C++ now

Reintroduction to Generic Programming

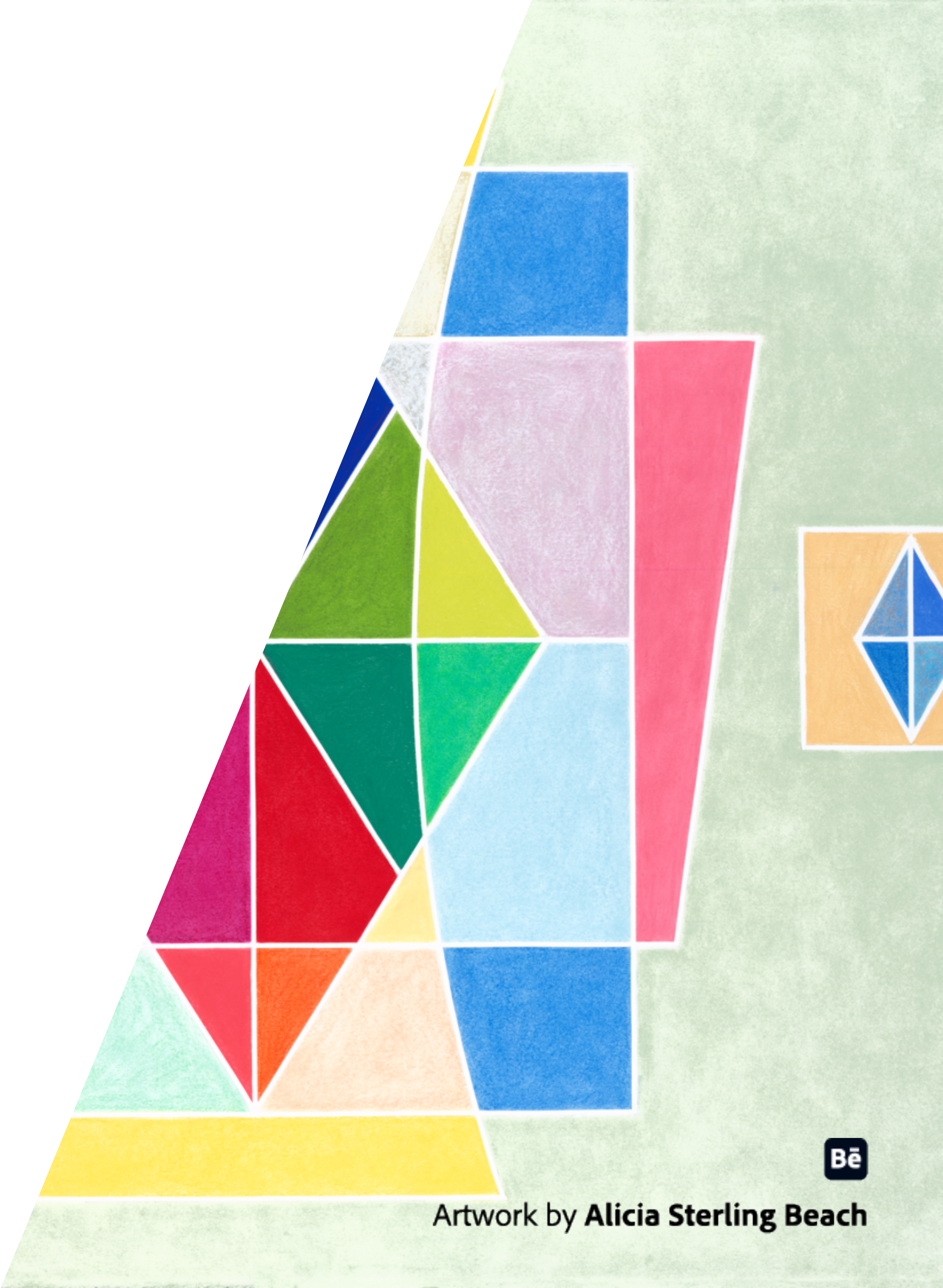
Nick DeMarco

2024

Adobe

Reintroduction to Generic Programming

Nick DeMarco | Senior Computer Scientist
Software Technology Lab



Artwork by **Alicia Sterling Beach**

All code is a liability.

All code is a liability.
Let's write less.

Make it generic!

```
/// Arranges the elements of `c` in non-decreasing order.
function sort(c: number[]) { /*...*/ }
```

Make it generic!

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }
```

The spectrum of generality

```
sort(c: number[])
```

One type is valid

```
sort<C>(c: C)
```

Any type is valid

Constrained **Generic**

The spectrum of generality

`sort(c: number[])`

One type is valid

`sort<T>(c: T[])`

Any built-in array is valid

`sort<C>(c: C)`

Any type is valid

Constrained

Generic

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }
```

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
```

TS

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray);
```

The spectrum of generality | That's too constrained!

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T>(c: T[]) { /*...*/ }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray);
```

```
Argument of type 'Int32Array' is not assignable to parameter of
type 'T[]'.

Type 'Int32Array' is missing the following properties from type
'T[]': pop, push, concat, shift, and 3 more
```

The spectrum of generality

```
sort(c: number[])
```
One type is valid

```
sort<T>(c: T[])
```
Any built-in array is valid

```
sort<C>(c: C)
```
Any type is valid

Constrained

Generic

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) { /*...*/ }
```

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
  // Bubble sort implementation to demonstrate.
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

The spectrum of generality | That's too generic!

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
  // Bubble sort implementation to demonstrate.
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

```
Property 'length' does not exist on type 'C'.
```

```
No index signature with a parameter
of type 'number' was found on type 'C'.
```

TS

The spectrum of generality

sort(c: number[])

One type is valid

sort<T>(c: T[])

Any built-in array is valid

sort<C>(c: C)

Any type is valid

Constrained

Generic

*Any type **with the required functionality** is valid*

The spectrum of generality

sort(c: number[])
One type is valid

sort<T>(c: T[])
Any built-in array is valid

sort<C>(c: C)
Any type is valid

Constrained

Generic

sort<???>(c: ???)
Any type ***with the required functionality*** *is valid*

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<???>(c: ???) { ... }
```

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) { ... }
```

TS

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C extends {}>(c: C) { ... }
```

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C extends {length: number}>(c: C) { ... }
```

TS

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C extends {length: number, [n: number]: ???}>(c: C) { ... }
```

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {length: number, [n: number]: T}>(c: C) { ... }
```

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {length: number, [n: number]: T}>(c: C) {
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) {
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

TS

The spectrum of generality | Precise constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) {
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

TS

The spectrum of generality | That's just right!

```typescript
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) { ... }

let numArray = new Int32Array([3, 2, 1]);
sort(numArray); // [1, 2, 3] ✅

let stringArray = ["c", "d", "a"];
sort(stringArray); // ["a", "c", "d"] ✅

let unsortable: Record<string, string> = { first: "Jane", last: "Doe" };
sort(unsortable); // Error: Property 'length' is missing... ✅
```

TS

The spectrum of generality

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}>(c: C) { ... }
```

*Any type **with the required functionality** is valid*

Key takeaway 1/8:

Maximize reusability with minimally* complete constraints

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> ...
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> ...


/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends {readonly length: number, [n: number]: T}> ...
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends {readonly length: number, [n: number]: T}> ...

/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends {readonly length: number, [n: number]: T}> ...

/// Partitions `c` around `nth`, leaving the value of `nth` as if `c` were entirely sorted.
function nthElement<T, C extends {readonly length: number, [n: number]: T}> ...
```

TS

Naming our constraints

```
interface ???<T> = {
  readonly length: number,
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends ??? > ...
```

Naming our constraints

```
interface ???<T> = {
  readonly length: number,
  [n: number]: T
};
```

MDN web docs

moz://a

Array.prototype.sort()

The sort() method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.

Naming our constraints

```
interface ???<T> = {
  readonly length: number,
  [n: number]: T
};
```

MDN web docs
moz://a

Array.prototype.sort()

The sort() method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.

Naming our constraints

```
interface ???<T> = {
  readonly length: number,
  [n: number]: T
};
```

1a421a7 TypeScript / src / lib / es5.d.ts

Code Blame 4545 lines (4005 loc) · 213 KB

```
interface ArrayLike<T> {
    readonly length: number;
    readonly [n: number]: T;
}
```

MDN web docs

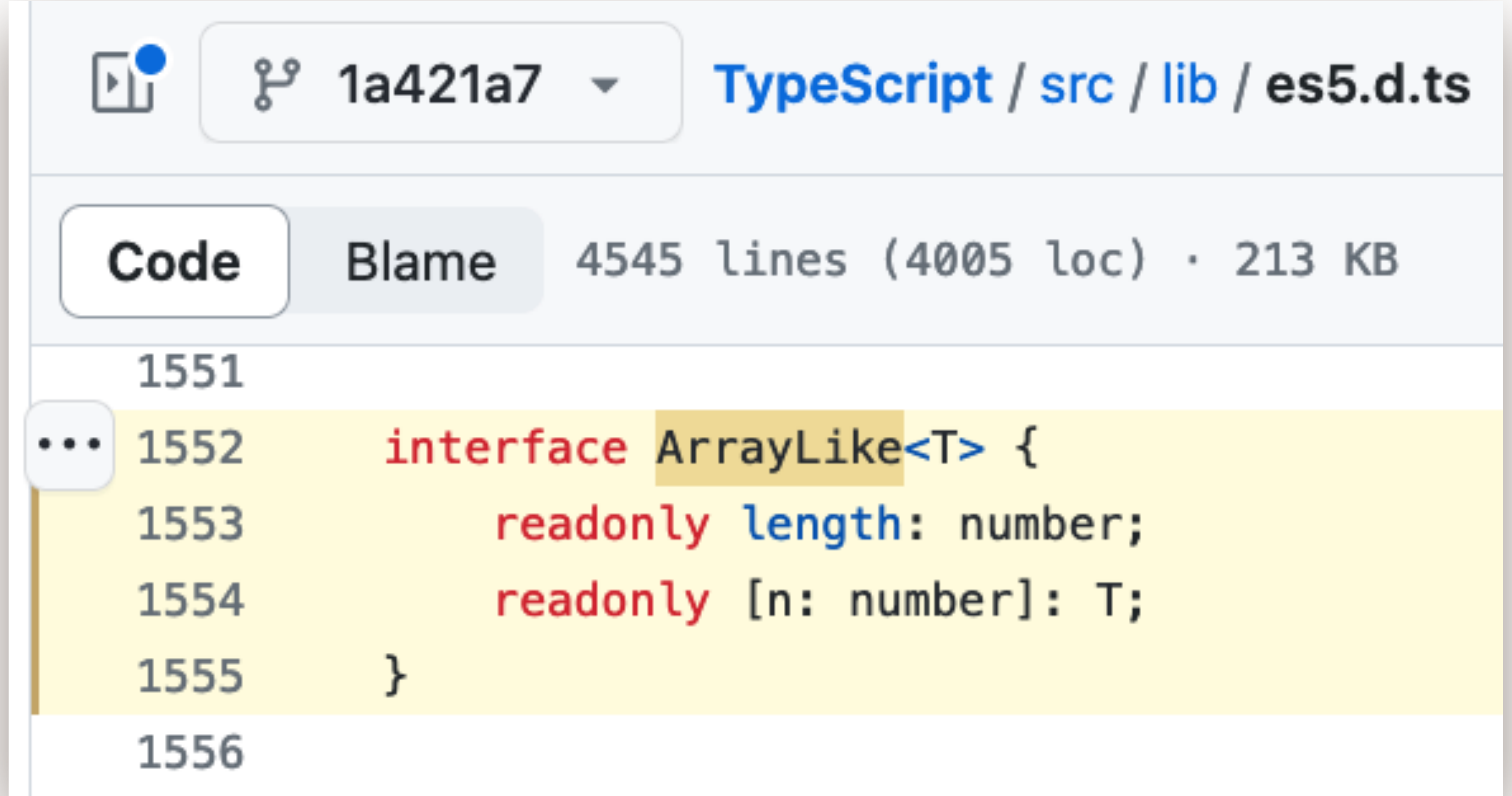
moz://a

Array.prototype.sort()

The sort() method is generic. It only expects the this value to have a length property and integer-keyed properties.

Although strings are also array-like, this method is not suitable to be applied on them, as strings are immutable.

