Implementing Swift Generics

Slava Pestov, Apple
John McCall, Apple

Swift Generics

- Bounded parametric polymorphism
  - Similar to Java, C#, Haskell, ML…
  - Constraints described in terms of “protocols”
- Separate static compilation
  - Specialization can still be done as an *optimization*
  - … unlike C++
- Efficient generic type layout without boxing
  - Type parameters can be substituted with any type
  - … like C++

Generic Functions

- Functions can be parameterized:

```
func f<T>(_ t: T) -> T {
  let copy = t
  print(copy)
  return copy
}
```

- Usable with any type:

```
f(value: 1)
f(value: (“Hello”, “LLVM”))
```

Generic Types

- Types can be parameterized:

```
struct Pair<T> {
  var first: T
  var second: T
}
```

- Reified generics:
  - Instantiations have identity at runtime
    - … the types `Pair<Int>` and `Pair<Double>` are distinct
  - Allows different layouts

Generic Type Layout

- Types can be parameterized:

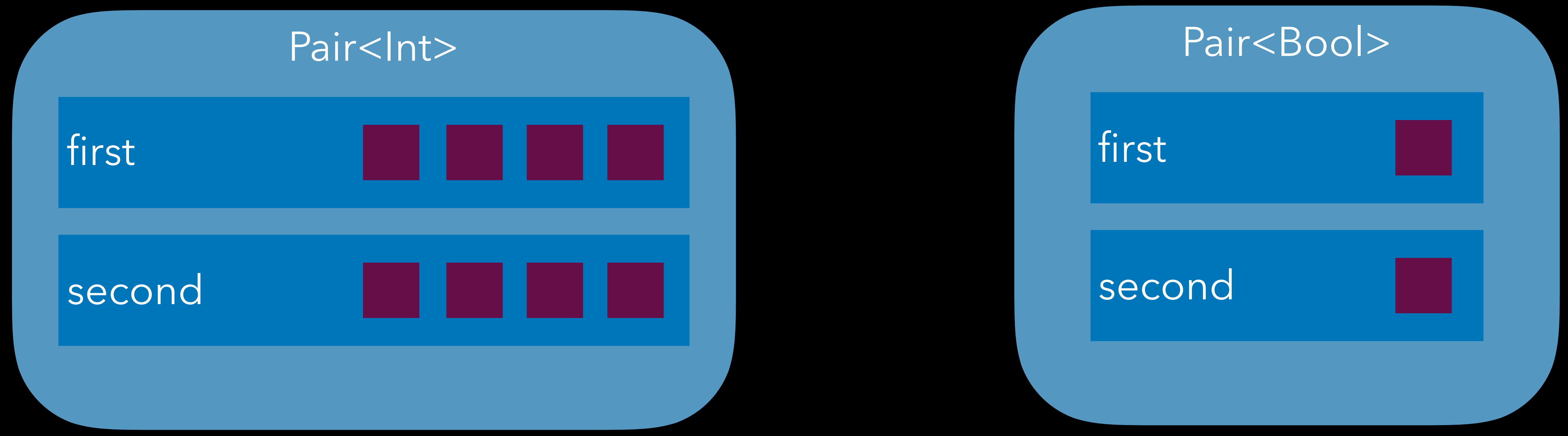
```
struct Pair<T> {
  var first: T
  var second: T
}
```

```
val: 0x00000045
     0x000001a4
```

```
let val = Pair(first: 69,
               second: 420)
```

```
val: 0x0001
```

```
let val = Pair(first: false,
               second: true)
```



Separate Compilation

- libPair.so:

```
public func makePair<T>(_ t1: T, _ t2: T)
    -> Pair<T> {
  return Pair(first: t1, second: t2)
}
```

- myprogram:

```
struct MyType { }
let p = makePair(MyType(), MyType())
```

Compiling Unconstrained Generics

Generic Functions

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

Generic Functions

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Steps:
  - Accept an (arbitrary) T
  - Copy a T into a local
  - Move local into the result
  - Destroy parameter

Generic Functions

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Steps:
  - Accept an (arbitrary) T
  - Copy a T into a local
  - Move local into the result
  - Destroy parameter

Generic Functions

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Steps:
  - Accept an (arbitrary) T
  - Copy a T into a local
  - Move local into the result
  - Destroy parameter

Generic Functions

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Steps:
  - Accept an (arbitrary) T
  - Copy a T into a local
  - Move local into the result
  - Destroy parameter

Value Witness Tables

- Strawman layout:

```
struct value_witness_table {
  size_t size, align;
  void (*copy_init)(opaque *dst, const opaque *src, type *T);
  void (*copy_assign)(opaque *dst, const opaque *src, type *T);
  void (*move_init)(opaque *dst, opaque *src, type *T);
  void (*move_assign)(opaque *dst, opaque *src, type *T);
  void (*destroy)(opaque *val, type *T);
};
```

- Just like copy ctor, move ctor, dtor, operator= in C++
  - …but generated by compiler

Value Witness Tables

value witness table

- size
- alignment
- copy(…)
- move(…)
- destroy(…)

Examples:

Trivial 4 byte type

- size: 4
- alignment: 4
- copy: memcpy
- move: memcpy
- destroy: no-op

Reference type

- size: 8
- alignment: 8
- copy: retain
- move: trivial
- destroy: release

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Generated Code

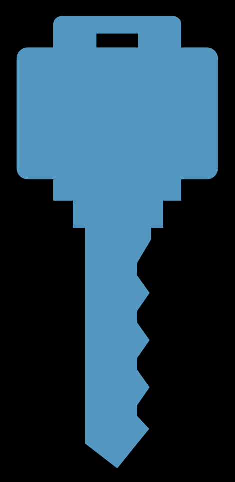
- Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

- Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

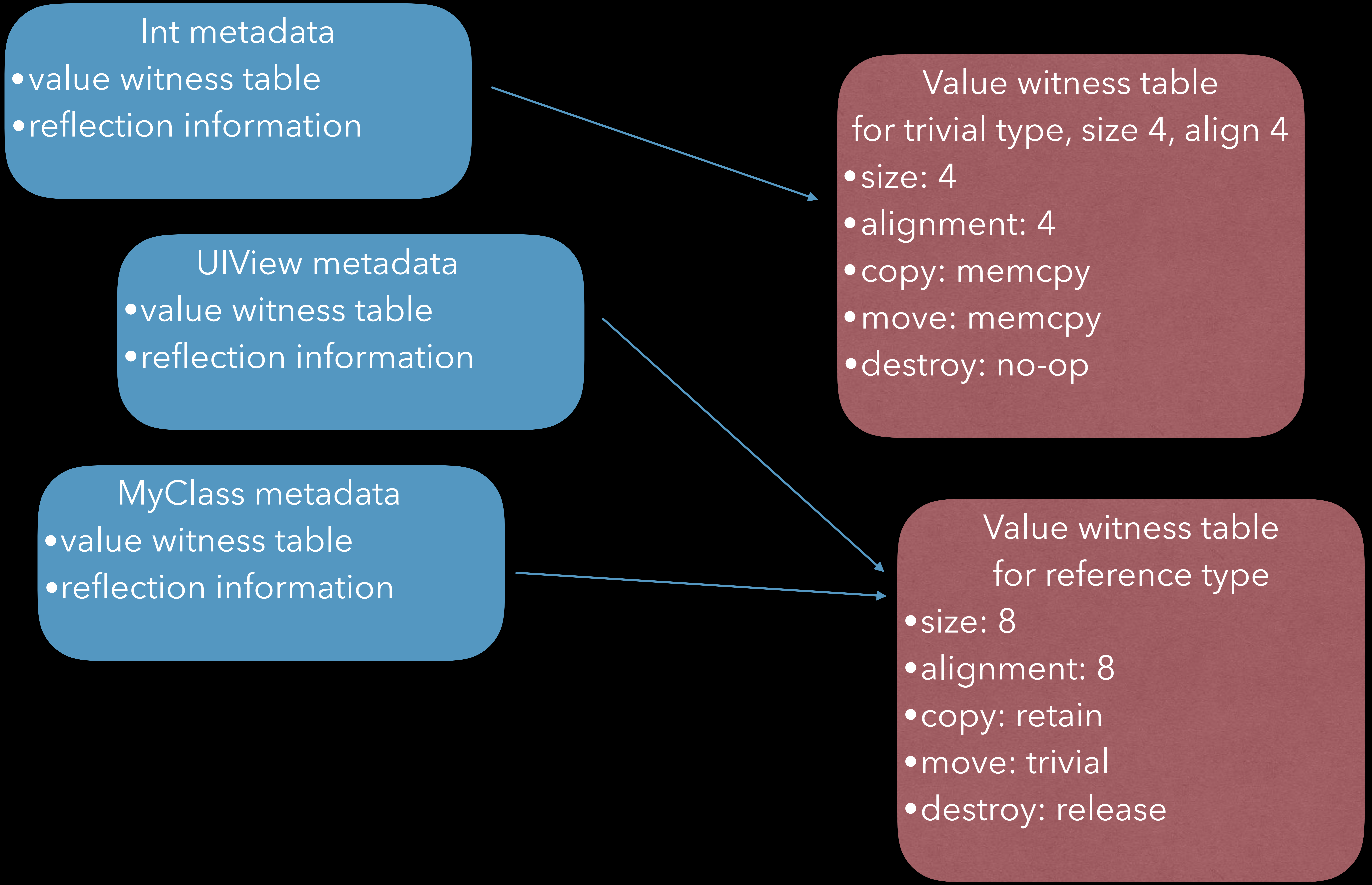
Type Metadata



Type Metadata

- How do you call a generic function with concrete types?
- Simple case: non-generic types, eg `Int`, `Bool`, `MyStruct`
  - Compiler emits fixed metadata
  - Value witness table is constant, fixed size and alignment

Type Metadata



Using Concrete Types

```
f(123)

struct MyStruct {
  var a, b, c, d: UInt8
}
f(MyStruct())
```

