C++ now

Generic Arity

Definition-Checked Variadics in Carbon

Geoffrey Romer

2024

Definition checking:

Type checking the definition

Definition checking:

Type checking the definition
Not the instantiations

Overview

- Carbon background
- Variadic syntax and semantics
- Type checking variadic code
- Type checking variadic function calls

Carbon background

The Carbon Language

- Carbon is an experimental successor language for C++

JavaScript → TypeScript

Java → Kotlin

C++ → **Carbon**

  - New language, not extension or superset of C++
  - High-fidelity interoperability with C++, and migration from C++
  - Permits design changes that are hard to do incrementally
- Experimental questions:
  - Can we build a C++ successor that meets our goals?
  - Can it reach a critical mass of community/industry interest?

⚠️ Under construction ⚠️

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool) {
  let limit: i32 = Math.Sqrt(n) as i32;
  var i: i32 = 2;
  while (i <= limit) {
    if (n % i == 0) {
      return (i, false);
    }
    i += 1;
  }
  return (n, true);
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool);

fn IsPrime(n: i32) {
  let (factor: i32, has_factor: bool) = SmallestFactor(n);
  return !has_factor;
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool);

fn IsPrime(n: i32) {
  let (factor: i32, has_factor: bool) = SmallestFactor(n);
  return !has_factor;
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool);

fn IsPrime(n: i32) {
  let (factor: i32, has_factor: bool) = SmallestFactor(n);
  return !has_factor;
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool);

fn IsPrime(n: i32) {
  let (factor: i32, has_factor: bool) = SmallestFactor(n);
  return !has_factor;
}
```

Carbon Crash Course

```
fn SmallestFactor(n: i32) -> (i32, bool);

fn IsPrime(n: i32) {
  let (factor: i32, has_factor: bool) = SmallestFactor(n);
  return !has_factor;
}
```

Carbon Crash Course

```
fn GenericSum[T:! Core.Add](x: T, y: T) -> T {
  return x + y;
}
```

Carbon Crash Course

```
fn GenericSum[T:! Core.Add](x: T, y: T) -> T {
  return x + y;
}
```

Carbon Crash Course

```
fn GenericSum[T:! Core.Add](x: T, y: T) -> T {
  return x + y;
}
```

Carbon Crash Course

```
fn Print[P:! Printable](arg: P);

fn GenericSum[T:! Core.Add](x: T, y: T) -> T {
  Print(x);
  return x + y;
}
```

Carbon Crash Course

```
fn Print[P:! Printable](arg: P);

fn GenericSum[T:! Core.Add](x: T, y: T) -> T {
  Print(x);
  return x + y;
}
```

❌ Error: `T` might not implement `Printable`

Carbon Crash Course

```
fn Print[P:! Printable](arg: P);

fn GenericSum[template T:! Core.Add](x: T, y: T) -> T {
  Print(x);
  return x + y;
}
```

Carbon Variadics

Requirements For Carbon Variadics

Goals:

- Support definition checking in variadic functions
- Support incremental migration from C++, anywhere in the call stack
- Prioritize iteration over recursion
- Address other C++ pain points

Requirements For Carbon Variadics

Constraint:

- Type checking should be efficient

Syntax and Semantics

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- `param` is a *pack* of all the arguments, which have type `i64`.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- `...` marks a *pack expansion*.
- A pack expansion is a kind of compile-time loop.
- There are several kinds, depending on context.

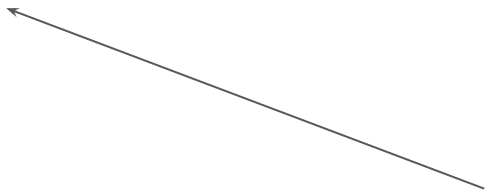
Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- A *pack expansion statement* is a pack expansion in a statement context.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```



- Consists of `...` followed by a statement called the *body*.
- Behaves like a loop: executes the body once for each element of the pack

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- An *each-name* is `each` followed by a pack name.
- Behaves like an iterator over the pack: on the Nth iteration, refers to Nth element.
- No way of referring to the pack as a whole.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64 {
  var sum: i64 = 0;
  ... sum += each param;
  return sum;
}
```

- Loops over the elements of `param`, adding each of them to `sum`.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```

- A pack expansion *expression* is a pack expansion in an expression context.
- Specifically, a tuple literal context