Naming our constraints

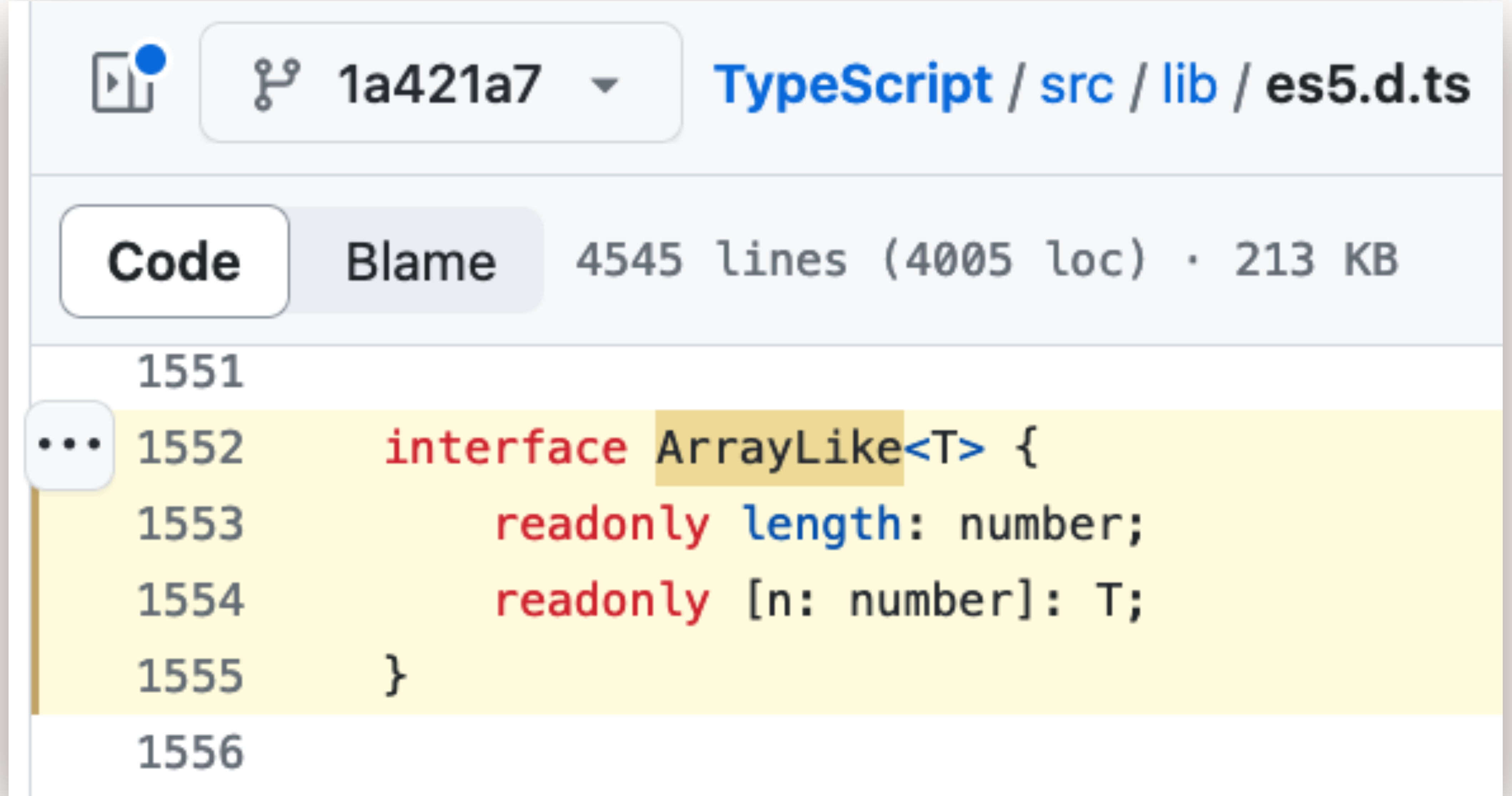



```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Documenting semantics

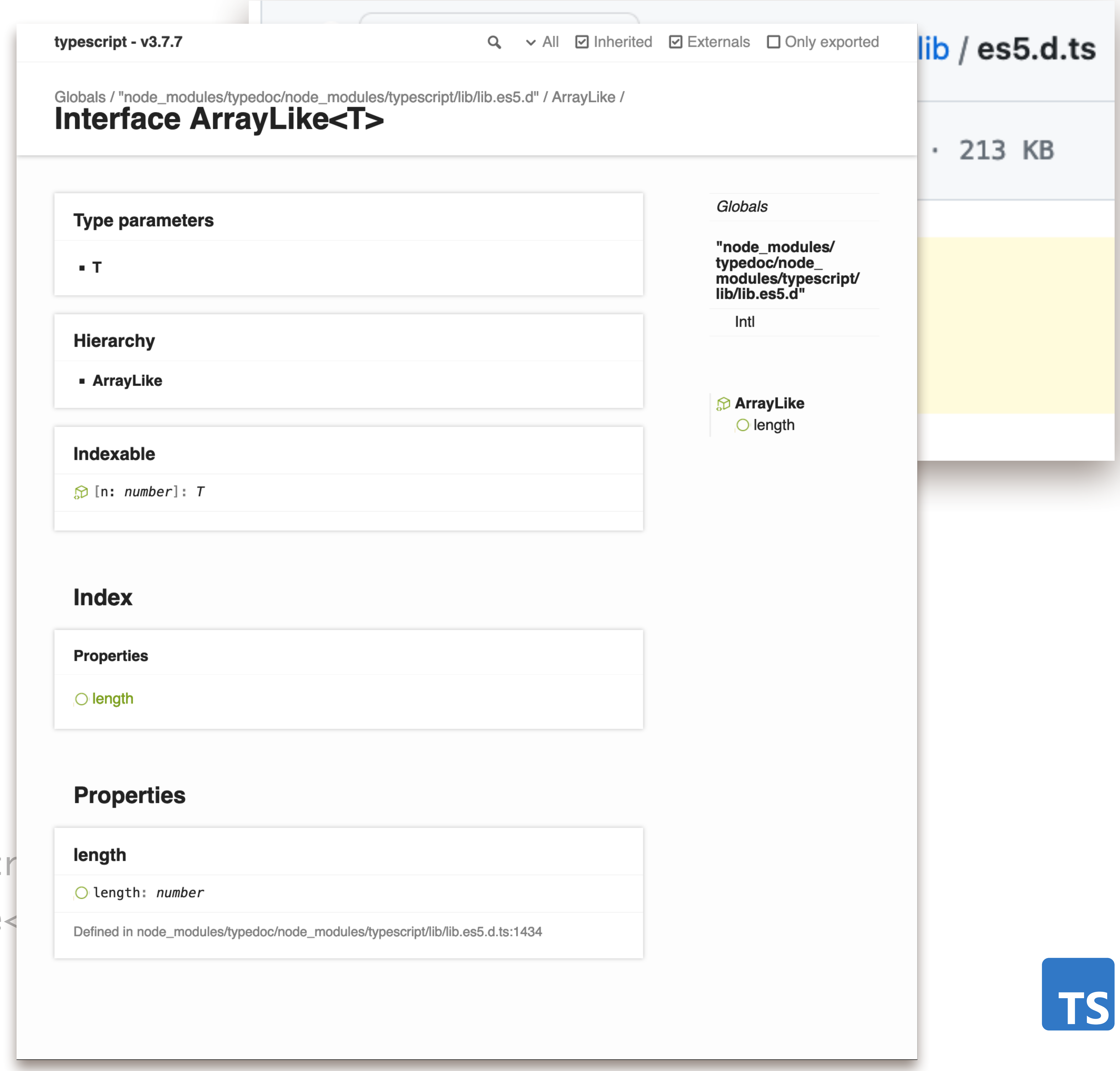
```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```




```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Documenting semantics

```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

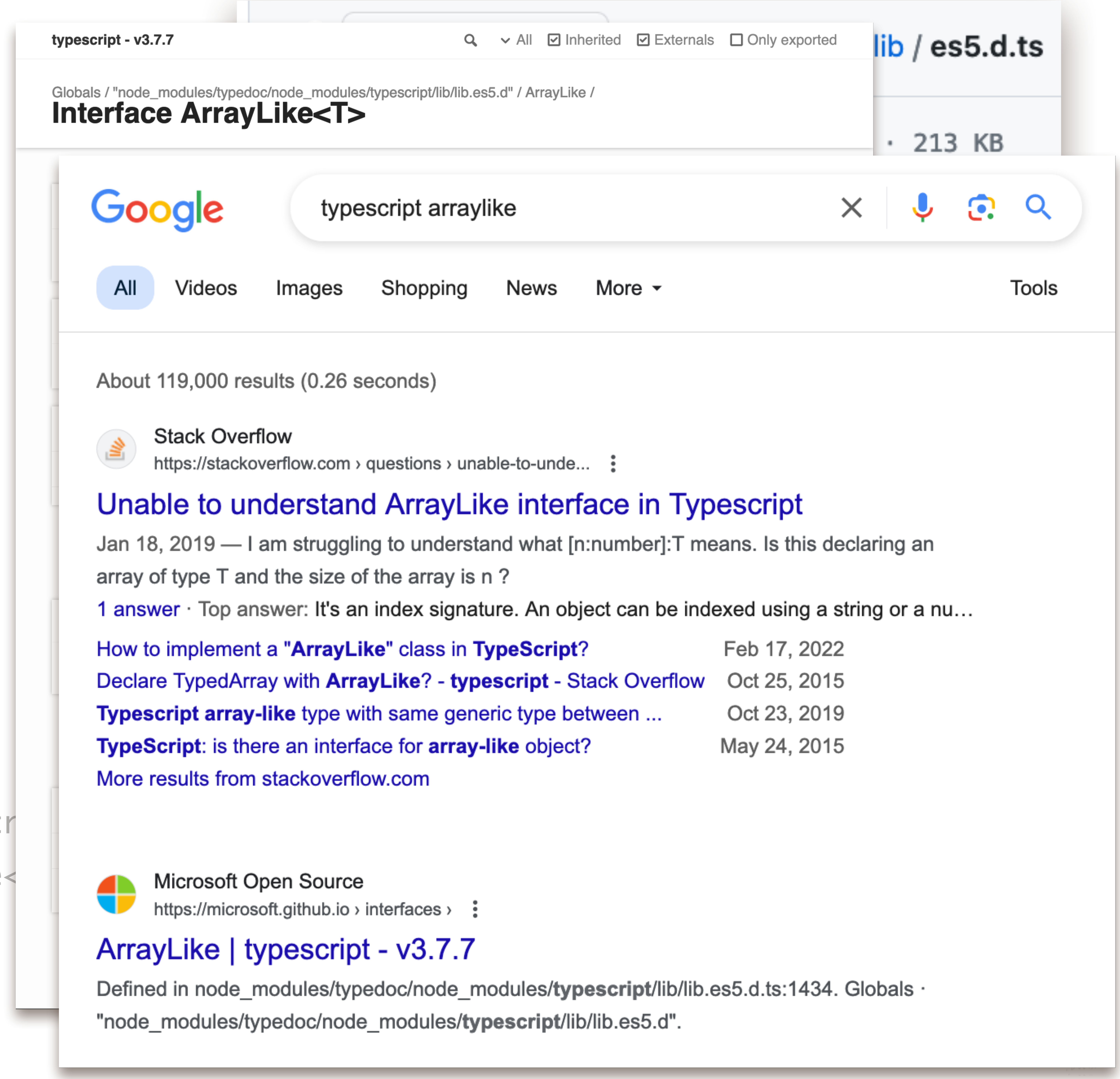



```
/// Arranges the elements of `c` in non-decr
function sort<T, C extends MutableArrayLike<
```

