Google

Optimizing generic code lowering to LLVM-IR through function equivalence coalescing

asbirlea@google.com

LLVM Developers Meeting

October 2025

Google

A.K.A. Merging equivalent functions ...
... in the front end

asbirlea@google.com

Google

TL; DR:

Templates / generics generate many functions.

Equivalent ones can be merged / coalesced in LLVM IR and the front end can do pretty good job at it.

Google

C++ has templates, Rust and Carbon have generics

Generics refers to the ability to generalize code by adding compile-time parameters.

- **Generic function**: function with at least a compile-time parameter (explicit or deduced)
- **Generic type**: type with a compile-time parameter
- **Generic interface**: interface with a compile-time parameter

Google

C++ has templates, Rust and Carbon have generics

Generics refers to the ability to generalize code by adding compile-time parameters.

- **Generic function**: function with at least a compile-time parameter (explicit or deduced)
- **Generic type**: type with a compile-time parameter
- **Generic interface**: interface with a compile-time parameter - **interfaces have functions**

In Carbon, function parameters:

- Regular parameters `fn F(x: Int) -> bool;`
- **Checked generics** `fn F[T:! type](x: T) -> T;`
- Template generics `fn F[template T:! type](x: T*) -> T*;`

https://docs.carbon-lang.dev/docs/design/generics/terminology.html#checked-versus-template-parameters

Google

Rust and Carbon's checked generics

Pros:

- **Performance: Monomorphization eliminates runtime overhead associated with generics (dynamic dispatch, type checking at runtime)**
- **Safety: Type checking at compile time**

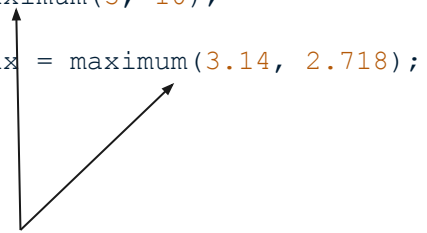
Cons:

- **Increase in code size and compile time**

Google

C++: template function -> argument deduction

```
// C++

template <typename T>

T maximum(T a, T b) {

   return (a > b) ? a : b;

}

void main() {

   int intMax = maximum(5, 10);

   double doubleMax = maximum(3.14, 2.718);

}
```

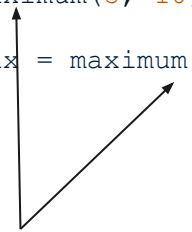


Function instantiations

Google

Carbon: checked generic function -> similar deduction

```cpp
// C++

template <typename T>

T maximum(T a, T b) {

   return (a > b) ? a : b;

}

void main() {

   int intMax = maximum(5, 10);

   double doubleMax = maximum(3.14, 2.718);

}
```



Function instantiations

```
// Carbon


fn maximum[T:! Ordered](a: T, b: T) -> T {

   return if a > b then a else b;

}

fn main() {

   var intMax: i32 = maximum(5, 10);

   var doubleMax: f64 = maximum(3.14, 2.718);

}
```

Function specifics

Google

Previous example revisited with pointers

```
// C++

template <typename T>

T maximum(T a, T b) {

   return (a > b) ? a : b;

}

void main() {

   int *i_1, *i_2;

   int *intMax = maximum(i_1, i_2);

   double *d_1, *d_2;

   double *doubleMax = maximum(d_1, d_2);

}
```

Function instantiations

```
// Carbon


fn maximum[T:! Ordered](a: T, b: T) -> T {

   return if a > b then a else b;

}

fn main() {

   var i_1: i32*; var i_2: i32*;

   var intMax: i32* = maximum(i_1, i_2);

   var d_1: f64*; var d_2: f64*;

   var doubleMax: f64* = maximum(d_1, d_2);

}
```

Function specifics

Google

Problem to solve

Function instantiations / specifics, where the arguments have different front-end types, but translate in LLVM to functions with arguments of the same LLVM type.

Google

Problem to solve

Function instantiations / specifics, where the arguments have different front-end types, but translate in LLVM to functions with arguments of the same LLVM type.

Goal: Deduplicate or coalesce such function instantiations / specifics in LLVM-IR.