Using Concrete Types

```
f(123)

struct MyStruct {
  var a, b, c, d: UInt8
}
f(MyStruct())
```

```
int val = 123;
extern type *Int_metadata;

_f(&val, Int_metadata);

MyStruct val;
type *MyStruct_metadata = { … };
f(&val, MyStruct_metadata);
```

Using Concrete Types

```
f(123)

struct MyStruct {
  var a, b, c, d: UInt8
}
f(MyStruct())
```

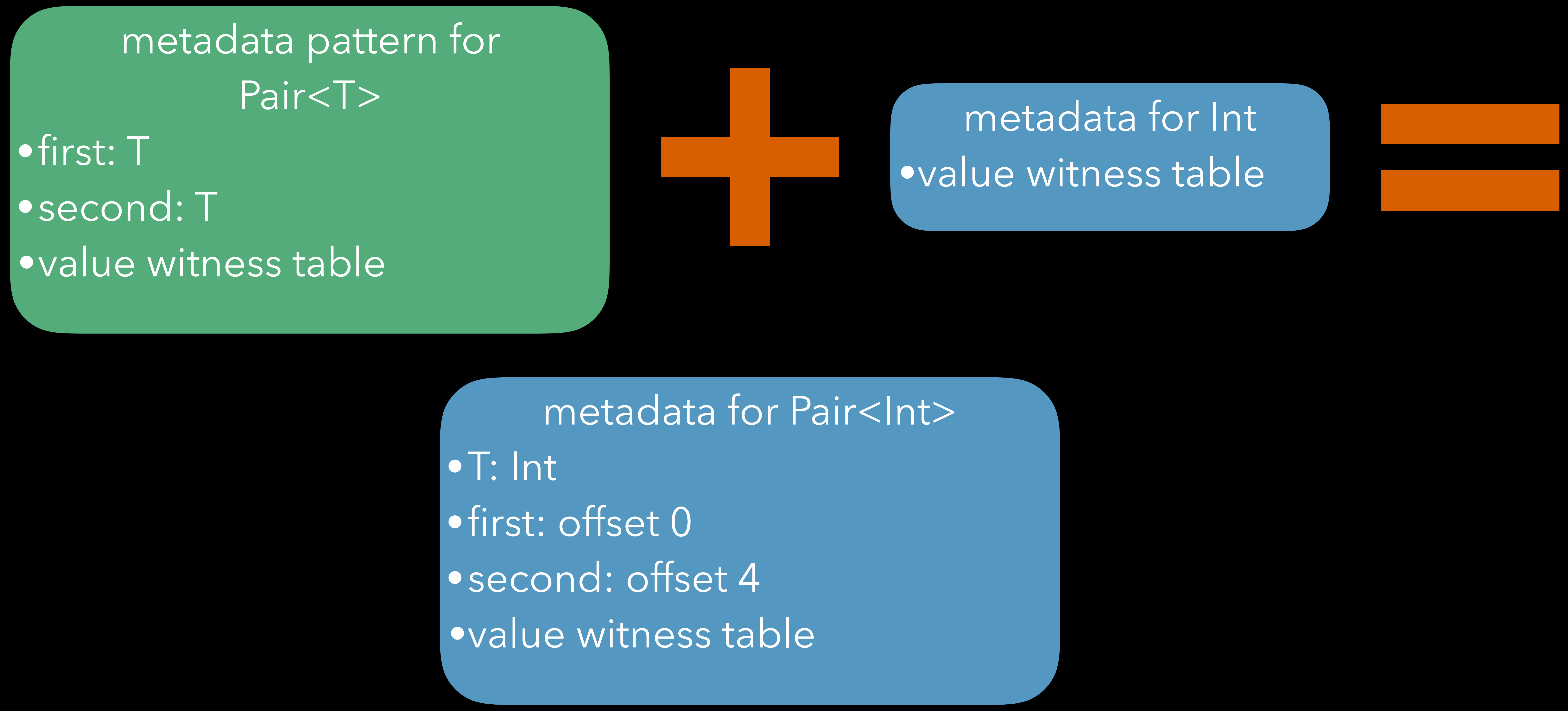
```
int val = 123;
extern type *Int_metadata;

_f(&val, Int_metadata);

MyStruct val;
type *MyStruct_metadata = { … };
f(&val, MyStruct_metadata);
```

Using Generic Types

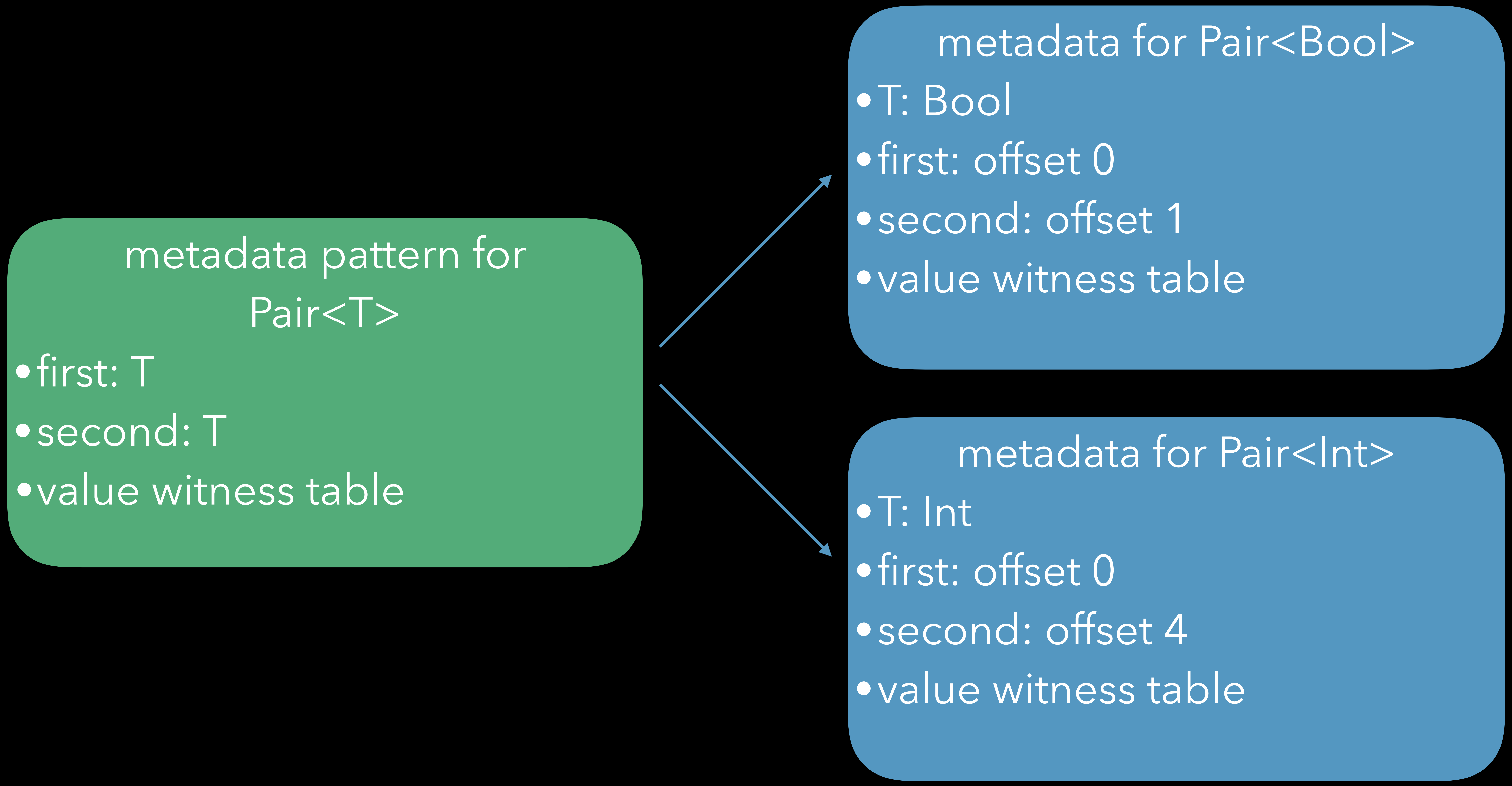
```
struct Pair<T> {
  var first: T
  var second: T
}
```

- We pack the fields in the same way as we would for a non-generic type
  - … and this must happen at runtime!
- `f(Pair<Int>())` vs. `f(Pair<Bool>())`
  - These two are distinct types and have unique type metadata

Generic Type Metadata



Generic Type Metadata



Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- The layout of `Pair<T>` depends on `T`
  - ‘`first`’ is still at offset 0
  - ‘`second`’ is at a *dynamic offset*
- Must instantiate `Pair<T>` to compute field offsets

Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Generic Property Access

- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Generic Property Access

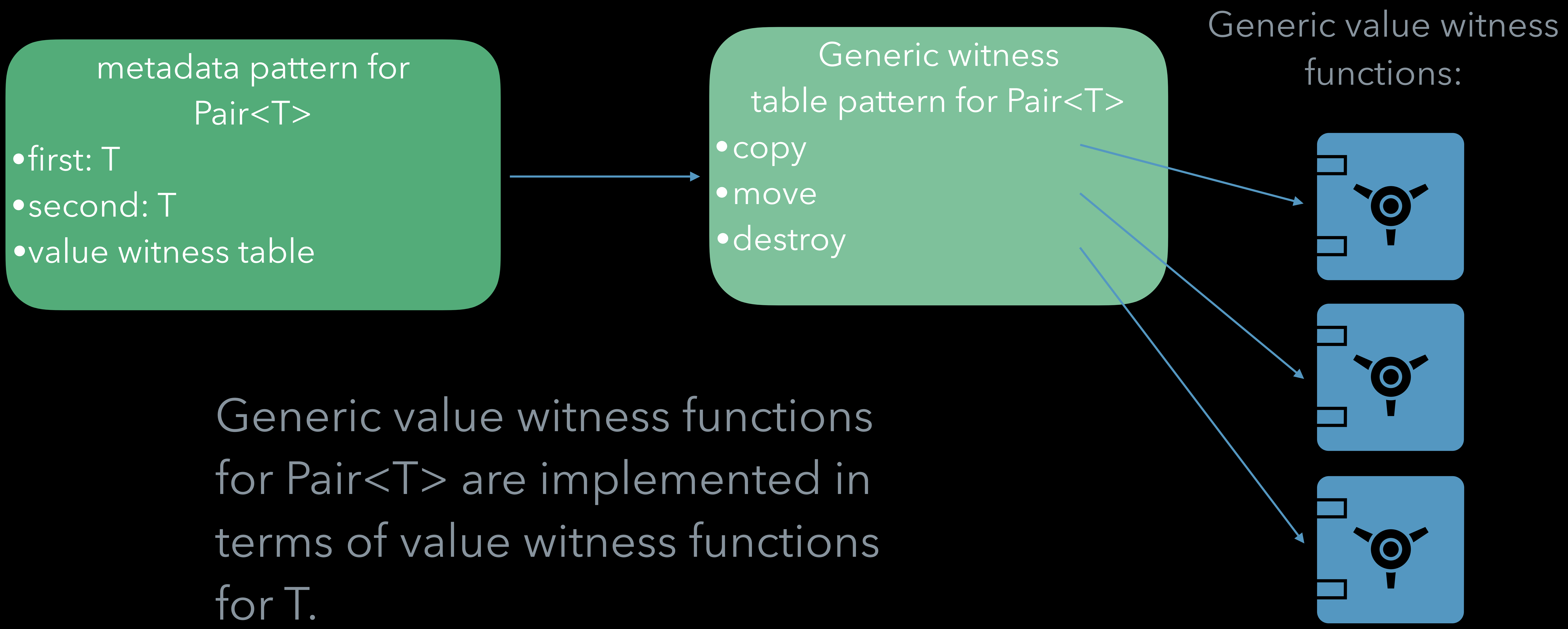
- Example:

```
func getSecond<T>(_ pair: Pair<T>) -> T {
  return pair.second
}
```

- Implementation:

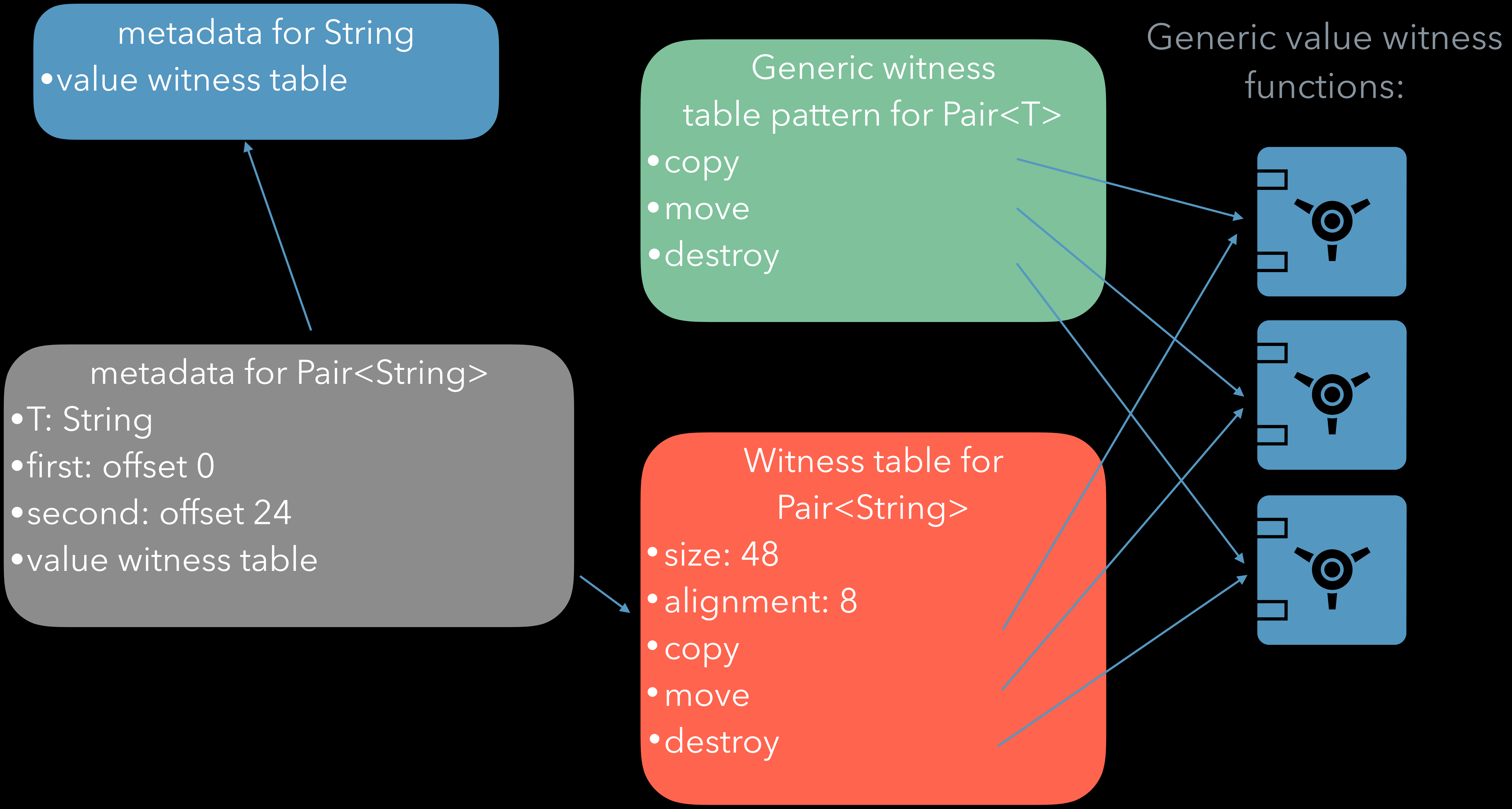
```
void getSecond(opaque *result, opaque *pair, type *T) {
  type *PairOfT = get_generic_metadata(&Pair_pattern, T);
  const opaque *second =
    (pair + PairOfT->fields[1]);
  T->vwt->copy_init(result, second, T);
  PairOfT->vwt->destroy(pair, PairOfT);
}
```

Value Witness Table for Pair<T>

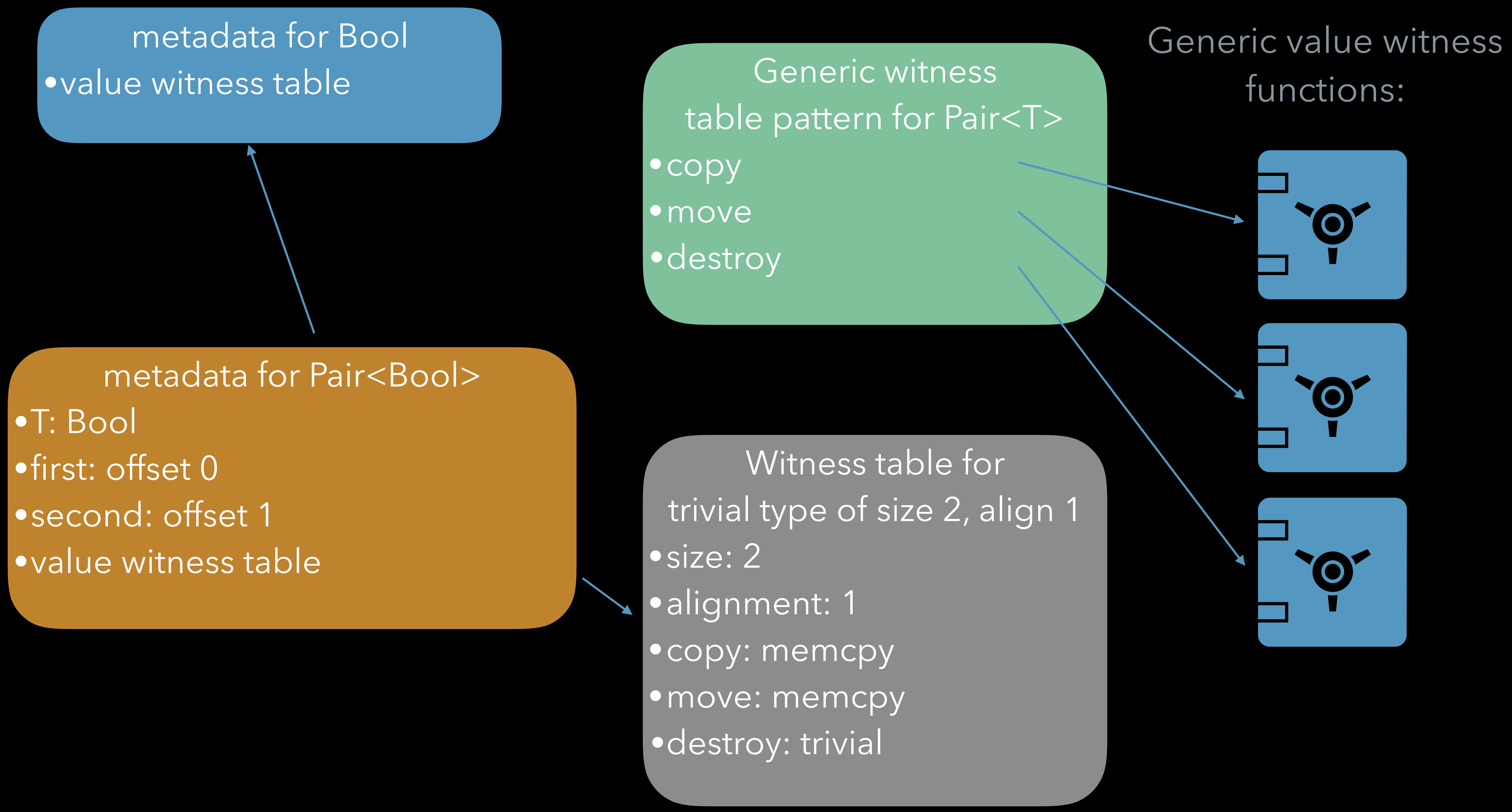


Generic value witness functions for Pair<T> are implemented in terms of value witness functions for T.