Documenting semantics

```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

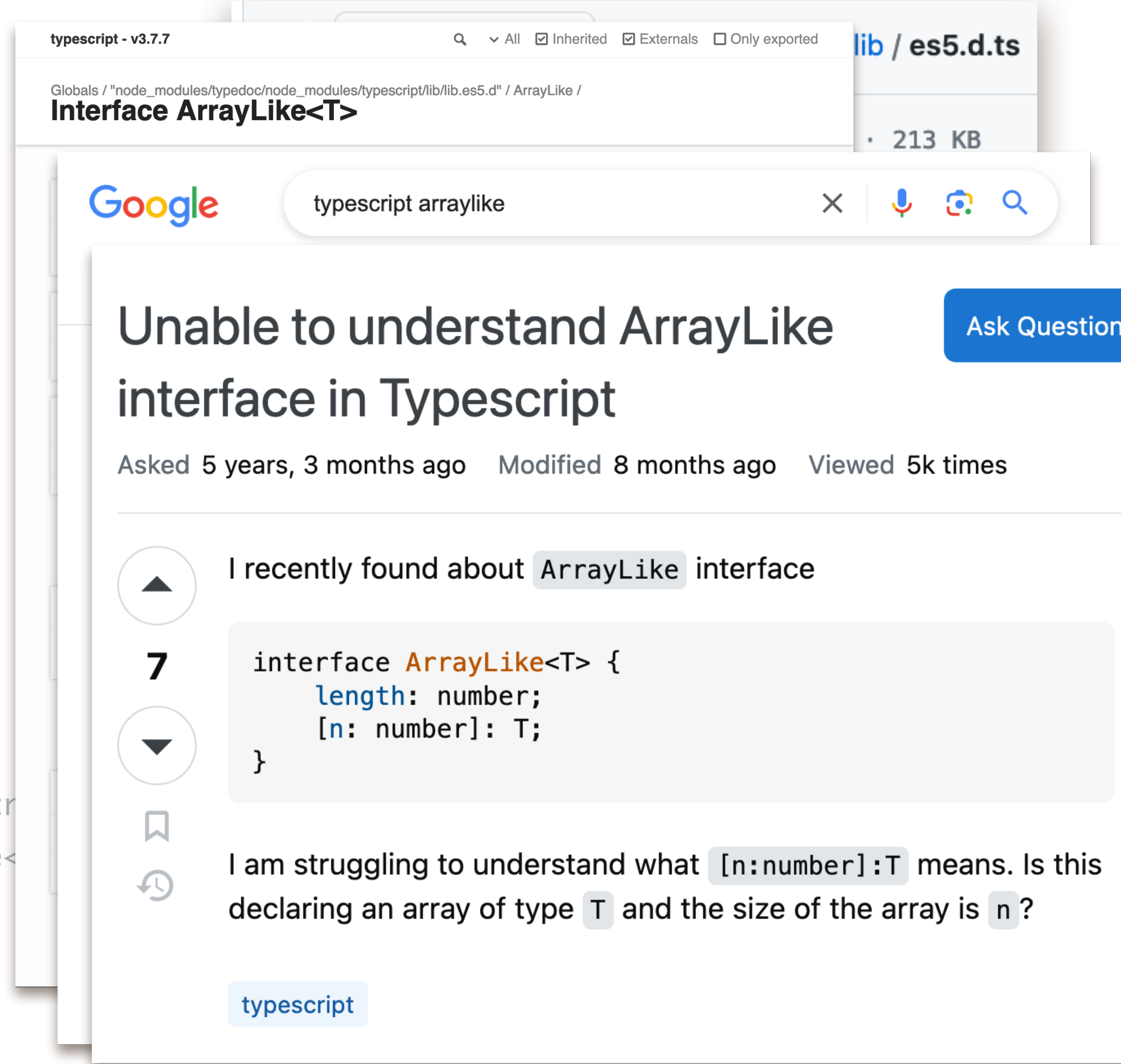
```
/// Arranges the elements of `c` in non-decr
function sort<T, C extends MutableArrayLike<
```



Documenting semantics

```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

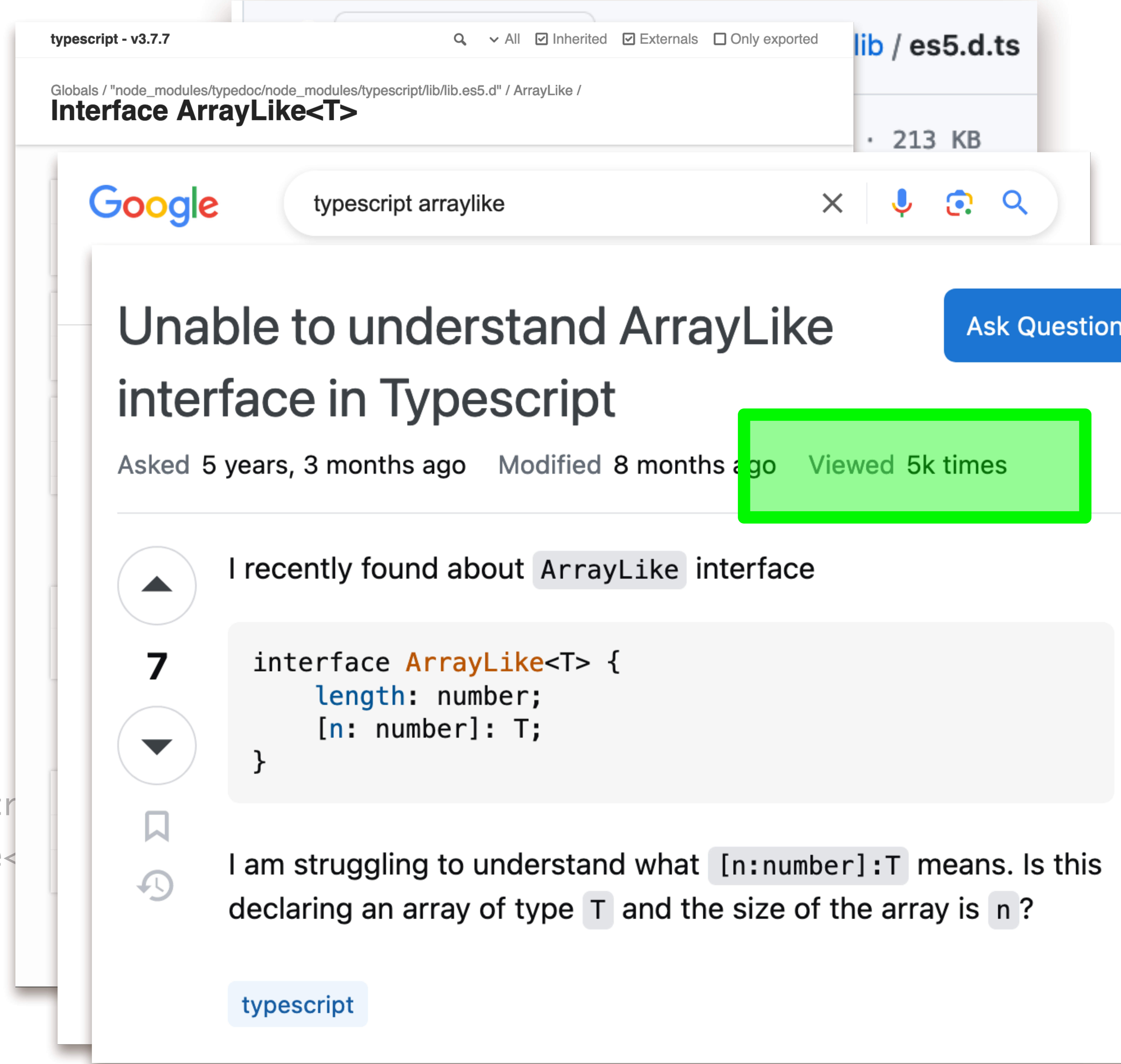
```
/// Arranges the elements of `c` in non-decr
function sort<T, C extends MutableArrayLike<
```



Documenting semantics

```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decr
function sort<T, C extends MutableArrayLike<
```



Documenting semantics

```
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Documenting semantics

```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  readonly length: number,
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Documenting semantics

```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Documenting semantics

