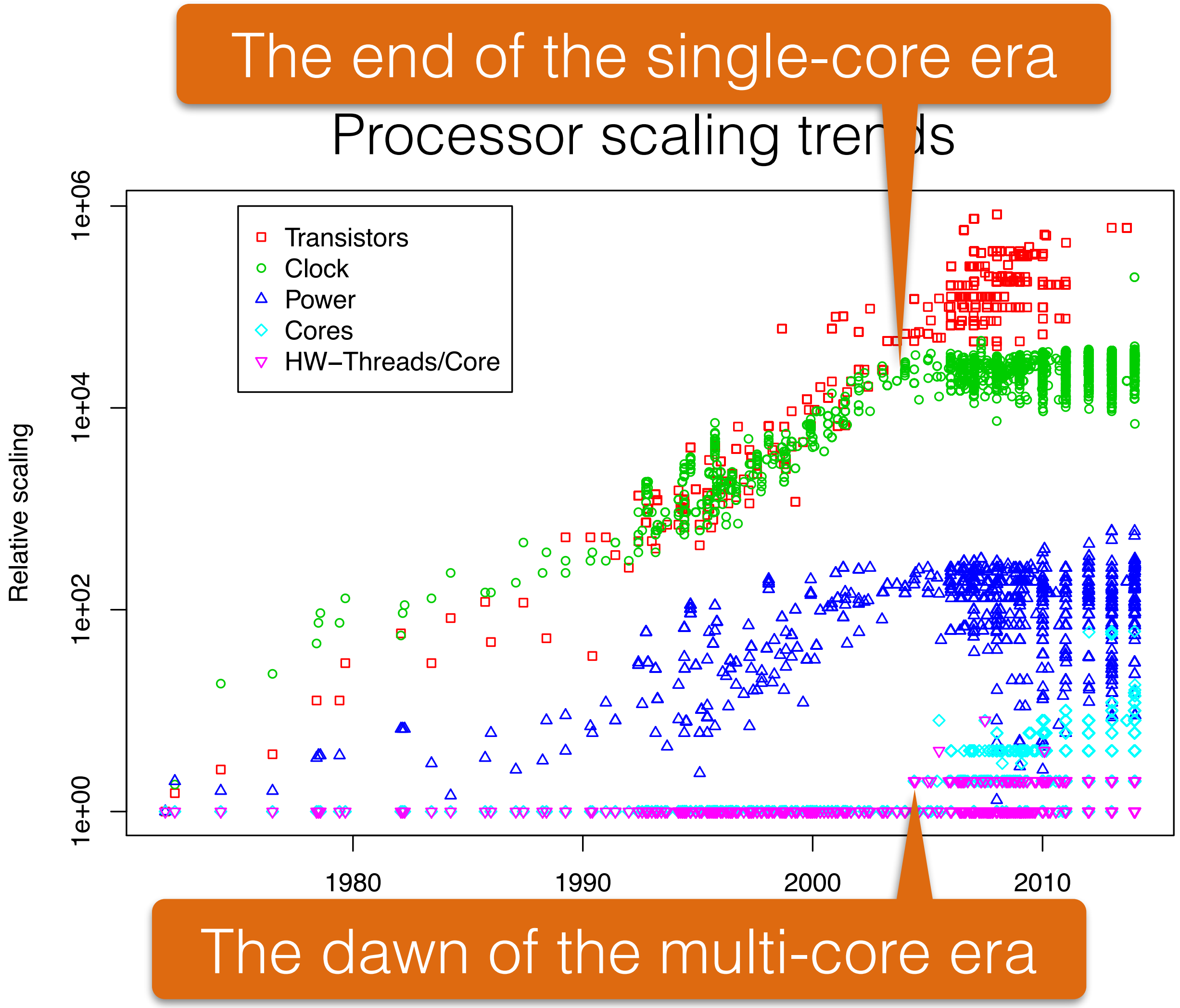
Sebastian Ertel

Dr.-Ing. Defense
Chair for Compiler Construction
TU Dresden, Germany

Supervisor: Prof. Dr.-Ing. Jeronimo Castrillon
Fachreferent : Prof. Dr.-Ing. Hermann Härtig

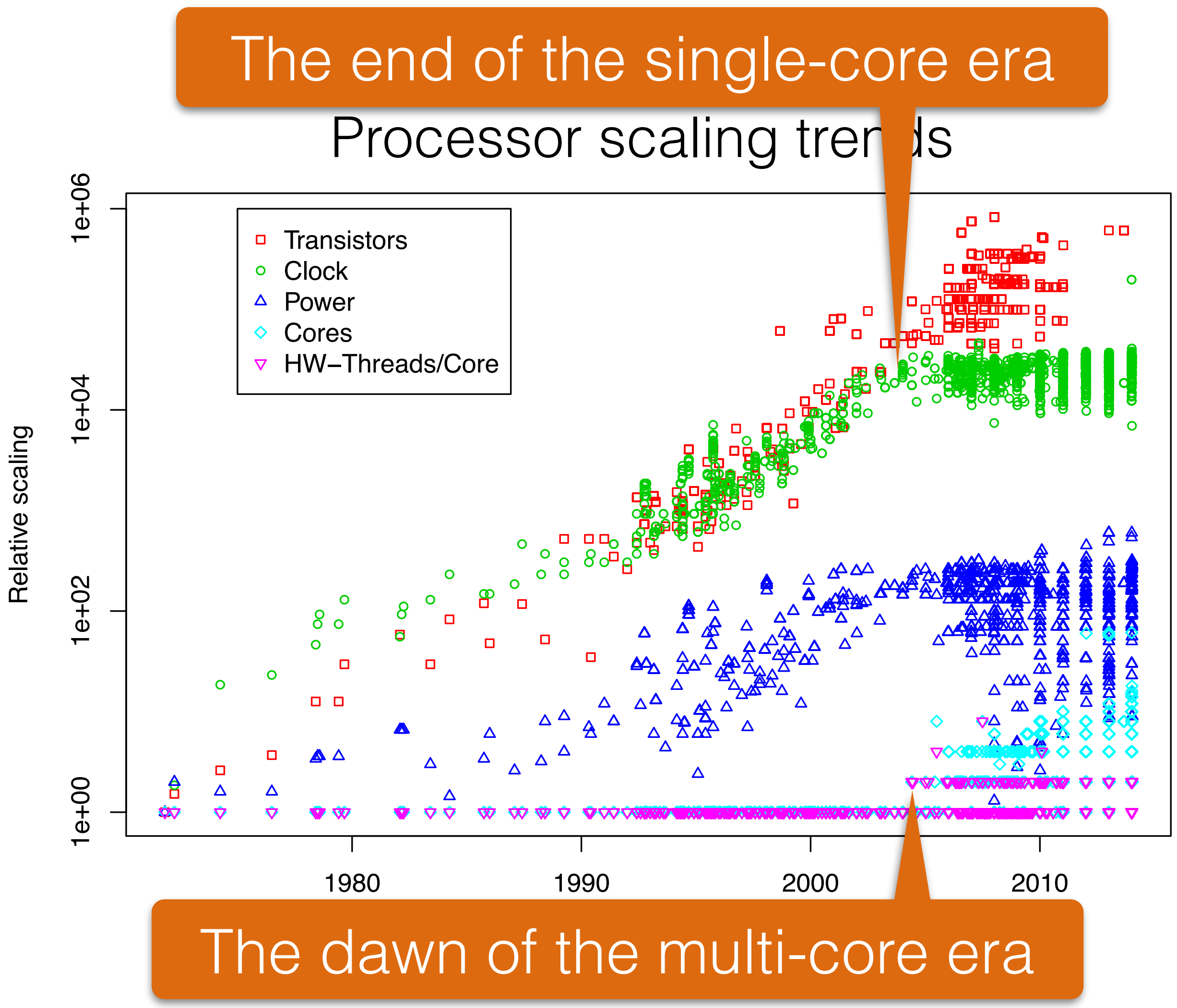
4.9.2019

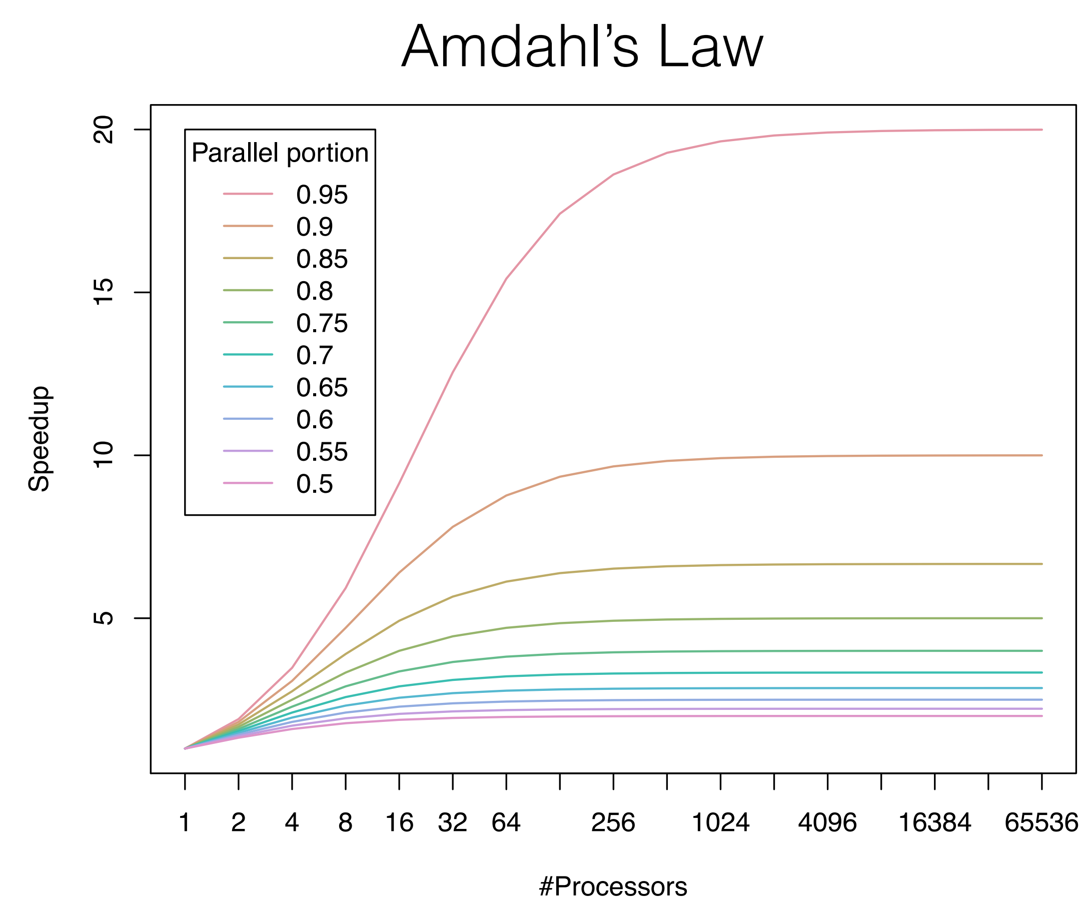
The Shape Of Things To Run



Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

The Shape Of Things To Run





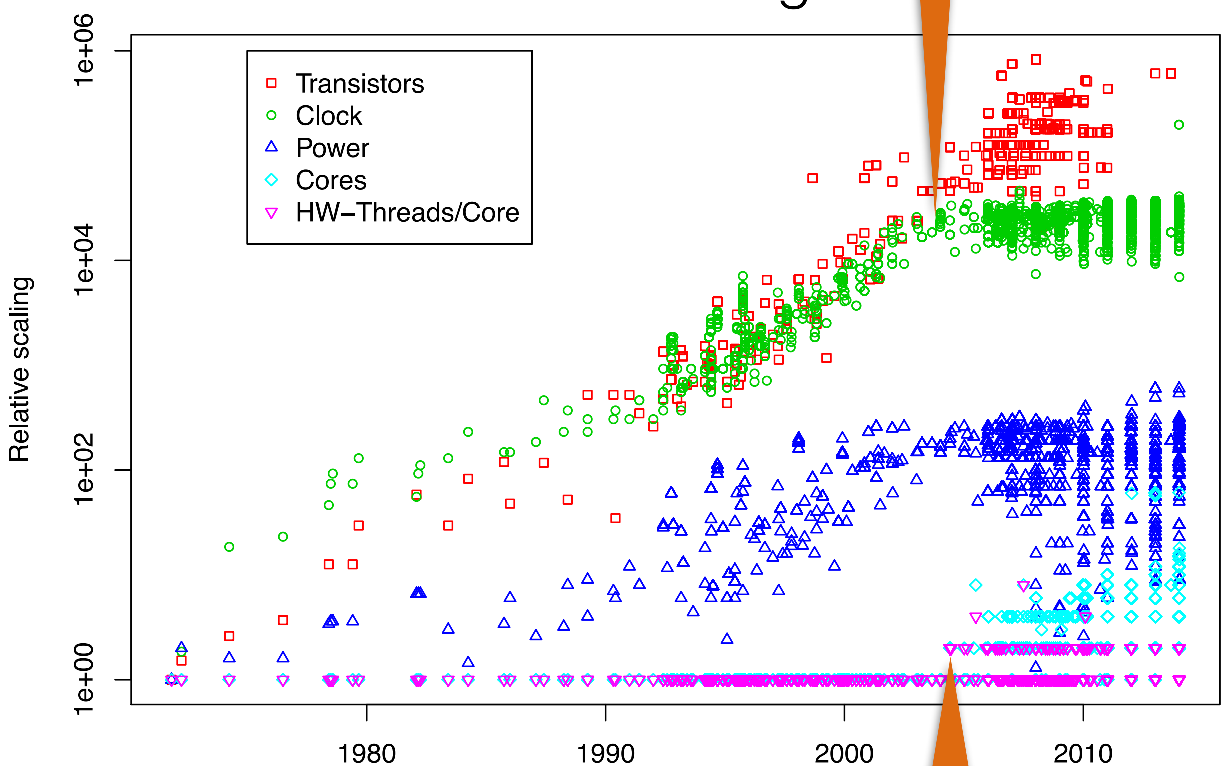
Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.

G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.

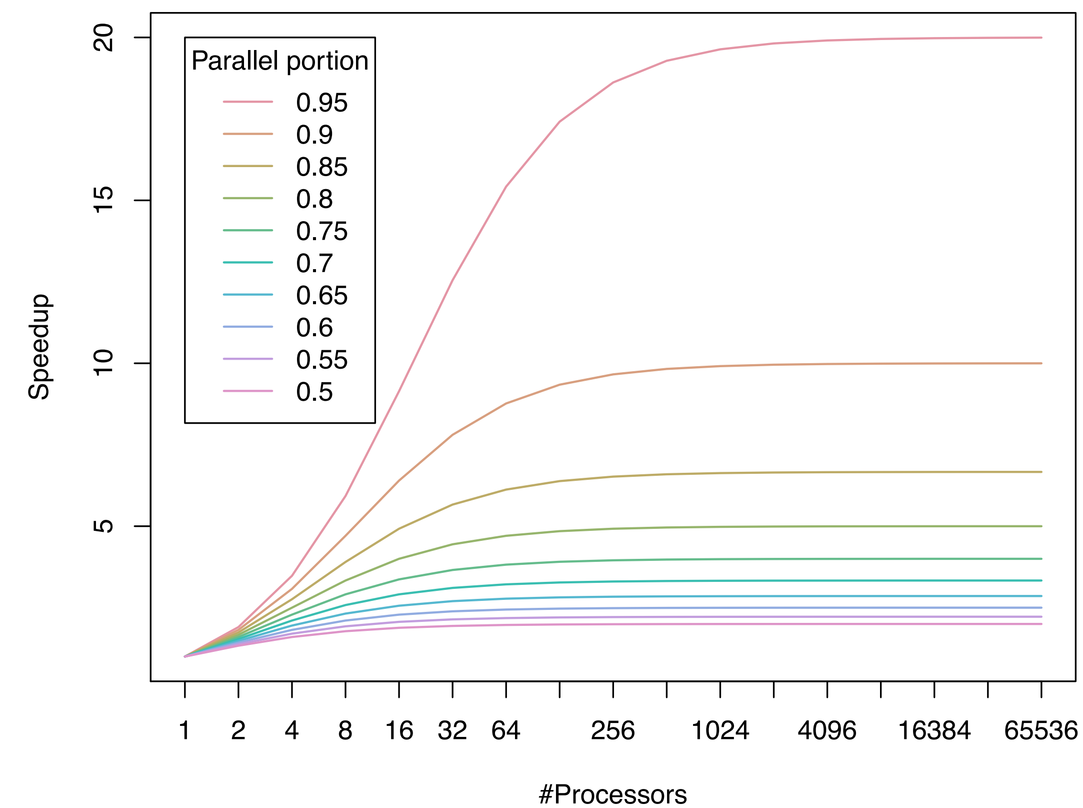
The Shape Of Things To Run

The end of the single-core era

Processor scaling trends



Amdahl's Law



The dawn of the multi-core era

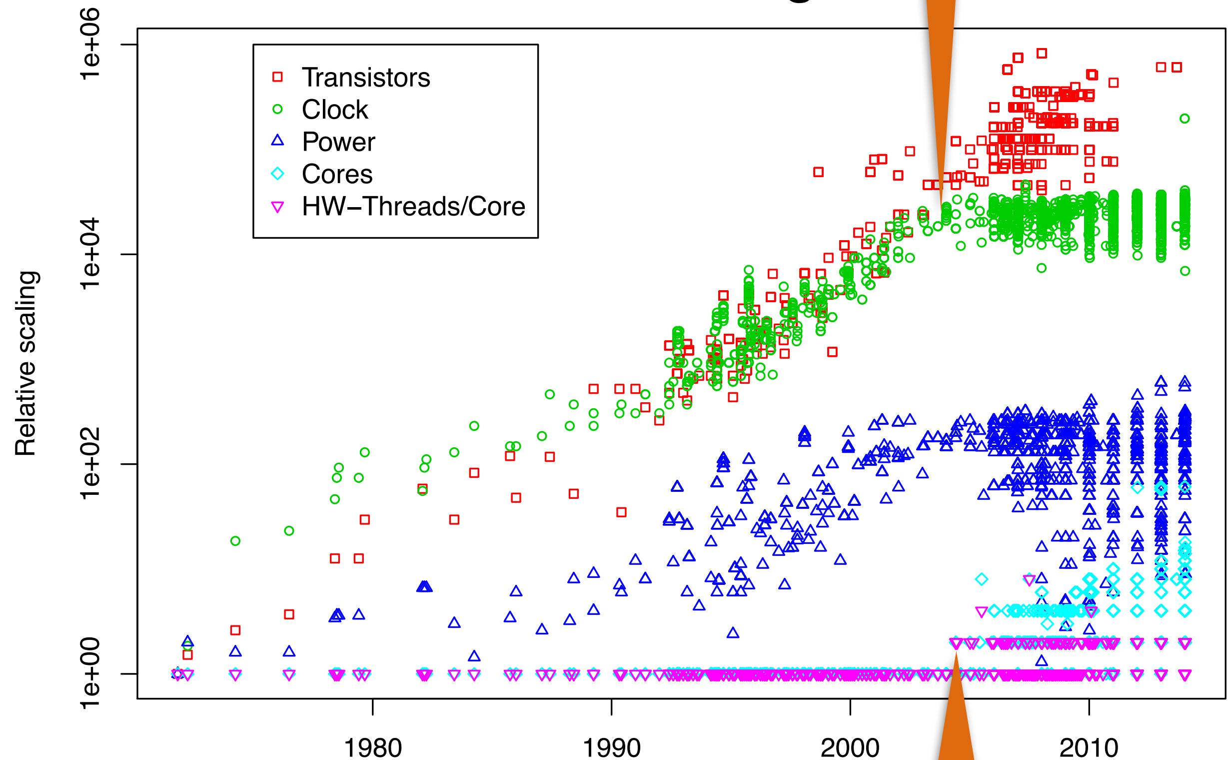
Challenge	Near-Term	Long-Term
1,000-fold software parallelism	Data parallel languages and "mapping" of operators, library and tool-based approaches	New high-level languages, compositional and deterministic frameworks

Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.
G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.
Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. Commun. ACM.