Value Witness Table for Pair<String>



Value Witness Table for Pair<Bool>



Value Witness Table for Pair<Bool>

metadata for UInt16

- value witness table

We *dynamically* detect generic type instantiations which are trivial (POD), and use simple value witnesses for them, instead of falling back to type-specific generic value witnesses.

metadata for Pair<Bool>

- T: Bool
- first: offset 0
- second: offset 1
- value witness table

Witness table for trivial type of size 2, align 1

- size: 2
- alignment: 1
- copy: memcpy
- move: memcpy
- destroy: trivial

Fully-substituted Case

```
func getSecond(_ pair: Pair<Int>) -> Int {
  return pair.second
}
```

- The layout of `Pair<Int>` is known at compile time
  - '`first`' is at offset 0
  - '`second`' is at offset 4
- We completely avoid passing type metadata, looking up field offsets, etc
- Optimizer generates specializations when source is available

Function Values

Higher-Order Functions

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Higher-Order Functions

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Higher-Order Functions

```
void apply(opaque *ret,
           opaque *value,
           ??? fn,
           type *T) {
  ...
}
```

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Higher-Order Functions

```
void apply(opaque *ret,
           opaque *value,
           ??? (*fn_invoke)(???, void *context),
           void *fn_context,
           type *T) {
  ...
}
```

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Higher-Order Functions

```
void apply(opaque *ret,
           opaque *value,
           void (*fn_invoke)(opaque *ret,
                             opaque *arg,
                             void *context),
           void *fn_context,
           type *T) {
  ...
}
```

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Higher-Order Functions

```
void apply(opaque *ret,
           opaque *value,
           void (*fn_invoke)(opaque *ret,
                             opaque *arg,
                             void *context),
           void *fn_context,
           type *T) {
  fn_invoke(ret, value, fn_context);
}
```

```
func apply<T>(value: T, fn: (T) -> T) -> T {
  return fn(value)
}
```

Concrete Application

```
apply(0, { $0 + 1 })
```

Concrete Application

```
apply(0, { $0 + 1 })
```

```
Int closure(Int $0) {
  return $0 + 1;
}

apply(..., closure, NULL, ...);
```

Concrete Application

```
apply(0, { $0 + 1 })
```

```
Int closure(Int $0) {
  return $0 + 1;
}

apply(..., closure, NULL, ...);

// We have: Int (*)(Int arg, void *ctxt)
// We need: void (*)(opaque *ret, opaque *arg,
//                   void *ctxt)
```

Options

- Compile all functions as if they were fully generic
- Use common representation for all values just in case
- ???

Options

- Compile all functions as if they were fully generic
- Use common representation for all values just in case
- Abstraction patterns

Abstraction Patterns

- One formal type, many lowered representations
- Need some way to agree on a representation:
  - Agreement only necessary when accessing a shared declaration
  - Derive abstraction pattern from generic pattern of that declaration
- Introduce thunks to translate between representations

Re-abstraction Thunks

```
apply(0, { $0 + 1 })

           Int closure(Int $0) {
             return $0 + 1;
           }

           apply(..., closure, NULL, ...);

           // We have: Int (*)(Int arg, void *ctxt)
           // We need: void (*)(opaque *ret, opaque *arg,
           //                   void *ctxt)
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {



}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Re-abstraction Thunks

```
Int closure(Int $0) {
  return $0 + 1;
}

void thunk(Int *ret, Int *arg, void *thunk_ctxt) {
  Int (*fn_invoke)(Int, void*) = thunk_ctxt->...;
  void *fn_context = thunk_ctxt->...;
  *ret = fn_invoke(*arg, fn_context);
}
void *thunk_ctxt = allocate(..., closure, NULL);

apply(..., thunk, thunk_ctxt, ...);
```

Constrained Generics

John McCall, Apple

Protocols Overview

Declarations

- Protocols declare a set of requirements on their conforming type

```
protocol Equatable {
  static func ==(l: Self, r: Self) -> Bool
}
```

Protocols Overview

Conformances

- Conformances define how a type conforms to a protocol

```
extension Person: Equatable {
  static func ==(l: Person, r: Person) -> Bool {
    return l.id == r.id
  }
}
```

Protocol Requirements

Associated types

- Protocol requirements sometimes need to refer to types other than the conforming type

```
protocol IteratorProtocol {
  mutating func next() -> ???
}
```

Protocol Requirements

Associated types

- Protocol requirements sometimes need to refer to types other than the conforming type

```
protocol IteratorProtocol {
  mutating func next() -> Optional<Element>
}
```

Protocol Requirements

Associated types

- Protocol requirements sometimes need to refer to types other than the conforming type

```
protocol IteratorProtocol {
  mutating func next() -> Optional<Element>

  associatedtype Element
}
```

Protocol Requirements

Associated conformances

- Protocols sometimes need to impose requirements on their associated types

```
protocol Sequence {
  func makeIterator() -> Iterator
  associatedtype Iterator
}
```

Protocol Requirements

Associated conformances

- Protocols sometimes need to impose requirements on their associated types

```
protocol Sequence {
  func makeIterator() -> Iterator
  associatedtype Iterator: IteratorProtocol
}
```

Protocol Requirements

Same-type requirements

- Protocols sometimes need to impose relationships between associated types

```
protocol Sequence {
  func makeIterator() -> Iterator
  associatedtype Iterator: IteratorProtocol
  associatedtype Element
}
```

Protocol Requirements

Same-type constraints

- Protocols sometimes need to state relationships between two associated types

```
protocol Sequence {
  func makeIterator() -> Iterator
  associatedtype Iterator: IteratorProtocol
  associatedtype Element
    where Element == Iterator.Element
}
```

Protocol Requirements

Same-type constraints

- Protocols sometimes need to state relationships between two associated types

```
protocol Sequence {
  func makeIterator() -> Iterator
  associatedtype Iterator: IteratorProtocol
  associatedtype Element
    where Element == Iterator.Element
}
```

- Creates redundant ways of identifying types

Protocol Requirements

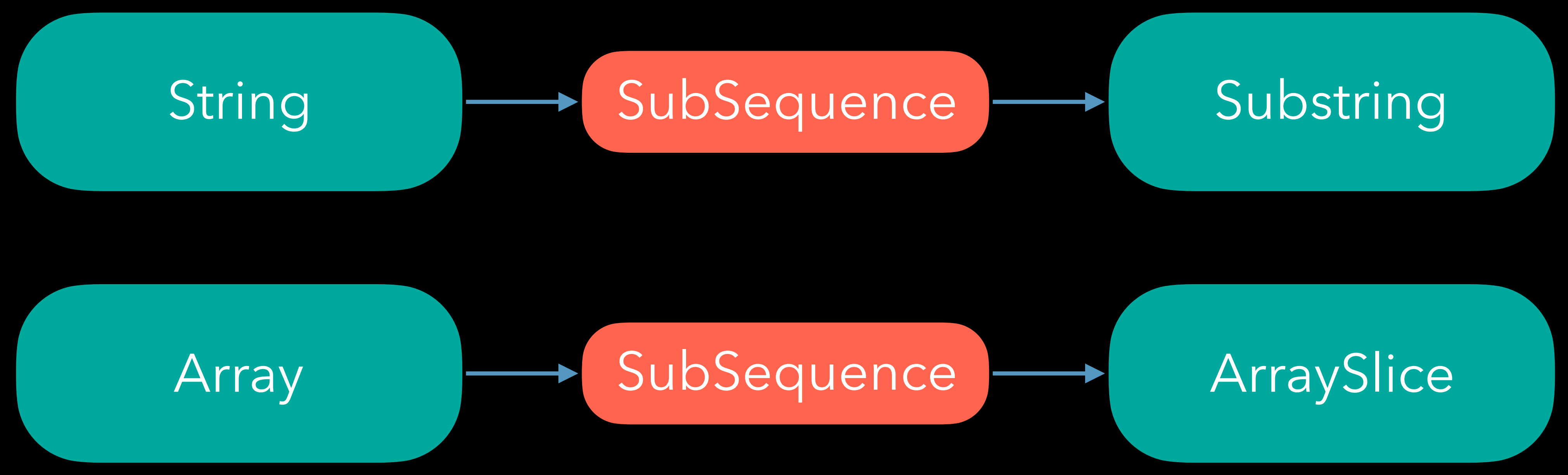
Recursive conformances