```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
```

Key takeaway 2/8:

Extract common constraints and document their semantics

What is this thing?

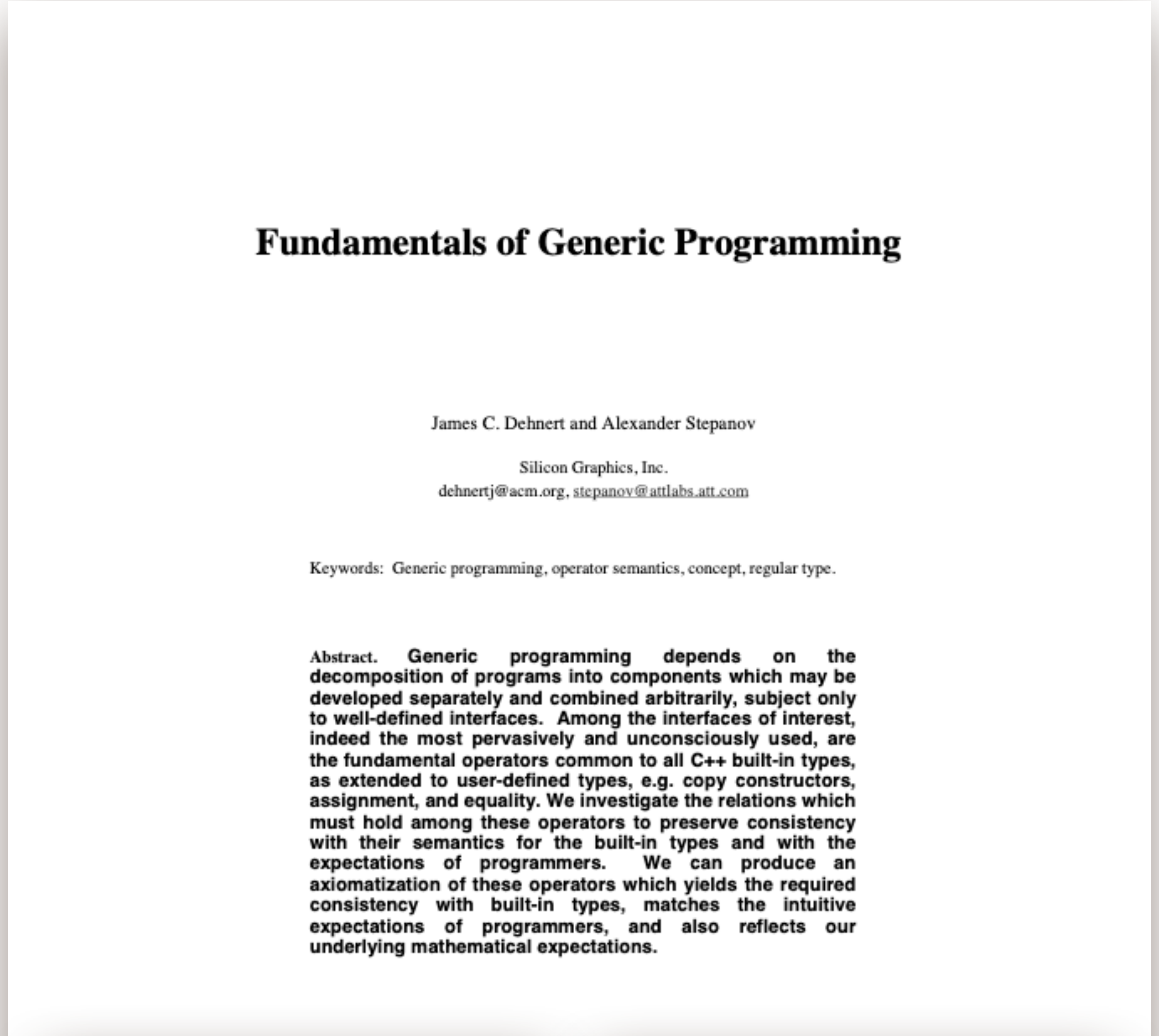
```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
```

Concepts

We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight… is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

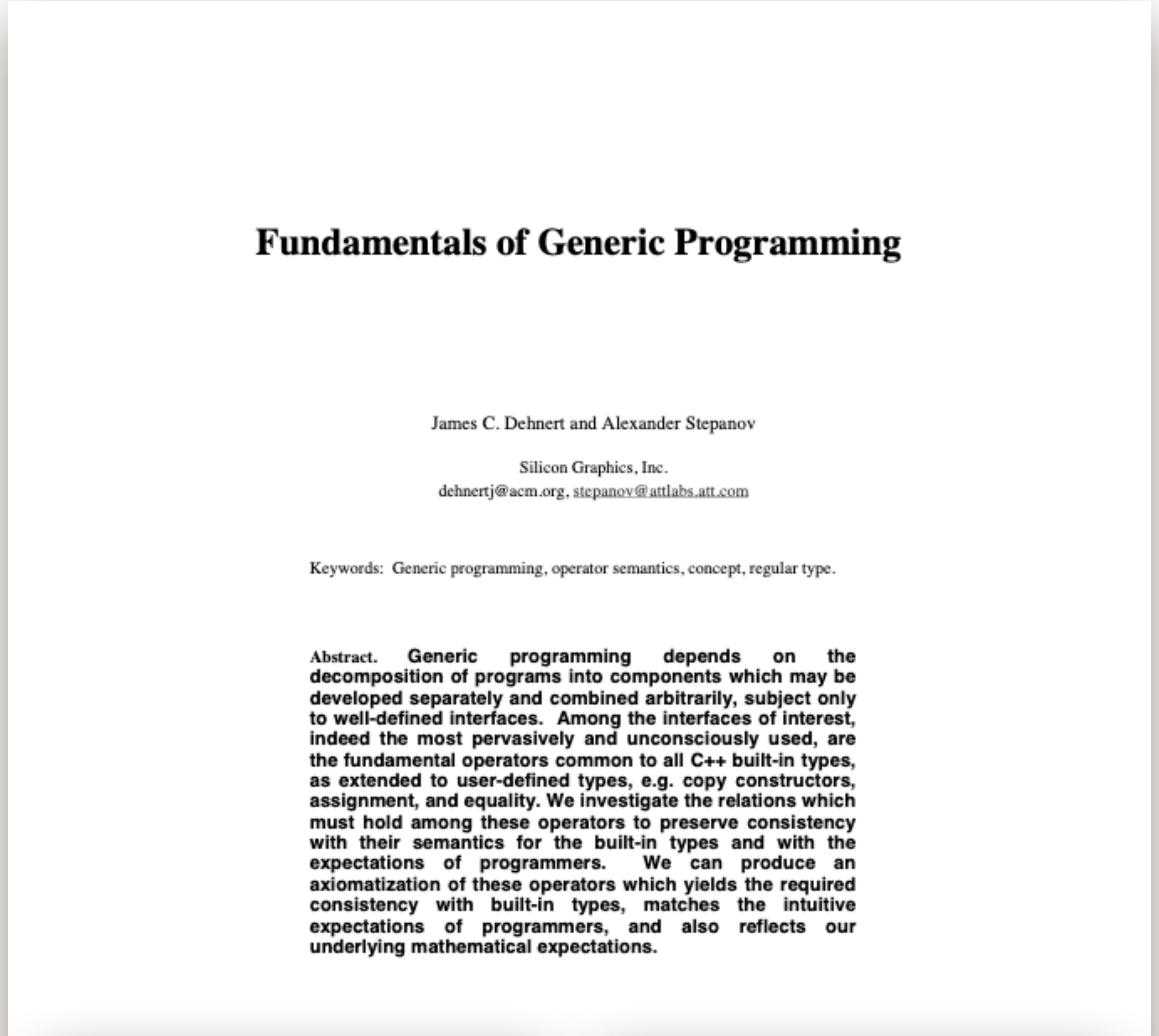
Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Concepts

We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight… is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

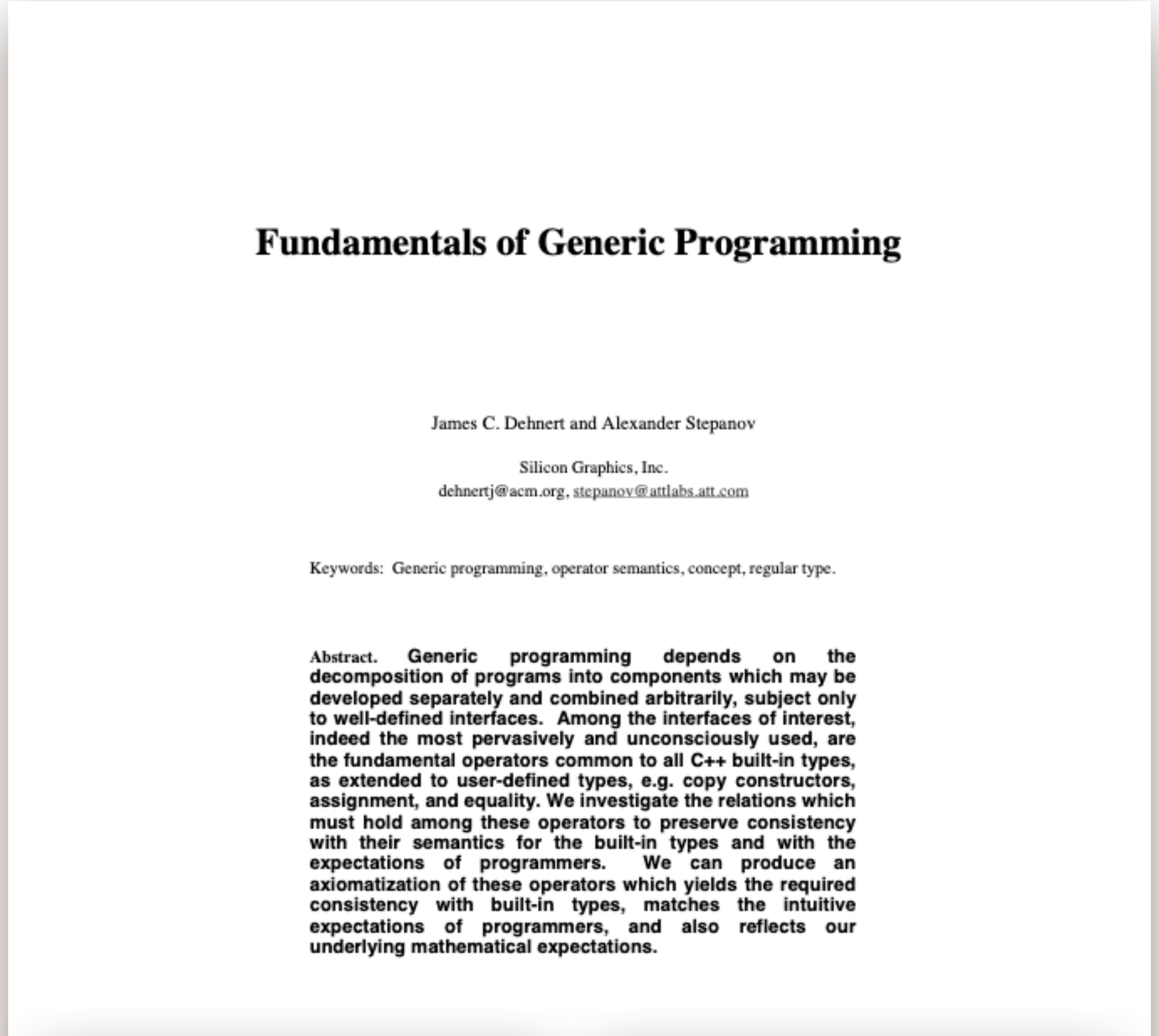
Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Concepts

We call the set of axioms satisfied by a data type and a set of operations on it a concept.

The critical insight… is that highly reusable components must be programmed assuming a minimal collection of such concepts, [which] must match as wide a variety of concrete program structures as possible.

Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components.



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Concepts

```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
```

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<T, C extends MutableArrayLike<T>> ...
/// Arranges the elements in `c` from `first` to `last` in non-decreasing order.
function sortSubrange<T, C extends MutableArrayLike<T>> ...
/// Partitions `c` around `nth`, leaving the value of `nth` as if `c` were entirely sorted.
function nthElement<T, C extends MutableArrayLike<T> ...
```

TS

Concepts | Finding common behavior

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list { ... };
```

Concepts | Finding common behavior

```cpp
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node {
    std::unique_ptr<node> _next = nullptr;
    T _value;
  };
  std::unique_ptr<node> _head = nullptr;
};
```

Concepts | Finding common behavior

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list { ... };
```

```
/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t { ... }
```

Concepts | Finding common behavior

```
/// A sequence of `T` stored non-contiguousl
template <class T>
struct linked_list { ... };
```

```
(T a, T b) {
   a == b; // is legal & meaningful
   a != b; // is legal & meaningful
}
```

```
/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t { ... }
```

Concepts | Finding common behavior

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list { ... };
```

```
/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t {
  std::size_t count{0};
  for (auto i = list._head.get(); i != nullptr; i = i = i->_next.get())
    if (i->_value == t) count++;
  return count;
}
```

Concepts | Finding common behavior

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input { ... };
```

Concepts | Finding common behavior

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input { ... };
```

```
(std::istream& stream, T t) {
  stream >> t; // is legal & meaningful
}
```

Concepts | Finding common behavior

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
  auto read() -> std::optional<T> { ... }
};
```

Concepts | Finding common behavior

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
  auto read() -> std::optional<T> { ... }
};


/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t { ... }
```

Concepts | Finding common behavior

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns one `T` read from std::cin, or `std::nullopt` on failure.
  auto read() -> std::optional<T> { ... }
};


/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t {
  std::size_t count{0};
  for (auto i = in.read(); i != std::nullopt; i = in.read())
    if (*i == t) count++;
  return count;
}
```

Concepts | Finding common behavior

```
/// Returns the frequency of `t` in the linked list.
template <std::equality_comparable T>
auto count(T t, linked_list<T>& list) -> std::size_t {
  std::size_t count{0};
  for (auto i = list._head.get(); i != nullptr; i = i->_next.get())
    if (i->_value == t) count++;
  return count;
}

/// Returns the frequency of `t` in the user input stream.
template <std::equality_comparable T>
auto count(T t, user_input<T>& in) -> std::size_t {
  std::size_t count{0};
  for (auto i = in.read(); i != std::nullopt; i = in.read())
    if (*i == t) count++;
  return count;
}
```

Lifting Concepts

```
// linked_list<T>
for (auto i = list._head.get(); i != nullptr; i = i->_next.get())
  if (i->_value == t) count++;

// user_input<T>
for (auto i = in.read(); i != std::nullopt; i = in.read())
  if (*i == t) count++;

// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) { ... };
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) { ... };
```

```
template <class I>
constexpr bool ??? =
  /* some predicates over I */;
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) { ... };
```

```
(I i) {
  /* ... these operations on i are
         legal & meaningful
  */
}
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) { ... };
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) {
  /// Returns the next valid position or an invalid sentinel for the end.
  { ++i } -> I&;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I>
concept ??? = requires(I i) {
  /// Returns the next valid position or an invalid sentinel for the end.
  { ++i } -> I&;
  /// Returns the value at this position; invalid iff this position is invalid.
  { *i } -> ???;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I, class T>
concept ??? = requires(I i) {
  /// Returns the next valid position or an invalid sentinel for the end.
  { ++i } -> I&;
  /// Returns the value at this position; invalid iff this position is invalid.
  { *i } -> T&;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I, class T>
concept ??? =
  std::equality_comparable &&
  requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
template <class I, class T>
concept ??? =
  std::equality_comparable &&
  requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Lifting Concepts

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> std::same_as<I&>;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> std::same_as<T&>;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator { ... };
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
// Generally,
for (auto i = the_beginning(); i != the_end(); get_next(i)) {
  if (get_value(i) == t) count++;
}
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator { ... };

  auto begin() { return list_iterator(_head.get()); }
  auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."

};

  // Generally,
  for (auto i = the_beginning(); i != the_end(); get_next(i)) {
    if (get_value(i) == t) count++;
  }
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator {
    node* _node;
    explicit list_iterator(node* n) : _node(n) {}
  };
  auto begin() { return list_iterator(_head.get()); }
  auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator {
    node* _node;
    explicit list_iterator(node* n) : _node(n) {}
    auto& operator++() { }
    auto& operator*() { }
    bool operator==(const list_iterator& rhs) const = default;
  };
  auto begin() { return list_iterator(_head.get()); }
  auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  struct node { unique_ptr<node> _next; T _value; };
  unique_ptr<node> _head = nullptr;

  struct list_iterator {
    node* _node;
    explicit list_iterator(node* n) : _node(n) {}
    auto& operator++() { _node = _node->next.get(); return *this; }
    auto& operator*() { return _node->value; }
    bool operator==(const list_iterator& rhs) const = default;
  };
  auto begin() { return list_iterator(_head.get()); }
  auto end()   { return list_iterator(nullptr); } // "invalid sentinel for the end."
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  auto begin() { return list_iterator(_head.get()); }
  auto end() { return list_iterator(nullptr); }
};


/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  auto begin() { return user_input_iterator(std::addressof(std::cin)); }
  auto end() { return user_input_iterator(nullptr); }
};
```

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  iterator<T> auto begin() { return list_iterator(_head.get()); }
  iterator<T> auto end() { return list_iterator(nullptr); }
};


/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  iterator<T> auto begin() { return user_input_iterator(std::addressof(std::cin)); }
  iterator<T> auto end() { return user_input_iterator(nullptr); }
};
```

See: https://tinyurl.com/GCCIstreamIterator

Modeling concepts

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  iterator<T> auto begin() { ... }
  iterator<T> auto end() { ... }
};
```

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  iterator<T> auto begin() { ... }
  iterator<T> auto end() { ... }
};
```

```
/// Returns the frequency of `t` in the sequence.
template <std::equality_comparable T, iterator<T> Iter>
auto count(T t, Iter begin, Iter end) -> std::size_t { ... }
```

Lifting concepts (again)

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  iterator<T> auto begin() { ... }
  iterator<T> auto end() { ... }
};
```

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  iterator<T> auto begin() { ... }
  iterator<T> auto end() { ... }
};
```