The Shape Of Things To Run

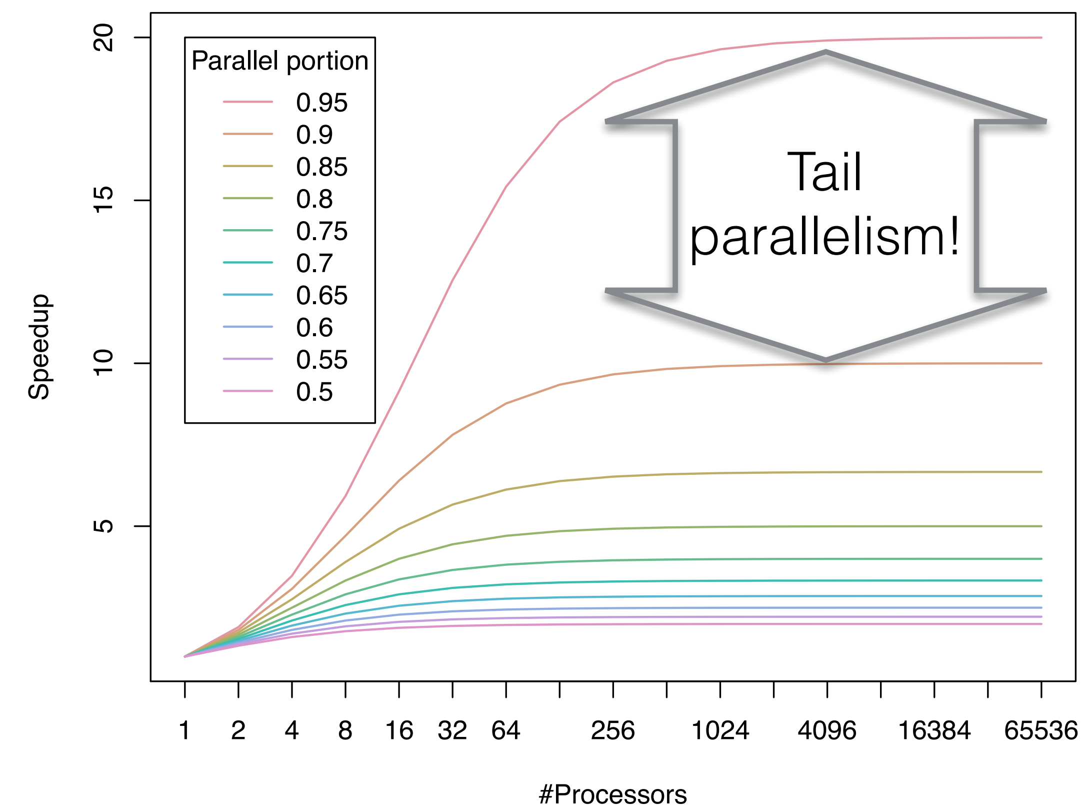
The end of the single-core era

Processor scaling trends



The dawn of the multi-core era

Amdahl's Law



Challenge	Near-Term	Long-Term
1,000-fold software parallelism	Data parallel languages and "mapping" of operators, library and tool-based approaches	New high-level languages, compositional and deterministic frameworks

Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs Journal.
G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67. ACM.
Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. Commun. ACM.

Programming Is Simple!

Programming

Variables	Conditionals	Classes
Functions	Loops	Objects
Function Calls		Methods

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

Concurrent/Parallel Programming

Message-Passing
Processes

Transactional Memory
Locks
compare-and-swap
Threads

Operating System

Fork/Join

LVars
MVars
IVars
Spawn

Tasks

Work-Stealing

Promises
Futures

Actors
Events

Coroutines
Fibers

Algorithmic Skeletons
OpenMP (loops)
Map/Reduce

Custom/optimized

SQL
Parallel Haskell (pH)
Reactive Programming
Streams
Dataflow

Low level → High level

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

Concurrent/Parallel Programming

Transactional Memory
Locks
Message-Passing
Processes
Threads
Operating System

LVars
MVars
Tasks
Futures
Work-Stealing

Events

Algorithmic
Map/Reduce
Custom/optimized

SQL
Parallel Haskell (pH)
Reactive Programming
Streams
Dataflow

Composition and state are key for systems!

Low level → High level

Parallel Programming Is Hard!

Programming

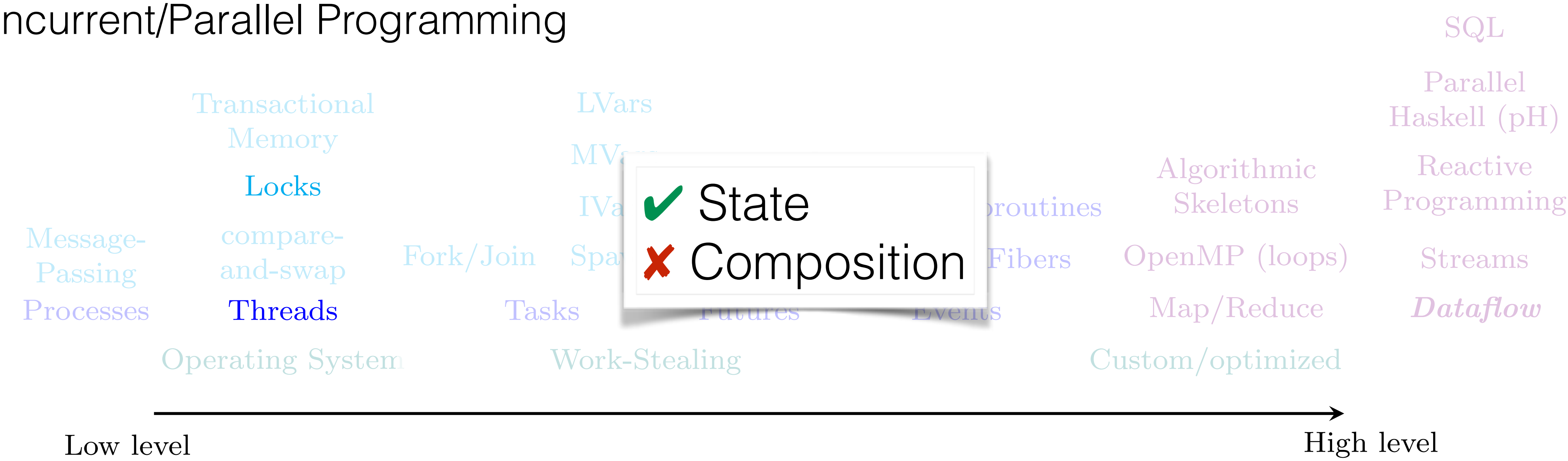
Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

⇩

Concurrent/Parallel Programming



Lee EA. The problem with threads. Computer. 2006.

Van Renesse R. Goal-oriented programming, or composition using events, or threads considered harmful. In ACM SIGOPS European Workshop 1998.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

⇩

Concurrent/Parallel Programming

Message-Passing Processes

Transactional Memory

Locks

compare-and-swap

Threads

Operating System

Fork/Join

LVars

MV

IVa

Spa

Tasks

Work

✔ State
✔ Composition

outines

ibers

Algorithmic Skeletons

OpenMP (loops)

Map/Reduce

Custom/optimized

SQL

Parallel Haskell (pH)

Reactive Programming

Streams

Dataflow

Low level → High level

Herlihy M, Moss JE. Transactional memory: Architectural support for lock-free data structures. ACM; 1993 Jun 1.

Shavit N, Touitou D. Software transactional memory. Distributed Computing. 1997 Feb 1;10(2):99-116.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

⇩

Concurrent/Parallel Programming

SQL

Transactional Memory

LVars

Parallel Haskell (pH)

MV

Locks

IVa

outines

Algorithmic Skeletons

Reactive Programming

Message-Passing Processes

compare-and-swap

Fork/Join

Spa

ibers

OpenMP (loops)

Streams

Threads

Tasks

Map/Reduce

Dataflow

Operating System

Work

Custom/optimized

✔ State
✔ Composition
✘ Composition

Low level → High level

J. Swalens, S. Marr, J. De Koster, and T. Van Cutsem. 2014. Towards Composable Concurrency Abstractions. In Proceedings of PLACES'14.

J. Swalens, J. De Koster, and W. De Meuter. 2016. Transactional Tasks: Parallelism in Software Transactions. In Proceedings of ECOOP'16.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

Concurrent/Parallel Programming

SQL

Parallel Haskell (pH)

Transactional Memory

LVars

MVars

Locks

IVa

Algorithmic Skeletons

Reactive Programming

oroutines

Message-Passing Processes

compare-and-swap

Fork/Join

Spa

Fibers

OpenMP (loops)

Streams

Threads

Tasks

ts

Map/Reduce

Dataflow

Operating System

Work-Stealing

Custom/optimized

? State
? Composition

Low level → High level

Mou ZG, Hudak P. An algebraic model for divide-and-conquer and its parallelism. The Journal of Supercomputing. 1988.

Lämmel R. Google's MapReduce programming model—Revisited. Science of computer programming. 2008.

Cole MI. Algorithmic skeletons: structured management of parallel computation. London: Pitman; 1989.

Buono D, Danelutto M, Lametti S. Map, reduce and mapreduce, the skeleton way. Procedia computer science. 2010

Dagum L, Menon R. OpenMP: An industry-standard API for shared-memory programming. Computing in Science & Engineering. 1998.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

⇩

Concurrent/Parallel Programming

Message-Passing Processes

Transactional Memory
Locks
compare-and-swap
Threads
Operating System

Fork/Join

LVars
MV
IVa
Spa
Tasks
Work

Algorithmic Skeletons
OpenMP (loops)
Map/Reduce
Custom/optimized

SQL
Parallel Haskell (pH)
Reactive Programming
Streams
Dataflow

✔ State
✔/✘ Composition
✔/✘ Systems

Low level → High level

Chamberlin DD, Boyce RF. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. ACM.

Nikhil RS. Implicit parallel programming in pH. Morgan Kaufmann; 2001.

Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N. FlumeJava: easy, efficient data-parallel pipelines. In ACM Sigplan Notices 2010. ACM.

Coutts D, Leshchinskiy R, Stewart D. Stream fusion: From lists to streams to nothing at all. In ACM SIGPLAN Notices 2007. ACM.

Thies W, Karczmarek M, Amarasinghe S. StreamIt: A language for streaming applications. In International Conference on Compiler Construction 2002. Springer.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

⇩

Concurrent/Parallel Programming

SQL
Parallel Haskell (pH)
Reactive Programming
Streams
Dataflow

Transactional Memory
Locks
LVars
MVars
Algorithmic

Message-Passing
Processes
Threads
Tasks
Futures
Events
Map/Reduce

Operating System
Work-Stealing
Custom/optimized

Composition and state are key for systems!

Low level → High level

Explicit

Implicit

Concurrency Bugs

Optimizations
(Sequential ➪ Parallel)

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. CGO '17.

Parallel Programming Is Hard!

Programming

Variables
Functions
Function Calls

Conditionals
Loops

Classes
Objects
Methods

Concurrent/Parallel Programming

Transactional Memory
Locks
compa
and-sw
Message-Passing Processes
Threads
Operating System

LVars
MVars
Tasks
Futures
Work-Stealing

Events

Algorithmic
ps)
Map/Reduce
Custom/optimized

SQL
Parallel Haskell (pH)
Reactive Programming
Streams
Dataflow

Composition and state are key for systems!

Low level — High level

Explicit

Implicit

Concurrency Bugs

PL/compiler co-design!

Optimizations
(Sequential ➪ Parallel)

Mary Hall, David Padua, and Keshav Pingali. 2009. Compiler research: the next 50 years. Commun. ACM.

Ingredients for Implicit Parallel Programming

Programming Model/ Language

- No concurrency abstractions.
- Practicality:
    - Integration into existing programming models (e.g., OOP)/ languages.
    - Gradual switch/ reuse of existing code base.

Ingredients for Implicit Parallel Programming

Programming Model/ Language → Compiler

Programming Model/ Language

- No concurrency abstractions.
- Practicality:
  - Integration into existing programming models (e.g., OOP)/ languages.
  - Gradual switch/ reuse of existing code base.

Compiler

- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Ingredients for Implicit Parallel Programming

Programming Model/ Language → Compiler → Runtime System

Programming Model/Language

- No concurrency abstractions.
- Practicality:
    - Integration into existing programming models (e.g., OOP)/ languages.
    - Gradual switch/ reuse of existing code base.

Compiler

- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Runtime System

- Flexible/dynamic runtime representation.
- Efficient parallel execution of independent computations.
- Deterministic execution.
- Concurrent I/O.

Programming Model/ Language → Compiler → Runtime System

Programming Model/ Language → Compiler → Runtime System

Programming Model/
Language

Compiler

Runtime System

- Parallelism
- Mapping/Scheduling
-

Database Systems

Data Management Systems

Data Streaming Systems

Server Systems

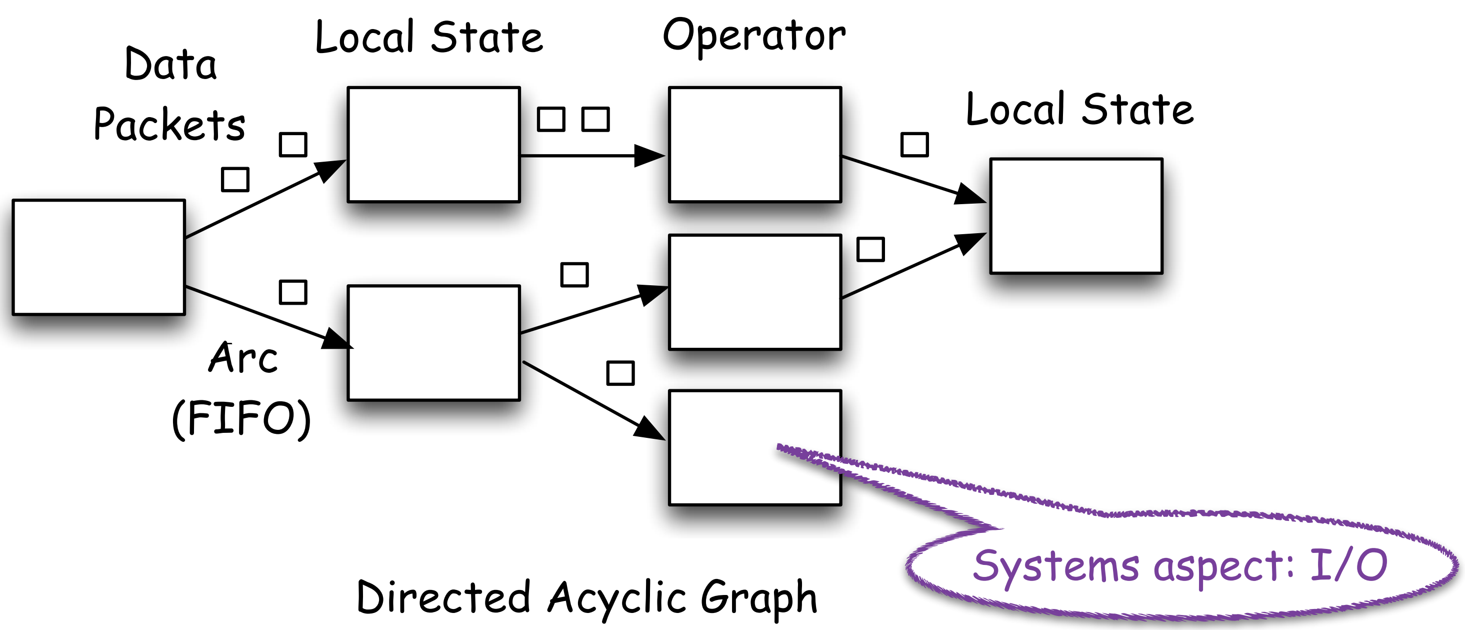
Embedded Systems

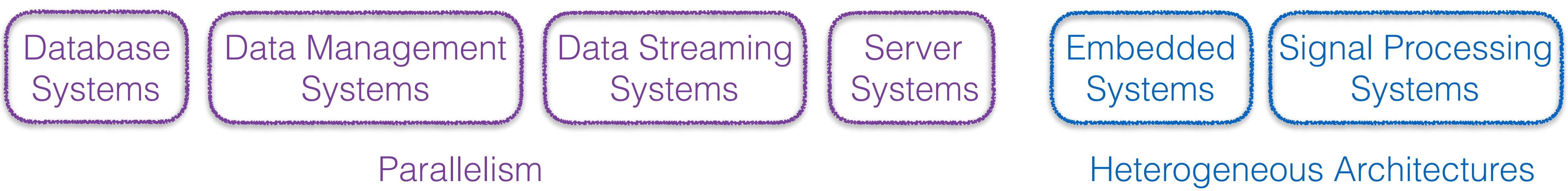
Signal Processing Systems

Parallelism

Heterogeneous Architectures

Dataflow





J. B. Dennis. Data flow supercomputers. Computer, 13(11):48–56, Nov. 1980.

J. P. Morrison. Flow-Based Programming. Nostrand Reinhold, 1994

Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures. Annual Reviews Inc.

Programming Model/
Language

Compiler

Runtime System

- Parallelism
- Mapping/Scheduling
-

Programming Model/ Language → Compiler → Runtime System

Dataflow

- Parallelism
- Mapping/Scheduling
- Dynamic Software Evolution

Sebastian Ertel and Pascal Felber. 2014. A framework for the dynamic evolution of highly-available dataflow programs. In Proceedings of the 15th International Middleware Conference (Middleware '14). ACM.

Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM.

Programming Model/
Language → Compiler → Runtime System

Dataflow
- Parallelism
- Mapping/Scheduling
- Dynamic Software Evolution

Programming Model/ Language → Compiler → Runtime System

Dataflow

- Flexible/dynamic runtime representation.
- Efficient parallel execution of independent computations.
- Deterministic execution.
- Concurrent I/O.