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```

- Consists of `...` followed by an expression called the *body*.
- Behaves like a loop: evaluates the body once for each element of the pack
  - And adds the values to the tuple

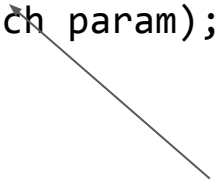
Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```

- A pack expansion *pattern* is a pack expansion in a pattern context.
- Specifically, a tuple pattern context.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```



- Consists of `...` followed by a pattern called the *body*.
- Behaves like a loop: matches the body against each element of the tuple being matched

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```

- Declares each element of `param` with type `i64`.
- Matches any value that's convertible to `i64`
- Behaves like an output iterator: appends matched values to the pack `param`.

Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```

- Takes an arbitrary number of arguments, all convertible to `i64`.
- Declares pack `param`, with all elements having type `i64`.
- Initializes elements of `param` with converted arguments.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

- The type of each element of `param` is the corresponding element of `T`.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

```
(... ((each param): (each T)))
```

- Each argument must be convertible to the corresponding element of `T`.
- Each element of `param` is initialized from the corresponding converted argument.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

- T is a deduced parameter, so "use" of T is actually deducing its contents.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

- Takes an arbitrary number of arguments, with types that implement `StringLike`.
- Binds `param` to the arguments, and `T` to their types.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

- These are calling methods on `each param`, not applying `each` to the result of a method call.

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

Syntax and Semantics

```
template <StringLike... Ts>
std::string StrCat(const Ts&... params) {
  std::string result;
  result.reserve((params.Length() + ... + 0));
  StrCatImpl(&result, params...);
  return result;
}

void StrCatImpl(std::string* out) { return; }

template <StringLike T, StringLike... Ts>
void StrCatImpl(std::string* out, const T& first, const Ts&... rest) {
  out->append(first.ToString());
  StrCatImpl(out, rest...);
}
```

Syntax and Semantics

```
template <StringLike... Ts>
std::string StrCat(const Ts&... params) {
  std::string result;
  result.reserve((params.Length() + ... + 0));
  (result.append(params.ToString()), ...);
  return result;
}
```

Syntax and Semantics

```
// Concatenates its arguments, which are all convertible to String
fn StrCat[... each T:! StringLike](... each param: each T) -> String {
  var len: i64 = 0;
  ... len += each param.Length();
  var result: String = "";
  result.Reserve(len);
  ... result.Append(each param.ToString());
  return result;
}
```

Syntax and Semantics

```
// Returns the minimum of its arguments, which must all have the same type T.
fn Min[T:! Ordered & Copyable](first: T, ... each next: T) -> T {
  var result: T = first;
  ... if (each next < result) {
    result = each next;
  }
  return result;
}
```

Syntax and Semantics

```
// Returns the minimum of its arguments, which must all have the same type T.
fn Min[T:! Ordered & Copyable](first: T, ... each next: T) -> T {
  var result: T = first;
  ... if (each next < result) {
    result = each next;
  }
  return result;
}
```

- T is deduced from each argument.
- All deductions must deduce identical types.

Syntax and Semantics

```
// Returns the minimum of its arguments, which must all have the same type T.
fn Min[T:! Ordered & Copyable](first: T, ... each next: T) -> T {
  var result: T = first;
  ... if (each next < result) {
    result = each next;
  }
  return result;
}
```

- Singular parameters can appear before and/or after a variadic parameter
- Can't have more than one variadic parameter

Syntax and Semantics

```
// Returns the minimum of its arguments, which must all have the same type T.
fn Min[T:! Ordered & Copyable](first: T, ... each next: T) -> T;
```

```
Min(... each x);                    ❌ Error: `first` might not match
Min(100, ... each x);               ✅
Min(... each x, 100);               ✅
Min(... each x, ... each y, 100);   ✅
```

Syntax and Semantics

Each-name: `each <name>`

- Refers to the Nth element of a pack

Pack expansion: `... <body>`

- Compile-time loop, with `<body>` as the loop body
- Expression expansion `(... each param * each param)`
  - Each-names in `<body>` → tuple elements
- Statement expansion `... sum += each param;`
  - Each-names in `<body>` → side effects
- Pattern expansion `(... each param: i32) = (1, 2, 3)`
  - Tuple elements → Each-names in `<body>`