```
/// Returns the frequency of `t` in the sequence.
template <std::equality_comparable T, iterator<T> Iter>
auto count(T t, Iter begin, Iter end) -> std::size_t { ... }
```

Lifting concepts (again)

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
template <class R, class T>
concept ??? = requires(R r) { ... };
```

Lifting concepts (again)

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
template <class R, class T>
concept ??? = requires(R r) {
  /// Returns an iterator to the first element.
  { r.begin() } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { r.end() } -> iterator<T>;
};
```

Lifting concepts (again)

```
/// A sequence of `T` stored non-contiguously
template <class T>
struct linked_list {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
/// A stream of `T` from stdin.
template <input_streamable T>
class user_input {
  /// Returns an iterator to the first element.
  iterator<T> auto begin() { ... }
  /// Returns a sentinel representing the end.
  iterator<T> auto end() { ... }
};
```

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { r.begin() } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { r.end() } -> iterator<T>;
};
```

Lifting concepts (again)

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { r.begin() } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { r.end() } -> iterator<T>;
};
```

```
static_assert(range<int[42], int>);
```

Lifting concepts (again)

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { r.begin() } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { r.end() } -> iterator<T>;
};
```

```
error: static assertion failed
   static_assert(range<int[], int>);
                 ^~~~~~~~~~~~~~~~~~~~
... because 'range<int[42], int>' evaluated to false
... because 'r.begin()' would be invalid: 'int *' is not a structure or union
{ r.begin() } -> iterator<T>;
```

```
static_assert(range<int[42], int>); // ❌ Sized array-of-int is not a range.
```

Lifting concepts (again)

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { begin(r) } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { end(r) } -> iterator<T>;
};
```

Lifting concepts (again)

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { begin(r) } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { end(r) } -> iterator<T>;
};


template <class R>
auto begin(R& r) -> decltype(r.begin()) { ... }

template <class T, std::size_t N>
auto begin( T(&array)[N] ) -> T* { ... }
```

Lifting concepts (again)

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { begin(r) } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { end(r) } -> iterator<T>;
};


template <class R>
auto begin(R& r) -> decltype(r.begin) { ... }

template <class T, std::size_t N>
auto begin( T(&array)[N] ) -> T* { ... }

static_assert(range<int[42], int>); // ✅ sized array-of-int is a range.
```

Generic algorithms in terms of concepts

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  { begin(r) } -> iterator<T>;
  { end(r) } -> iterator<T>;
};
```

```
/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t { ... }
```

Generic algorithms in terms of concepts

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  { begin(r) } -> iterator<T>;
  { end(r) } -> iterator<T>;
};
```

```
/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t {
  std::size_t count{0};
  for (iterator<T> auto i = begin(r); i != end(r); ++i)
    if (*i == t) count++;
  return count;
}
```

Generic algorithms in terms of concepts

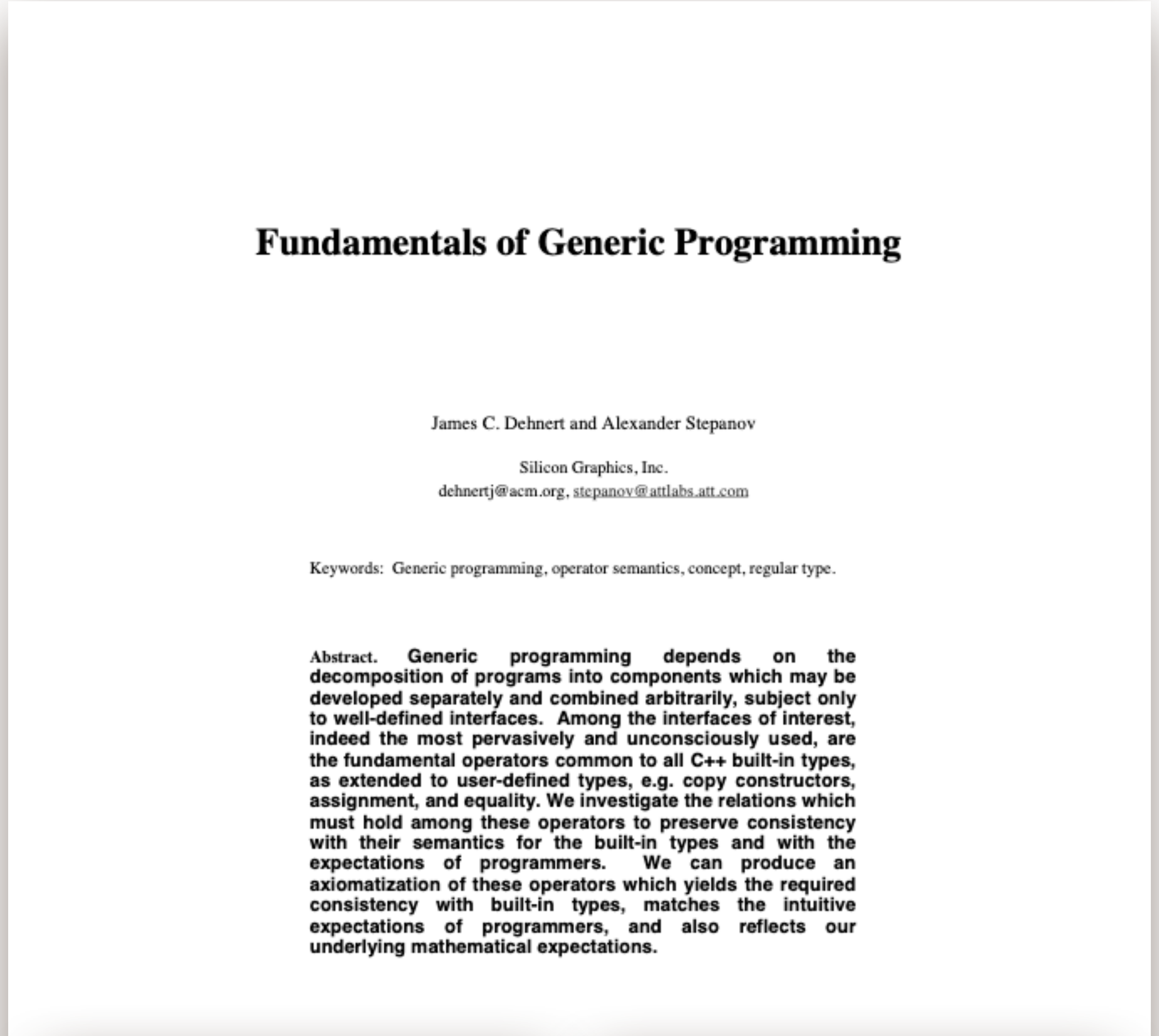
```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  { begin(r) } -> iterator<T>;
  { end(r) } -> iterator<T>;
};
```

```
/// Returns the frequency of `t` in the range.
template <std::equality_comparable T, range<T> Range>
auto count(T t, Range r) -> std::size_t {
  std::size_t count{0};
  for (iterator<T> auto i = begin(r); i != end(r); ++i)
    if (*i == t) count++;
  return count;
}
```

Key takeaway 3/8:

Discover *Concepts* by analyzing common semantics in algorithms



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

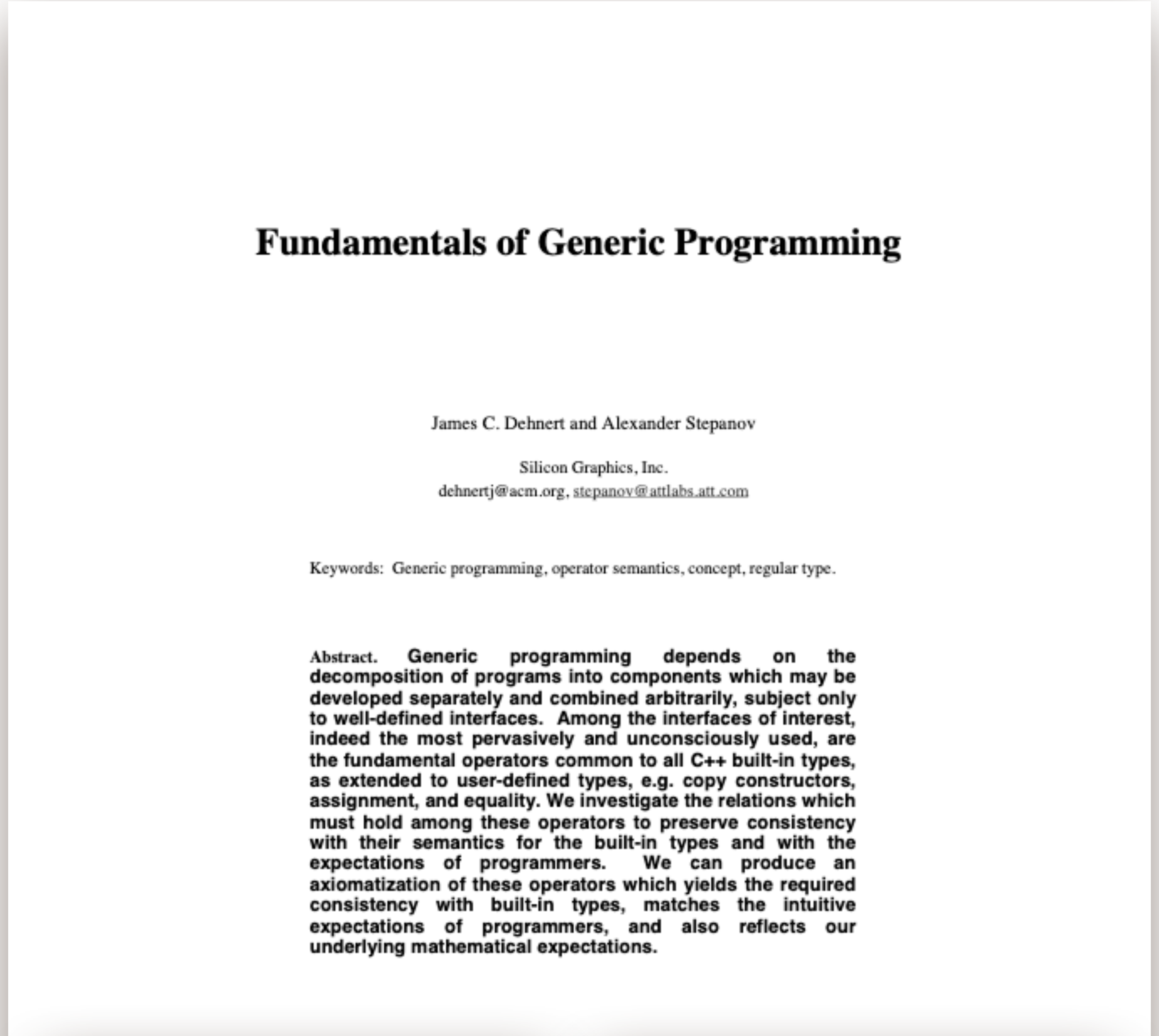
Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Regular types

If we hope to reuse code [which uses] the standard C++ operators, and apply it to both built-in and user-defined types, we must extend the semantics as well as the syntax of the standard operators to user-defined types.

… concepts which match the semantics of built-in types and operators provide an excellent foundation for generic programming.

… a **regular type** matches the built-in type semantics, thereby making our user-defined types *behave* like built-in types as well.



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Regular types have value semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
Ordering, e.g.	`a < b`

Fundamentals of Generic Programming

Regular types have value semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic Programming

Regular types have value semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic Programming

```
template <class T>
concept regular =
    std::semiregular<T> &&
    std::equality_comparable<T>;

template <class T>
concept semiregular =
    std::copyable<T> &&
    std::default_initializable<T>;

// ... taxonomy 4-5 layers deep
```

Intertwined semantics

```
/// A random access sequence of mutable elements.
interface MutableArrayLike<T> = {
  /// The number of elements
  readonly length: number,
  /// Gets the `n`th element
  [n: number]: T
};
```

Intertwined semantics

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    /// Returns the next valid position or an invalid sentinel for the end.
    { ++i } -> I&;
    /// Returns the value at this position; invalid iff this position is invalid.
    { *i } -> T&;
};
```

Intertwined semantics

```
/// A sequence of `T`
template <class R, class T>
concept range = requires(R r) {
  /// Returns an iterator to the first element.
  { begin(r) } -> iterator<T>;
  /// Returns a sentinel representing the end.
  { end(r) } -> iterator<T>;
};
```

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic Programming

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

```
T a = b;
assert(a == b);
```

Fundamentals of Generic Programming

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

```
T a = b;
assert(a == b);
assert(f(a) == f(b))
```

Fundamentals of Generic Programming

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic

```
T a = c;
T b = c;
always_mutates(a);
assert(b==c && a != b);
```

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic Programming

Intertwined semantics

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
~~Ordering, e.g.~~	~~`a < b`~~

Fundamentals of Generic Programming

```
template <class T>
concept regular =
    std::semiregular<T> &&
    std::equality_comparable<T>;

template <class T>
concept semiregular =
    std::copyable<T> &&
    std::default_initializable<T>;

// ... taxonomy 4-5 layers deep
```

cppreference.com

Log in Search

Page Discussion Standard revision: Diff View Edit History

C++ / Concepts library

std::copy_constructible

Defined in header <concepts>

```
template< class T >
concept copy_constructible =
    std::move_constructible<T> &&
    std::constructible_from<T, T&> && std::convertible_to<T&, T> &&
    std::constructible_from<T, const T&> && std::convertible_to<const T&, T> &&
    std::constructible_from<T, const T> && std::convertible_to<const T, T>;
```

(since C++20)

The concept `copy_constructible` is satisfied if `T` is an lvalue reference type, or if it is a `move_constructible` object type where an object of that type can constructed from a (possibly const) lvalue or const rvalue of that type in both direct- and copy-initialization contexts with the usual semantics (a copy is constructed with the source unchanged).