```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
}
```

Protocol Requirements

Recursive conformances

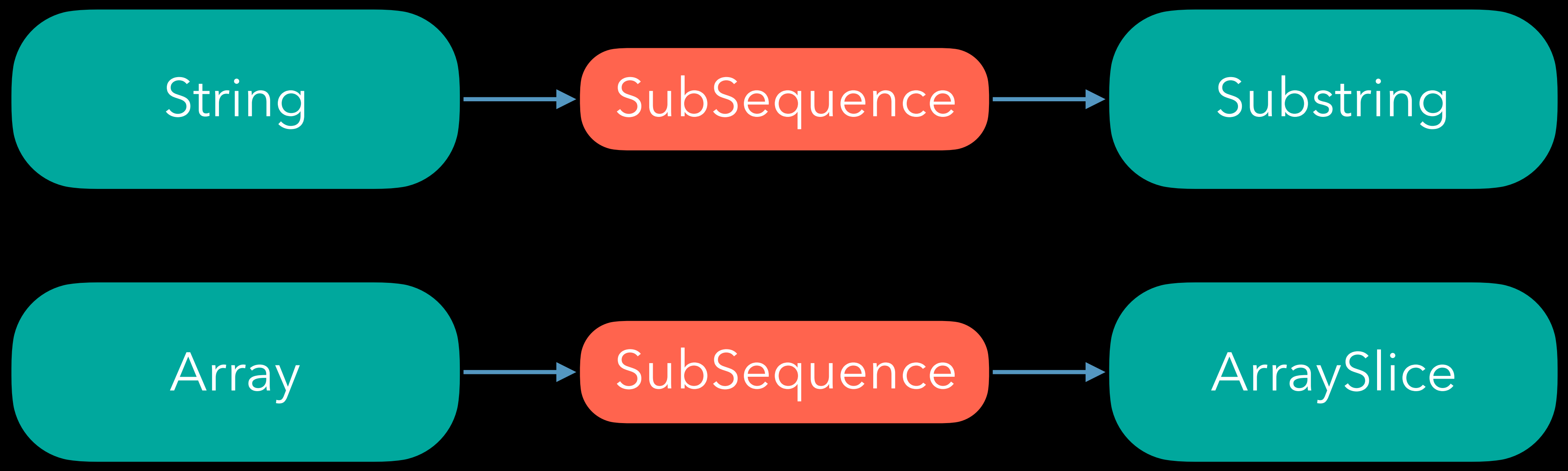
```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }

  subscript(r: Range<Int>) -> SubSequence { get }
  associatedtype SubSequence
}
```



Protocol Requirements

Recursive conformances

```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }

  subscript(r: Range<Int>) -> SubSequence { get }
  associatedtype SubSequence : Collection
    where SubSequence.Element == Element
}
```



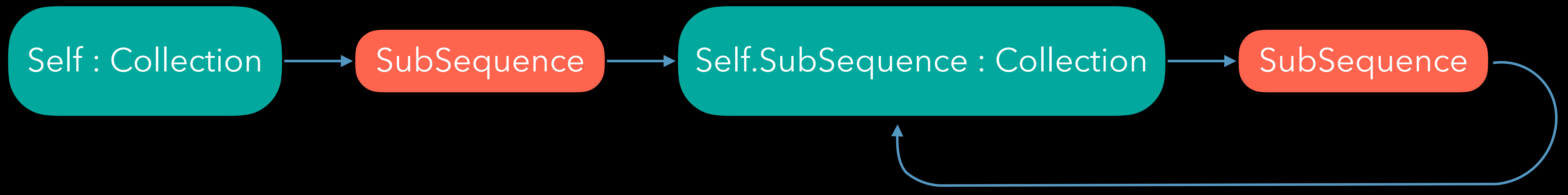
Protocol Requirements

Recursive conformances

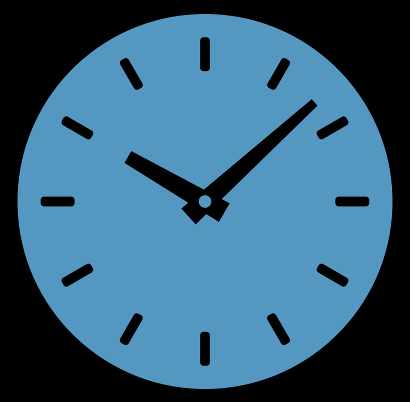
```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }

  subscript(r: Range<Int>) -> SubSequence { get }
  associatedtype SubSequence : Collection
    where SubSequence.Element == Element
}
```

Recursive conformances

```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }

  subscript(r: Range<Int>) -> SubSequence { get }
  associatedtype SubSequence : Collection
    where SubSequence.Element == Element,
          SubSequence.SubSequence == SubSequence
}
```



Compiling Constrained Generics



Type-Checking Generic Contexts

Example

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Type-Checking Generic Contexts

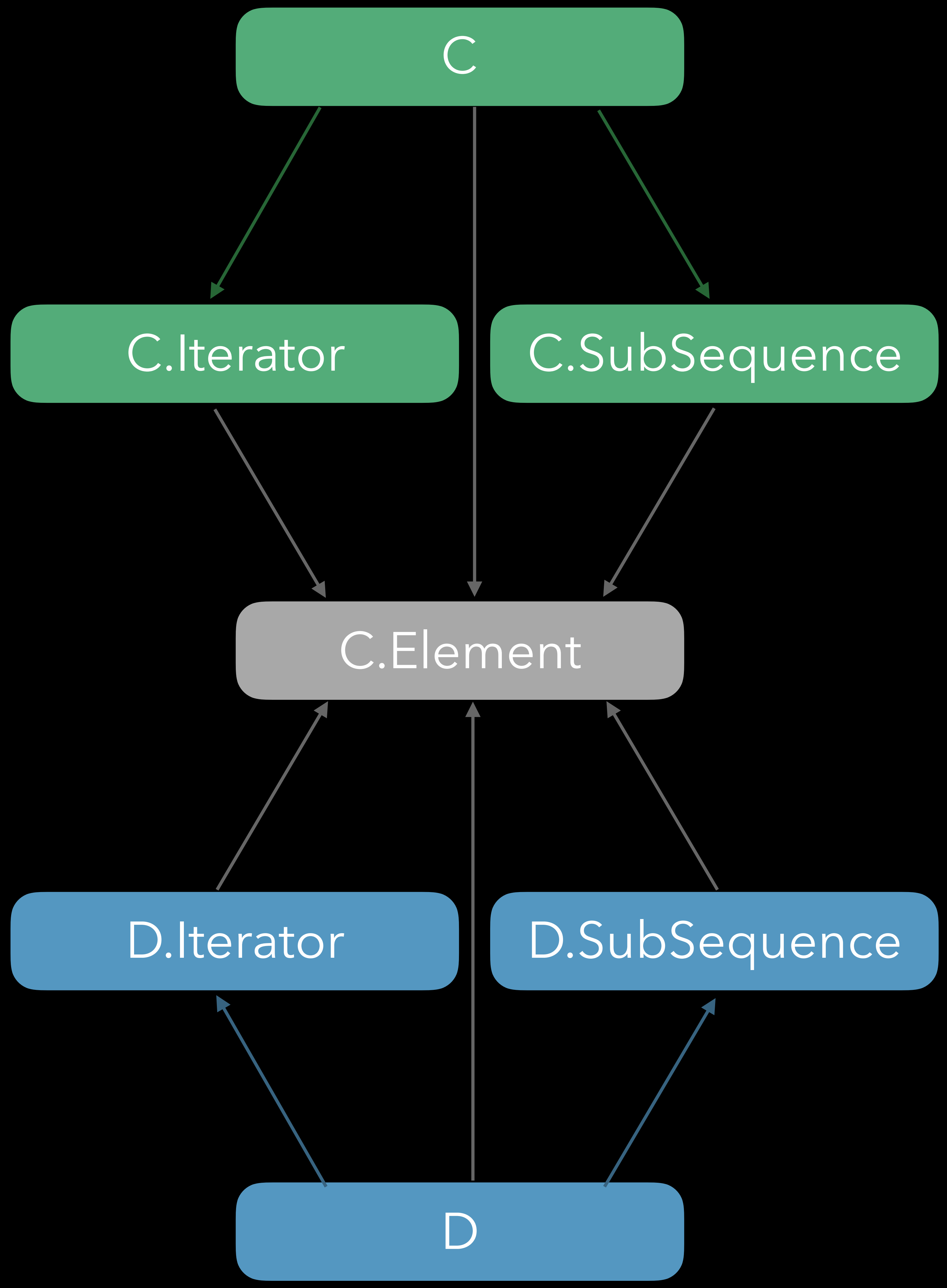
Generic signature checking

- What are the different unknown types?
- What do we know about those types in this context?
- Why do we know those things?

Type-Checking Generic Contexts

Unknown types

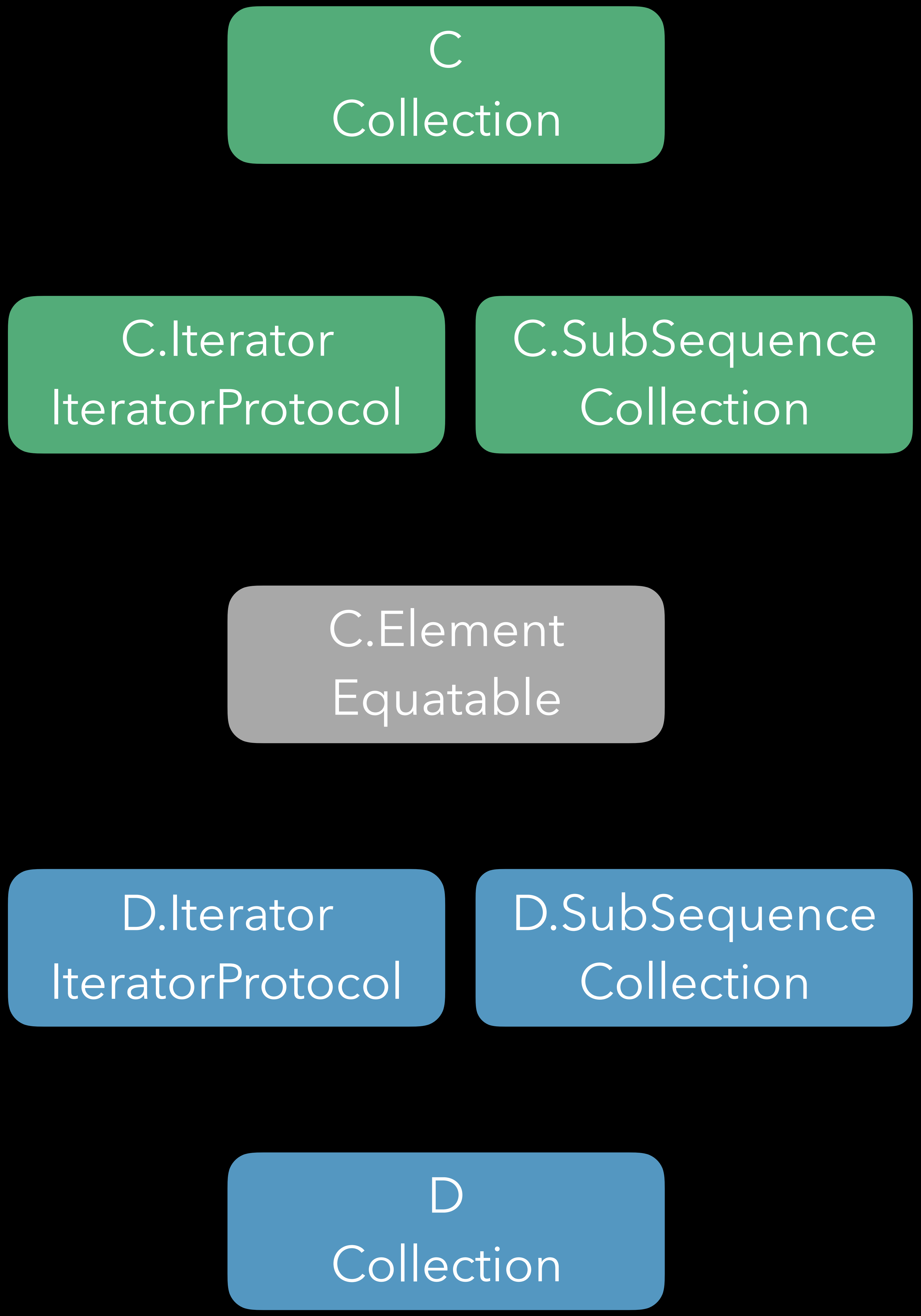
```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```