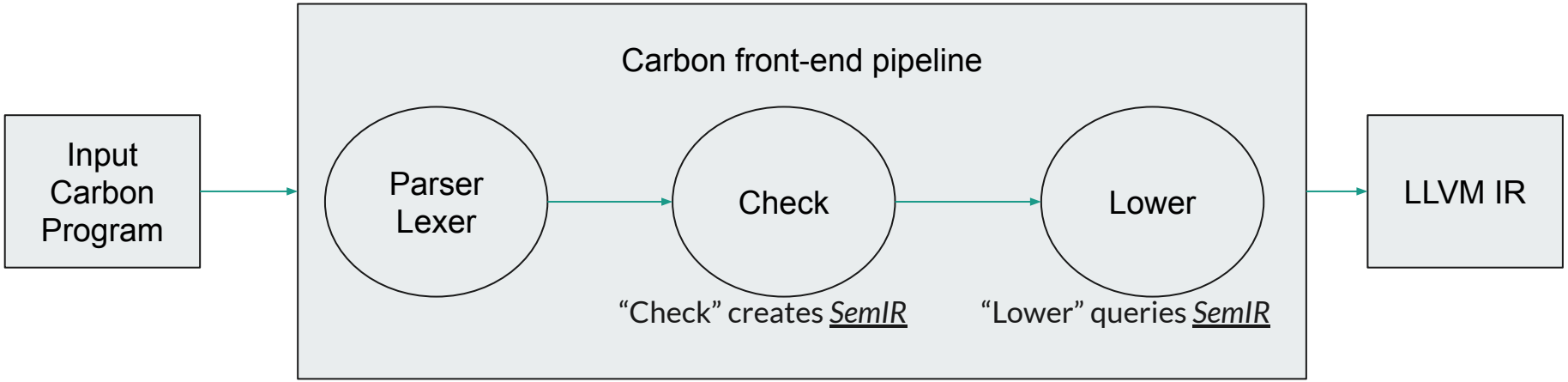
Google

Why is this important?

Code size and **compile time**.