Semantic requirements

If `T` is an object type, then `copy_constructible<T>` is modeled only if given

- `v`, an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`,

the following are true:

- After the definition `T u = v;`, `u` is equal to `v` and `v` is not modified;
- `T(v)` is equal to `v` and does not modify `v`.

Key takeaway 4/8:

Define concepts with intertwined semantics.

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter { };
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.
}
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2);
}
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto&
  std::s
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
```

```
error: no matching function for call to object of type 'const __advance_fn'
    std::ranges::advance(c, 10);

... because 'counter' does not satisfy 'input_or_output_iterator'
... because 'counter' does not satisfy 'weakly_incrementable'
... no type named 'difference_type' in 'std::incrementable_traits<counter>'
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2);
}
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  using difference_type = std::ptrdiff_t;
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
  requires(I i) {
    { ++i } -> std::same_as<I&>;
    { *i } -> std::same_as<T&>;
};
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2);
}
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  using difference_type = std::ptrdiff_t;
  friend bool operator==(counter, counter) = default;
  auto&
  auto&
  std::s
};
```

```
/// A position in a sequence of `T`.
template <class I, class T>
concept iterator =
  std::equality_comparable &&
```

```
error: no matching function for call to object of type 'const __advance_fn'
    std::ranges::advance(c, 10);

... because 'counter' does not satisfy 'input_or_output_iterator'
... because 'counter' does not satisfy 'weakly_incrementable'
... because '__i++' would be invalid: cannot increment value of type 'counter'
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2);
}
```

The spectrum of generality

Any type ***with the required functionality*** *is valid*

std::input_or_output_iterator

Defined in header <iterator>

```
template< class I >
    concept input_or_output_iterator =
        requires(I i) {
            { *i } -> /*can-reference*/;
        } &&
        std::weakly_incrementable<I>;
```

(since C++20)

The input_or_output_iterator concept forms the basis of the iterator concept taxonomy; every iterator type satisfies the input_or_output_iterator requirements.

std::weakly_incrementable

Defined in header <iterator>

```
template< class I >
    concept weakly_incrementable =
        std::movable<I> &&
        requires(I i) {
            typename std::iter_difference_t<I>;
            requires /*is-signed-integer-like*/<std::iter_difference_t<I>>;
            { ++i } -> std::same_as<I&>; // not required to be equality-preserving
            i++;                          // not required to be equality-preserving
        };
```

(since C++20)

Concepts are abstract entities

```
template <class T>
concept Addable = requires(T t) {
  { t + t } -> std::same_as<T>;
};

template <class T>
concept Subtractable = requires(T t) {
  { t - t } -> std::same_as<T>;
};

template <class T>
concept Multipliable = requires(T t) {
  { t * t } -> std::same_as<T>;
};

...etc
```

Concepts are abstract entities

```
template <class T>
concept Addable = requires(T t) {
  { t + t } -> std::same_as<T>;
};

template <class T>
concept Subtractable = requires(T t) {
  { t - t } -> std::same_as<T>;
};

template <class T>
concept Addable = requires(T t) {
  { t / t } -> std::same_as<T>;
};

...etc
```

```
template <class T>
concept Numeric = requires(T t) {
  { t + t } -> std::same_as<T>;
  { t - t } -> std::same_as<T>;
  { t * t } -> std::same_as<T>;
  { t / t } -> std::same_as<T>;
  ...etc
};
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  using difference_type = std::ptrdiff_t;
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};
```

```
/// A position in a sequence.
template <class I, class T>
concept iterator = requires(I i) {
  { ++i } -> std::same_as<I&>;
  { *i } -> std::same_as<T&>;
  { i == i } -> std::same_as<bool>;
  { i != i } -> std::same_as<bool>;
};
```

```
int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
```

Using our `iterator`

```
/// A monotonically increasing integer count.
struct counter {
  using difference_type = std::ptrdi
  friend bool operator
  auto& operator
  auto& opera
  std::size
};



int main() {
  static_assert(iterator<counter>

  auto c = counter{40};
  std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
```

```
/// A position in a sequence.
template <class I, class T>
concept iterator = requires(I i) {
  td::same_as<I&>;
  e_as<T&>;
  me_as<bool>;
  _as<bool>;

iterator over std::size_t.
```

Semantic consistency >>> Maximum reusability

Adobe

The spectrum of generality

Any type ***with the required functionality*** *is valid*

The spectrum of generality

*Any type **satisfying the required <u>concepts</u>** is valid*

~~*Any type **with the required functionality** is valid*~~

Key takeaway 5/8:

[Over-]constrain algorithms with *Concepts* that model abstract entities.

Concepts FTW

```
/// A monotonically increasing integer count.
struct counter {
  using difference_type = std::ptrdiff_t;
  friend bool operator==(counter, counter) = default;
  auto& operator++() { ++count; return *this; }
  auto& operator*() { return count; }
  std::size_t count = 0;
};


int main() {
  static_assert(iterator<counter, std::size_t>); // ✅ counter is an iterator over std::size_t.

  auto c = counter{40};
  std::ranges::advance(c, 2); // '__i++' invalid: cannot increment value of type 'counter'
}
```

WTF Concepts

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}
```

WTF Concepts

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}


int main() {
  std::vector<int> v{1, 2, 3};
  find(begin(v), end(v), 2); // ✅ finds the 2
}
```

WTF Concepts

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}


int main() {
  std::vector<int> v{1, 2, 3};
  find(begin(v), end(v), 2); // ✅ finds the 2

  std::istream_iterator first(std::cin), last;
  find(first, last, 42);
}
```

WTF Concepts

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}


int main
  std::v
  find(b

  std::istream_iterator first(std::cin), last;
  find(first, last, 42);
}
```

```
In 'auto find(I, I, T) [with I = std::istream_iterator<char>; T = int]':
no match for 'operator<' (operand types are 'std::istream_iterator<char>'...)
  while (first < last && !(*first == value))
         ~~~~~~^~~~~~
```

The spectrum of generality | That's too generic!

```
/// Arranges the elements of `c` in non-decreasing order.
function sort<C>(c: C) {
  // Bubble sort implementation to demonstrate.
  for (let i = 0; i < c.length; i++) {
    for (let j = 0; j < c.length - i - 1; j++) {
      if (!(c[j] < c[j + 1])) {
        [c[j], c[j + 1]] = [c[j + 1], c[j]];
      }
    }
  }
}
```

```
Property 'length' does not exist on type 'C'.
```

```
No index signature with a parameter
of type 'number' was found on type 'C'.
```

TS

WTF Concepts

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}
```

```
In 'auto find(I, I, T) [with I = std::istream_iterator<char>; T = int]':
no match for 'operator<' (operand types are 'std::istream_iterator<char>'...)
  while (first < last && !(*first == value))
         ~~~~~~^~~~~~
```

```
int main
  std::v
  find(b

  std::istream_iterator first(std::cin), last;
  find(first, last, 42);
}
```

WTF Concepts | Stealthy Requirements

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}


int main() {
  std::vector<int> v{1, 2, 3};
  find(begin(v), end(v), 2); // ✅ finds the 2

  std::istream_iterator first(std::cin), last;
  find(first, last, 42); // No operator `<`...
}
```

Archetypes

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}

inline void test_find(input_iterator_archetype f,
                      input_iterator_archetype l,
                      equality_comparable_archetype x) {
  find(f, l, x); // No operator `<`... but caught by the author of `find`.
}
```

Archetypes

https://www.boost.org/doc/libs/1_84_0/libs/concept_check/concept_covering.htm

```
template <class T>
class input_iterator_archetype
{
private:
  typedef input_iterator_archetype self;
public:
  typedef std::input_iterator_tag iterator_category;
  typedef T value_type;
  struct reference {
    operator const value_type&() const { return static_object<T>::get(); }
  };
  typedef const T* pointer;
  typedef std::ptrdiff_t difference_type;
  self& operator=(const self&) { return *this;  }
  bool operator==(const self&) const { return true; }
  bool operator!=(const self&) const { return true; }
  reference operator*() const { return reference(); }
  self& operator++() { return *this; }
  self operator++(int) { return *this; }
};
```

Archetypes

- Boost Concept Check Library
    - https://www.boost.org/doc/libs/1_84_0/libs/concept_check/concept_check.htm
- Andrzej Krzemieński Blog Post: Concept Archetypes
    - https://akrzemi1.wordpress.com/2020/09/02/concept-archetypes/
- "I'm skeptical that we should expect early-checked generics to land in a C++ compiler, given their inflexibility." -- Sean Baxter
    - https://github.com/seanbaxter/circle/blob/master/new-circle/README.md#generic

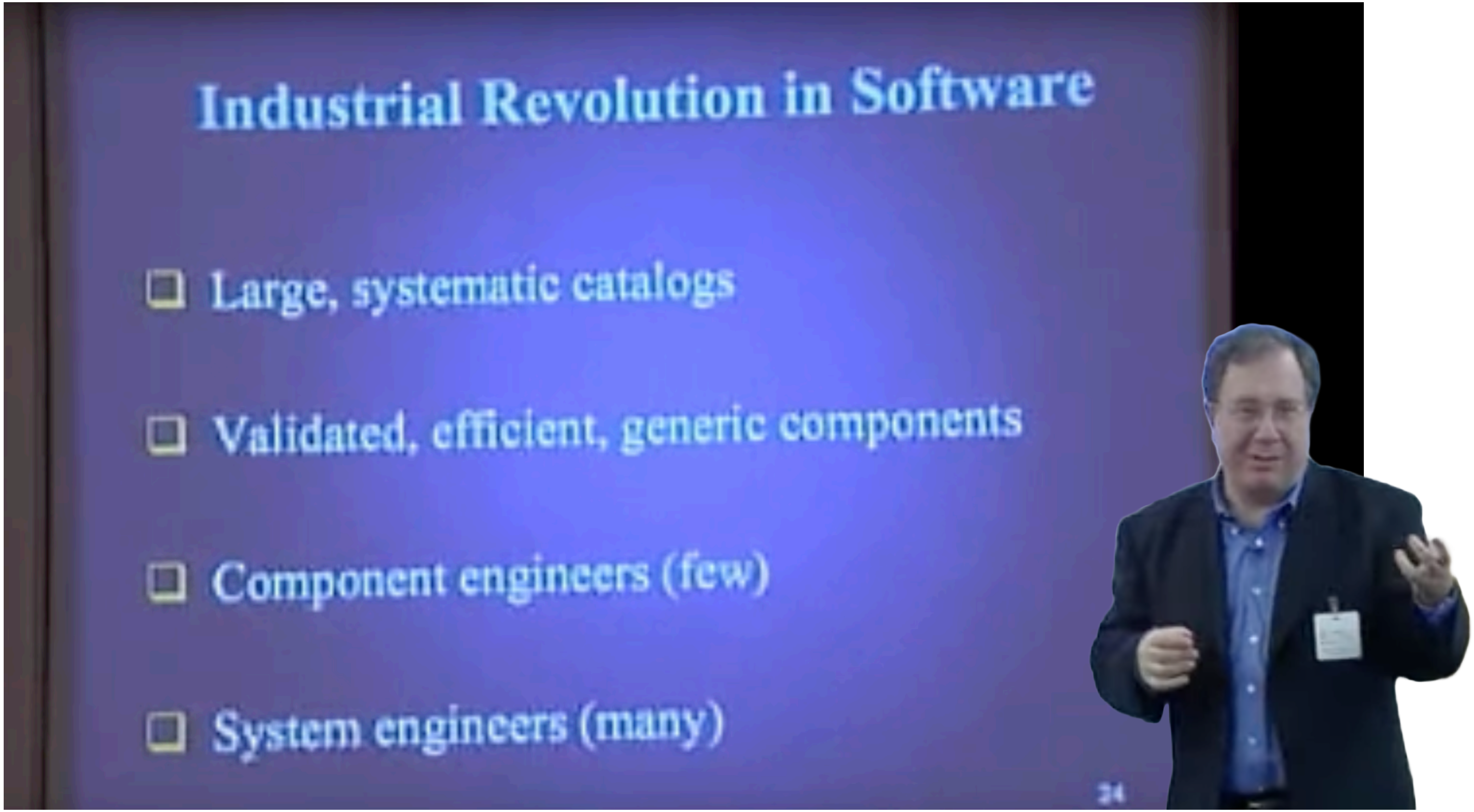
Stealthy Requirements

```
/// Returns an iterator to the first element equal to `x`.
template <std::input_iterator I, std::equality_comparable T>
auto find(I first, I last, const T& x) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}


int main() {
  std::vector<int> v{1, 2, 3};
  find(begin(v), end(v), 2); // ✅ finds the 2

  std::istream_iterator first(std::cin), last;
  find(first, last, 42); // No operator `<`...
}
```

Key takeaway 6/8:

⚠️ **Beware of *stealthy requirements***

"Making maximally reusable functions isn't what's needed to ship product."