Type-Checking Generic Contexts

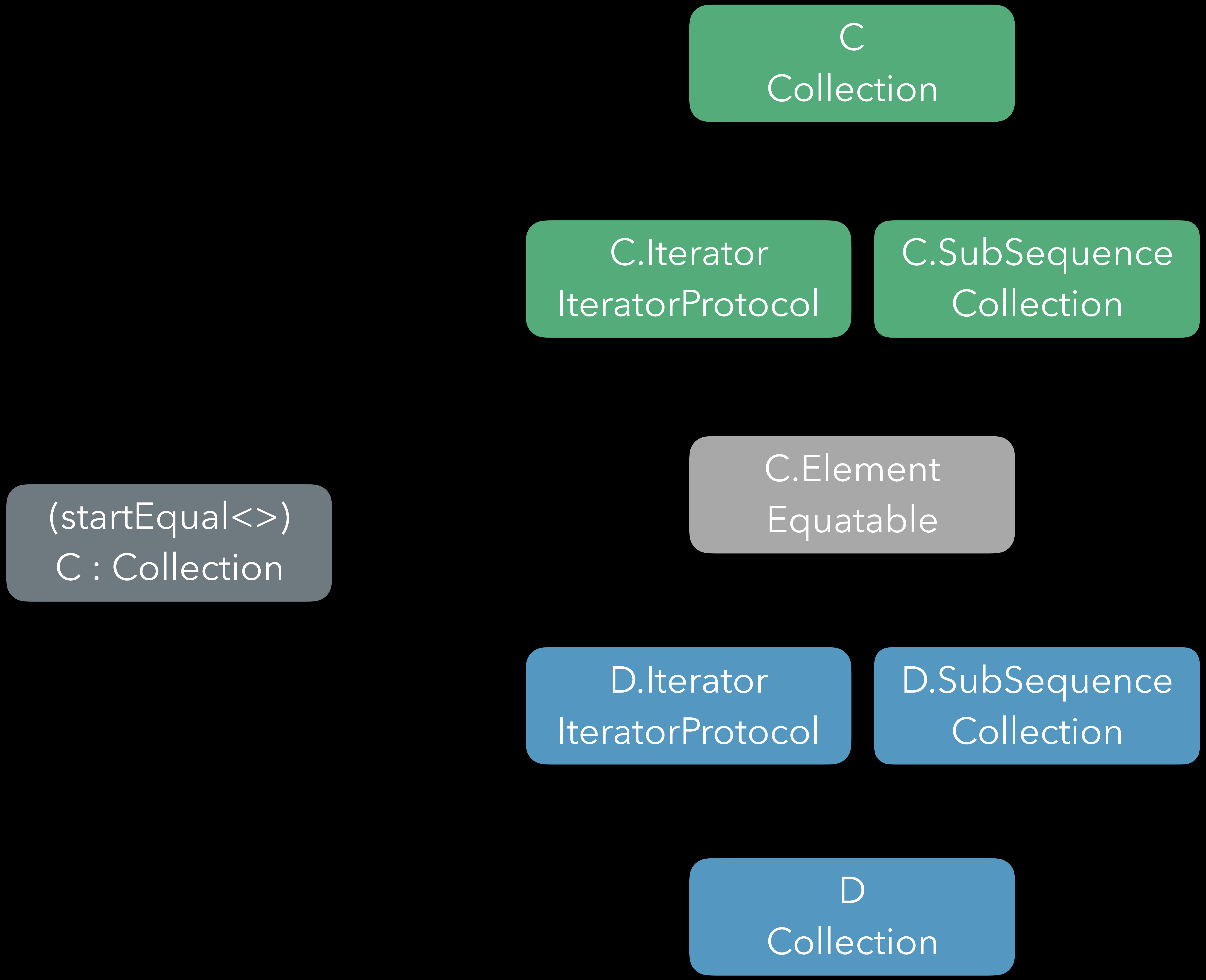
Requirements on unknown types

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```



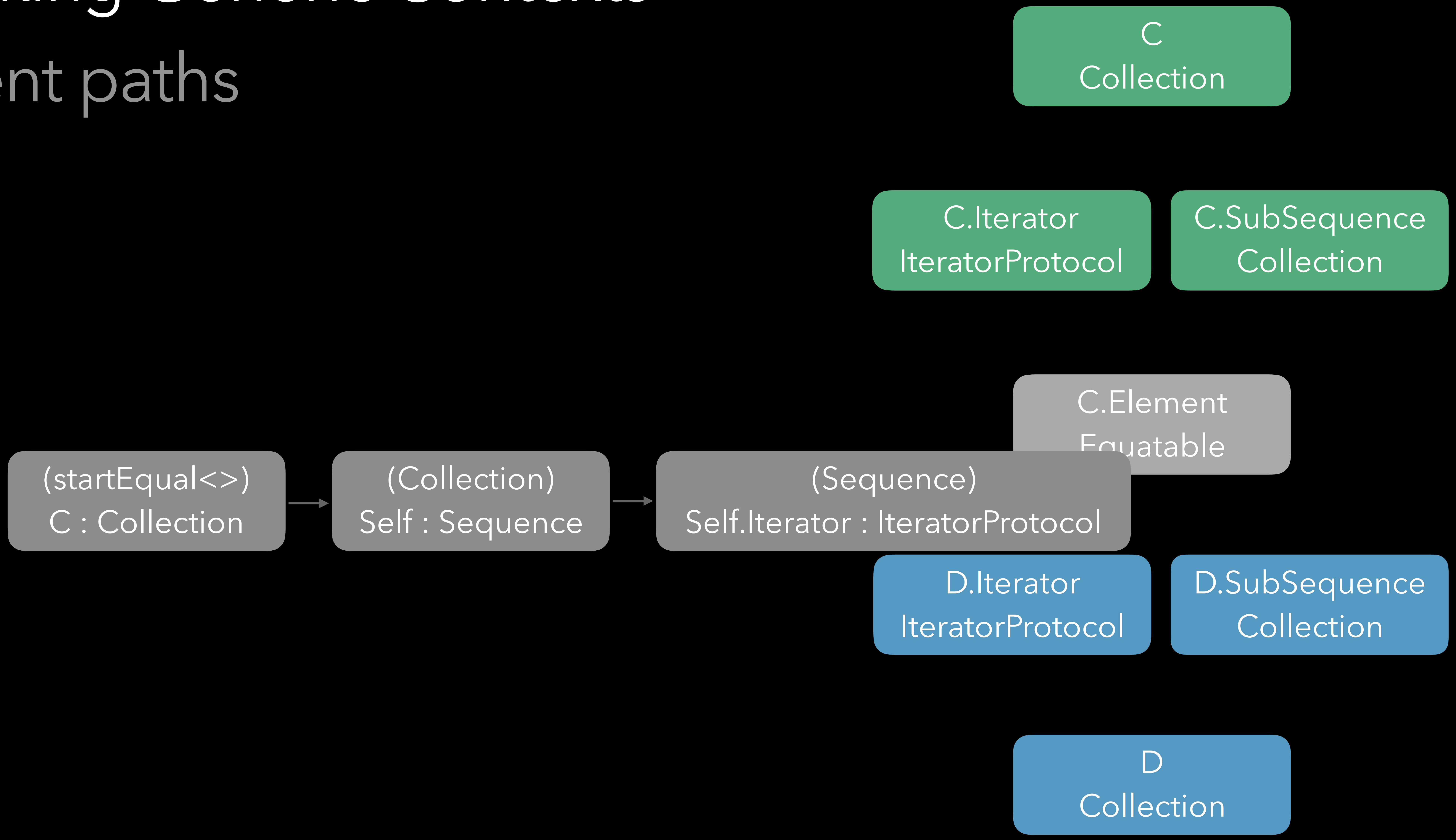
Type-Checking Generic Contexts

Requirement paths



Type-Checking Generic Contexts

Requirement paths



Type-Checking Generic Contexts

Canonicalized signature

- Primary type parameters:

```
<C, D>
```

- Canonicalized, minimized top-level requirements

```
C: Collection
D: Collection
C.Element : Equatable
C.Element == D.Element
```