Type checking pack expansions

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
      i32,
      ???,
      f32
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

`i32`,
`|param|` repetitions of `i64`,
`f32`

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
      i32,
      <i64; |param|>,
      f32
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
    x:      {<i32; 1>, <i64; |param|>, <f32; 1>}
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
    x:      {<i32; 1>, <i64; |param|>, <f32; 1>}
```

- This is not Carbon syntax!

Type checking pack expansions

```
fn F[... each T: Printable](... each param: each T) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(each x);
}
```

Type checking pack expansions

```
fn F[... each T: Printable](... each param: each T) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(each x);
}
```

```
    x:      {i32, ... each T, f32}
```

Type checking pack expansions

```
fn F[... each T: Printable](... each param: each T) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(each x);
}
```

```
    x:      {<i32; 1>, <each T; |param|>, <f32; 1>}
```

Type checking pack expansions

```
fn F[... each T: Printable](... each param: each T) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(each x);
}
```

Type checking pack expansions

```
fn F[... each T: Printable](... each param: each T) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(each x);
}
```

```
    (<i32; 1>, <each T; |param|>, <f32; 1>)
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
    x:          {<i32; 1>, <i64; |param|>, <f32; 1>}
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param, 1.0 as f32);
  ... Print(2 * each x);
}
```

```
    x:           {<i32; 1>, <i64; |param|>, <f32; 1>}
    2 * each x:  {<i32; 1>, <i64; |param|>, <f32; 1>}
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param);
  let (... each y: auto) = (1 as f64, ... each param);
  ... Print(each x * each y);
}
    x:          {<i32, 1>, <i64, |param|>}
    y:          {<f64, 1>, <i64, |param|>}

    each x * each y: ???
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param);
  let (... each y: auto) = (1 as f64, ... each param);
  ... Print(each x * each y);
}
    x:          {<i32, 1>, <i64, |param|>}
    y:          {<f64, 1>, <i64, |param|>}

    each x * each y: {<f64, 1>,
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param);
  let (... each y: auto) = (1 as f64, ... each param);
  ... Print(each x * each y);
}
    x:          {<i32, 1>, <i64, |param|>}
    y:          {<f64, 1>, <i64, |param|>}

    each x * each y: {<f64, 1>, <i64, |param|>}
```

Type checking pack expansions

- Pack and tuple types are sequences of *segments*
  - A segment consists of an expression and an arity
  - Lets us represent repetition and arity explicitly but generically
- Inside a pack expansion, all expressions and patterns have pack types
  - And they must all have the same shape
- Typecheck by looping over the segments of the pack types

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: auto) = (1 as i32, ... each param);
  let (... each y: auto) = (1 as f64, ... each param);
  ... Print(each x * each y);
}
```

Type checking pack expansions

```
fn F(... each param: i64) {
  let (... each x: i64) = (1 as i32, ... each param);
  let (... each y: i64) = (1 as f64, ... each param);
  ... Print(each x * each y);
}
```

❌

Type checking pattern matching

Type checking function calls

Type checking function calls: easy cases

```
fn IntSum(... each param: i64) -> i64;

fn Average(... each x: i32) ->  i32 {
  let sum: i64 = IntSum(... each x);
  return (sum as f64)/Count(... each x);
}
```

Type checking function calls: easy cases

```
fn IntSum(... each param: i64) -> i64;

fn Average(... each x: i32) ->  i32 {
  let sum: i64 = IntSum(... each x);
  return (sum as f64)/Count(... each x);
}
```

```
{<i64; |param|>}


{<i32; |x|}
```

Type checking function calls: easy cases

```
fn IntSum(... each param: i64) -> i64;

fn Average(first: i32, ... each next: i32) -> {
  let sum: i64 = IntSum(first, ... each next);
  return (sum as f64)/(1 + Count(... each next));
}
```

```
{<i64; |param|>}


{<i32; 1>, <i32; |next|>}
```

Type checking function calls: easy cases

```
fn StrCat[... each T:! StringLike]
    (... each param: each T) -> String;

fn PrintAll[... each U:! StringLike]
    (... each x: each U) {
  Print(StrCat(... each x));
}
```

```
{<StringLike; |param|}
{<each T; |param|>}



{<each U; |x|>}
```