Alexander Stepanov: STL and its Design Principles (Jan 2002)

https://youtu.be/COuHLky7E2Q

Product
Code

Library
Code

UI Layouts

`std::vector`

Product Code

Library Code

UI Layouts

Composite Graph

File Parsers

`std::vector`

Product Code

Library Code

User preferences

Event Handling

Mouse tracker

All code is a liability.

UI Layouts

Composite Graph

File Parsers

`std::vector`

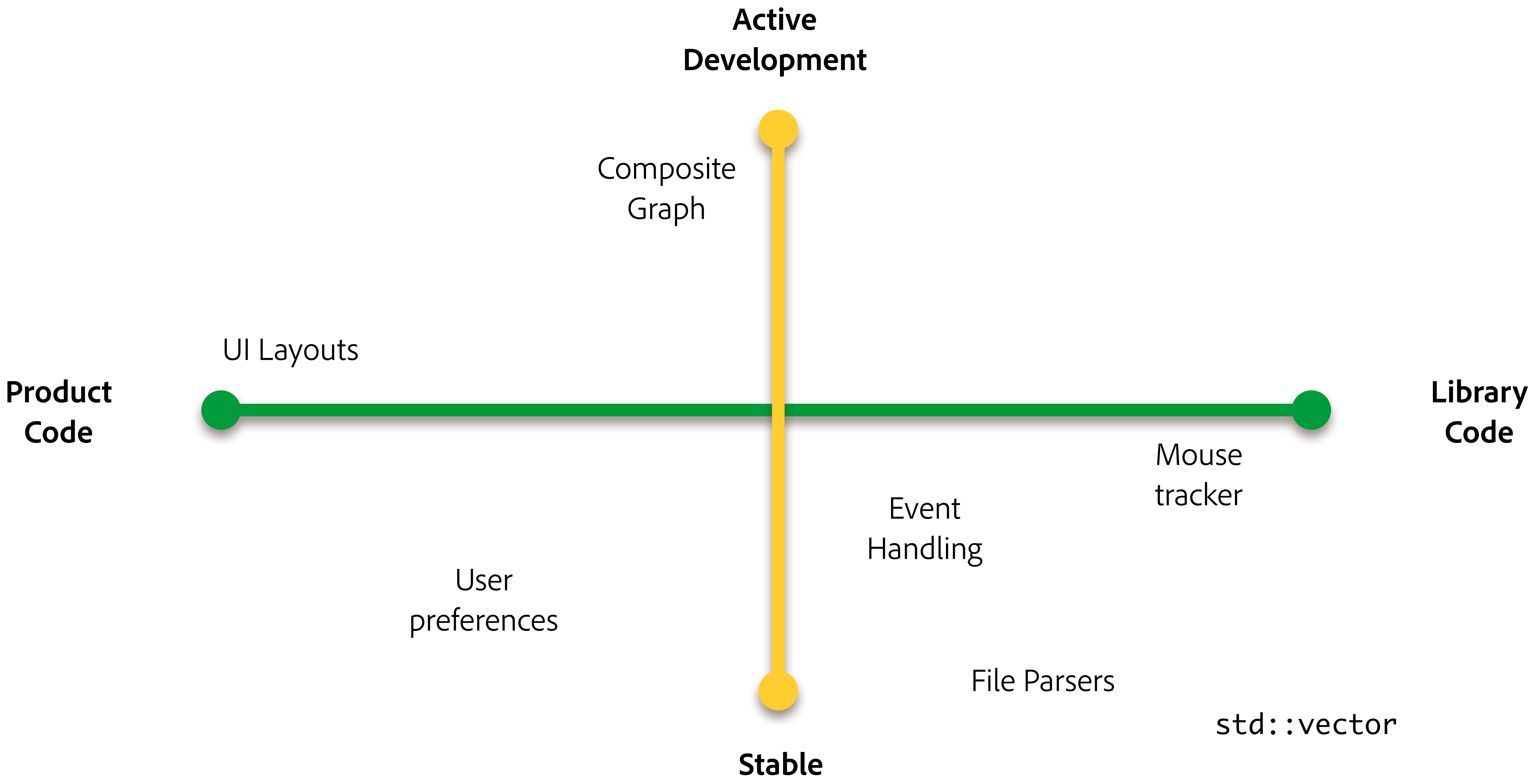
Product Code

Library Code

User preferences

Event Handling

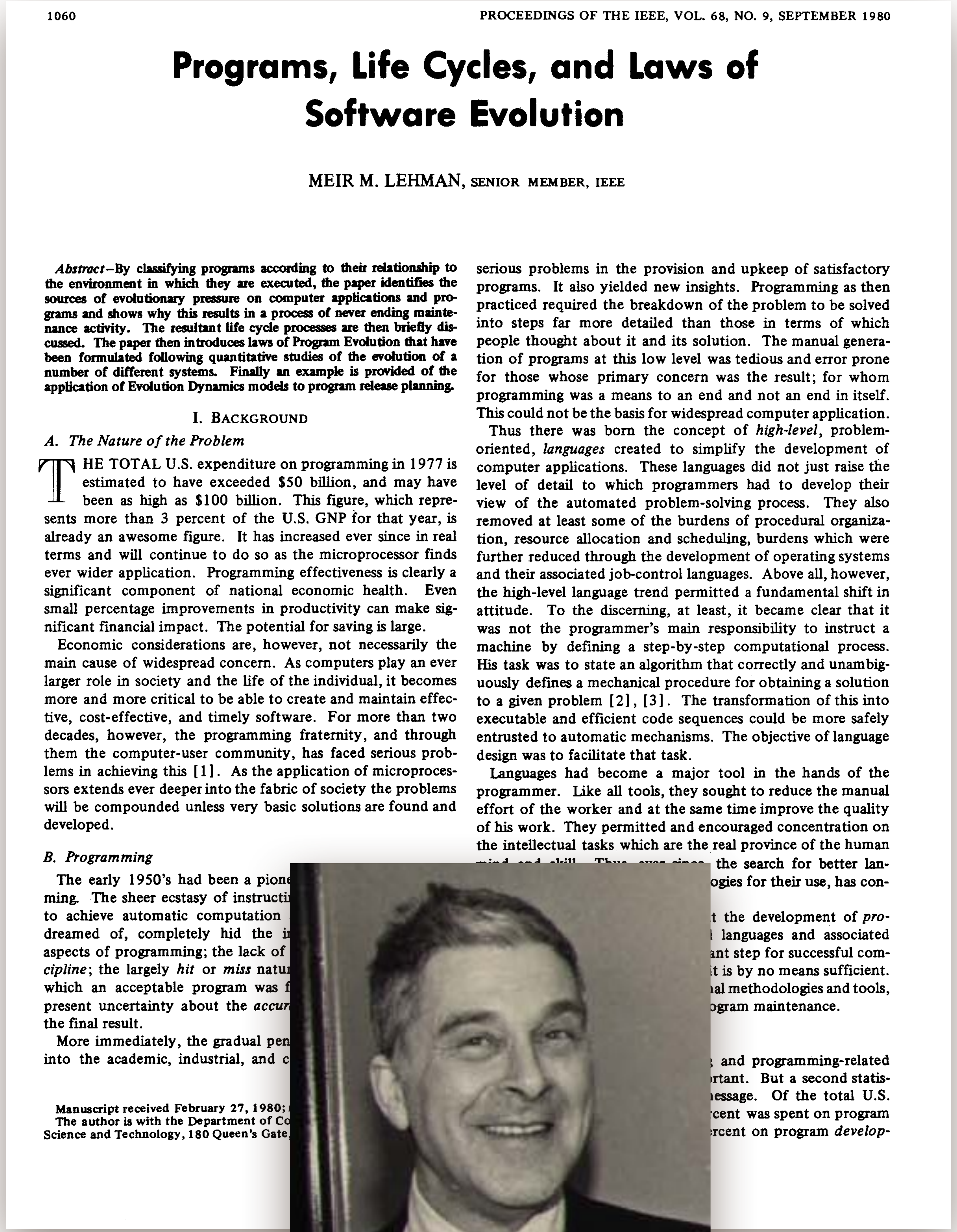
Mouse tracker


Active Development
Composite Graph
UI Layouts
Product Code
Library Code
Mouse tracker
Event Handling
User preferences
File Parsers
std::vector
Stable

Key takeaway 7/8:

Generalize your stable code before it rots.

Lehman's laws of software evolution



1060 PROCEEDINGS OF THE IEEE, VOL. 68, NO. 9, SEPTEMBER 1980

Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

Abstract–By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

I. Background

A. The Nature of the Problem

THE TOTAL U.S. expenditure on programming in 1977 is estimated to have exceeded $50 billion, and may have been as high as $100 billion. This figure, which represents more than 3 percent of the U.S. GNP for that year, is already an awesome figure. It has increased ever since in real terms and will continue to do so as the microprocessor finds ever wider application. Programming effectiveness is clearly a significant component of national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations are, however, not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective, and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [1]. As the application of microprocessors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

B. Programming

The early 1950's had been a pione
ming. The sheer ecstasy of instructi
to achieve automatic computation
dreamed of, completely hid the i
aspects of programming; the lack of
cipline; the largely *hit* or *miss* natu
which an acceptable program was f
present uncertainty about the *accur*
the final result.