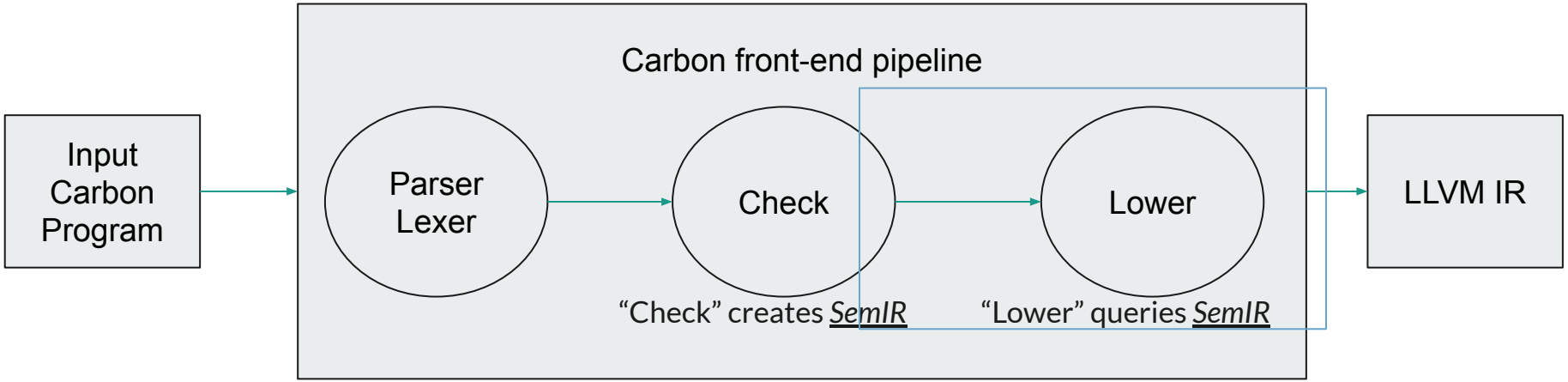
Google

Current implementation: Carbon's front-end IR



Google

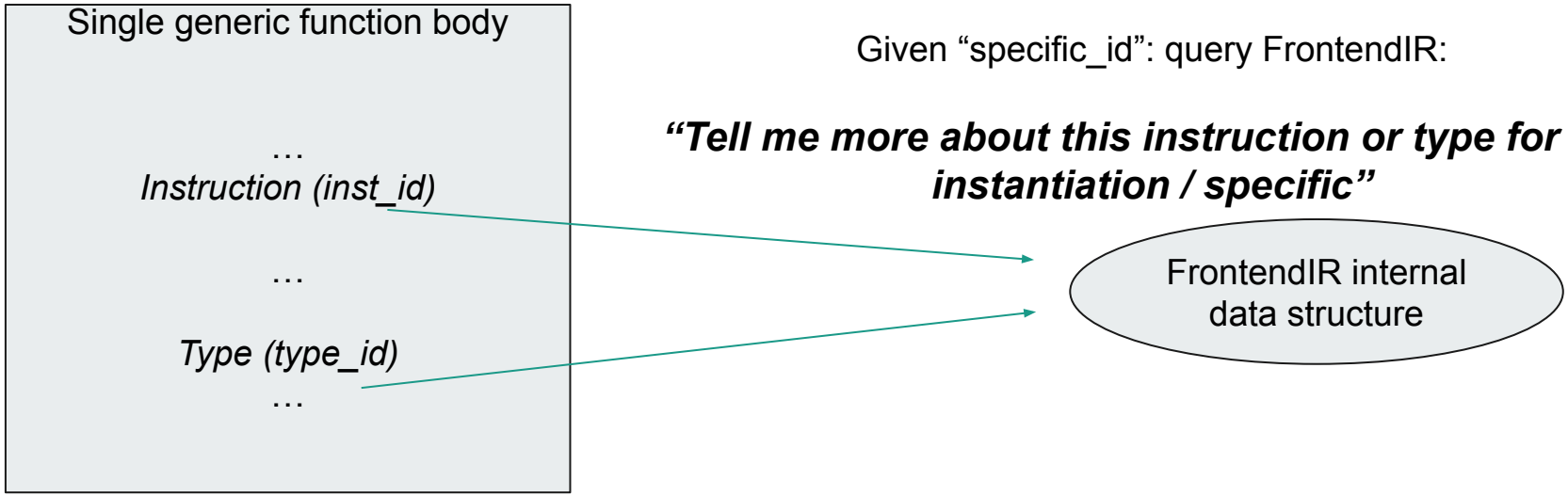
Carbon's front-end IR: *SemIR* - focus for this talk



Note: A front end IR vs an AST? ... great idea!

Google

Representation: generic function in FrontendIR



Google

Easy: compare function bodies! Done! Right?

Single generic function body

…

Instruction(specific_id1) == Instruction(specific_id2) ?