Type checking function calls: easy cases

```
fn StrCat[... each T:! StringLike]
    (... each param: each T) -> String;

fn PrintAll[... each U:! StringLike]
    (... each x: each U) {
  Print(StrCat("Everything: ", ... each x));
}
```

```
T: {<StringLike; |param|}
param: {<each T; |param|>}



{<String; 1>, <each U; |x|>}
```

Type checking function calls: easy cases

```
fn StrAppend[... each T:! StringLike]
    (out: String* ... each param: each T);

fn StrCat[... each U:! StringLike]
    (... each x: each U) -> String {
  returned var result: String;
  StrAppend(&result, ... each x);
  return result;
}
```

```
{<String*; 1>, <each T; |param|>}

{<String*; 1>, <each U; |x|>}
```

Type checking function calls: easy cases

```
fn Min[T:! Comparable & Value]
    (first: T, ... each next: T) -> T;

fn Normalize(... each x: f64*) {
  let min: f64 = Min(... *each x);
  ... *each x -= min;
}
```

❌

```
{<T; 1>, <T; |next|>}

{<f64; |x|>}
```

Type checking function calls: easy cases

```
fn Min[T:! Comparable & Value]
    (first: T, ... each next: T) -> T;

fn Normalize(... each x: f64*) {
  let min: f64 = Min(... *each x);
  ... *each x -= min;
}
```

❌

```
    1 + |next| = |x|
```

```
{<T; 1>, <T; |next|>}

{<f64; |x|>}
```

Type checking function calls: easy cases

```
fn Min[T:! Comparable & Value]
    (first: T, ... each next: T) -> T;

fn Normalize(... each x: f64*) {
  let min: f64 = Min(... *each x);
  ... *each x -= min;
}
```

❌

```
    1 + |next| = |x|
    |next| = |x| - 1
```

```
{<T; 1>, <T; |next|>}

{<f64; |x|>}
```

Type checking function calls: easy cases

```
fn Min[T:! Comparable & Value]
    (first: T, ... each next: T) -> T;

fn Normalize(... each x: f64*, y: f64*) {
  let (... each arg: f64*) = (... each x, y);
  let min: f64 = Min(... *each arg);
  ... *each arg -= min;
}
```

```
{<T; 1>, <T; |next|>}


{<f64; |x|+1>}
```

Type checking function calls: easy cases

```
fn Min[T:! Comparable & Value]
    (first: T, ... each next: T) -> T;

fn Normalize(... each x: f64*, y: f64*) {
  let (... each arg: f64*) = (... each x, y);
  let min: f64 = Min(... *each arg);
  ... *each arg -= min;
}
```

```
1+|next| = |x|+1
|next| = |x|
```

```
{<T; 1>, <T; |next|>}
```

```
{<f64; |x|+1>}
```

Type checking function calls: easy cases

- Each argument has only one parameter that can match it
  - This happens if each singular parameter has a corresponding singular argument
- Each parameter has only one argument that it can match
  - This happens if each singular argument has a corresponding singular parameter

Type checking function calls: hard cases

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));

fn F[... each T:! type](... each x: Vector(each T), y: Vector(i32)) {
  var z: auto = Zip(... each x, y);
}
```

Type checking function calls: hard cases

```
fn F[X:! I, ... each Y:! I](x: X, ... each y: each Y);

fn G[... each A:! I, B:! I](... each a: each A, b: each B) {
  F(... each a, b);
}
```

	X	Y
A	✅	✅
B	✅	✅

Type checking function calls: hard cases

```
fn F[W:! I, X:! I, ... each Y:! I](w: W, x: X, ... each y: each Y);

fn G[... each A:! I, B:! I, C:! I](... each a: each A, b: B, c: C) {
  F(... each a, b, c);
}
```

	W	X	Y
A	✅	✅	✅
B	✅	✅	✅
C	❌	✅	✅

Type checking function calls: hard cases

```
fn F[V:! I, W:! I, X:! I, ... each Y:! I](v: V, w: W, x: X, ... each y: each Y);

fn G[... each A:! I, B:! I, C:! I, D:! I](... each a: each A, b: B, c: C, d: D) {
  F(... each a, b, c, d);
}
```

	V	W	X	Y
A	✅	✅	✅	✅
B	✅	✅	✅	✅
C	❌	✅	✅	✅
D	❌	❌	✅	✅

Type checking function calls: hard cases