More immediately, the gradual pen
into the academic, industrial, and c

Manuscript received February 27, 1980;
The author is with the Department of Co
Science and Technology, 180 Queen's Gate,

serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practiced required the breakdown of the problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

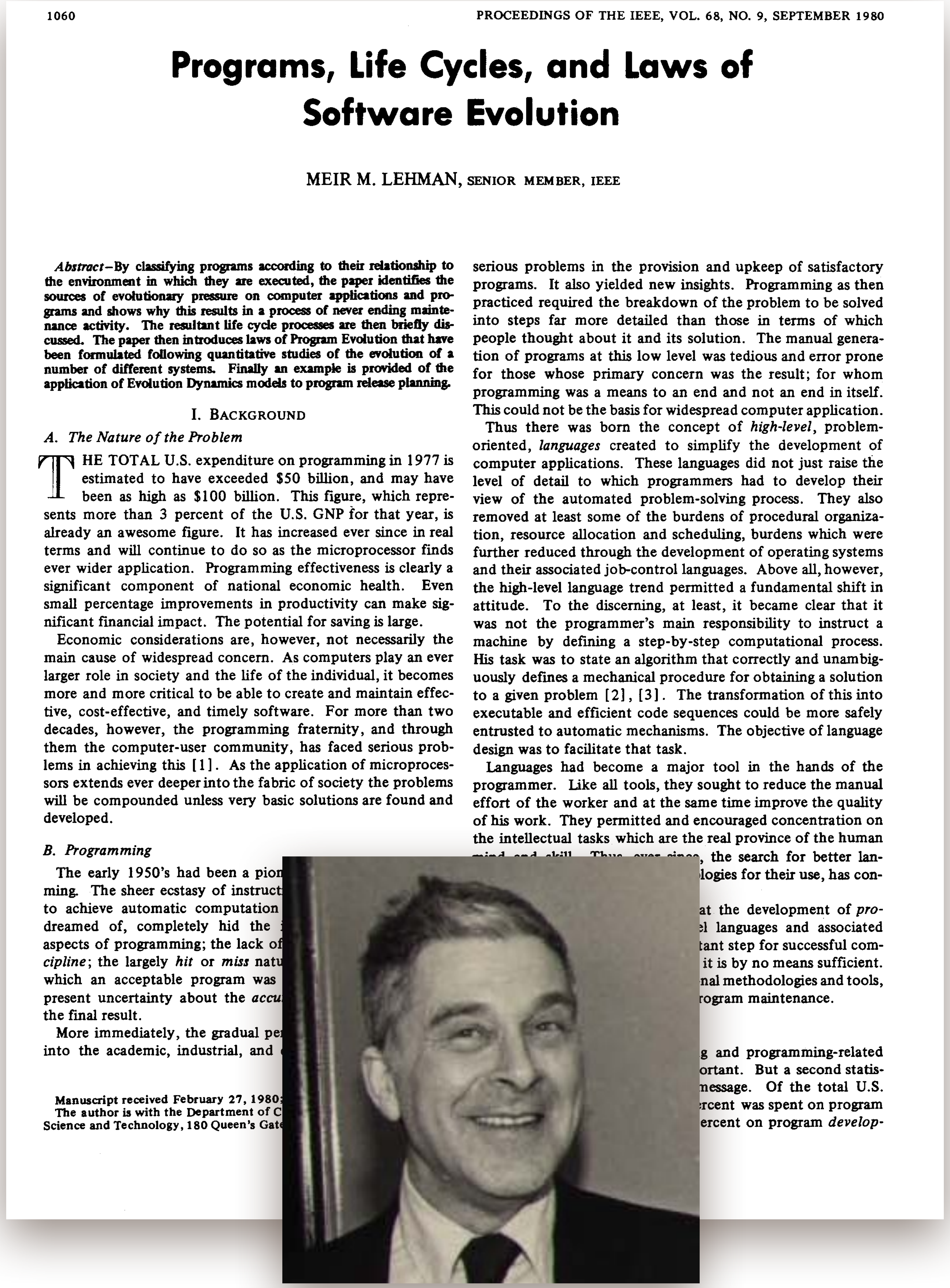
Thus there was born the concept of *high-level*, problem-oriented, *languages* created to simplify the development of computer applications. These languages did not just raise the level of detail to which programmers had to develop their view of the automated problem-solving process. They also removed at least some of the burdens of procedural organization, resource allocation and scheduling, burdens which were further reduced through the development of operating systems and their associated job-control languages. Above all, however, the high-level language trend permitted a fundamental shift in attitude. To the discerning, at least, it became clear that it was not the programmer's main responsibility to instruct a machine by defining a step-by-step computational process. His task was to state an algorithm that correctly and unambiguously defines a mechanical procedure for obtaining a solution to a given problem [2], [3]. The transformation of this into executable and efficient code sequences could be more safely entrusted to automatic mechanisms. The objective of language design was to facilitate that task.

Languages had become a major tool in the hands of the programmer. Like all tools, they sought to reduce the manual effort of the worker and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real province of the human
the search for better lan-
ogies for their use, has con-

t the development of *pro-*
l languages and associated
ant step for successful com-
it is by no means sufficient.
al methodologies and tools,
ogram maintenance.

and programming-related
rtant. But a second statis-
essage. Of the total U.S.
rcent was spent on program
rcent on program *develop-*

Lehman's laws of software evolution



1060 PROCEEDINGS OF THE IEEE, VOL. 68, NO. 9, SEPTEMBER 1980

Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

Abstract—By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

I. Background

A. The Nature of the Problem

THE TOTAL U.S. expenditure on programming in 1977 is estimated to have exceeded $50 billion, and may have been as high as $100 billion. This figure, which represents more than 3 percent of the U.S. GNP for that year, is already an awesome figure. It has increased ever since in real terms and will continue to do so as the microprocessor finds ever wider application. Programming effectiveness is clearly a significant component of national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations are, however, not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective, and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [1]. As the application of microprocessors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

B. Programming

The early 1950's had been a pion... ming. The sheer ecstasy of instruct... to achieve automatic computation... dreamed of, completely hid the ... aspects of programming; the lack of... *cipline*; the largely *hit* or *miss* natu... which an acceptable program was... present uncertainty about the *accu*... the final result.

More immediately, the gradual pe... into the academic, industrial, and ...

Manuscript received February 27, 1980;...
The author is with the Department of C...
Science and Technology, 180 Queen's Gat...

serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practiced required the breakdown of the problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

Thus there was born the concept of *high-level*, problem-oriented, *languages* created to simplify the development of computer applications. These languages did not just raise the level of detail to which programmers had to develop their view of the automated problem-solving process. They also removed at least some of the burdens of procedural organization, resource allocation and scheduling, burdens which were further reduced through the development of operating systems and their associated job-control languages. Above all, however, the high-level language trend permitted a fundamental shift in attitude. To the discerning, at least, it became clear that it was not the programmer's main responsibility to instruct a machine by defining a step-by-step computational process. His task was to state an algorithm that correctly and unambiguously defines a mechanical procedure for obtaining a solution to a given problem [2], [3]. The transformation of this into executable and efficient code sequences could be more safely entrusted to automatic mechanisms. The objective of language design was to facilitate that task.

Languages had become a major tool in the hands of the programmer. Like all tools, they sought to reduce the manual effort of the worker and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real province of the human ..., the search for better lan... ...logies for their use, has con...

...at the development of *pro*... ...l languages and associated ...tant step for successful com... it is by no means sufficient. ...nal methodologies and tools, ...rogram maintenance.

...g and programming-related ...ortant. But a second statis... ...message. Of the total U.S. ...rcent was spent on program ...ercent on program *develop*-

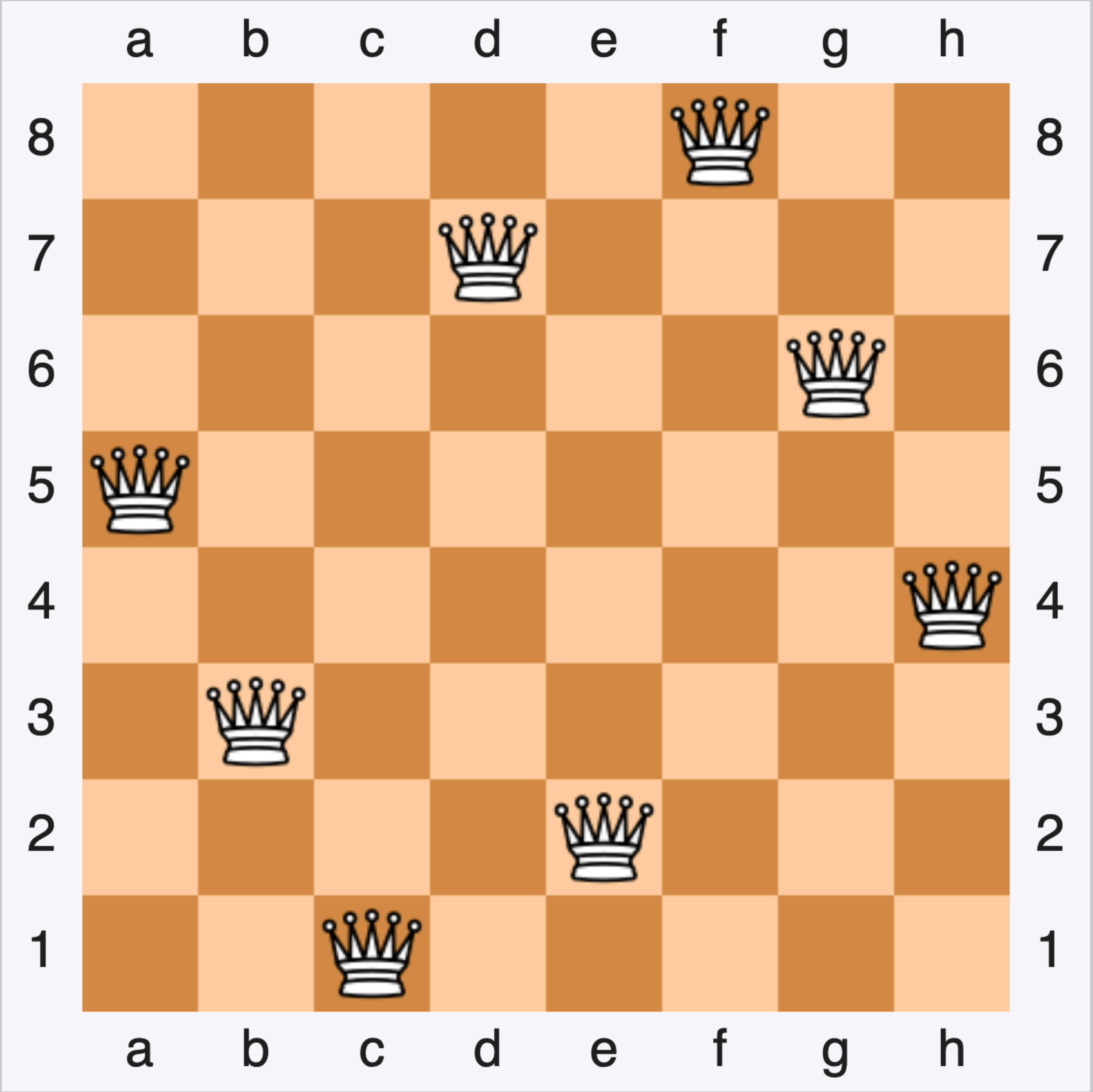
[The classifications of software in this paper are] based on a recognition of the fact that, at the very least, any program is *a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse.*

Lehman's laws of software evolution

- E-Programs
  - Outcomes specified, and the program is **embedded** in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.
- P-Programs
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.
- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.

Lehman's laws of software evolution

- E-Programs
  - Outcomes specified, and the progra uck scheduling & orchestration)
  - Change frequently in response to c
- P-Programs
  - Outcome specified, but implement kfish)
  - Programs are evaluated subjectivel
  - Spec doesn't change, but model of
- **S-Programs**
  - Completely specified (e.g. 8-Queens
  - Programs are evaluated entirely in re



The only symmetrical solution to the eight queens puzzle (up to rotation and reflection)

Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is **embedded** in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.
- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.
- **S-Programs**
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.

Lehman's laws of software evolution

- E-Programs
  - Outcomes specified, and the program is **embedded** in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.
- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.
- S-Programs
  - Completely specified (e.g. 8-Queens Problem)
  - Programs are evaluated entirely in relation to their spec, and rarely change.

Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is **embedded** in the problem space (e.g. delivery truck scheduling & orchestration)
  - Change frequently in response to changing environment.
- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.
- **S-Programs**
- Completely specified (e.g. 8-Queens Problem)
- Programs are evaluated entirely in relation to their spec, and rarely change.

Lehman's laws of software evolution

- **Continuing Change**
  - E-Programs must be continually adapted or they become progressively less satisfactory.
- **Continuing Growth**
  - the functional content of an E-Program must be continually increased to maintain user satisfaction over its lifetime.
- **Increasing Complexity**
  - as E-Programs evolve, their complexity increases unless work is done to maintain or reduce it.
- **Declining Quality**
  - the quality of an E-Program will appear to be declining unless it is rigorously maintained

Lehman's laws of software evolution

- **Continuing Change**
  - E-Programs must be continually adapted or they become progressively less satisfactory.
- **Continuing Growth**
  - the functional content of an E-Program must be continually increased to maintain user satisfaction over its lifetime.
- **Increasing Complexity**
  - as E-Programs evolve, their complexity increases unless work is done to maintain or reduce it.
- **Declining Quality**
  - the quality of an E-Program will appear to be declining unless it is rigorously maintained

Active Development

- E-Programs
- P-Programs
- S-Programs

Stable

Lehman's laws of software evolution

- **E-Programs**
  - Outcomes specified, and the program is **embedded** in the problem space (delivery truck scheduling)
  - Change frequently in response to changing environment.
- **P-Programs**
  - Outcome specified, but implementation is a models the problem. (e.g. chess player / stockfish)
  - Programs are evaluated subjectively; how well do they do?
  - Spec doesn't change, but model of the problem might.
- **S-Programs**
- Completely specified (e.g. 8-Queens Problem)
- Programs are evaluated entirely in relation to their spec, and rarely change.

Active Development

- E-Programs
- P-Programs
- S-Programs

Stable

Key takeaway 8/8:

Identify stable specifications and generalize their implementations.

Adobe