Programming Model/ Language → Compiler → Runtime System

Dataflow

Dataflow:

- Not concise
- Limited scalability
- Concurrency abstraction

Programming Model/
Language

Compiler

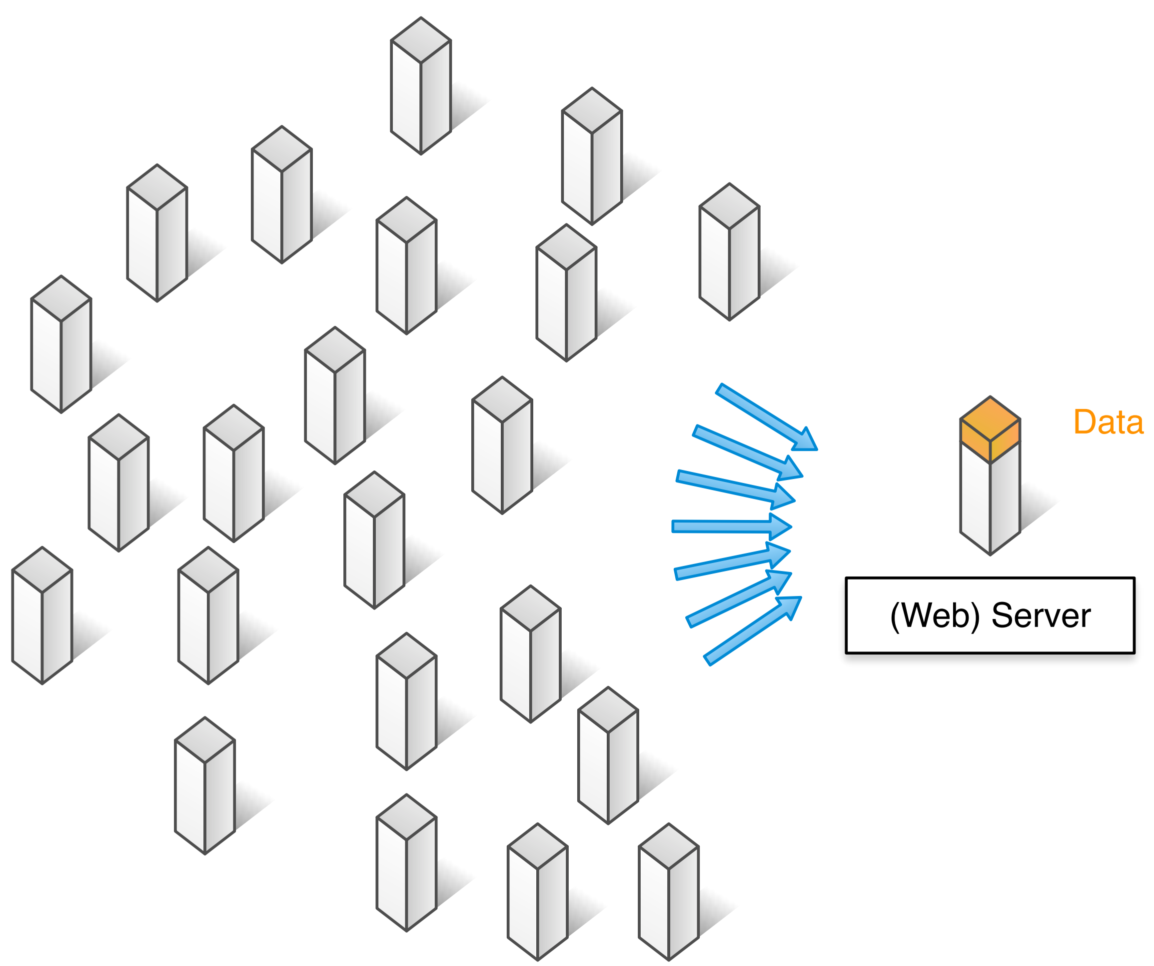
Runtime System

Dataflow

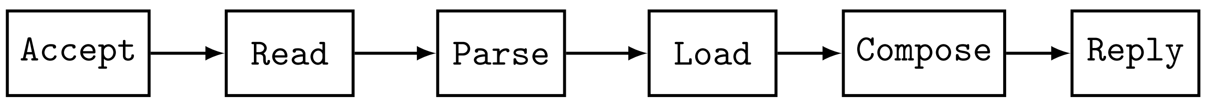
SFP:

- Variables
- Functions
- State

Server Architecture



Explicit Dataflow Construction



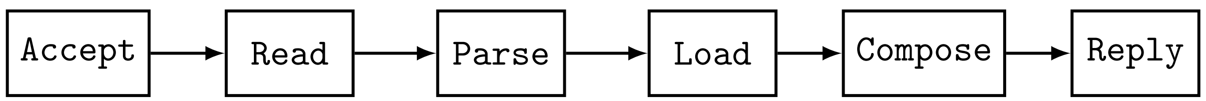
Explicit Dataflow Construction

Accept → Read → Parse → Load → Compose → Reply

```
public class WebServer extends StreamFlexGraph {
  private Filter a, r, p, l, c, rep;

  public WebServer() {
    a   = makeFilter(Accept.class);
    r   = makeFilter(Read.class);
    p   = makeFilter(Parse.class);
    l   = makeFilter(Compose.class);
    rep = makeFilter(Reply.class);
    connect(a, r);
    connect(r, p);
    connect(p, l);
    connect(l, c);
    connect(c, rep);
    validate();
  }

  public void start() {
    new Synthesizer(a).start();
    super.start();
  }
}
```

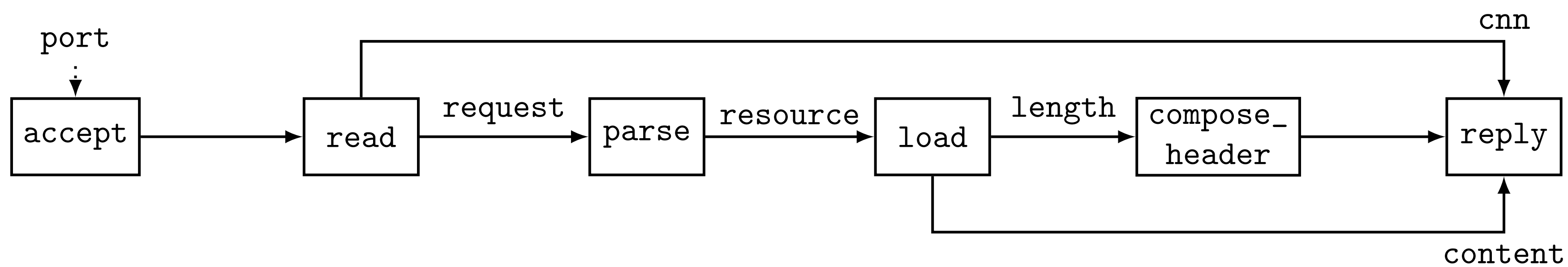
Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

Implicit Dataflow Construction




```
public class WebServer extends StreamFlexGraph {
  private Filter a, r, p, l, c, rep;

  public WebServer() {
    a   = makeFilter(Accept.class);
    r   = makeFilter(Read.class);
    p   = makeFilter(Parse.class);
    l   = makeFilter(Compose.class);
    rep = makeFilter(Reply.class);
    connect(a, r);
    connect(r, p);
    connect(p, l);
    connect(l, c);
    connect(c, rep);
    validate();
  }

  public void start() {
    new Synthesizer(a).start();
    super.start();
  }
}
```

```
fn server(port) {
  let (cnn, request)    = read(accept(port));
  let resource          = parse(request);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

Classic Operators

```
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }







}
```

⇨ Explicit dataflow

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

Classic Operators

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

⇨ Explicit dataflow

⇨ Computation

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: high-throughput stream programming in java. OOPSLA '07

Stateful Functions

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

⇨

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

⇧

Computation

Stateful Functions

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

⇨

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

⇧

Computation

Computation is (almost) a function:

- ✘ Some loss of dataflow expressiveness.
- ✔ Solid theoretical foundation.
- ✔ Deterministic.

Stateful Functions

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

Java

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

Stateful Functions

```
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

```
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

```
struct FileLoad {
   cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

Stateful Functions

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

```rust
struct FileLoad {
   cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

```c
char* load(char* resource) {
  static GHashTable* cache = g_hash_table_new();
  char* contents = NULL;
  // load file data from disk or cache (omitted)
  return contents;
}
```

Stateful Functions

?

```
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

```
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

```
struct FileLoad {
    cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

```
char* load(char* resource) {
  static GHashTable* cache = g_hash_table_new();
  char* contents = NULL;
  // load file data from disk or cache (omitted)
  return contents;
}
```

Stateful Functions

?

```
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

State encapsulation

```
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

```
struct FileLoad {
   cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

```
char* load(char* resource) {
  static GHashTable* cache = g_hash_table_new();
  char* contents = NULL;
  // load file data from disk or cache (omitted)
  return contents;
}
```

Local State

```
fn server(port) {
  let (cnn, req)          = read(accept(port));
  let resource            = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose(length), content)
}
```

```
struct FileLoad {
   cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

State is local to each call-site!

Programming Model/
Language

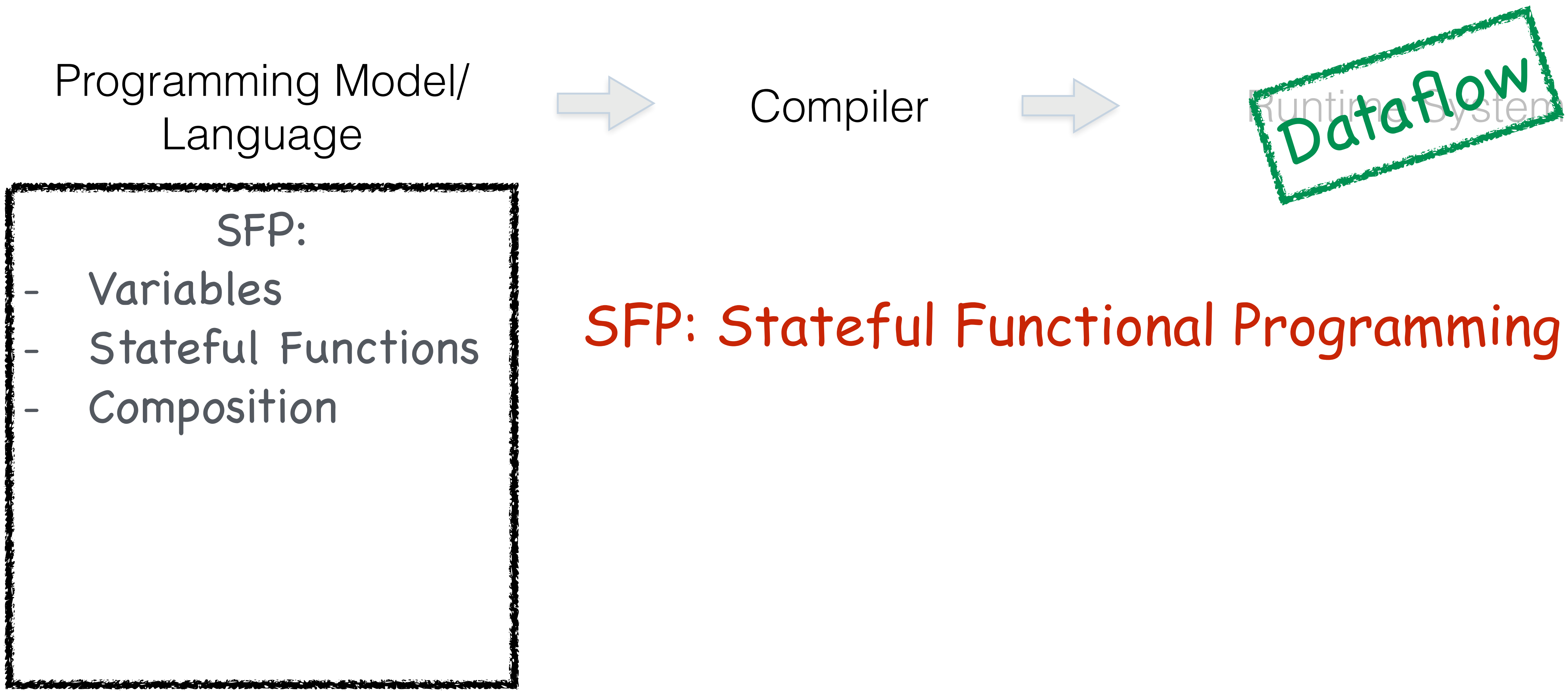
Compiler

Runtime System

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition



SFP: Stateful Functional Programming

Programming Model/
Language

Compiler

Runtime System

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Code reuse!

SFP: Stateful Functional Programming

Algorithms

```
fn server(port) {
  let (cnn, req)        = read(accept(port));
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

Top-level Algorithms

Abstraction

Lambda Algorithms

```
fn respond(cnn, content, length){
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let (cnn, req)        = read(accept(port));
  let resource          = parse(req);
  let (content, length) = load(resource);
  respond(cnn, content, length)
}
```

```
fn server(port) {
  let respond = |cnn, content, length|
                reply(cnn, compose_header(length), content);
  let (cnn, req)        = read(accept(port));
  let resource          = parse(req);
  let (content, length) = load(resource);
  respond(cnn, content, length)
}
```

Application

Programming Model/
Language

Compiler

Runtime System

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms

Programming Model/ Language

Compiler

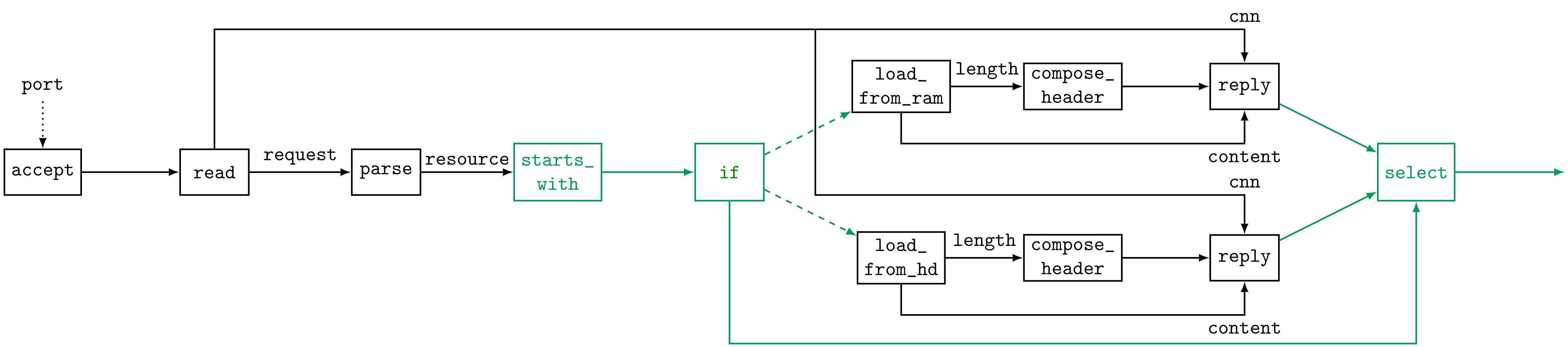
Runtime System

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow

Control Flow




```
fn respond(cnn, content, length){
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let (cnn, req)          = read(accept(port));
  let resource            = parse(req);
  let (content, length) = load(resource);
  respond(cnn, content, length)
}
```

```
fn respond(cnn, content, length){
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let (cnn, req) = read(accept(port));
  let resource   = parse(req);
  if startsWith(resource, "news/") {
    let (content, length) = load_from_ram(resource);
    respond(cnn, content, length)
  } else {
    let (content, length) = load_from_hd(resource);
    reply(cnn, content, length)
  }
}
```

Programming Model/
Language

Compiler

Runtime System

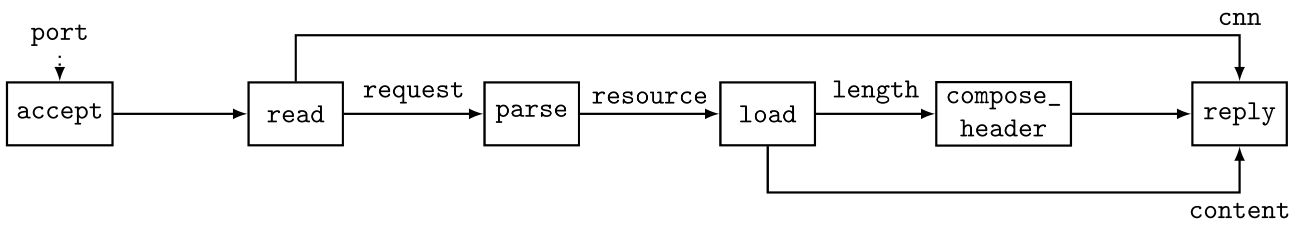
Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow

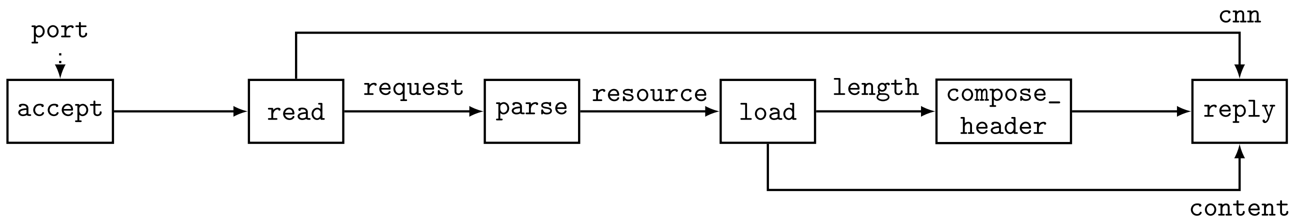
Semantic Gap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Semantic Gap

```
fn server(port) {
  let cnn                = accept(port);
  let (cnn, req)         = read(cnn);
  let resource           = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Typical PL semantics:

The ***server*** function executes **once** for each invocation.

Dataflow semantics:

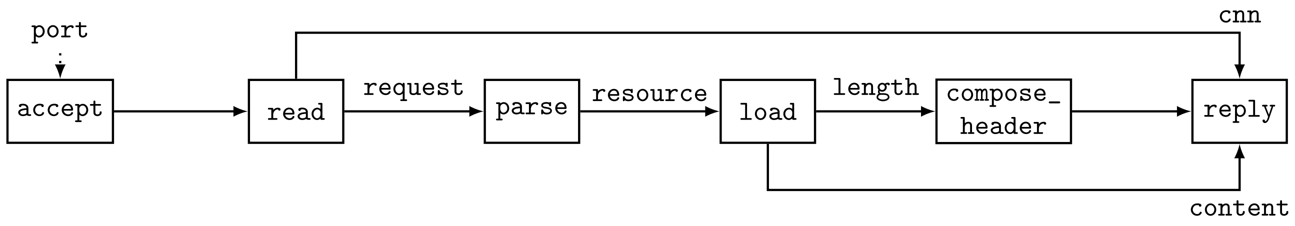
The ***server*** algorithm executes **forever** for each invocation.

⇩

The ***accept*** node is an I/O source operator, i.e., it finishes when all requests have been exhausted!

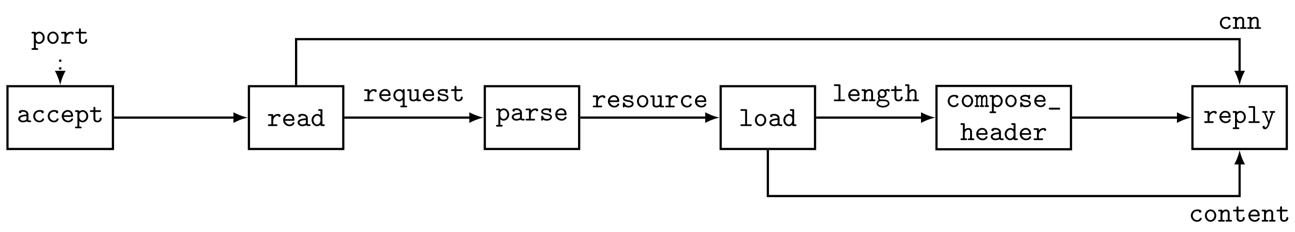
SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

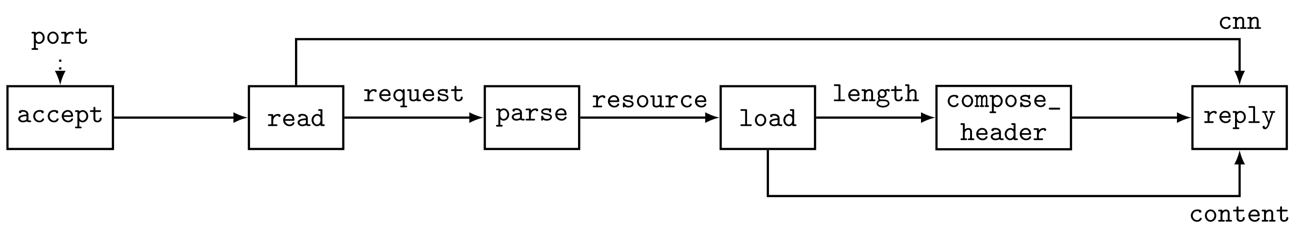


Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

SMap

```
fn server(port) {
  let cnn             = accept(port);
  let (cnn, req)      = read(cnn);
  let resource        = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

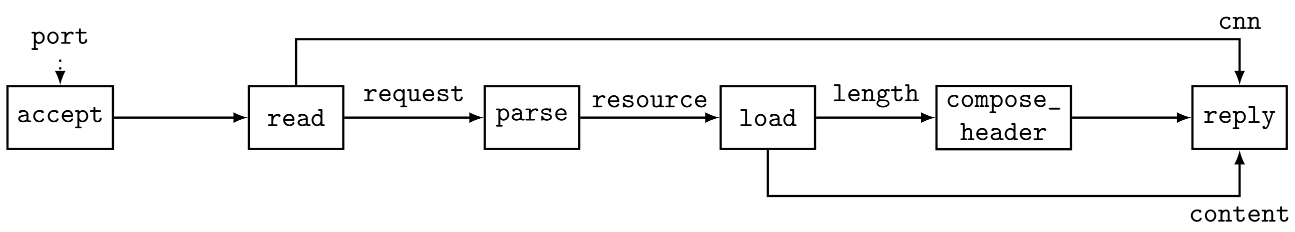
```
fn request_handling(cnn) {
  let (cnn, req)      = read(cnn);
  let resource        = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

f

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

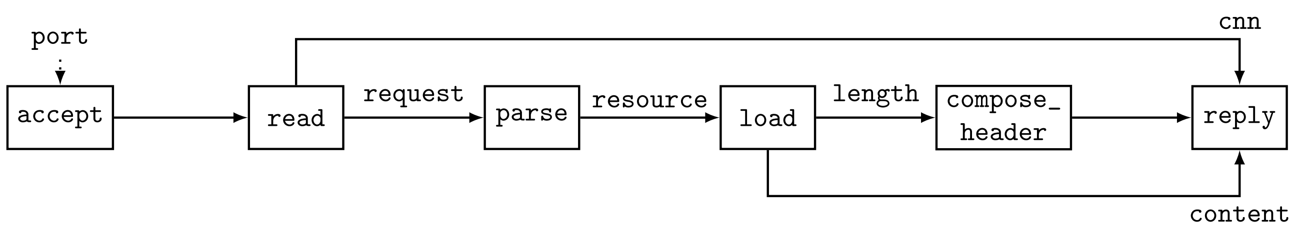
$I = [a_1, a_2]$

$\mathtt{f}$

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I)$$
$$b_1 = \mathtt{f}(a_1)$$
$$b_2 = \mathtt{f}(a_2)$$