…

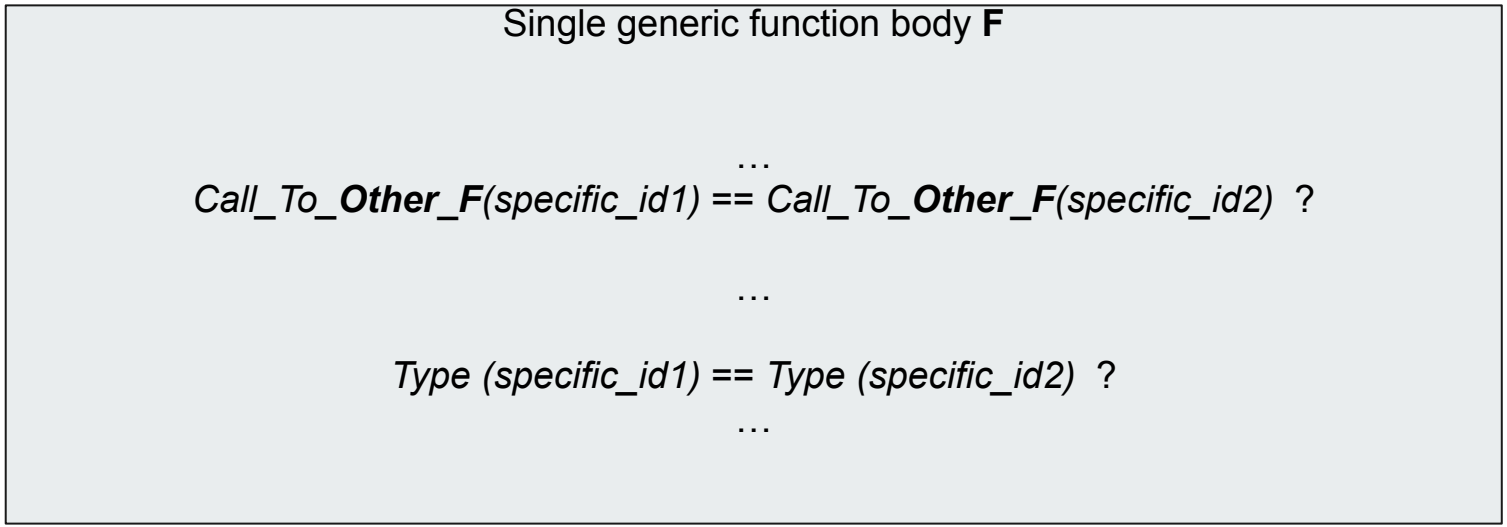
Type (specific_id1) == Type (specific_id2) ?

…

Google

First consideration: the instruction can be a call.

To determine equivalence, we need to evaluate the body of *Other_F* => Build the definition of *Other_F* for the two specific_ids => Applies for transitive calls, so **need the full call graph!**



Google

Second consideration: recursion.

To determine equivalence in the presence of recursion: need to compare specific_ids.

Single generic function body **F**

…

*Call_To_**F**(specific_id1) == Call_To_**F**(specific_id2) ?*

…

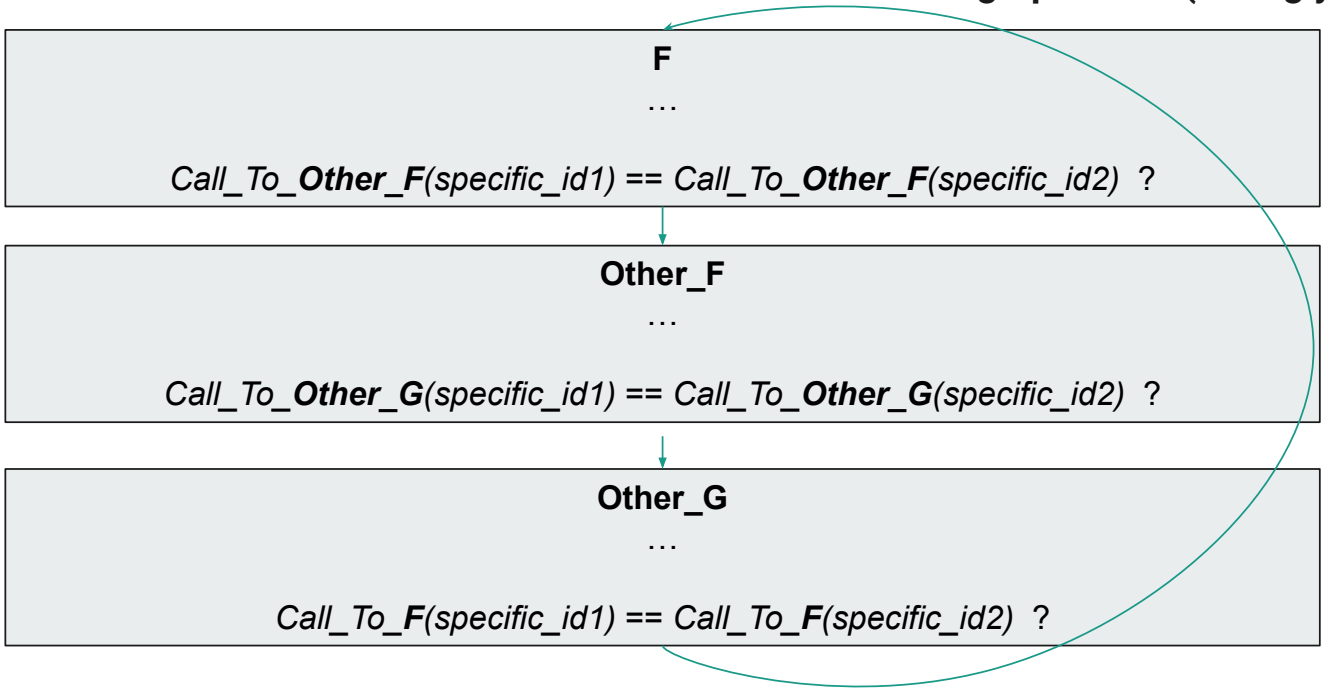
Type (specific_id1) == Type (specific_id2) ?