Type-Checking Generic Contexts

Queries on canonicalized signatures

- Map a particular path to an unknown type into a canonical path

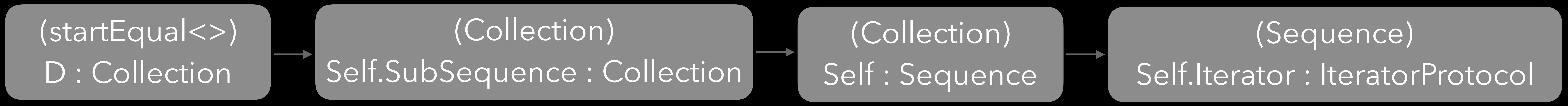
```
D.SubSequence.Iterator.Element => C.Element
```

- Map an unknown type to a set of requirements

```
D.SubSequence.Iterator.Element => { Equatable }
```

- Map a requirement to a requirement path

```
D.SubSequence.Iterator: IteratorProtocol =>
```



Constrained Generic Functions

Example

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Function Signature Lowering

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Function Signature Lowering

Formal parameters and results

```
bool startEqual(opaque *c, opaque *d);
```

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Function Signature Lowering

Type parameters

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D);
```

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Function Signature Lowering

Conformance parameters

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                ??? C_is_Collection,
                ??? D_is_Collection,
                ??? C_Element_is_Equatable);
```

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Protocol Witness Tables

- Runtime representation of a protocol conformance
- Like type metadata: created when needed, live forever
- Unlike type metadata, not guaranteed to be unique
- Like value witness tables, can sometimes shared between similar instances

Protocol Witness Tables

Requirement signature

```
protocol Collection : Sequence {
  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }

  subscript(r: Range<Int>) -> SubSequence { get }
  associatedtype SubSequence : Collection
    where SubSequence.Element == Element,
          SubSequence.SubSequence == SubSequence
}
```

Protocol Witness Tables

Requirement signature

```
protocol Collection {
  associatedtype SubSequence

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Witness table layout

```
struct CollectionWT {
  associatedtype SubSequence

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Associated type accessors

```
struct CollectionWT {
  associatedtype SubSequence

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Associated type accessors

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Base conformances

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Base conformances

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  where Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Associated conformances

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  where Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Associated conformances

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  CollectionWT *(SubSequence_is_Collection)(...);
  where Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Same-type constraints

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  CollectionWT *(SubSequence_is_Collection)(...);
  where Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Same-type constraints

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  CollectionWT *(SubSequence_is_Collection)(...);



  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Member requirements

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  CollectionWT *(SubSequence_is_Collection)(...);



  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Protocol Witness Tables

Member requirements

```
struct CollectionWT {
  type *(*SubSequence)(type *Self, CollectionWT *wt);

  SequenceWT *Self_is_Sequence;
  CollectionWT *(SubSequence_is_Collection)(...);



  Int (*startIndex)(opaque *self, type *Self, ...);
  Int (*endIndex)(opaque *self, type *Self, ...);
  void (*subscript_Int)(opaque *out, ...);
  void (*subscript_Range)(opaque *out, ...);
};
```

Function Signature Lowering

Conformance parameters

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                ??? C_is_Collection,
                ??? D_is_Collection,
                ??? C_Element_is_Equatable);
```

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Function Signature Lowering

Conformance parameters

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable);
```

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.startIndex] == d[d.startIndex]
}
```

Compiling Generic Functions

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  ...
}
```

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  ...
}
```

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  ...
}
```

(startEqual<>)
C : Collection

→

(Collection)
c.startIndex : Int

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Call c.startIndex

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  ...
}
```


```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Call c.subscript_Int

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  ...
}
```

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Call c.subscript_Int

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  ...
}
```

(startEqual<>)
C : Collection

→

(Collection)
c.subscript_Int : (Int) -> C.Element

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Call c.subscript_Int

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  ...
}
```

(startEqual<>)
C : Collection

→

(Collection)
c.subscript_Int : (Int) -> C.Element

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Allocate memory for a temporary C.Element

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  ...
}
```

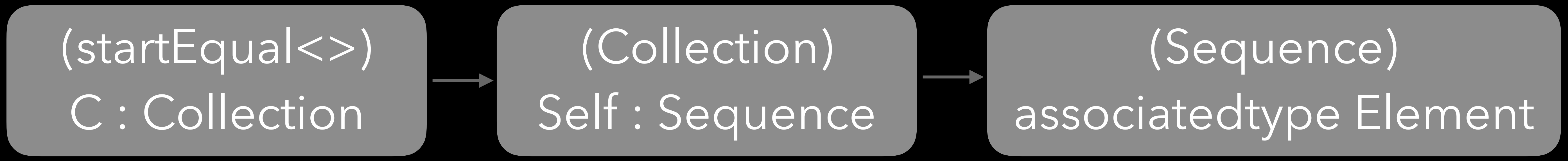
(startEqual<>) C : Collection → (Collection) Self : Sequence → (Sequence) associatedtype Element

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Get the protocol witness table for C: Sequence

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  SequenceWT *C_is_Sequence = C_is_Collection->Self_is_Sequence;
  ...
}
```

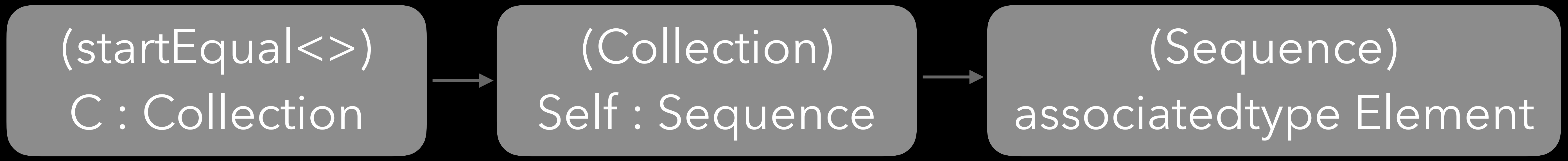



```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Get the type metadata for C.Element

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  SequenceWT *C_is_Sequence = C_is_Collection->Self_is_Sequence;
  type *Element = C_is_Sequence->Element(C, C_is_Sequence);
  ...
}
```

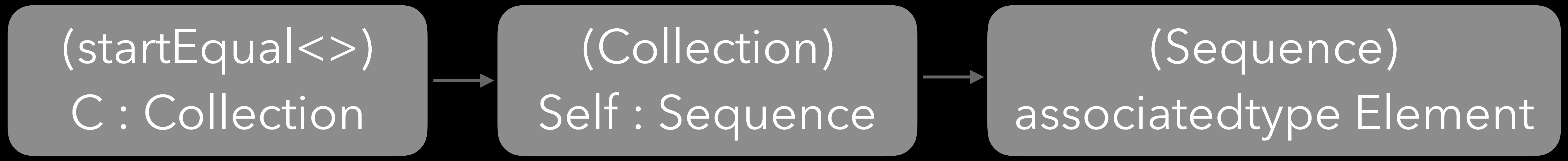



```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Allocate memory for a temporary C.Element

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  SequenceWT *C_is_Sequence = C_is_Collection->Self_is_Sequence;
  type *Element = C_is_Sequence->Element(C, C_is_Sequence);
  opaque *l = alloca(Element->vwt->size);
  ...
}
```




```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

Call c.subscript_Int

```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  SequenceWT *C_is_Sequence = C_is_Collection->Self_is_Sequence;
  type *Element = C_is_Sequence->Element(C, C_is_Sequence);
  opaque *l = alloca(Element->vwt->size);
  C_is_Collection->subscript_Int(l, c, cStart, C, C_is_Collection);
  ...
}
```

```
return c[c.startIndex] == d[d.startIndex]
```

Compiling Generic Functions

How far have we gotten?

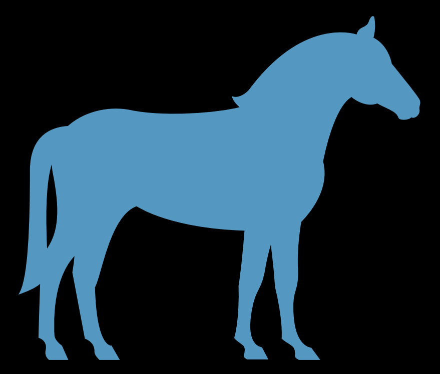
```
bool startEqual(opaque *c, opaque *d,
                type *C, type *D,
                CollectionWT *C_is_Collection,
                CollectionWT *D_is_Collection,
                EquatableWT *C_Element_is_Equatable) {
  Int cStart = C_is_Collection->startIndex(c, C, C_is_Collection);
  SequenceWT *C_is_Sequence = C_is_Collection->Self_is_Sequence;
  type *Element = C_is_Sequence->Element(C, C_is_Sequence);
  opaque *l = alloca(Element->vwt->size);
  C_is_Collection->subscript_Int(l, c, cStart, C, C_is_Collection);
  ...
}
```

```
return c[c.startIndex] == d[d.startIndex]
```

Conclusions

- Core design goal: be able to run generic code
- Generic code can be specialized *as an optimization*
  - Interoperation is key
  - No major compromises for specialized code

Questions?



Postmatter

More about abstraction patterns

Examples

```
struct MyTransform<T,U> {
  let fn: (T) -> U
}

// struct MyTransform {
//   void (*fn_invoke)(opaque *ret,
//                     opaque *arg,
//                     void*);
//   void *fn_context;
// };
```

Examples

```
struct MyPredicate<T> {
  let fn: (T) -> Bool
}

// struct MyPredicate {
//   bool (*fn_invoke)(opaque *arg, void*);
//   void *fn_context;
// };
```

Examples

```
struct MyGenerator<T> {
  let fn: () -> T
}

// struct MyGenerator {
//   void (*fn_invoke)(opaque *ret, void*);
//   void *fn_context;
// };
```

Substitution

```
struct MyValue<T> {
  let v: T
}
let fn: MyValue<(Int) -> Int>
```

How is this function value represented in a context that doesn't know it's storing a function?

Substitution

