A “third way” of compiling polymorphism

Isaac Elliott

13 May, 2025

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Let's start with some code generation fundamentals.

Here's a simple function that does addition.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Plausible machine code for add

```
add:
mov rax, [rsp + 8]
mov rbx, [rsp + 16]
add rax, rbx
mov [rsp + 24], rax
ret
```

Plausible machine code for *calling* add

```
push QWORD 42
push QWORD 99
call add
mov [rbp], rax
add rsp, 16
```

Let's start with some code generation fundamentals.

Here's a simple function that does addition.

Here's some plausible machine code for the function. Let's step through it so you can get a feel for what it does.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (calling add)

	Instruction
→	`push QWORD 42`
	`push QWORD 99`
	`call add`
	`add rsp, 16`

Address	Data
`0x0`	free
`...`	free
`rbp - 8 (= rsp)`	space for z
`rbp`	?

On the left we have the program counter. The arrow points to the instruction that will be executed next.

On the right is the stack, which is a primitive data struture provided to the program by the operating system. It's just a slab of memory, plus a register `rsp` that holds the address of the top of the stack.

By convention, more recent stack entries are stored at lower addresses. The maximum possible stack location isn't usually `0x0`; there are usually other parts of the program that live at those low address. But I thought that putting `0x0` as the "ceiling" of the stack was very illustrative.

The `rbp` register holds the "frame pointer". Every function call gets to own part of the stack, and promises not to mess with the portion of the stack owned by its caller. The frame pointer tells us where our caller's stack ends and ours begins. Local variables are addressed relative to the frame pointer.

So you can see that before we call `add` we've allocated memory for its return value.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (calling add)

	Instruction	Address	Data
	`push QWORD 42`	`0x0`	free
→	`push QWORD 99`	`...`	free
	`call add`	`rbp - 16 (= rsp)`	42
	`add rsp, 16`	`rbp - 8`	space for z
		`rbp`	?

We just pushed the 8 byte value 42 onto the stack.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (calling add)

	Instruction
	`push QWORD 42`
	`push QWORD 99`
→	`call add`
	`add rsp, 16`

Address	Data
`0x0`	free
`...`	free
`rbp - 24 (= rsp)`	99
`rbp - 16`	42
`rbp - 8`	space for z
`rbp`	?

We just pushed the 8 byte value 99 onto the stack.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (running add)

	Instruction
→	`mov rax, [rsp + 8]`
	`mov rbx, [rsp + 16]`
	`add rax, rbx`
	`mov [rsp + 24], rax`
	`ret`

Address	Data
`0x0`	free
`...`	free
`rbp - 32 (= rsp)`	return addr
`rbp - 24`	99
`rbp - 16`	42
`rbp - 8`	space for z
`rbp`	?

We've called `add`, which pushes the return address onto the stack and then jumps to the function's address.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (running add)

	Instruction
	`mov rax, [rsp + 8]`
→	`mov rbx, [rsp + 16]`
	`add rax, rbx`
	`mov [rsp + 24], rax`
	`ret`

Address	Data
`0x0`	free
`...`	free
`rbp - 32 (= rsp)`	return addr
`rbp - 24`	99
`rbp - 16`	42
`rbp - 8`	space for z
`rbp`	?

99 was loaded into `rax`

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (running add)

	Instruction
	`mov rax, [rsp + 8]`
	`mov rbx, [rsp + 16]`
→	`add rax, rbx`
	`mov [rsp + 24], rax`
	`ret`

Address	Data
0x0	free
...	free
rbp - 32 (= rsp)	return addr
rbp - 24	99
rbp - 16	42
rbp - 8	space for z
rbp	?

42 was loaded into rbx

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (running add)

	Instruction
	`mov rax, [rsp + 8]`
	`mov rbx, [rsp + 16]`
	`add rax, rbx`
→	`mov [rsp + 24], rax`
	`ret`

Address	Data
`0x0`	free
`...`	free
`rbp - 32 (= rsp)`	return addr
`rbp - 24`	99
`rbp - 16`	42
`rbp - 8`	space for z
`rbp`	?