```
fn F[U:! I, V:! I, W:! I, X:! I, ... each Y:! I]
    (u: U, v: V, w: W, x: X, ... each y: each Y);

fn G[... each A:! I, B:! I, C:! I, D:! I, E:! I]
    (... each a: each A, b: B, c: C, d: D, e: E) {
  F(... each a, b, c, d, e);
}
```

	V	W	X	Y	Z
A	✅	✅	✅	✅	✅
B	✅	✅	✅	✅	✅
C	❌	✅	✅	✅	✅
D	❌	❌	✅	✅	✅
E	❌	❌	❌	✅	✅

Type checking function calls: hard cases

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));

fn F[... each T:! type](... each x: Vector(each T), y: Vector(i32)) {
  var z: auto = Zip(... each x, y);
}
```

- Tell programmers to design their APIs differently?

Type checking function calls: hard cases

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));

fn F[... each T:! type](... each x: Vector(each T), y: Vector(i32)) {
  var z: auto = Zip(... each x, y);
}
```

- Tell programmers to design their APIs differently?
- No, because APIs have to migrate from C++ to Carbon.

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(each Param))
    -> Vector((... each Param));
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (???)
    -> ???;
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: ???)
    -> ???;
```

```
(first, ... each next) == (... each param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: ???)
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: ???)
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: Vector(???))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: Vector(???))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[???]
    (... each param: Vector(???))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! ???]
    (... each param: Vector(Param))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! ???]
    (... each param: Vector(Param))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector(???);
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector(???);
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param));
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param));
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector((First, ... each Next));
```



```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param));
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Type checking function calls: solution

```
fn Zip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param));
```

```
|param| >= 1
```

```
fn F[... each T:! type](... each x: Vector(each T), y: Vector(i32)) {
  var z: auto = Zip(... each x, y);
}
```

Type checking function calls: solution

```
// Returns -first + (sum of each next)
fn F[T:! Core.Negate, ... each U:! Core.AddWith(T)]
    (first: T, each next: U) -> T;

fn G(... each p: i64, q: i64) -> i64 {
  return F(... each p, q);
}
```

Type checking function calls: solution

```
fn G(... each p: i64, q: i64) -> i64 {
  return F(... each p, q);
}
```



```
fn G(... each p: i64, q: i64) -> i64 {
  let (... each arg: ???) = (... each p, q);
  return F(... each arg);
}
```

Type checking function calls: solution

```
fn G(... each p: i64, q: i64) -> i64 {
  return F(... each p, q);
}
```



```
fn G(... each p: i64, q: i64) -> i64 {
  let (... each arg: ???) = (... each p, q);
  return F(... each arg);
}
```

Type checking function calls: solution

```
fn G(... each p: i64, q: i64) -> i64 {
  return F(... each p, q);
}
```



```
fn G(... each p: i64, q: i64) -> i64 {
  let (... each arg: i64) = (... each p, q);
  return F(... each arg);
}
```

Type checking function calls: solution

```
// Returns -first + (sum of each next)
fn F[T:! Core.Negate, ... each U:! Core.AddWith(T)]
    (first: T, each next: U) -> T;


fn G(... each p: i64, q: i64) -> i64 {
  let (... each arg: i64) = (... each p, q);
  return F(... each arg);
}
```

Type checking function calls: solution

At the callee declaration:

- Merge as many parameters as possible

At the callsite:

- Deduce parameter arity from argument arities
- If each argument has only one parameter that can match it: easy case
- If not, merge as many arguments as possible
- If each parameter has only one argument that it can match: easy case
- If not, type checking fails

Conclusion

Conclusion

Goal: Support definition checking in variadic functions

- Packs and tuples are sequences of segments
  - Generic representation of arity and repetition
- Expressions and patterns in pack expansions have pack types
  - Generic representation of how types change across iterations

Conclusion

Goal: Support incremental migration from C++, anywhere in the call stack

Function call type checking:

- C++ API conventions don't give us the structure we need
- We try to infer that structure my merging parameters and merging arguments
  - Can be done in linear time
  - Hypothesis: sufficient for real-world use cases.

Conclusion

Goal: Prioritize iteration over recursion

- Pack expansion statements let you directly express variadic loops
- Pack expansion expressions and patterns are also fundamentally iterative
- Future work: declaration pack expansions

Conclusion

Goal: Address other pain points of C++ variadics

- Variadic parameters can be homogeneous
- Variadic parameters don't have to go last

See also

- Language design proposal: https://github.com/carbon-language/carbon-lang/pull/2240
- The Carbon Language: https://github.com/carbon-language/carbon-lang
- Other talks:
  - Carbon Language: An experimental successor to C++
  - Carbon's Successor Strategy
  - Definition-Checked Generics: The Why and How [Part 1, Part 2]

Backup slides

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);

fn F(... each x: Vector(String), y: Vector(i32)) {
  var z: auto = WeirdZip(... each x, y);
}
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);


fn WeirdZip[???]
    (???)
    -> ???
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);
```



```
fn WeirdZip[... each Param:! type]
    (... each param: Vector(Param))
    -> ???;
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);
```



```
fn WeirdZip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param).0);
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Rewriting the return type