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

$\mathtt{f}$

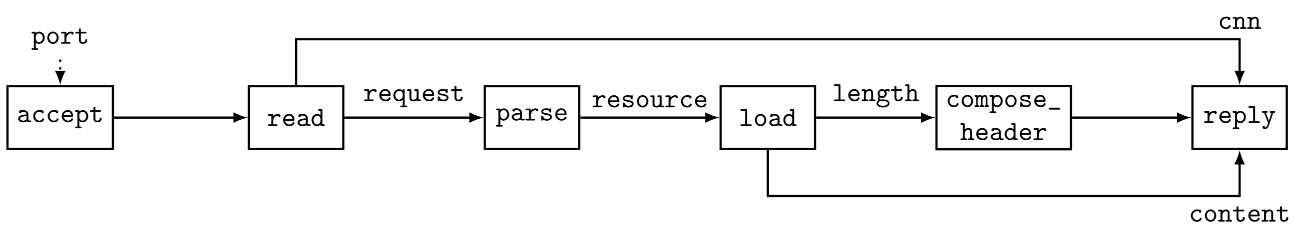
$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \quad = \quad I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I)$$

$$b_1 = \mathtt{f}(a_1) \qquad (s'_f, b_1) = \mathtt{f}(s_f, a_1)$$

$$b_2 = \mathtt{f}(a_2) \qquad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

$\mathtt{f}$

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \quad = \quad I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I)$$

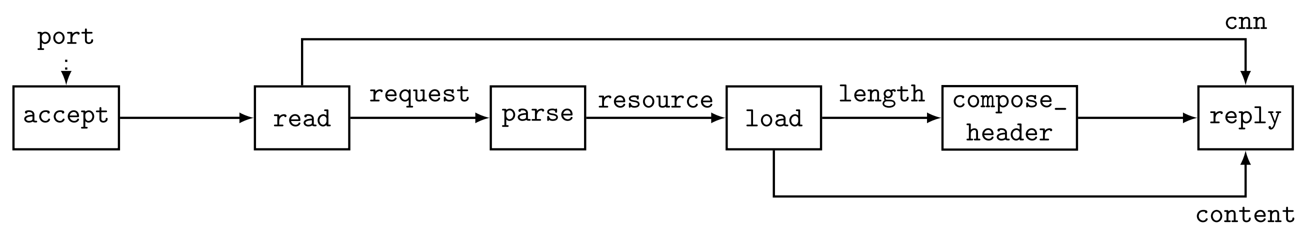
$$b_1 = \mathtt{f}(a_1) \qquad (s'_f, b_1) = \mathtt{f}(s_f, a_1)$$

$$b_2 = \mathtt{f}(a_2) \qquad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

data parallel   strictly sequential

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

$\mathtt{f}$

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \quad = \quad I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I)$$

$$b_1 = \mathtt{f}(a_1) \qquad (s'_f, b_1) = \mathtt{f}(s_f, a_1)$$

$$b_2 = \mathtt{f}(a_2) \qquad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

```
fn f(a)}{
  let x = g(a);
  h(x)
}
```

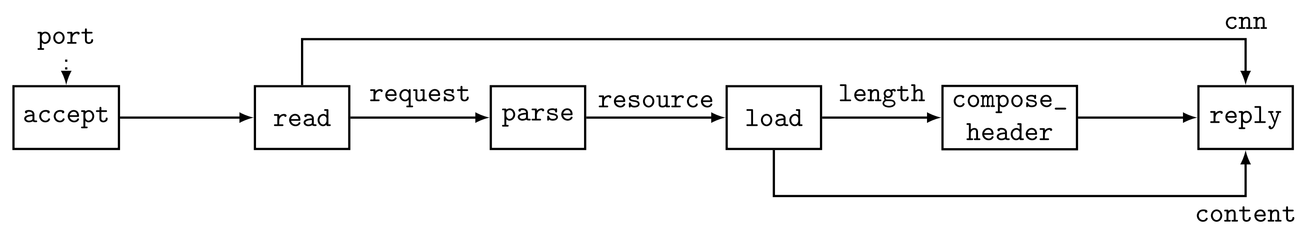
$\mathtt{f} = \mathtt{h} \circ \mathtt{g}$

$s_f = (s_g, s_h)$

data parallel strictly sequential

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

f

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \quad b_1 = \mathtt{f}(a_1) \quad b_2 = \mathtt{f}(a_2)$$

=

$$I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I) \quad (s'_f, b_1) = \mathtt{f}(s_f, a_1) \quad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

```
fn f(a){
  let x = g(a);
  h(x)
}
```

$\mathtt{f} = \mathtt{h} \circ \mathtt{g}$

$s_f = (s_g, s_h)$

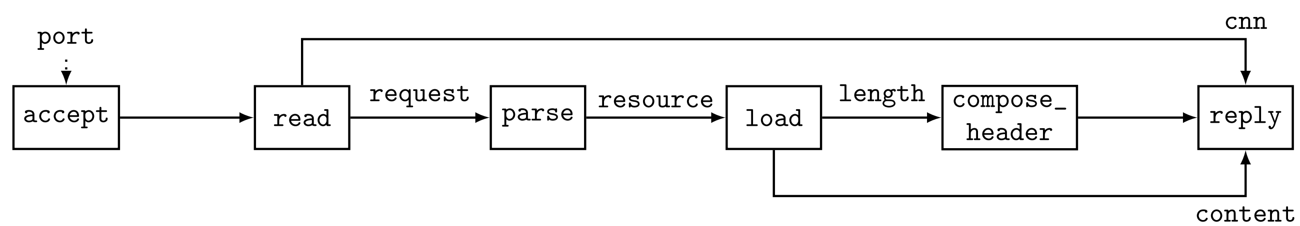
$$(s'_g, x_1) = \mathtt{g}(s_g, a_1)$$
$$(s''_g, x_2) = \mathtt{g}(s'_g, a_2)$$
$$(s'_h, b_1) = \mathtt{h}(s_h, x_1)$$
$$(s''_h, b_2) = \mathtt{h}(s'_h, x_2)$$

data parallel

strictly sequential

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

$\mathtt{f}$

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \qquad b_1 = \mathtt{f}(a_1) \qquad b_2 = \mathtt{f}(a_2)$$

=

$$I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I) \qquad (s'_f, b_1) = \mathtt{f}(s_f, a_1) \qquad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

```
fn f(a)}{
  let x = g(a);
  h(x)
}
```

⇨

$\mathtt{f} = \mathtt{h} \circ \mathtt{g}$

$s_f = (s_g, s_h)$

$(s'_g, x_1) = \mathtt{g}(s_g, a_1)$

$(s''_g, x_2) = \mathtt{g}(s'_g, a_2)$

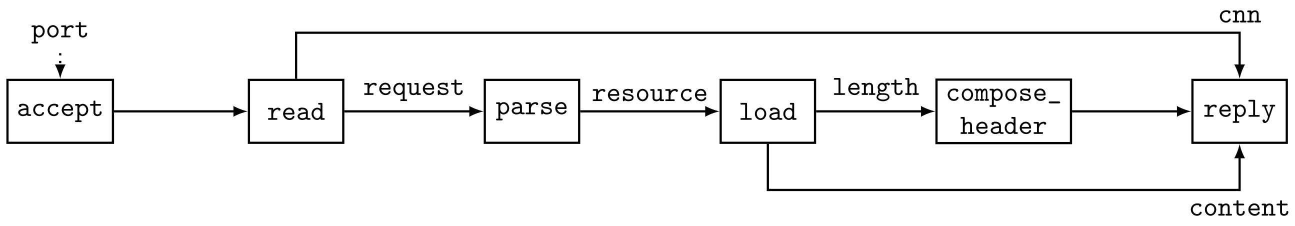
$(s'_h, b_1) = \mathtt{h}(s_h, x_1)$

$(s''_h, b_2) = \mathtt{h}(s'_h, x_2)$

data parallel strictly sequential

SMap

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}

fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

f

$$I' = [b_1, b_2] = \mathtt{map}(\mathtt{f}, I) \quad b_1 = \mathtt{f}(a_1) \quad b_2 = \mathtt{f}(a_2)$$

=

$$I' = [b_1, b_2] = \mathtt{smap}(\mathtt{f}, I) \quad (s'_f, b_1) = \mathtt{f}(s_f, a_1) \quad (s''_f, b_2) = \mathtt{f}(s'_f, a_2)$$

```
fn f(a)}{
  let x = g(a);
  h(x)
}
```

$\mathtt{f} = \mathtt{h} \circ \mathtt{g}$

$s_f = (s_g, s_h)$

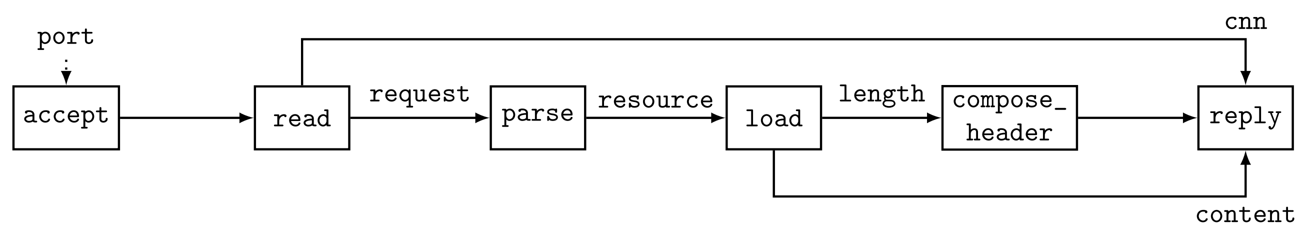
$$(s'_g, x_1) = \mathtt{g}(s_g, a_1)$$
$$(s''_g, x_2) = \mathtt{g}(s'_g, a_2)$$
$$(s'_h, b_1) = \mathtt{h}(s_h, x_1)$$
$$(s''_h, b_2) = \mathtt{h}(s'_h, x_2)$$

$$(s'_g, x_1) = \mathtt{g}(s_g, a_1)$$
$$(s'_h, b_1) = \mathtt{h}(s_h, x_1)$$
$$(s''_g, x_2) = \mathtt{g}(s'_g, a_2)$$
$$(s''_h, b_2) = \mathtt{h}(s'_h, x_2)$$

data parallel | strictly sequential | pipeline parallel

SMap

```
fn server(port) {
  let cnn              = accept(port);
  let (cnn, req)       = read(cnn);
  let resource         = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```



Abstract over processing:

```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnn = accept(port);
  request_handling(cnn)
}
```

$I = [a_1, a_2]$

f

$I' = [b_1, b_2] = \mathrm{map}(\mathrm{f}, I)$ = $I' = [b_1, b_2] = \mathrm{smap}(\mathrm{f}, I)$

$b_1 = \mathrm{f}(a_1)$ $(s'_f, b_1) = \mathrm{f}(s_f, a_1)$

$b_2 = \mathrm{f}(a_2)$ $(s''_f, b_2) = \mathrm{f}(s'_f, a_2)$

data parallel strictly sequential

Abstract over data:

Lists
- Finite
- Infinite (stream)

Generator
- Location independent

```
fn server(port) {
  let cnns:List<Cnn>    = infinite_generator(|| accept(port));
  let results:List<()> = smap(request_handling, cnns);
  ()
}
```

```
fn f(a){
  let x = g(a);
  h(x)
}
```

$\mathrm{f} = \mathrm{h} \circ \mathrm{g}$

$s_f = (s_g, s_h)$

$(s'_g, x_1) = \mathrm{g}(s_g, a_1)$ $(s'_g, x_1) = \mathrm{g}(s_g, a_1)$

$(s''_g, x_2) = \mathrm{g}(s'_g, a_2)$ $(s'_h, b_1) = \mathrm{h}(s_h, x_1)$

$(s'_h, b_1) = \mathrm{h}(s_h, x_1)$ $(s''_g, x_2) = \mathrm{g}(s'_g, a_2)$

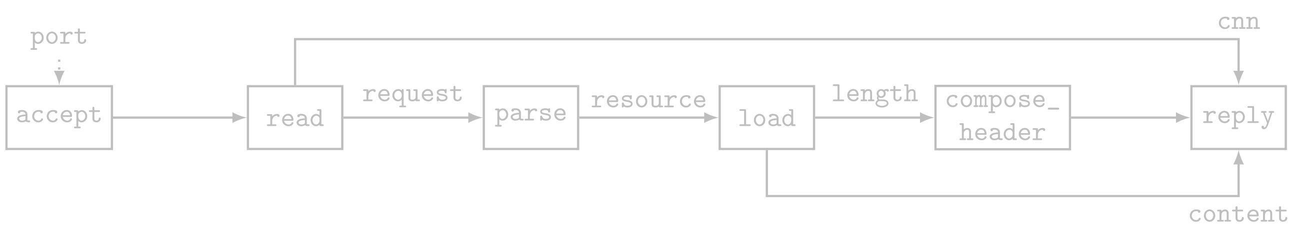
$(s''_h, b_2) = \mathrm{h}(s'_h, x_2)$ $(s''_h, b_2) = \mathrm{h}(s'_h, x_2)$

pipeline parallel

Allison L. Continuations implement generators and streams. The Computer Journal. 1990.

SMap

```
fn server(port) {
  let cnn                 = accept(port);
  let (cnn, req)          = read(cnn);
  let resource            = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

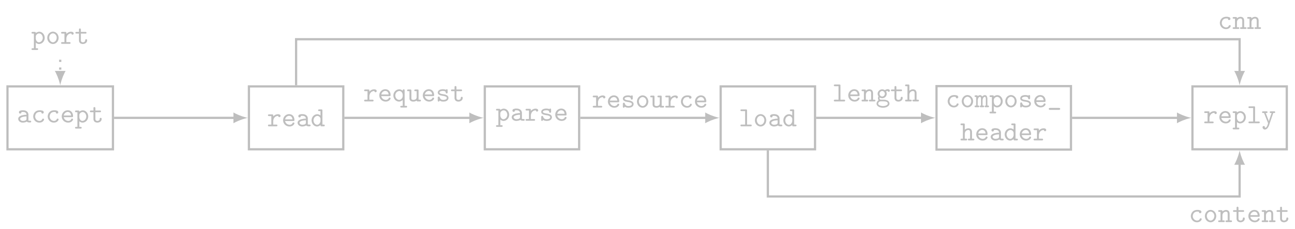



```
fn request_handling(cnn) {
  let (cnn, req)          = read(cnn);
  let resource            = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnns:List<Cnn>    = infinite_generator(|| accept(port));
  let results:List<()> = smap(request_handling, cnns);
  ()
}
```

SMap as a Loop

```
fn server(port) {
  let cnn                = accept(port);
  let (cnn, req)         = read(cnn);
  let resource           = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```




```
fn request_handling(cnn) {
  let (cnn, req)         = read(cnn);
  let resource           = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

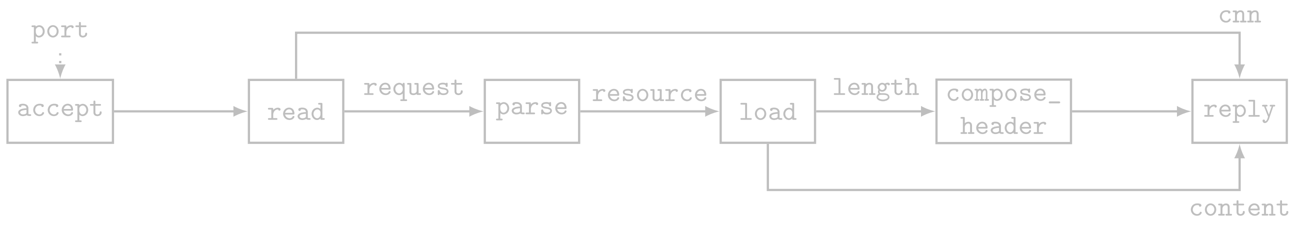
```
fn server(port) {
  let cnns:List<Cnn>    = infinite_generator(|| accept(port));
  let results:List<()> = for cnn in cnns {
                            request_handling(cnn)
                          };
  ()
}
```

=

```
fn server(port) {
  let cnns:List<Cnn>    = infinite_generator(|| accept(port));
  let results:List<()> = smap(request_handling, cnns);
  ()
}
```

From SMap to Dataflow

```
fn server(port) {
  let cnn               = accept(port);
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

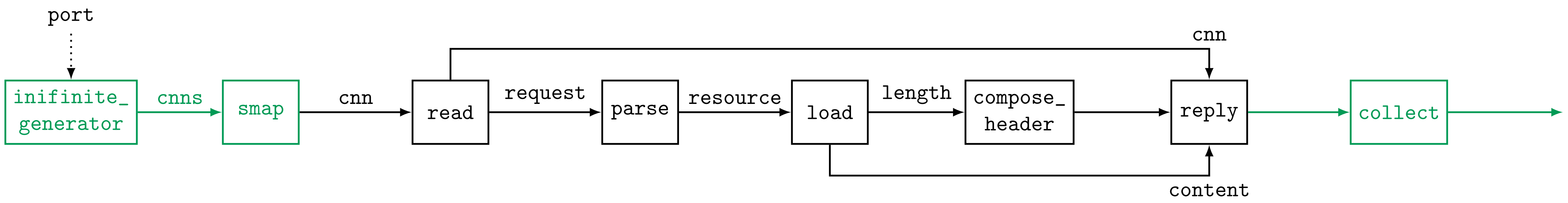



```
fn request_handling(cnn) {
  let (cnn, req)        = read(cnn);
  let resource          = parse(req);
  let (content, length) = load(resource);
  reply(cnn, compose_header(length), content)
}
```

```
fn server(port) {
  let cnns:List<Cnn>   = infinite_generator(|| accept(port));
  let results:List<()> = for cnn in cnns {
                           request_handling(cnn)
                         };
  ()
}
```

=