`rax` was set to $99 + 42 = 141$

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (running add)

	Instruction
	`mov rax, [rsp + 8]`
	`mov rbx, [rsp + 16]`
	`add rax, rbx`
	`mov [rsp + 24], rax`
→	`ret`

Address	Data
0x0	free
...	free
rbp - 32 (= rsp)	return addr
rbp - 24	99
rbp - 16	42
rbp - 8	141
rbp	?

The space for z was filled with that value

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (returned from add)

	Instruction
	`push QWORD 42`
	`push QWORD 99`
	`call add`
→	`add rsp, 16`

Address	Data
`0x0`	free
`...`	free
`rbp - 24 (= rsp)`	99
`rbp - 16`	42
`rbp - 8`	141
`rbp`	?

We returned from add, which popped the return address from the top of the stack and then jumped to it.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Program state (returned from add)

Instruction
`push QWORD 42`
`push QWORD 99`
`call add`
`add rsp, 16`
→

Address	Data
`0x0`	free
`...`	free
`rbp - 8 (= rsp)`	141
`rbp`	?

Two 8-byte values were “popped” from the stack. They’re technically still there (because we didn’t overwrite them) but for all intents and purposes they’re garbage.

Compiling a function

A simple function

```
int64_t add(int64_t x, int64_t y) {
    return x + y;
}
```

A simple function call

```
int64_t z = add(42, 99);
```

Plausible machine code for add

```
add:
mov rax, [rsp + 8]
mov rbx, [rsp + 16]
add rax, rbx
mov [rsp + 24], rax
ret
```

Plausible machine code for *calling* add

```
push QWORD 42
push QWORD 99
call add
mov [rbp], rax
add rsp, 16
```

Production compilers avoid passing data via memory where possible, preferring registers, because accessing memory can be hundreds of times slower.

My example uses memory instead of registers to make *size requirements* very explicit. At some point, every type must have a known size because it's possible that a value of that type will end up in memory. In this example values are exclusively placed on the stack, but programs may also store values in the heap (another canonical memory region that I won't explain here).

There is no way to get around size requirements, even when a compiler uses registers effectively. Registers have fixed sizes, so the compiler needs to know the size of a value to determine which, if any, register can hold the value.

Compiling a polymorphic function

A simple polymorphic function (C++)

```
template <class T>
T id(T x) { return x; }
```

What is the size of T?

A simple polymorphic function (Haskell)

```
id :: a -> a
id x = x
```

What is the size of a?

Polymorphic functions (also known as generics) are defined over *all* types.

In the examples to the left, the names T and a are known as type *variables*. Since a type variable stands for any possible type, it has no definitive size.

I just said that every type must have a known size. So how to you compile a function that involves types with *unknown* sizes? This sounds like a contradiction.

I've chosen C++ and Haskell for my code examples because these languages have very different solutions to this problem. Let's look at C++ first because C++ comes before Haskell in the dictionary.

Compiling a polymorphic function (monomorphisation)

A simple polymorphic function (C++)

```
template <class T>
T id(T x) { return x; }
```

C++ compiles polymorphic functions by *not actually compiling polymorphic functions*.

The definition of a polymorphic function doesn't result in any code generation. Instead, specialised copies of the polymorphic function are generated whenever it's called with a concrete type. This process is called *monomorphisation*.

Compiling a polymorphic function (monomorphisation)

A simple polymorphic function (C++)

```
template <class T>
T id(T x) { return x; }
```

```
bool x = id(true);
/*
bool id(bool x) {
    return x;
}
*/
```

C++ compiles polymorphic functions by *not actually compiling polymorphic functions*.

The definition of a polymorphic function doesn't result in any code generation. Instead, specialised copies of the polymorphic function are generated whenever it's called with a concrete type. This process is called *monomorphisation*.

Compiling a polymorphic function (monomorphisation)

A simple polymorphic function (C++)

```
template <class T>
T id(T x) { return x; }
```

```
bool x = id(true);
/*
bool id(bool x) {
    return x;
}
*/
```

```
int32_t x = id(20);
/*
int32_t id(int32_t x) {
    return x;
}
*/
```

C++ compiles polymorphic functions by *not actually compiling polymorphic functions*.

The definition of a polymorphic function doesn't result in any code generation. Instead, specialised copies of the polymorphic function are generated whenever it's called with a concrete type. This process is called *monomorphisation*.