…

Google

Putting the two together:

Evaluate transitive calls + Consider recursion => Need call graph SCCs (strongly connected components)



Google

So far, checking LLVM type equivalence sufficies

Google

Third consideration: LLVM types may be insufficient

- Calling a function on the same “type” may not always call the same function.

```
interface I {
 fn F();
}
class X {}
impl X as I {
 fn F() { Core.Print(1); }
}
class Y {}
impl Y as I {
 fn F() { Core.Print(2); }
}

fn G(T:! I) { T.F(); }

fn Run() {
 G(X);
 G(Y);
}
```

Google

Other frontend-specific considerations

- Carbon may define generic constants in interfaces too
  => need to consider different LLVM-IR `allocas`.

```
interface I {
 let T:! type;
}

fn G(U:! I) {
 var x: U.T;
}

class C {}

impl C as I where .T = bool {}

class D {}
impl D as I where .T = i32 {}

fn H() {
 G(C);
 G(D);
}
```

Google

All correctness considerations

For each instantiation/specific, consider:

- anything that translates to an `llvm::Type`
- any frontend specific feature that can generate different LLVM IR

Google

Final consideration: performance

Opt for hashing aggregated information into function “fingerprints” using LLVM’s BLAKE3.

- reduce storage
- ease of comparison

Google

High level algorithm for coalescing generic function definitions

At the top level:

1. **Generate all function definitions and collect data for each one.**
2. **Perform coalescing logic and cleanup duplicates.**

Google

High level algorithm for coalescing generic function definitions

1. **Generate all function definitions:**

- Collect **function type** for each function definition into a `common fingerprint` (hash)
- Generate LLVM-IR for each instruction in the function
- If an **instruction/type is specific-dependent**, add this info into the `common fingerprint` (hash)
- When a **call** to another **specific of a generic** is found, that function will need a function definition (creates declaration, and marks it as needing a definition, will come back later)
  - Collect this info to a `specific fingerprint` (hash)
  - Collect the *non-hashed* specific_id.

Google

High level algorithm for coalescing generic function definitions

2. Perform coalescing logic

- For each two specifics of the same generic
  - 3. **Check fingerprint equivalence**
  - If step 3. returns equivalent, a list of all **equivalent functions** in the **call graph SCC** will be given.
  - Define a canonical specific to use for each equivalence found, and perform replacements
  - Once replaced, the duplicated may be deleted from LLVM IR

Google

High level algorithm for coalescing generic function definitions

3. Check fingerprint equivalence

- If `common fingerprints` are different or already known not equivalent => not equivalent
- If `specific fingerprints` are the same => equivalent
- If already assumed equivalent (in a cycle) => equivalent
- Check each of the calls to other specifics (non-hashed specific_ids):
  - If all are equivalent or assumed equivalent => equivalent
  - If any are known non-equivalent => not equivalent
  - SCC resolution: assume they're equivalent and recurse.

Google

Reminder: Why is this important?

Code size and **compile time**.

Alternatives considered:

- Should it be done in the front-end or the middle end?
  - LLVM has a MergeFunctions Pass
  - When this was implemented, the guestimate was “in the front-end is likely to yield better results”.
- When to do the coalescing in the front-end?
  - Should we do a pre-processing before actually generating LLVM-IR?
  - Should we firstbuild all the call-graph to determine which specifics are expected to have a definition
  - This will duplicate most of the lowering logic, minus the LLVM-IR creation itself
- Should this be done at all?
  - We guessed yes!