```
fn server(port) {
  let cnns:List<Cnn>   = infinite_generator(|| accept(port));
  let results:List<()> = smap(request_handling, cnns);
  ()
}
```



Programming Model/
Language

SFP:
- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

Compiler

Runtime System

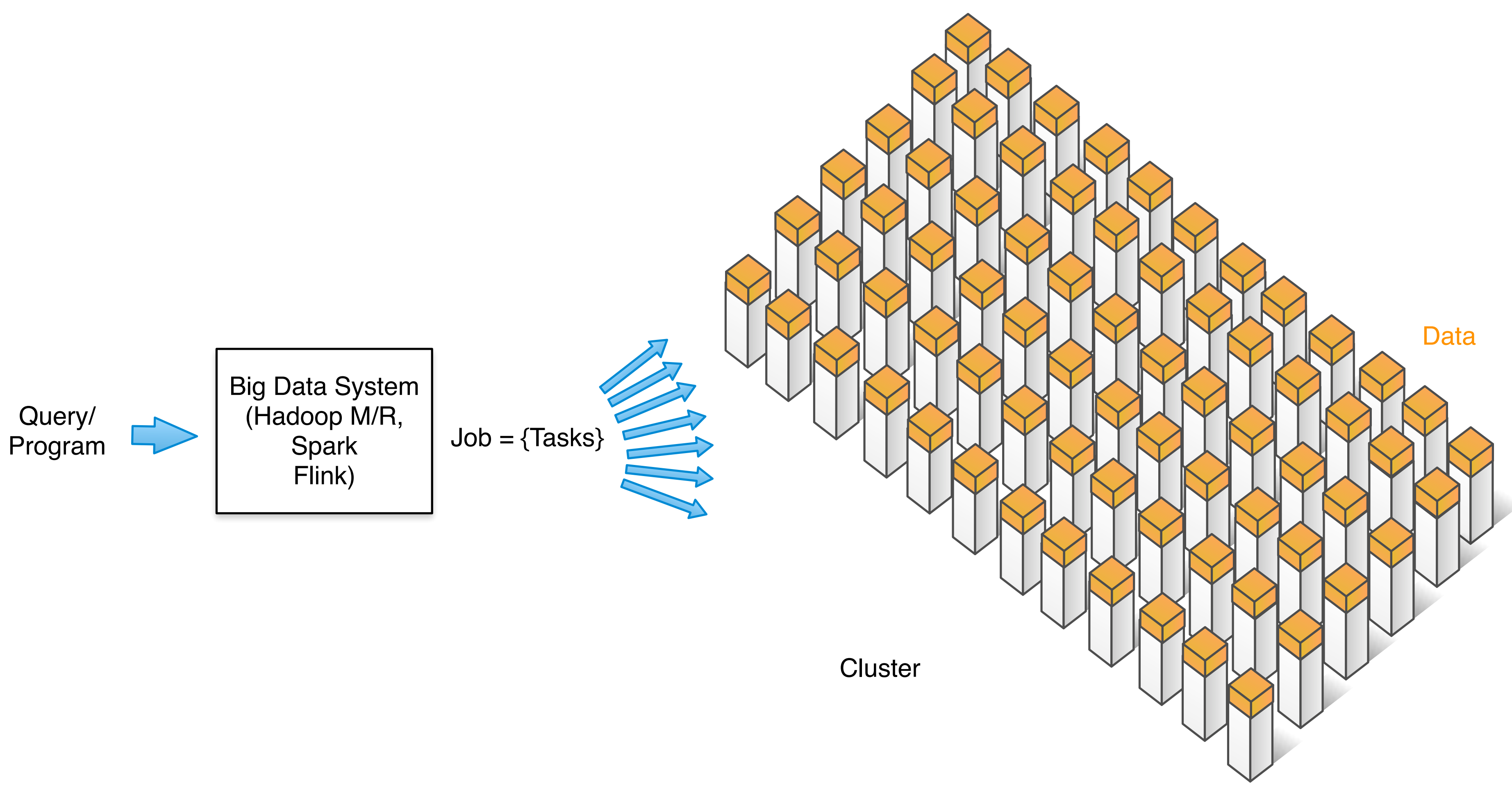
Dataflow

Programming Model/ Language → Compiler → ~~Runtime System~~ Dataflow

- No concurrency abstractions.
- Practicality:
  - Integration into existing programming models (e.g., OOP)/ languages.
  - Gradual switch/ reuse of existing code base.

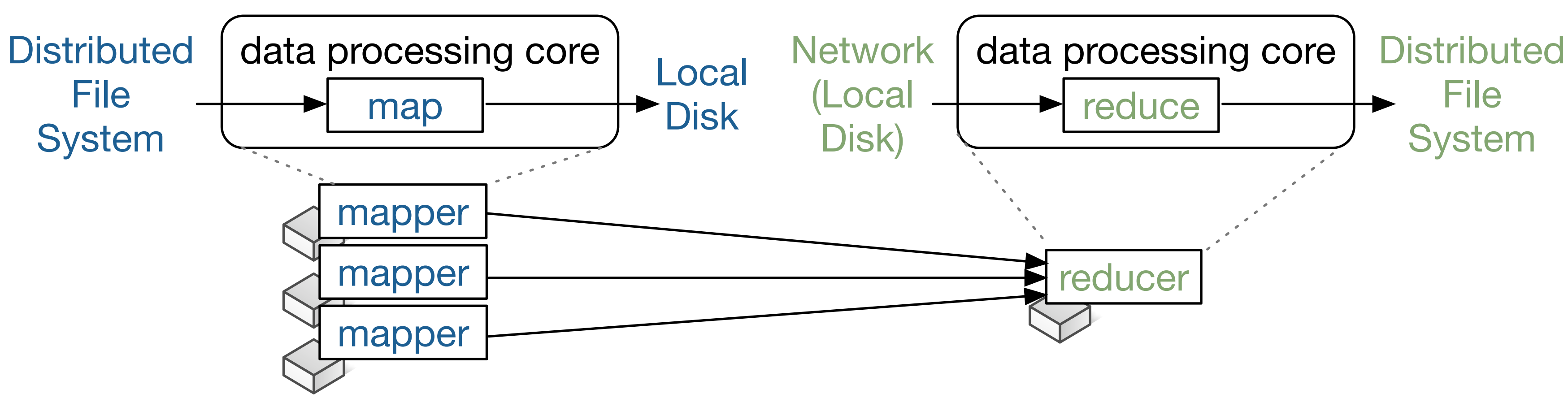
Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster
2. General wisdom: The main bottleneck is I/O (disk and network)



Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster for independent tasks.
2. General wisdom: The main bottleneck is I/O (disk and network)



Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM. 2008.

Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster for independent tasks.
2. General wisdom: The main bottleneck is I/O (disk and network) for simple data.

Jobs are CPU-bound!

WordCount, Sort		Analytics Queries (in Hive)
Simple Data Formats	vs.	Complex Data Formats (Parquet, Tables, JSON)
Uncompressed		Compressed

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks.(NSDI'15).

Big Data Systems

1. Big Data Systems (BDSs) scale with the number of cores in the cluster for independent tasks.
2. General wisdom: The main bottleneck is I/O (disk and network) for simple data.

Jobs are CPU-bound!

WordCount, Sort		Analytics Queries (in Hive)
Simple Data Formats	vs.	Complex Data Formats (Parquet, Tables, JSON)
Uncompressed		Compressed

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks.(NSDI'15).

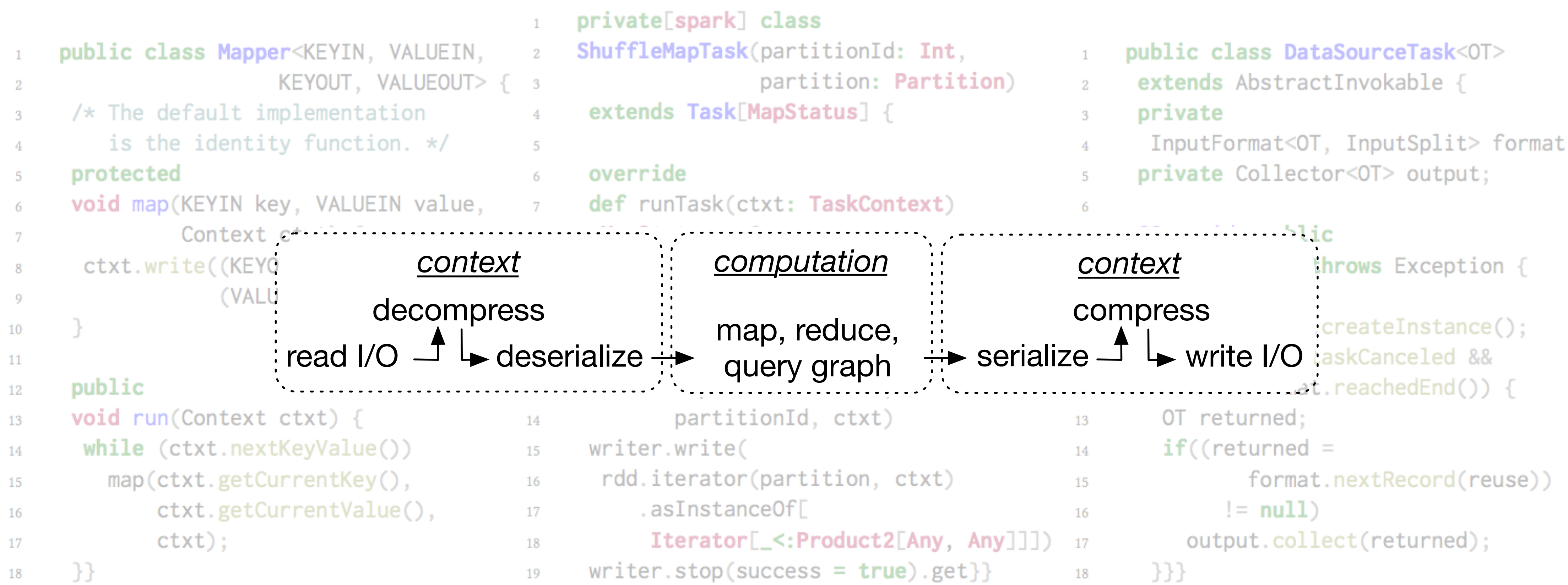
BDS cores/data processing pipelines do not scale well on multicores!

- Local optimizations do not solve this problem.
- BDSs do not benefit from new network HW.

➡ Rewrite data processing cores!

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On the [ir]relevance of network performance for data processing. (HotCloud'16).

Data Processing Cores



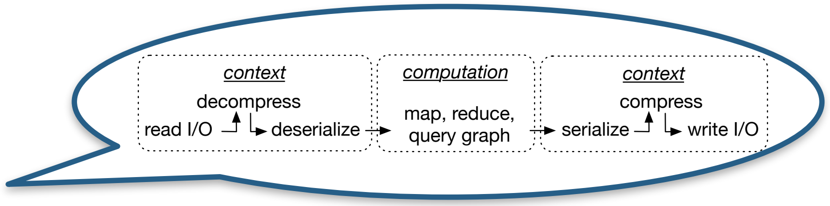
(a) Hadoop (b) Spark (c) Flink

Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
  output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
  let (line, content) = split;
  let kv_pairs = hmr_map(mapper, line, content);
  for kv_pair in kv_pairs {
    let (k, v) = kv_pair;
    output_side(writer, k, v)
  }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
  let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
  for split in splits {
    compute_and_output(mapper, writer, split)
  }
}
```

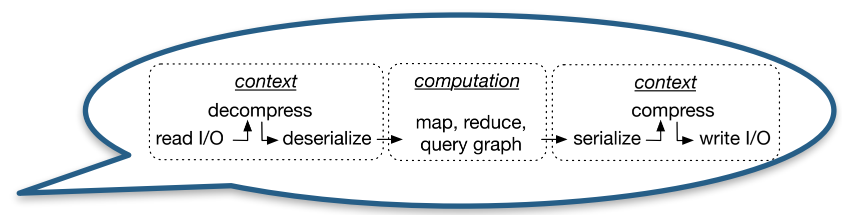
Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
  output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
  let (line, content) = split;
  let kv_pairs = hmr_map(mapper, line, content);
  for kv_pair in kv_pairs {
    let (k, v) = kv_pair;
    output_side(writer, k, v)
  }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
  let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
  for split in splits {
    compute_and_output(mapper, writer, split)
  }
}
```



Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
  output(writer, key, value)
}

fn compute_and_output(mapper, writer, split) {
  let (line, content) = split;
  let kv_pairs = hmr_map(mapper, line, content);
  for kv_pair in kv_pairs {
    let (k, v) = kv_pair;
    output_side(writer, k, v)
  }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
  let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
  for split in splits {
    compute_and_output(mapper, writer, split)
  }
}
```

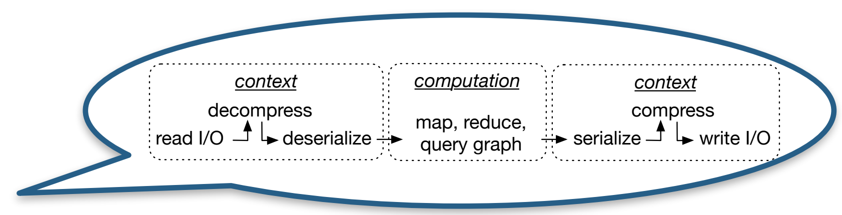


Rewriting Hadoop Map/Reduce

```
fn output_side(writer, key, value) {
  output(writer, key, value)
}
```

```
writer.write(key, value)
```

```
fn compute_and_output(mapper, writer, split) {
  let (line, content) = split;
  let kv_pairs = hmr_map(mapper, line, content);
  for kv_pair in kv_pairs {
    let (k, v) = kv_pair;
    output_side(writer, k, v)
  }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
  let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
  for split in splits {
    compute_and_output(mapper, writer, split)
  }
}
```

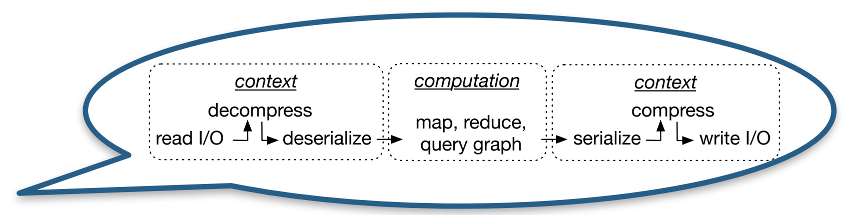


Rewriting Hadoop Map/Reduce

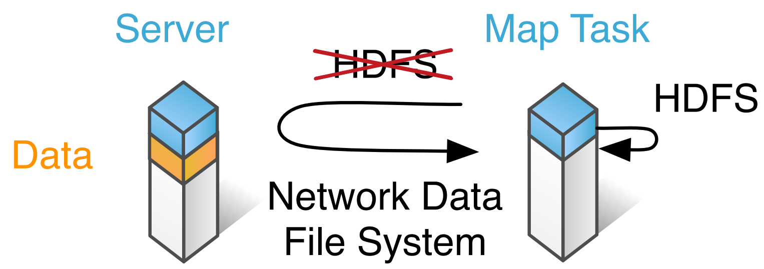
```
fn output_side(writer, key, value) {
  output(writer, key, value)
}
                    writer.write(key, value)

fn compute_and_output(mapper, writer, split) {
  let (line, content) = split;
  let kv_pairs = hmr_map(mapper, line, content);
  for kv_pair in kv_pairs {
    let (k, v) = kv_pair;
    output_side(writer, k, v)
  }
}

fn coarse(reader: org.apache.hadoop.mapreduce.Mapper$Context,
          mapper: org.apache.hadoop.mapreduce.Mapper,
          writer: org.apache.hadoop.mapreduce.Mapper$Context) {
  let splits = finite_generator(|| load_split(reader),
                                || has_more_data(reader));
  for split in splits {
    compute_and_output(mapper, writer, split)
  }
}
```

4 variants:

- Coarse (C)
- Coarse Input Fine Output (CIFO)
- Fine Input Coarse Output (FICO)
- Fine (F)



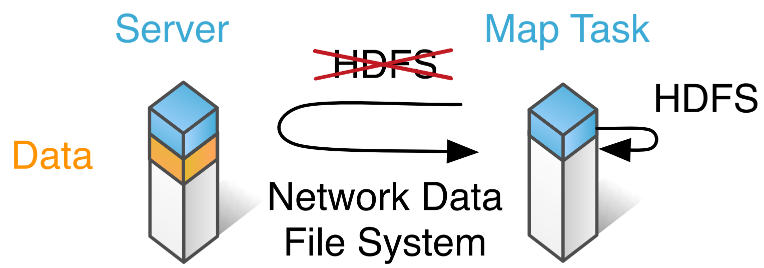
Evaluation



Almost the same setup as for a reduce task!

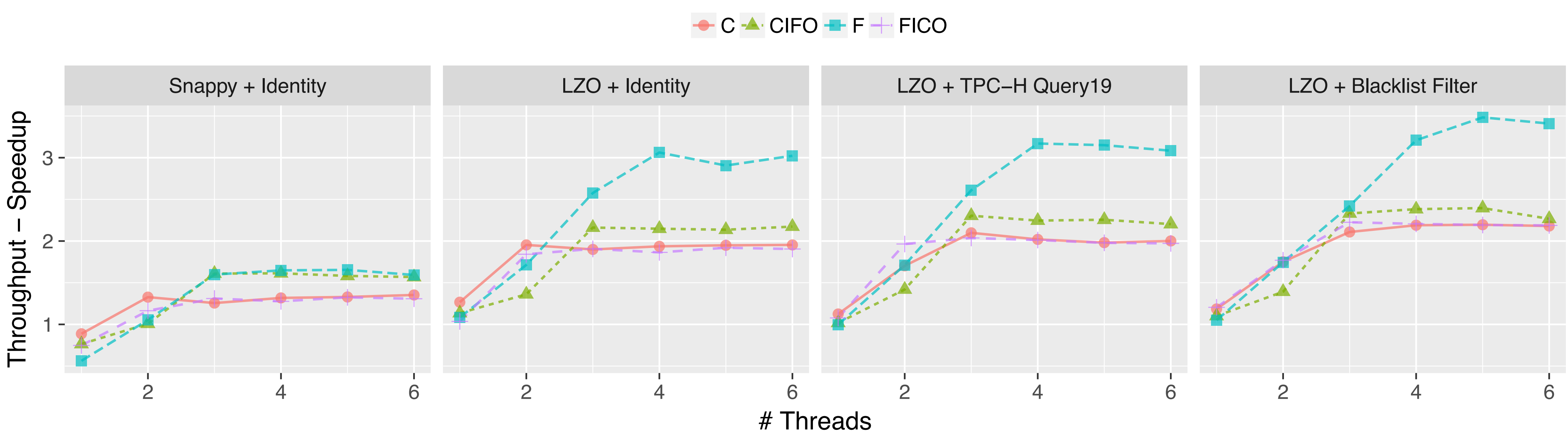
- SequenceFile
- JSON
- Snappy, LZO
- TPC-H Parts table

Throughput Analysis



Almost the same setup as for a reduce task!

- SequenceFile
- JSON
- Snappy, LZO
- TPC-H Parts table



Speedup of up to 3.5x for compute-intensive configurations!

Programming Model/
Language

Compiler

~~Runtime System~~

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM.

Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'18). ACM.

Programming Model/ Language → Compiler → Runtime System

SFP

Dataflow

- No concurrency abstractions.
- Practicality:
  - Integration into existing programming models (e.g., OOP)/ languages.
  - Gradual switch/ reuse of existing code base.

Programming Model/ Language

SFP

Compiler

Runtime System

Dataflow

Dataflow Model (Graph):

- Missing established framework for rewrites
- Rewrites not verified for correctness
- Missing library/research for rewrites
- Very domain-specific

PL/Compiler Co-Design

Programming Model/ Language

Compiler

Runtime System

Dataflow

SFP:

- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

J. Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. Commun. ACM, 1978.

Barendregt, Hendrik Pieter. "Functional programming and lambda calculus." *Formal models and semantics*. Elsevier, 1990.