```
let fn: MyValue<(Int) -> Int>

func useFnValue1<T>(_: MyValue<(Int) -> Int>)
useFnValue1(fn)

func useFnValue2<T>(_: MyValue<(T) -> Int>)
useFnValue2(fn)

func useFnValue3<T, U>(_: MyValue<(T) -> U>)
useFnValue3(fn)
```

Substitution

- Could recursively reabstract all generic types
  - Very expensive
  - Requires us to know all the members statically, which would break library evolution plans
  - Not possible for reference types without breaking semantics

Most General Representation

- Only recursively reabstract specific structural types: tuples, functions, optionals
- Otherwise types have a single representation, with member declaration determining member representation
- Multi-representation types substituted for generic parameters use *most general representation*

Most General Representation

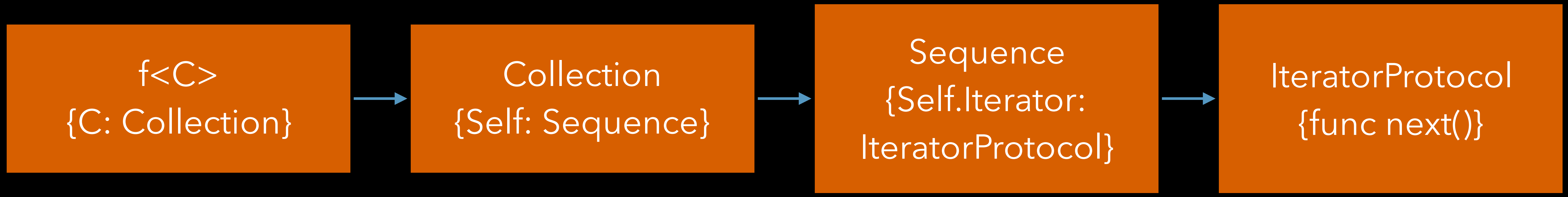
- Multi-representation types substituted for generic parameters use *most general representation*
- Determined as if every substitutable position within the type was an unconstrained generic parameter:
  - `(Int, Int) -> Int` ==> `(T, U) -> V`
  - i.e. `void (Int *out, Int *arg1, Int *arg2)`
- Note representation preserved under partial substitution

Type-checking Protocol Conformances

Requirement Paths

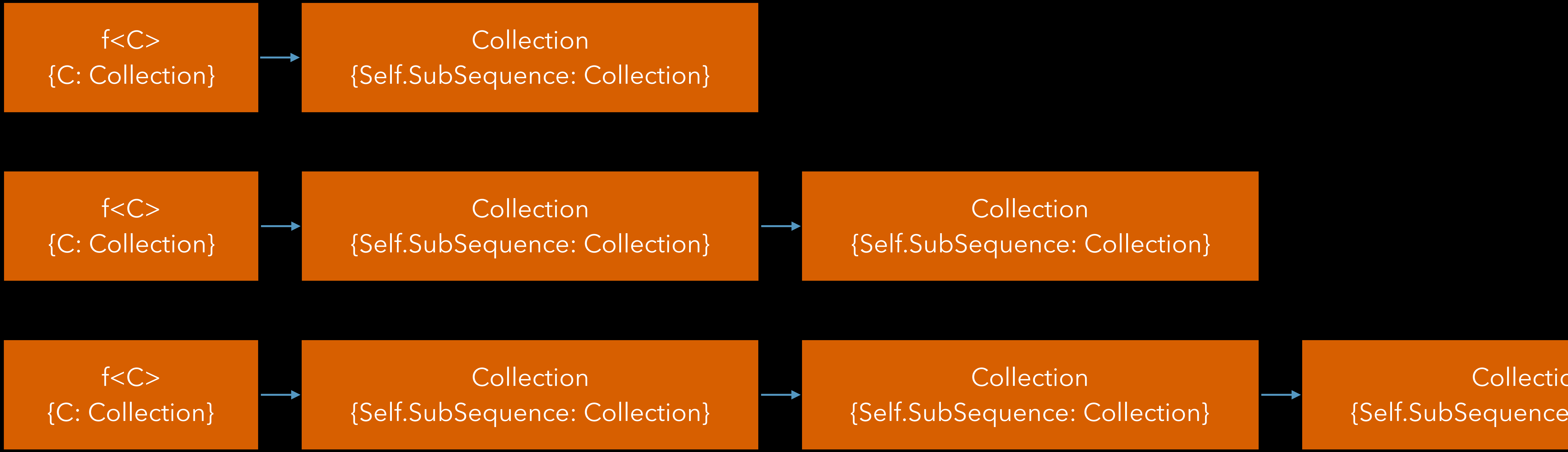
```
func f<C: Collection>(c: C) {
  let firstValue = c.makeIterator().next()!
}
```


```
func f<C: Collection>(c: C) {
  let firstValue = c.makeIterator().next()!
}
```

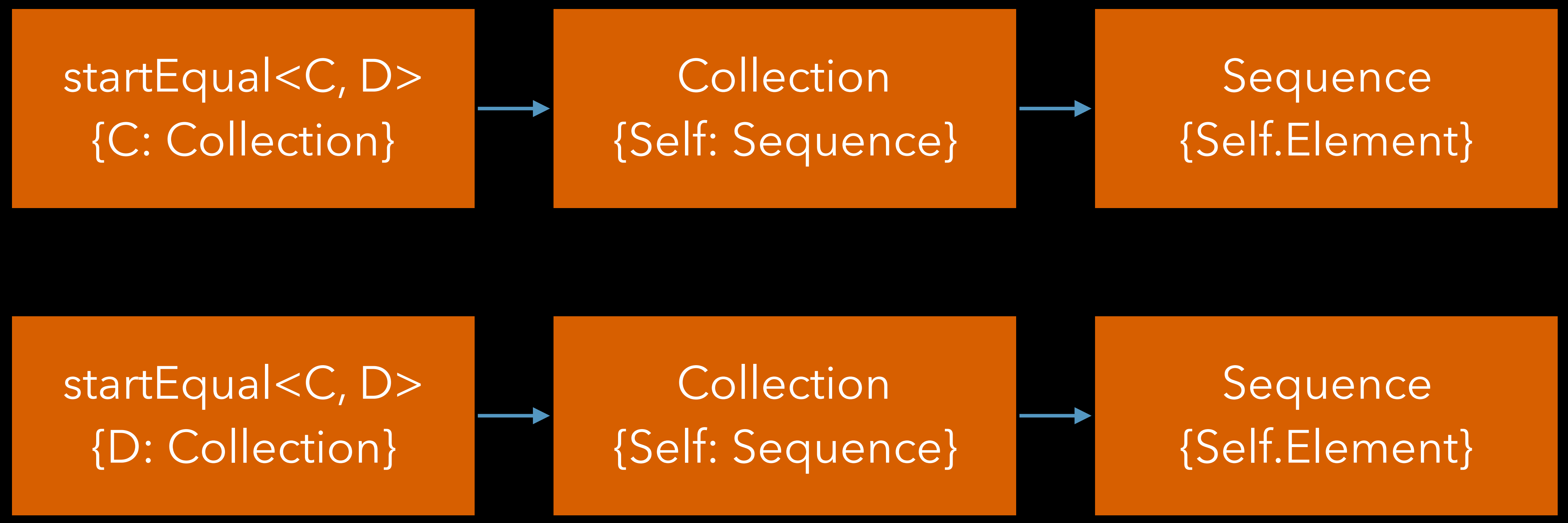
Requirement Paths

```
func f<C: Collection>(c: C) {
  let slice: C.SubSequence = c[0..<10]
}
```



Canonicalization

```
func startEqual<C, D>(c: C, d: D) -> Bool
    where C: Collection, D: Collection,
          C.Element == D.Element,
          C.Element : Equatable {
  return c[c.beginIndex] == d[d.beginIndex]
}
```



Type-Checking Generic Signatures

- Collect all the constraints
- Check for problems
- Minimize and canonicalize

Type-Checking
Protocol Conformances

```
extension Array<T> : Collection {
  var startIndex: Int { return 0 }
  var endIndex: Int { return count }
  subscript(i: Int) -> T { ... }
  subscript(r: Range<Int>) -> ArraySlice<T> { ... }
}
```

Type-Checking Protocol Conformances

- Start with the minimized requirements of the protocol:

```
protocol Collection {
  associatedtype SubSequence

  where Self: Sequence,
        Self.SubSequence: Collection,
        Self.SubSequence.Element == Self.Element,
        Self.SubSequence.SubSequence == Self.SubSequence

  var startIndex: Int { get }
  var endIndex: Int { get }
  subscript(i: Int) -> Element { get }
  subscript(r: Range<Int>) -> SubSequence { get }
}
```

Type-Checking
Protocol Conformances

- Determine the concrete associated types

```
extension Array<T> : Collection {
  var startIndex: Int { return 0 }
  var endIndex: Int { return count }
  subscript(i: Int) -> T { ... }
  subscript(r: Range<Int>) -> ArraySlice<T> { ... }
}
```

Type-Checking Protocol Conformances

- Determine the concrete associated types

```
extension Array<T> : Collection {
  var startIndex: Int { return 0 }
  var endIndex: Int { return count }
  subscript(i: Int) -> T { ... }
  subscript(r: Range<Int>) -> ArraySlice<T> { ... }

  typealias Element = T
  typealias SubSequence = ArraySlice<T>
}
```

Type-Checking Protocol Conformances

- Check type requirements
- Recurse on conformance constraints:

```
Self: Sequence
Self.SubSequence: Collection
```

- Check equality constraints:

```
Self.SubSequence.Element == Self.Element
Self.SubSequence.SubSequence == Self.SubSequence
```

Type-Checking Protocol Conformances

- Look for matching value requirements:

```
var startIndex: Int { get }
var endIndex: Int { get }
subscript(i: Int) -> Element { get }
subscript(r: Range<Int>) -> SubSequence { get }
```

Algorithms?

- Type-checking problems are… exciting
- Associated type hierarchies can be infinite
  - Must validate *enough* constraints to be correct
  - Must not validate *too many* constraints or won't halt