Google

Preliminary performance results

Google

Methodology

- Generating **stress tests** in Carbon - not representative of average compilations! (https://github.com/carbon-language/carbon-lang/tree/trunk/testing/base)
- Extended current code generator with generic function definitions.
- Testing on test sizes of 512 - 1024 lines of Carbon code.

```
fn MynIyDnM [TTTTT:! type, UUUUU:! type, VVVVV:! type] (p0sHhfwj : TTTTT, jn8GlOYy : UUUUU, LjXhpIoTWTTPP : VVVVV) {
  var tKPjletOVDF: bool;
  var xFiskfTKdLaZGfihMZP8y_MZGSAifOoV6nfRZ18mfjHcd: i32;
  var u_AkEg44: bool;
  var LdQXJ9_9wpI: i32;

  ED7ouKpZJY5(jn8GlOYy, LjXhpIoTWTTPP, p0sHhfwj);
  MRXxe3IoV(LdQXJ9_9wpI, u_AkEg44, LjXhpIoTWTTPP);
  HoLg8shLaE(tKPjletOVDF, jn8GlOYy, u_AkEg44);
  xe446oHmj(xFiskfTKdLaZGfihMZP8y_MZGSAifOoV6nfRZ18mfjHcd, LdQXJ9_9wpI, jn8GlOYy);
  YolYO3v79B(p0sHhfwj, tKPjletOVDF, LdQXJ9_9wpI);
}

fn Main_method_no_template () {
  var lcP4TvtzNDvE: i32*;
  var BNanx7bT: xSQqf;
  var oYrqnvDiL_k: bool;
  var ev68jlB4ZSkQQYb: i5msb;

  HoLg8shLaE(oYrqnvDiL_k, BNanx7bT, ev68jlB4ZSkQQYb);
  Rczn8GQVgxX(BNanx7bT, lcP4TvtzNDvE, oYrqnvDiL_k);
  ZKvqJ37kXbqvBItbA_IpdfeS3cRY3O7gBzBRm(lcP4TvtzNDvE, ev68jlB4ZSkQQYb, BNanx7bT);
  KzXHhj0sPzkhQ(ev68jlB4ZSkQQYb, oYrqnvDiL_k, lcP4TvtzNDvE);
  YolYO3v79B(oYrqnvDiL_k, BNanx7bT, ev68jlB4ZSkQQYb);
}
```

Google

Code size impact with function coalescing logic

- **Balanced types** (primitive and pointers): ~44-56% reduction in:
  - LLVM-IR size before and after optimizations
  - assembly size
  - # of functions generated

- **Most pointer types**: ~98% reduction in:
  - LLVM-IR size before and after optimizations
  - assembly size
  - # of functions generated

- **MergeFunctions pass** makes a difference in **~12% of tests**
  - maximum of ~4% reduction in # of functions, average ~0.1% reduction

Google

Compile time impact with function coalescing logic

- **Balanced types** (primitive and pointers):
  - ~0.9% increase to ~2% reduction in Carbon's **lowering** stage.
  - ~25-51% reduction in Carbon's **opt and codegen** stages (-O3), with inlining disabled.

- **Most pointer types**:
  - ~7-9% reduction in Carbon's **lowering** stage!
  - ~97% reduction in Carbon's **opt and codegen** stages (-O3), with inlining disabled.

- Note: With inlining enabled, similar compile times with the coalescing logic enabled, but compile-time explosion without it, due to aggressive inlining in the non-coalesced functions.

Google

Status and more information

The algorithm is part of Carbon's lowering to LLVM-IR stage.
(https://github.com/carbon-language/carbon-lang/blob/trunk/toolchain/lower)

Algorithm and other considerations are documented in the docs:
https://github.com/carbon-language/carbon-lang/blob/trunk/toolchain/docs/coalesce_generic_lowering.md

Lots of room for further improvements!

Google

Questions? Let's talk!