PL/Compiler Co-Design

Programming Model/ Language → Compiler → Runtime System

Dataflow

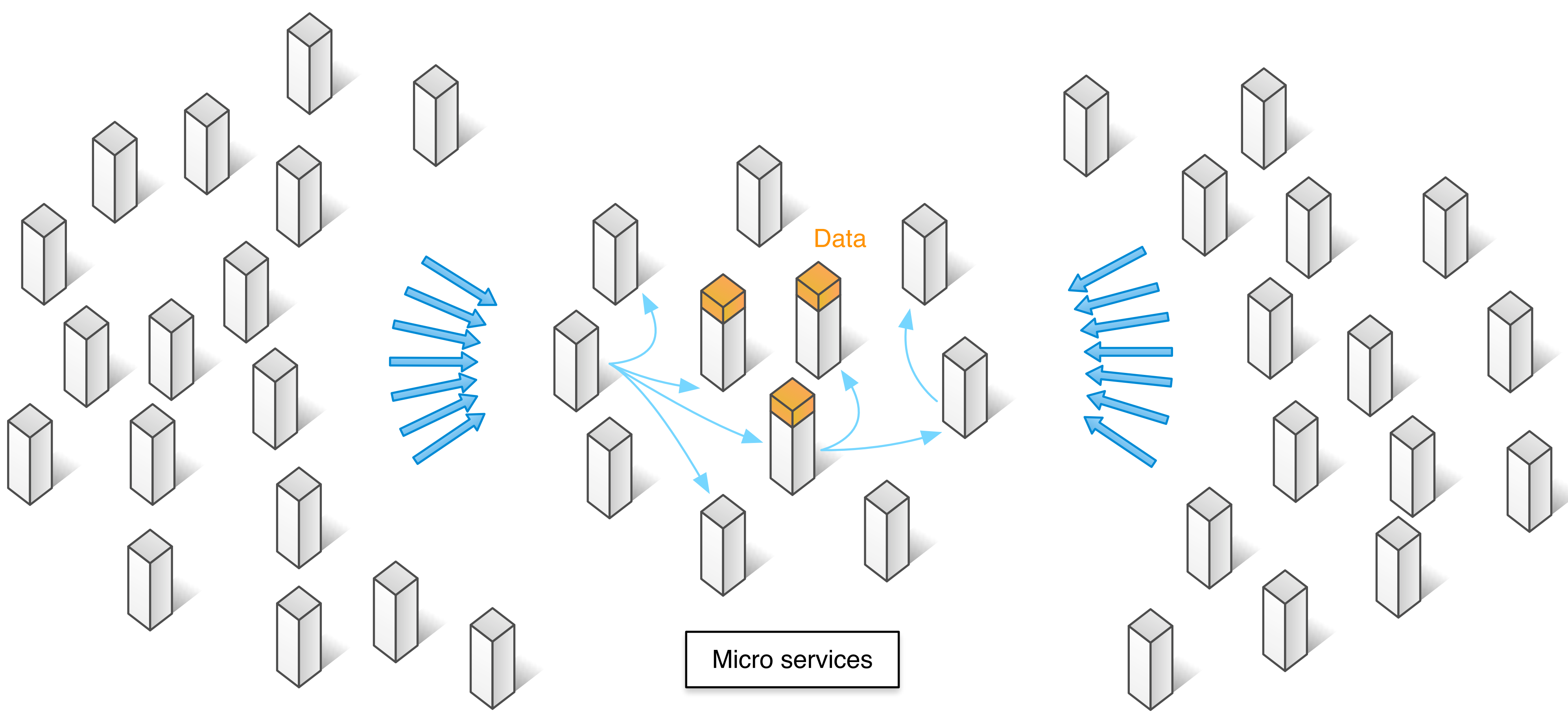
SFP:
- Variables
- Stateful Functions
- Composition
- Algorithms
- Control Flow
- SMap

Lambda calculus:
- Variables
- Abstraction
- Application

J. Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. Commun. ACM, 1978.

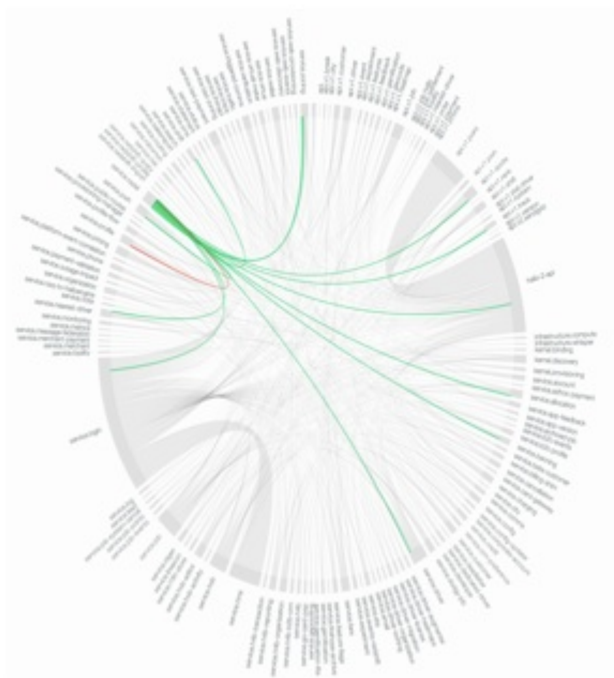
Barendregt, Hendrik Pieter. "Functional programming and lambda calculus." *Formal models and semantics*. Elsevier, 1990.

Microservice Architectures



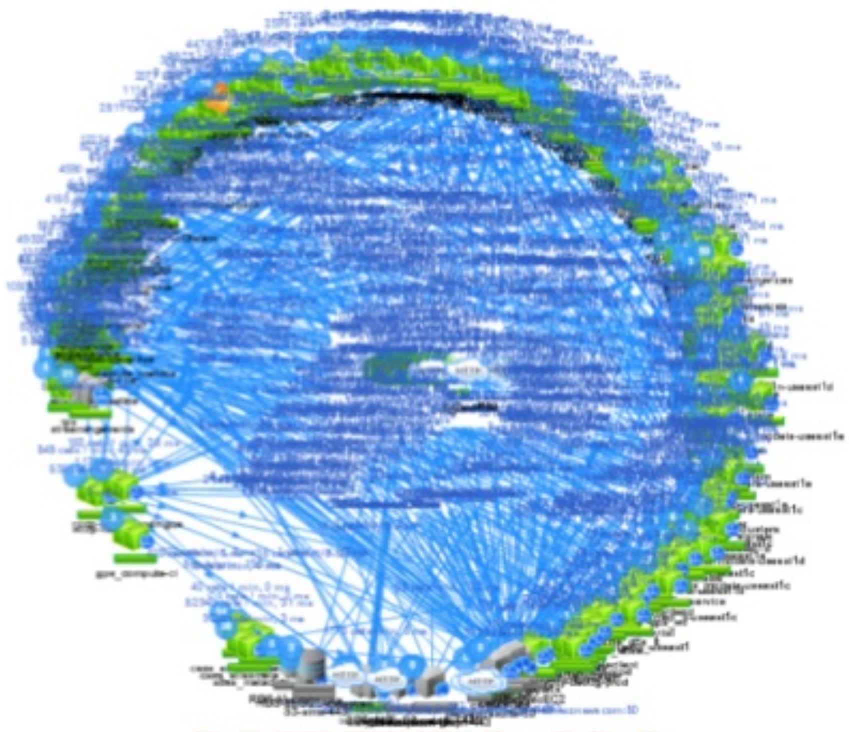
Redefining the Death Star

450 microservices



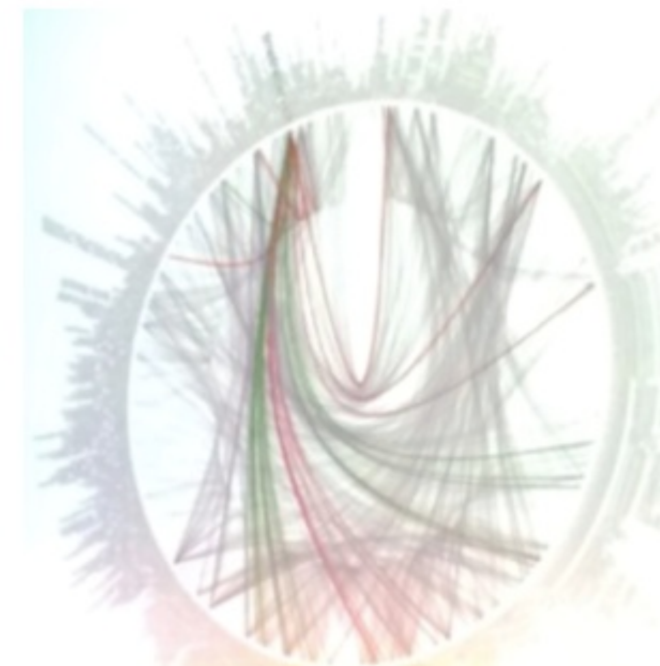
HAIL O

500+ microservices



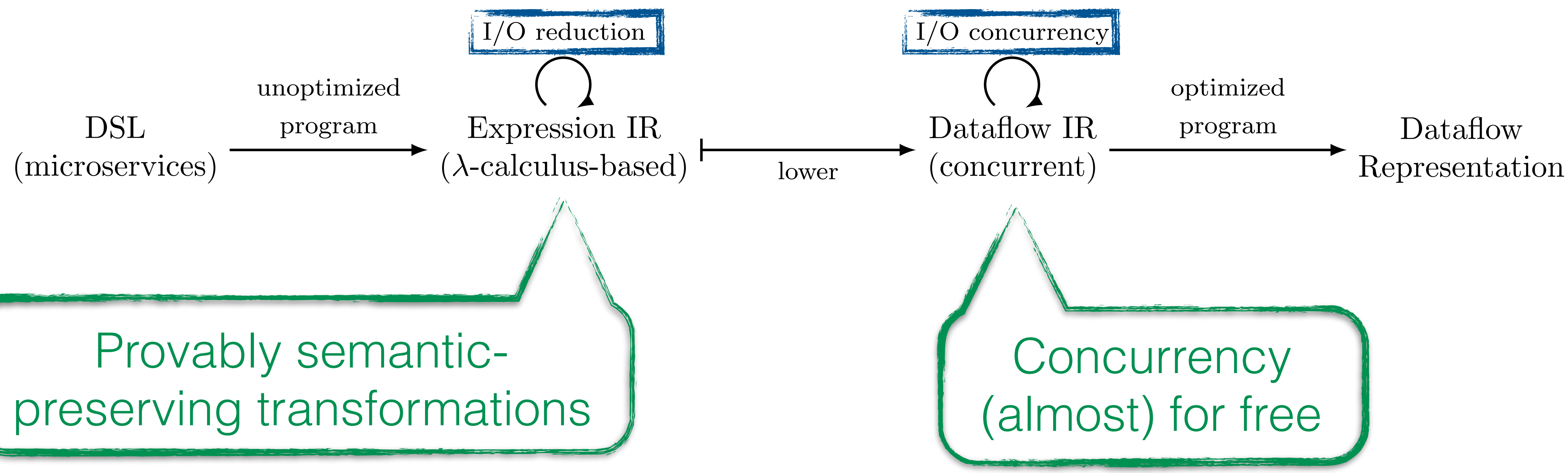
NETFLIX

500+ microservices

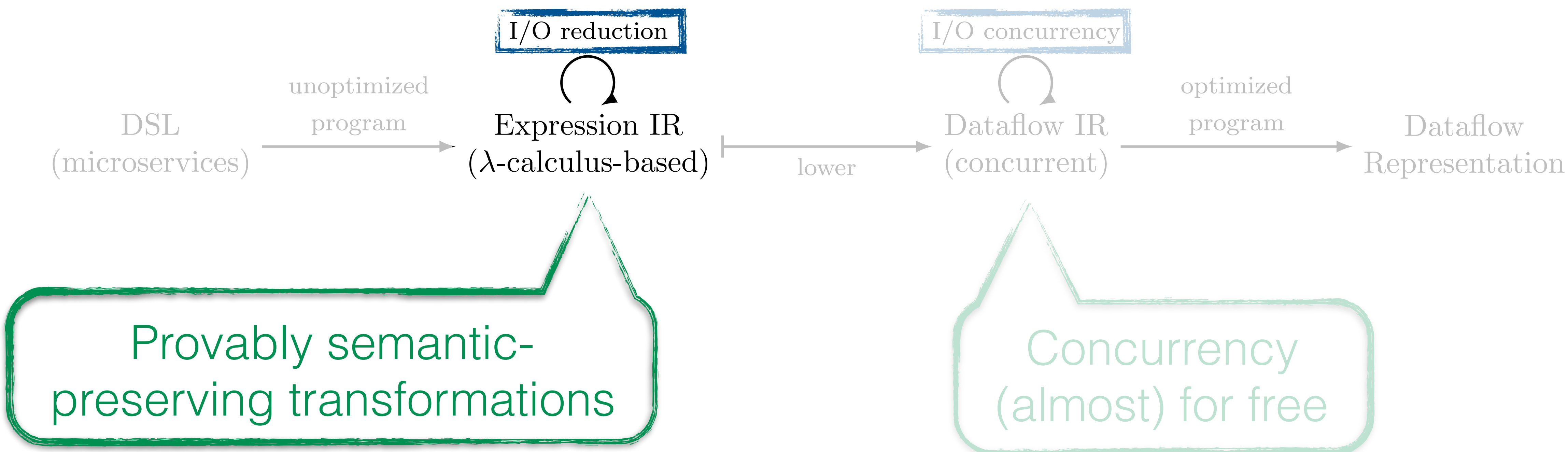


Source:
Netflix: http://www.slideshare.net/BruceWong3/the-case-for-chaos
Twitter: https://twitter.com/adrianco/status/441883572618948608
Hail-o: https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/

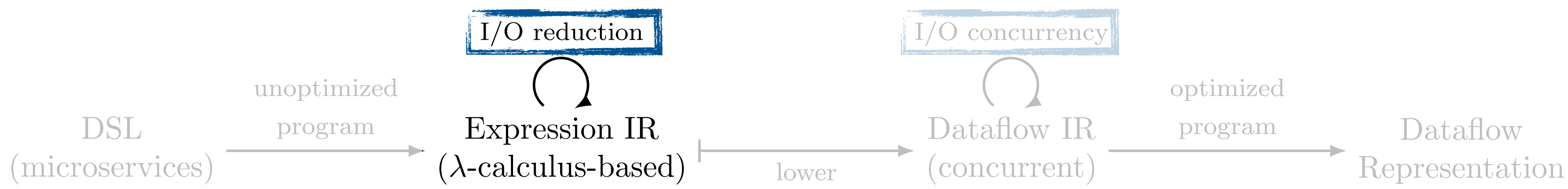
Compiler


I/O reduction
I/O concurrency
DSL
(microservices)
unoptimized
program
Expression IR
(λ-calculus-based)
lower
Dataflow IR
(concurrent)
optimized
program
Dataflow
Representation
Provably semantic-
preserving transformations
Concurrency
(almost) for free

Compiler


I/O reduction
I/O concurrency
unoptimized program
optimized program
DSL (microservices)
Expression IR (λ-calculus-based)
lower
Dataflow IR (concurrent)
Dataflow Representation
Provably semantic-preserving transformations
Concurrency (almost) for free

Foundation: Call-by-need Lambda Calculus

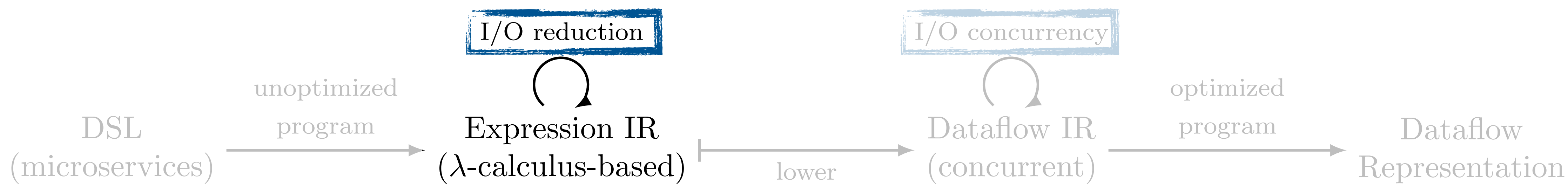


t ::= v
| λv.t
| t t
| let v=t in t

- Variables
- Abstraction
- Application
- (Bindings)

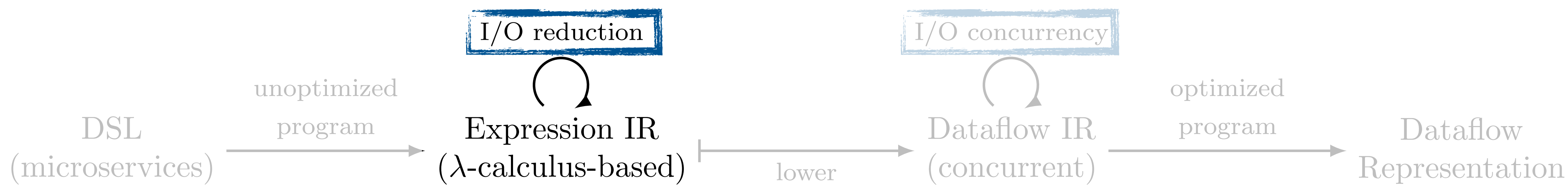
Ariola ZM, Felleisen M. The call-by-need lambda calculus. Journal of functional programming. 1997.
Maraist J, Odersky M, Wadler P. The call-by-need lambda calculus. Journal of functional programming. 1998.
Chang S, Felleisen M. The call-by-need lambda calculus, revisited. In European Symposium on Programming 2012. Springer.

Foundation: Call-by-need Lambda Calculus



t ::= v
  | λv.t
  | t t
  | let v=t in t
  | if (t t t)

Domain-specific Aspects



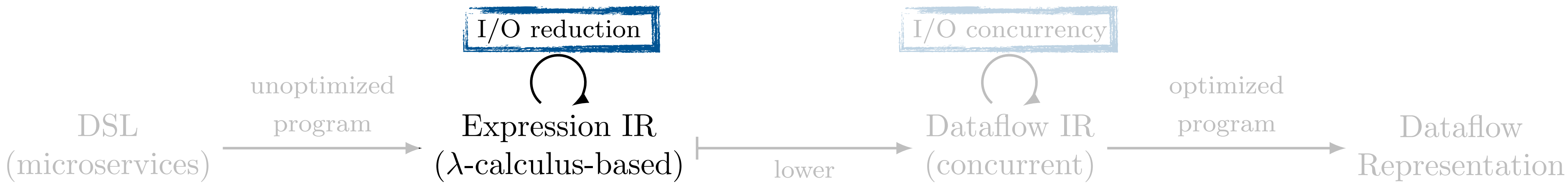
$$
\begin{array}{rl}
t ::= & v \\
| & \lambda v.t \\
| & t\ t \\
| & \text{let } v{=}t \text{ in } t \\
| & \text{if } (t\ t\ t) \\
| & ff_f(x_1 \ldots x_n) \\
| & io(x)
\end{array}
$$

$map(\lambda v.t\ [v_1 \ldots v_n])$

- stateless/stateful functions
- I/O (read) request

- SMap

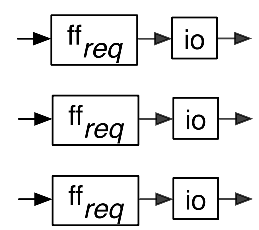
I/O Transformations



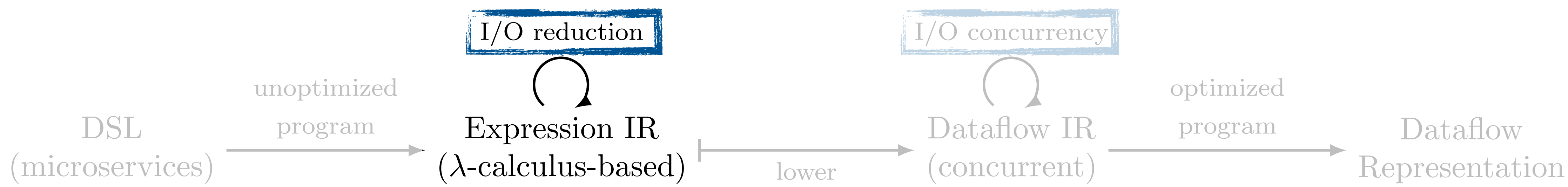
t ::= v
| $\lambda v.t$
| $t\ t$
| let v=t in t
| if ($t\ t\ t$)
| $ff_f(x_1 \dots x_n)$
| io(x)

map($\lambda v.t$ [$v_1 \dots v_n$])

- Grouping independent I/O calls:



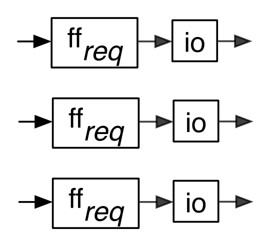
I/O Transformations



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

map(λv.t [$v_1 \ldots v_n$])