```
fn WeirdZip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param).0);
```

```
|param| >= 1
```

```
fn F(... each x: Vector(String), y: Vector(i32)) {
  var z: auto = Zip(... each x, y);
}
```

Rewriting the return type

```
fn WeirdZip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param).0);
```

```
|param| >= 1
```

```
fn F(... each x: Vector(String), y: Vector(i32)) {
  var z: Vector((... String, i32).0) = Zip(... each x, y);
}
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);
```



```
fn WeirdZip[... each Param:! type]
    (... each param: Vector(Param))
    -> Vector((... each Param).0);
```

```
(first, ... each next) == (... each param)
|param| >= 1
(First, ... each Next) == (... each Param)
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);
```



```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);
```

Rewriting the return type

```
fn WeirdZip[First:! type, ... each Next:! type]
    (first: Vector(First), ... each next: Vector(each Next))
    -> Vector(First);

fn F(... each x: Vector(String), y: Vector(i32)) {
  var z: auto = WeirdZip(... each x, y);  ❌
}
```

Apply and ...expand

```
// Invokes f, with the tuple `args` as its arguments.
fn Apply[... each T:! type, F:! CallableWith(... each T)]
    (f: F, args: (... each T)) -> auto {
  let (... each arg: auto) = args;
  return f(... each arg);
}
```

Apply and ...expand

```
// Invokes f, with the tuple `args` as its arguments.
fn Apply[... each T:! type, F:! CallableWith(... each T)]
    (f: F, args: (... each T)) -> auto {
  let (... each arg: auto) = args;
  return f(... each arg);
}
```

Apply and ...expand

```
// Invokes f, with the tuple `args` as its arguments.
fn Apply[... each T:! type, F:! CallableWith(... each T)]
    (f: F, args: (... each T)) -> auto {
  let (... each arg: auto) = args;
  return f(... each arg);
  return f(...expand args);
}
```

Zip and . . . and

```
fn Zip[... each ElementType:! type](... each vector: Vector(each ElementType))
      -> Vector((... each ElementType)) {
  ... var each iter: auto = each vector.Begin();
  var result: Vector((... each ElementType));
  while (...and each iter != each vector.End()) {
    result.push_back((... each iter));
    ... each iter++;
  }
  return result;
}
```

Zip and . . . and

```
fn Zip[... each ElementType:! type](... each vector: Vector(each ElementType))
      -> Vector((... each ElementType)) {
  ... var each iter: auto = each vector.Begin();
  var result: Vector((... each ElementType));
  while (...and each iter != each vector.End()) {
    result.push_back((... each iter));
    ... each iter++;
  }
  return result;
}
```

Zip and . . . and

```
fn Zip[... each ElementType:! type](... each vector: Vector(each ElementType))
      -> Vector((... each ElementType)) {
  ... var each iter: auto = each vector.Begin();
  var result: Vector((... each ElementType));
  while (...and each iter != each vector.End()) {
    result.push_back((... each iter));
    ... each iter++;
  }
  return result;
}
```

Zip and . . . and

```
fn Zip[... each ElementType:! type](... each vector: Vector(each ElementType))
      -> Vector((... each ElementType)) {
  ... var each iter: auto = each vector.Begin();
  var result: Vector((... each ElementType));
  while (...and each iter != each vector.End()) {
    result.push_back((... each iter));
    ... each iter++;
  }
  return result;
}
```

Pack expansions vs loops

```
var sum: i64 = 0;
let (... each x: i64) =
    (1, 2, 3, 4);
... sum += each x;
```

```
var sum: i64 = 0;
for (x: i64 in
    (1, 2, 3, 4)) {
  sum += x;
}
```

expand as an operator

```
...if (Condition()) {
  var x: auto = expand F(y);
}
```

Vectorizing pack expansion

```
(... if (each cond) then F(each param) else G(each param))
```

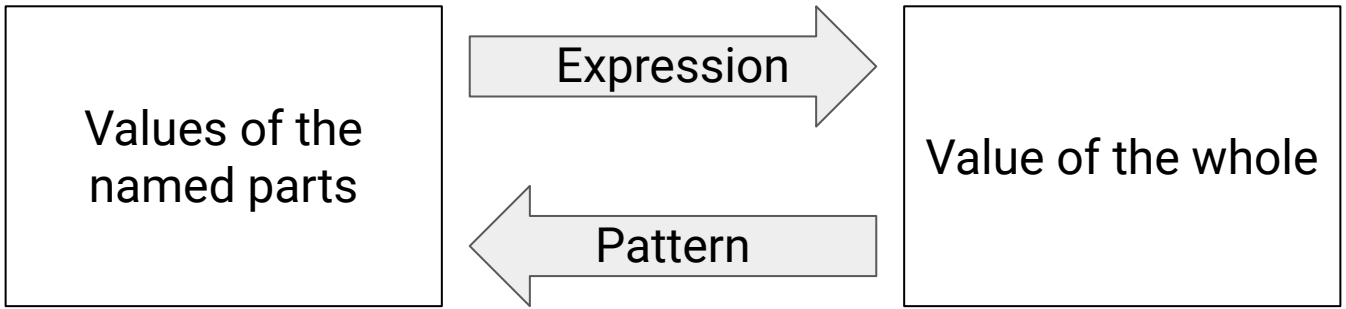
Goals for Carbon

Carbon is designed to support:

1. Performance-critical software
2. Software and language evolution
3. Code that is easy to read, understand, and write
4. Practical safety and testing mechanisms
5. Fast and scalable development
6. Modern OS platforms, hardware architectures, and environments
7. Interoperability with and migration from existing C++ code

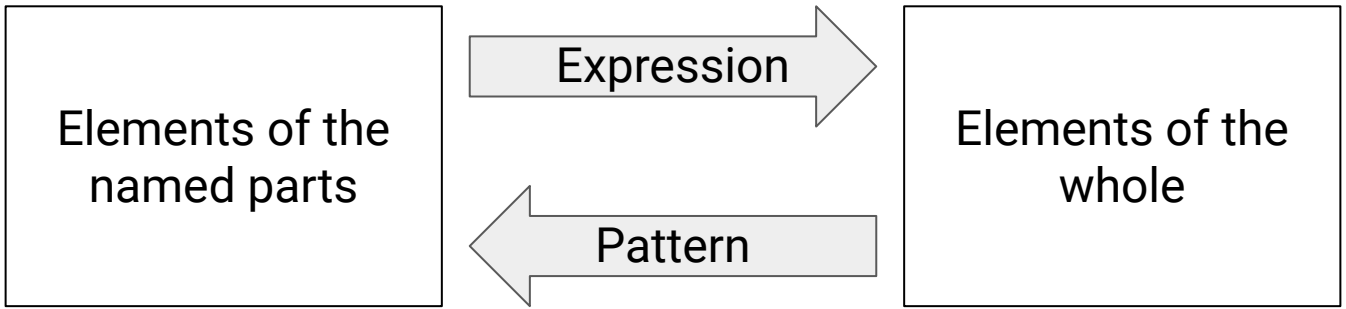
Patterns and Expressions

```
let x: i64  = foo;
    x      == foo

let (first: i32, second: i64)  = bar;
    (first     , second     ) == bar
```



Patterns and Expressions

```
let (... each x: i64)  = foo;
    (... each x     ) == foo

let (... (each first: i32, each second: i64))  = bar;
    (... (each first     , each second     )) == bar
```



Syntax and Semantics

```
// Computes the sum of its arguments
fn IntSum(... each param: i64) -> i64;

// Computes the sum of the squares of its arguments
fn SumOfSquares(... each param: i64) -> i64 {
  return IntSum(... each param * each param);
}
```