- Grouping independent I/O calls:



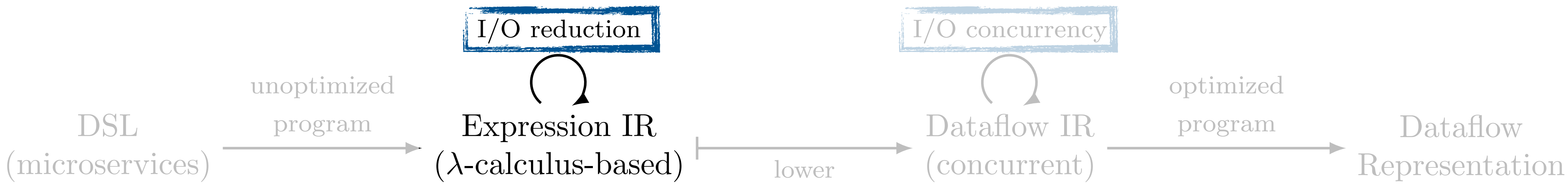
Remember SMap

$$\begin{aligned}(s'_g, x_1) &= \mathtt{g}(s_g, a_1)\\ (s''_g, x_2) &= \mathtt{g}(s'_g, a_2)\\ (s'_h, b_1) &= \mathtt{h}(s_h, x_1)\\ (s''_h, b_2) &= \mathtt{h}(s'_h, x_2)\end{aligned}$$

$$\begin{aligned}(s'_g, x_1) &= \mathtt{g}(s_g, a_1)\\ (s'_h, b_1) &= \mathtt{h}(s_h, x_1)\\ (s''_g, x_2) &= \mathtt{g}(s'_g, a_2)\\ (s''_h, b_2) &= \mathtt{h}(s'_h, x_2)\end{aligned}$$

pipeline parallel

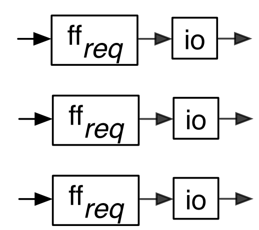
I/O Transformations



$$
\begin{aligned}
t ::=\ & v \\
|\ & \lambda v.t \\
|\ & t\ t \\
|\ & \text{let } v = t \text{ in } t \\
|\ & \text{if } (t\ t\ t) \\
|\ & ff_f(x_1 \dots x_n) \\
|\ & io(x)
\end{aligned}
$$

$\text{map}(\lambda v.t\ [v_1 \dots v_n])$

- Grouping <u>independent</u> I/O calls:



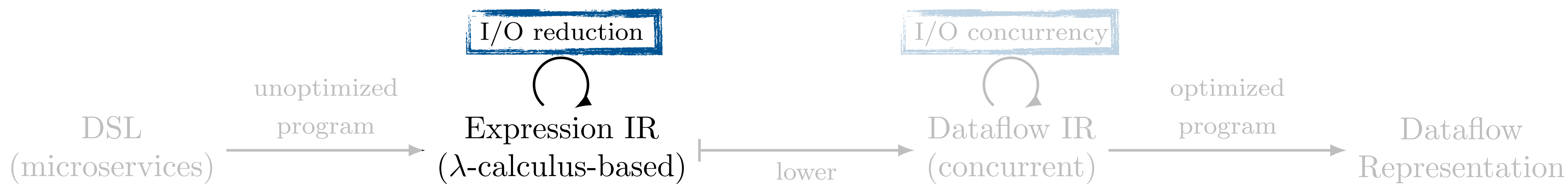
Let-floating

$$
\begin{aligned}
\texttt{smap}(\texttt{f}, [a_1, a_2]) = & \texttt{ let } (s'_g, x_1) = \texttt{g}(s_g, a_1) \texttt{ in} & = \quad & \texttt{let } (s'_g, x_1) = \texttt{g}(s_g, a_1) \texttt{ in} \\
& \texttt{ let } (s''_g, x_2) = \texttt{g}(s'_g, a_2) \texttt{ in} & & \texttt{let } (s'_h, b_1) = \texttt{h}(s_h, x_1) \texttt{ in} \\
& \texttt{ let } (s'_h, b_1) = \texttt{h}(s_h, x_1) \texttt{ in} & & \texttt{let } (s''_g, x_2) = \texttt{g}(s'_g, a_2) \texttt{ in} \\
& \texttt{ let } (s''_h, b_2) = \texttt{h}(s'_h, x_2) \texttt{ in} & & \texttt{let } (s''_h, b_2) = \texttt{h}(s'_h, x_2) \texttt{ in} \\
& \quad [b_1, b_2] & & \quad [b_1, b_2]
\end{aligned}
$$

pipeline parallel

Peyton Jones S, Partain W, Santos A. Let-floating: moving bindings to give faster programs. In ACM SIGPLAN Notices 1996. ACM.

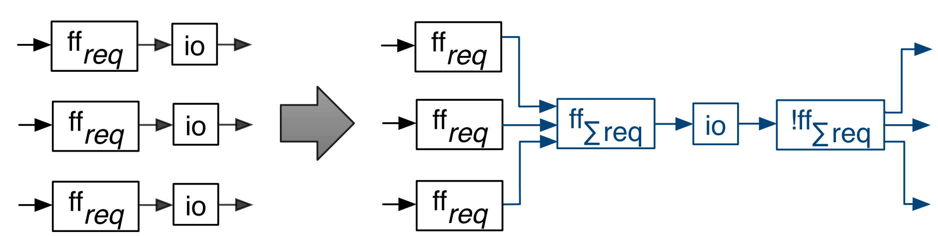
I/O Transformations



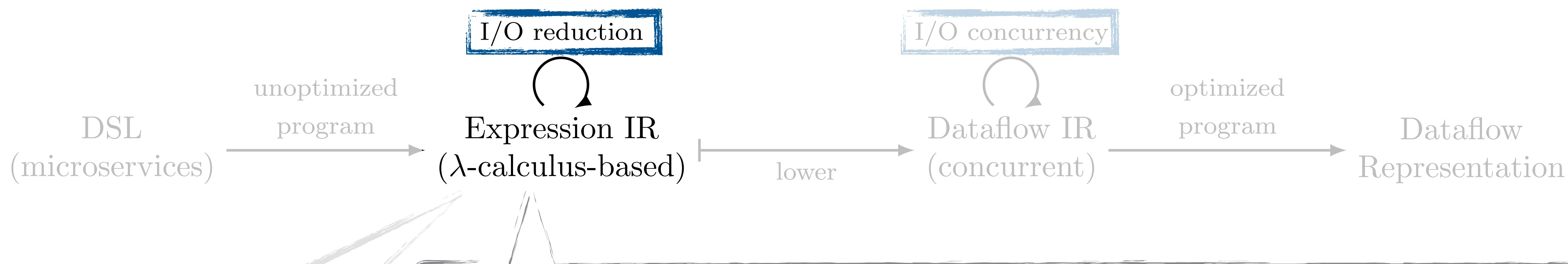
$$
\begin{aligned}
t ::=\ & v \\
|\ & \lambda v.t \\
|\ & t\ t \\
|\ & \text{let } v{=}t \text{ in } t \\
|\ & \text{if } (t\ t\ t) \\
|\ & ff_f(x_1 \ldots x_n) \\
|\ & io(x)
\end{aligned}
$$

$map(\lambda v.t\ [v_1 \ldots v_n])$

- Grouping independent I/O calls:



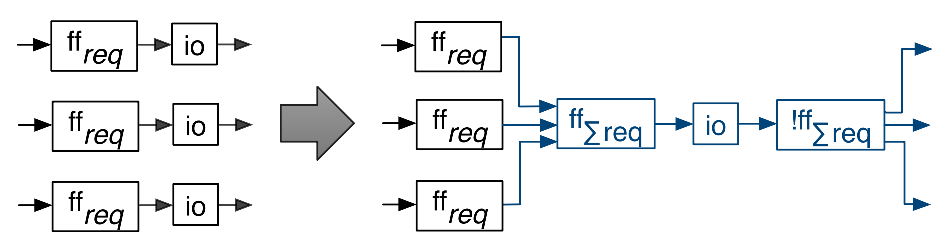
I/O Transformations



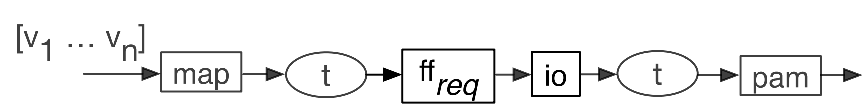
t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

map(λv.t $[v_1 \dots v_n]$)

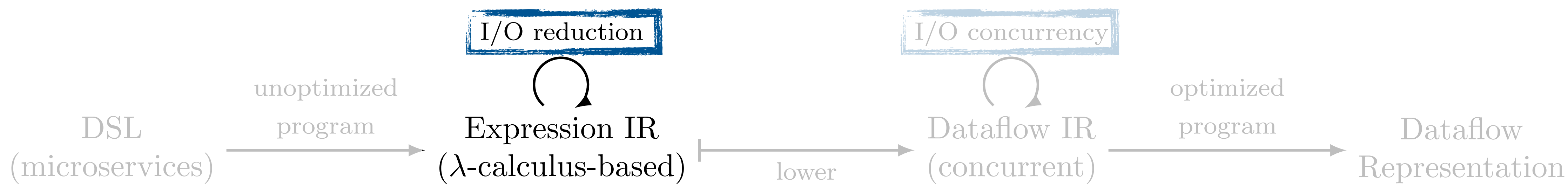
- Grouping independent I/O calls:



- Lifting I/O calls:



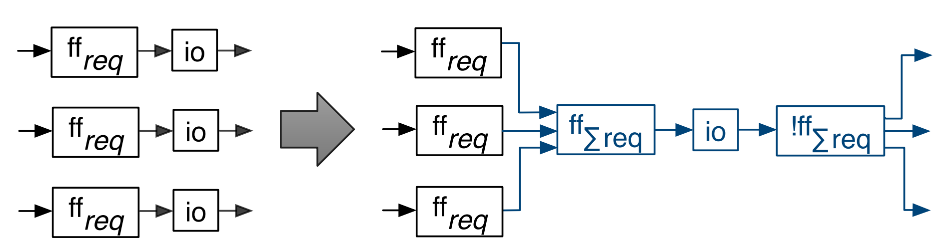
I/O Transformations



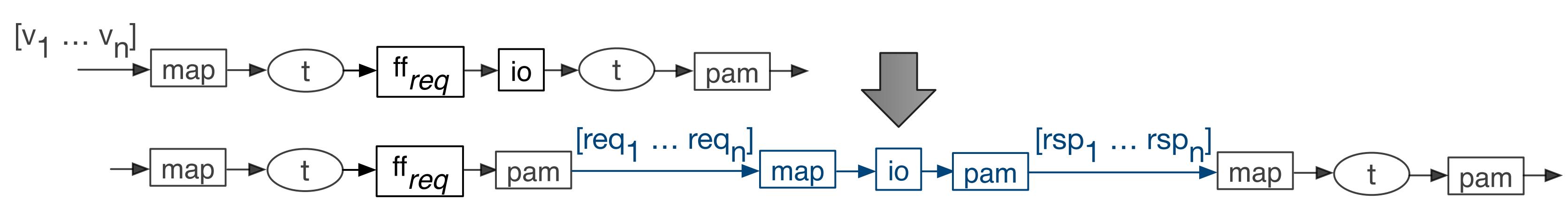
$$
\begin{aligned}
t ::=\ & v \\
 |\ & \lambda v.t \\
 |\ & t\ t \\
 |\ & \text{let } v = t \text{ in } t \\
 |\ & \text{if } (t\ t\ t) \\
 |\ & ff_f(x_1 \ldots x_n) \\
 |\ & io(x)
\end{aligned}
$$

$map(\lambda v.t\ [v_1 \ldots v_n])$

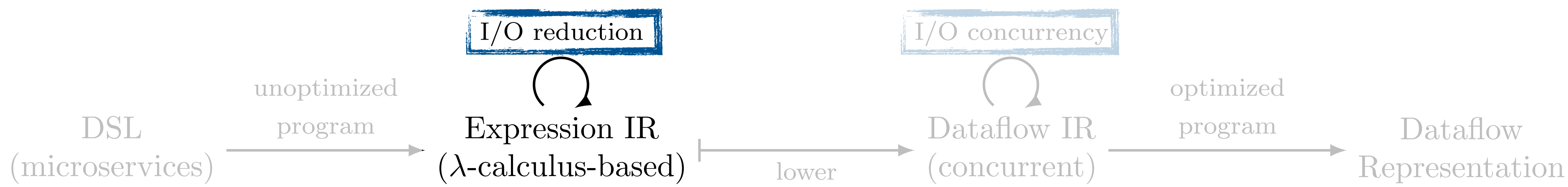
- Grouping independent I/O calls:



- Lifting I/O calls:



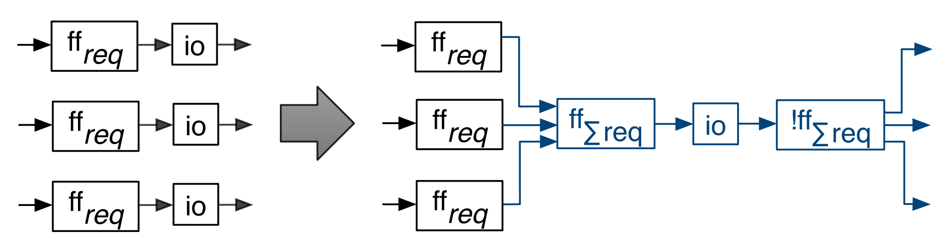
I/O Transformations



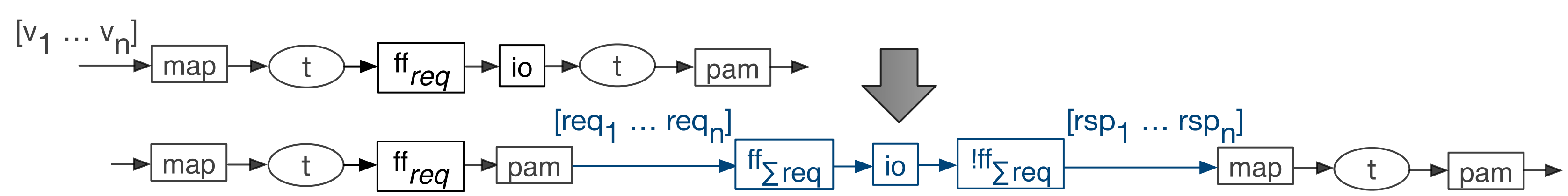
$$
\begin{aligned}
t ::=\ & v \\
\mid\ & \lambda v.t \\
\mid\ & t\ t \\
\mid\ & \text{let } v=t \text{ in } t \\
\mid\ & \text{if } (t\ t\ t) \\
\mid\ & ff_f(x_1 \ldots x_n) \\
\mid\ & io(x)
\end{aligned}
$$

$map(\lambda v.t\ [v_1 \ldots v_n])$

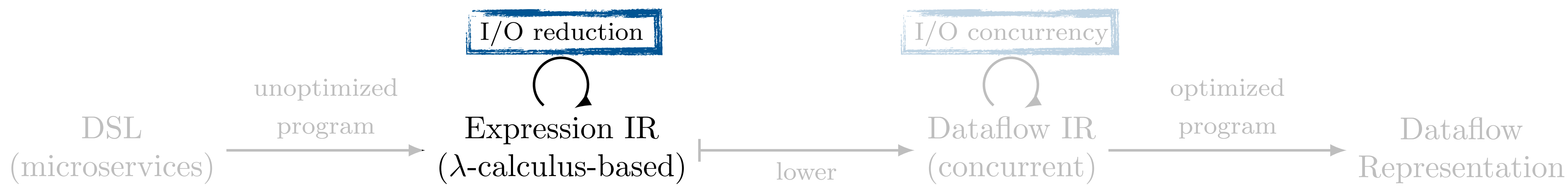
- Grouping <u>independent</u> I/O calls:



- Lifting I/O calls:



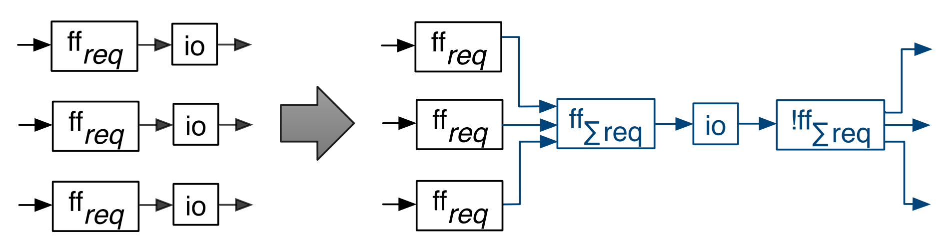
I/O Transformations



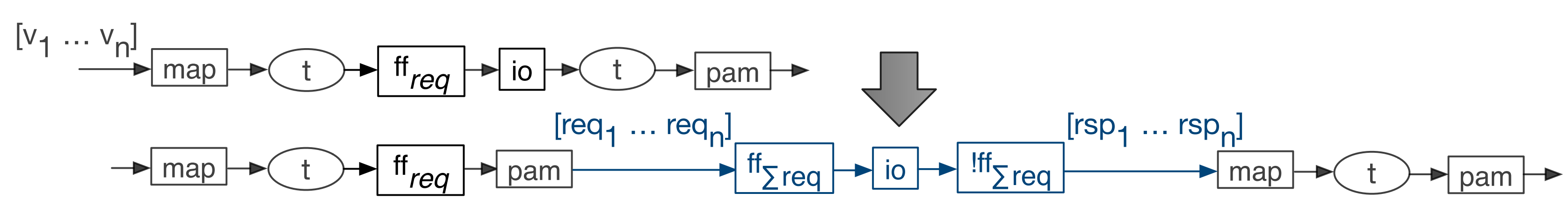
t ::= v
| $\lambda v.t$
| $t\ t$
| let v=t in t
| if ($t\ t\ t$)
| $ff_f(x_1 \dots x_n)$
| io(x)

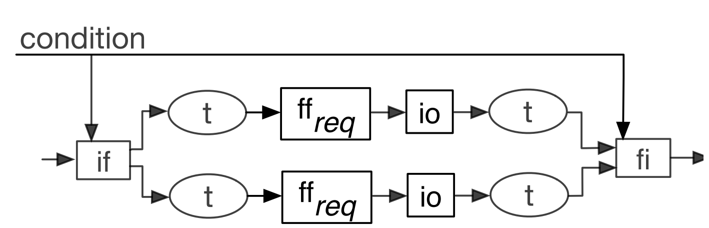
map($\lambda v.t$ [$v_1 \dots v_n$])

- Grouping independent I/O calls:

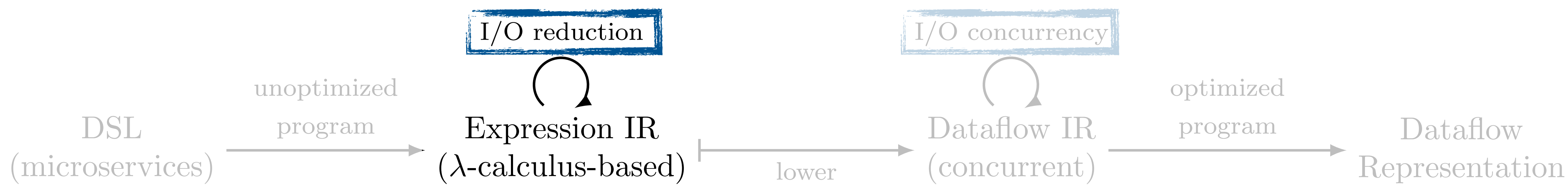


- Lifting I/O calls:





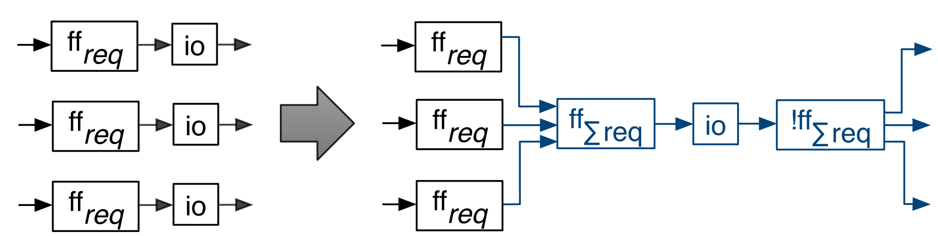
I/O Transformations



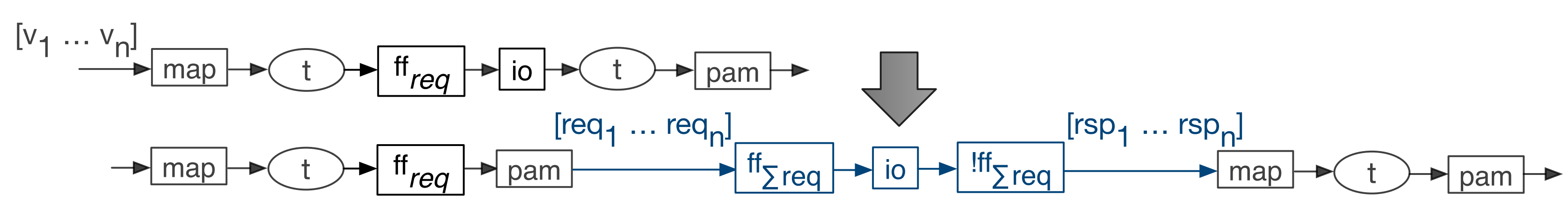
$$
\begin{aligned}
t ::=\ & v \\
\mid\ & \lambda v.t \\
\mid\ & t\ t \\
\mid\ & \text{let } v{=}t \text{ in } t \\
\mid\ & \text{if } (t\ t\ t) \\
\mid\ & ff_f(x_1 \ldots x_n) \\
\mid\ & io(x)
\end{aligned}
$$

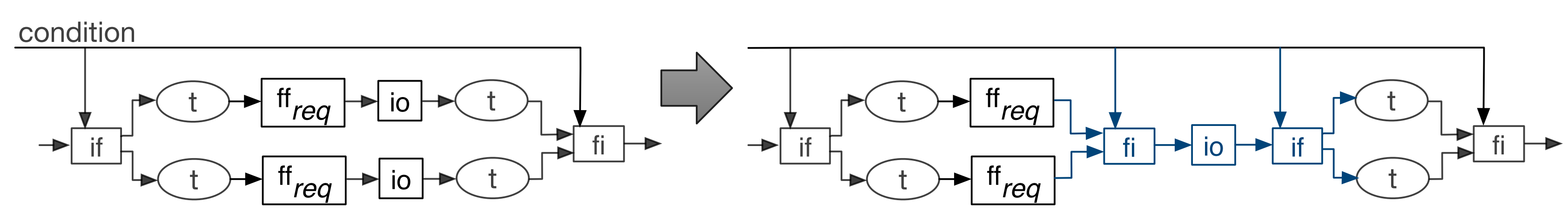
$map(\lambda v.t\ [v_1 \ldots v_n])$

- Grouping independent I/O calls:



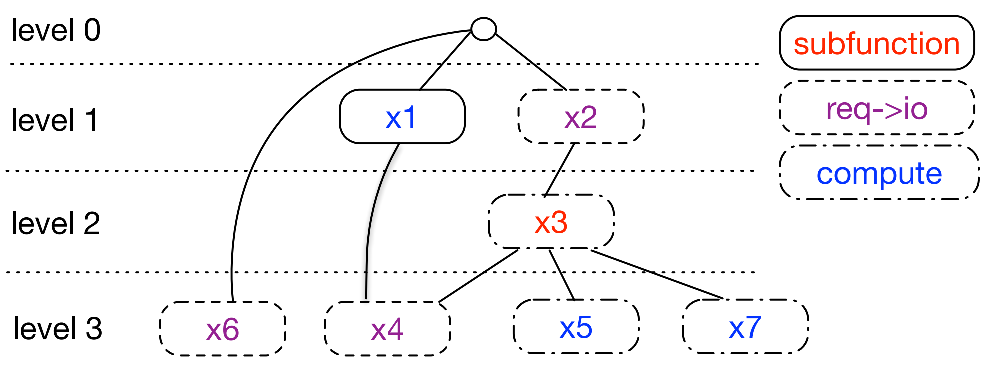
- Lifting I/O calls:





Evaluation

Level-Graphs:

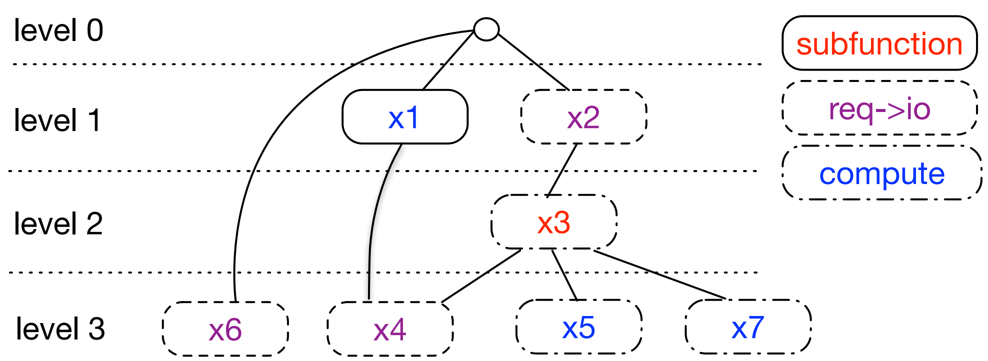


Programs:

Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers. MULTIPROG'18.

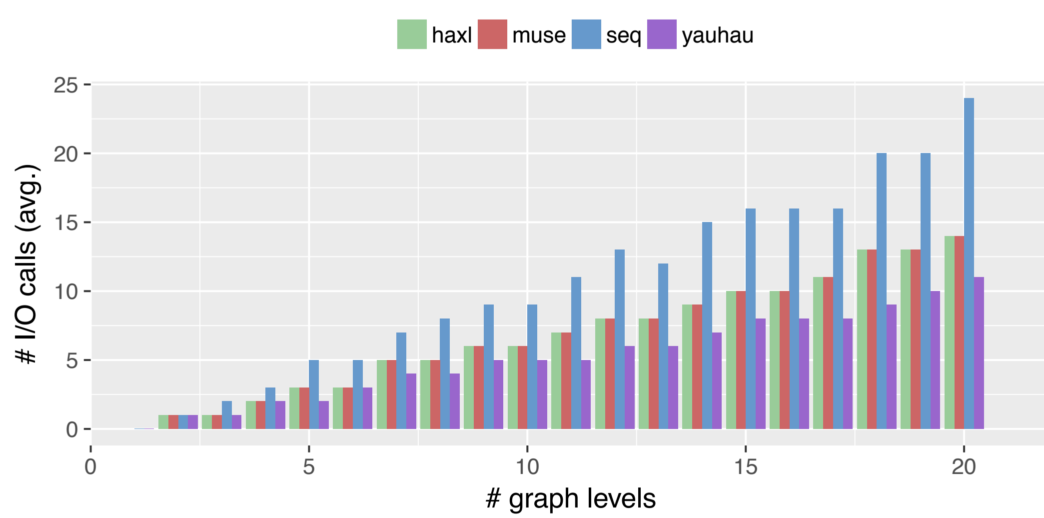
Evaluation

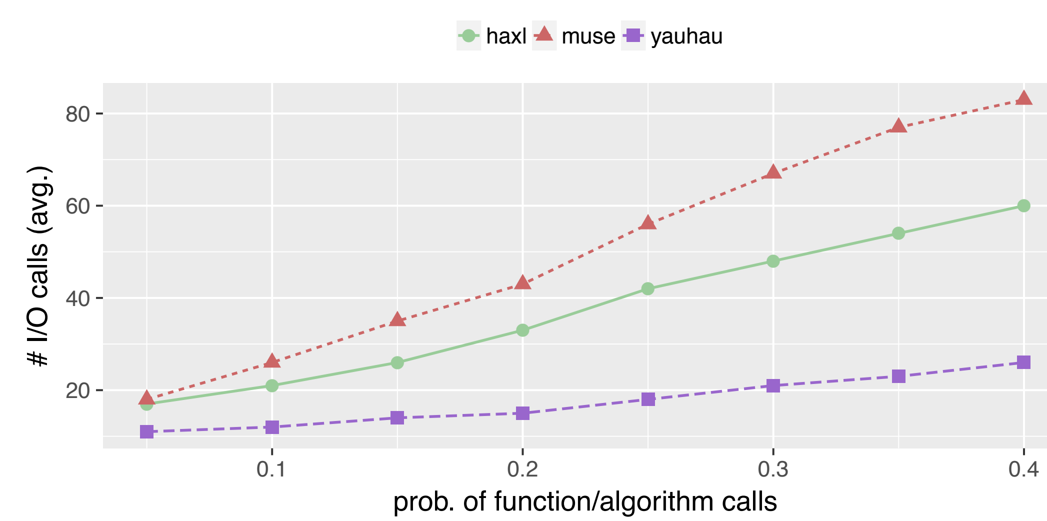
Level-Graphs:



Programs:

- Haxl
- Muse
- Seq
- Ÿauhau





Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. ICFP '14.

Alexey Kachayev. 2015. Reinventing Haxl: Efficient, Concurrent and Concise Data Access. Presentation at EuroClojure 2015.

Programming Model/ Language

SFP

Compiler

Runtime System

Dataflow

Lambda calculus:
- Variables
- Abstraction
- Application

Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for concise code and efficient I/O. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). ACM.

Programming Model/ Language

SFP

Compiler

Runtime System

Dataflow

- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Programming Model/ Language

Compiler

~~Runtime System~~ Dataflow

SFP:
- Variables
- Stateful Functions
- Composition
- Algorithms
- Conditionals
- SMap

Lambda calculus:
- Variables
- Abstraction
- Application
- Combinators

Stateful Functions

?

```java
class FileLoad extends Filter {

  private Channel<String> in, out;

  public void work() {
    // explicit channel control
    String resource = in.take();
    String contents = load(resource);
    out.put(contents);
  }

  private Map<String, String> cache = new HashMap<>();

  private String load(String resource){
    String content = null;
    if(!cache.containsKey(resource)){
      content = new String(
          Files.readAllBytes(Paths.get(resource))
        );
      cache.put(resource,content);
      // cache eviction emitted for brevity
    } else {
      content = cache.get(resource);
    }
    return content;
  }

}
```

State encapsulation

```java
class FileLoad {
  Map<String, String> cache = new HashMap<>();

  String load(String resource) {
    String contents = null;
    // load file data from disk or cache (omitted)
    return contents;
  }
}
```

```rust
struct FileLoad {
    cache : HashMap,
};

impl FileLoad {
  fn load(&self, resource:String) -> String {
    let contents : String = {
    // load file data from disk or cache (omitted)
    };
    contents
  }
}
```

```c
char* load(char* resource) {
  static GHashTable* cache = g_hash_table_new();
  char* contents = NULL;
  // load file data from disk or cache (omitted)
  return contents;
}
```

Foundations of Stateful Functions

Foundations of Stateful Functions

State Threads:

```
load :: String -> State (HashMap String String)
                        String
load :: String -> ST (MHashMap s String String)
                     String
```

Foundations of Stateful Functions

State Threads:

```
load :: String -> State (HashMap String String)
                        String
load :: String -> ST (MHashMap s String String)
                     String
```

$$f_{st} = (a, s) \to (b, s)$$

Composition:

$$f = a \to b$$
$$g = b \to c$$
$$g \circ f = a \to c$$

$$f_{st} = (a, s_f) \to (b, s_f)$$
$$g_{st} = (a, s_g) \to (c, s_g)$$
$$\boxed{g_{st} \circ f_{st} = ?}$$

Foundations of Stateful Functions

State Threads:

```
load :: String -> State (HashMap String String)
                          String
load :: String -> ST (MHashMap s String String)
                      String
```

$$f_{st} = (a, s) \to (b, s)$$

Composition:

$$f = a \to b$$
$$g = b \to c$$
$$g \circ f = a \to c$$

$$f_{st} = (a, s_f) \to (b, s_f)$$
$$g_{st} = (a, s_g) \to (c, s_g)$$
$$\boxed{g_{st} \circ f_{st} = ?}$$

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of "STCLang: state thread composition as a foundation for monadic dataflow parallelism". arXiv preprint arXiv:1906.12098

Foundations of Stateful Functions

State Threads:

```
load :: String -> State (HashMap String String)
                  String
load :: String -> ST (MHashMap s String String)
                String
```

$$f_{st} = (a, s) \rightarrow (b, s)$$

Composition:

$$f = a \rightarrow b$$
$$g = b \rightarrow c$$
$$g \circ f = a \rightarrow c$$

$$f_{st} = (a, s_f) \rightarrow (b, s_f)$$
$$g_{st} = (a, s_g) \rightarrow (c, s_g)$$
$$\boxed{g_{st} \circ f_{st} = ?}$$

SMap:

- Formal analysis on determinism and deadlocks.
- Not only **map**s but also **fold**s!
- Not only pipeline parallelism but also task-level and data parallelism!

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of "STCLang: state thread composition as a foundation for monadic dataflow parallelism". arXiv preprint arXiv:1906.12098

Foundations of Stateful Functions

State Threads:

```
load :: String -> State (HashMap String String)
                    String
load :: String -> ST (MHashMap s String String)
                  String
```

$$f_{st} = (a, s) \to (b, s)$$

Composition:

$$f = a \to b$$
$$g = b \to c$$
$$g \circ f = a \to c$$

$$f_{st} = (a, s_f) \to (b, s_f)$$
$$g_{st} = (a, s_g) \to (c, s_g)$$
$$\boxed{g_{st} \circ f_{st} = ?}$$

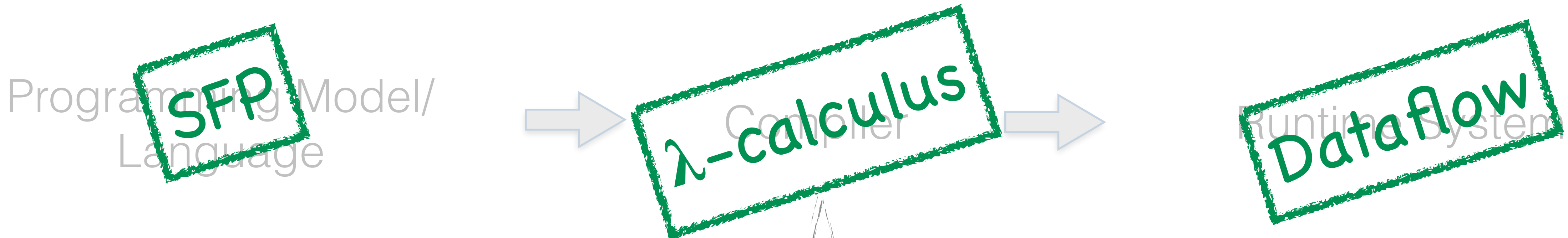
SMap:

- Formal analysis on determinism and deadlocks.
- Not only **map**s but also **fold**s!
- Not only pipeline parallelism but also task-level and data parallelism!

```
fn map_reduce() {
  let stateG = init_G();
  for data in list {
    let x = f(data);
    stateG.g(x)
  };
  stateG
}
```

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: state thread composition as a foundation for monadic dataflow parallelism. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). ACM.

Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of "STCLang: state thread composition as a foundation for monadic dataflow parallelism". arXiv preprint arXiv:1906.12098



- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Ingredients for Implicit Parallel Programming

Programming Model/ Language — SFP → Compiler — λ-calculus → Runtime System — Dataflow

SFP

- No concurrency abstractions.
- Practicality:
  - Integration into existing programming models (e.g., OOP)/ languages.
  - Gradual switch/ reuse of existing code base.

λ-calculus

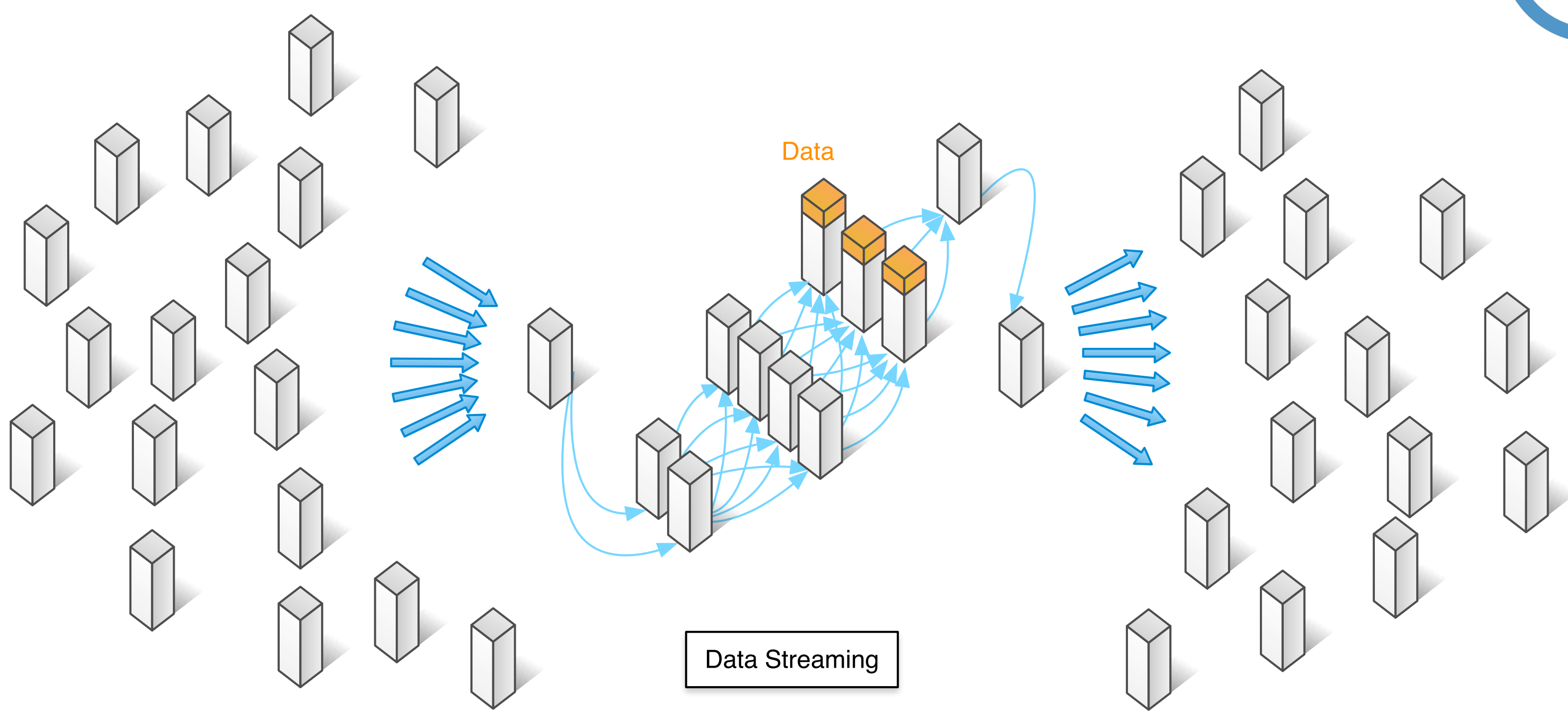
- Intermediate Representation (IR) needs a solid well-studied theoretical foundation for
- provable semantic-preserving transformations
- even for programs with side-effects.

Dataflow

- Flexible/dynamic runtime representation.
- Efficient parallel execution of independent computations.
- Deterministic execution.
- Concurrent I/O.

Changing the Game

- Functional Reactive Programming (FRP) based on stateful functions.
- FRP instead of streams for big data systems.
  https://github.com/ohua-dev/stc-lang



- Current Main Focus: Rust integration ➪ Safety
  https://github.com/ohua-dev/ohua-rust-runtime
- STM alternative.
- User-defined functions and NoSQL (in Noria).

Thanks for your attention!

https://ohua-dev.github.io/