Compiling a polymorphic function (monomorphisation)

A simple polymorphic function (C++)

```
template <class T>
T id(T x) { return x; }
```

```
bool x = id(true);
/*
bool id(bool x) {
    return x;
}
*/
```

```
int32_t x = id(20);
/*
int32_t id(int32_t x) {
    return x;
}
*/
```

```
string x = id("hello");
/*
string id(string x) {
    return x;
}
*/
```

C++ compiles polymorphic functions by *not actually compiling polymorphic functions*.

The definition of a polymorphic function doesn't result in any code generation. Instead, specialised copies of the polymorphic function are generated whenever it's called with a concrete type. This process is called *monomorphisation*.

Pros and cons of monomorphisation

Pros	Cons

Pros and cons of monomorphisation

Pros	Cons
Generates efficient code	

The main advantage of this approach is that it generates the same code that you'd write if you used C and copy-pasted a lot of code. Each monomorphic version of the polymorphic function performs as well as if you'd written is by hand. This is what's known as a "zero-cost abstraction".

Of course, monomorphisation *does* have costs, just not at the level of individual monomorphic functions.

Pros and cons of monomorphisation

Pros	Cons
Generates efficient code	Generates a lot of code

If you use a function like `id` with 5 different concrete types, then your final program contains 5 slightly different versions of the same function. Introducting type variables to a function multiplies this effect. Large programs that use polymorphism liberally tend to exhibit *code bloat* when compiled using monomorphisation.

Pros and cons of monomorphisation

Pros	Cons
Generates efficient code	Generates a lot of code
	No separate compilation

A related issue is that modules containing polymorphic functions can't be separately compiled. Separate compilation reduces code bloat by placing a function's code in a single object file that can be referenced by other parts of the program. Separate compilation also speeds up the compilation process because code for a function definition is not repeatedly generated.

Pros and cons of monomorphisation

Pros	Cons
Generates efficient code	Generates a lot of code
	No separate compilation
	Complicates advanced type systems

The final drawback I want to mention is how monomorphisation interacts with advanced type system features like higher-kinded types, existential types, generalised algebraic datatypes (GADTs), and dependent types. My impression as a compiler implementor is that it makes implementing some of these features more difficult, and for other features it's just the wrong approach.

Pros and cons of monomorphisation

Pros	Cons
Generates efficient code	Generates a lot of code
	No separate compilation
	Complicates advanced type systems

While I've listed few pros relative to the number of cons, I often find that runtime efficiency is worth the trade.

Compiling a polymorphic function (uniform representation)

A simple polymorphic function (Haskell)

```haskell
id :: a -> a
id x = x
```

C equivalent

```c
void* id(void* x) {
    return x;
}
```

In Haskell, polymorphic functions actually are compiled. The requirement that "every type is has a known size" (including type variables) is satisfied by *making every type (including type variables) have the same size*. Haskell values are boxed by default, which means they're represented by pointers to heap-allocated memory.

This approach is called *uniform representation*.

Pros and cons of uniform representation

Pros	Cons

Pros and cons of uniform representation

Pros	Cons
Generates compact code	

Polymorphic functions are compiled once, reducing code size.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	
Allows separate compilation	

This allows modules to be compiled once and reused throughout the program.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	
Allows separate compilation	
Simplifies advanced type systems	

Code generation for advanced type system features is no more difficult than the fundamental features. Functions take and return pointers, from humble booleans to complex dependent types.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	Generates allocation-heavy (slow) code
Allows separate compilation	
Simplifies advanced type systems	

The biggest drawback of the uniform representation approach is its reliance on memory. As I said earlier, accessing memory can be hundreds of times slower than accessing registers. Overuse of memory and underuse of registers is a potentially massive handicap.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	Generates allocation-heavy (slow) code
Allows separate compilation	
Simplifies advanced type systems	

Why are (heap) allocations slow?

Pros and cons of uniform representation

Pros	Cons
Generates compact code	Generates allocation-heavy (slow) code
Allows separate compilation	
Simplifies advanced type systems	

Why are (heap) allocations slow?

- Reduced data locality

Modern CPUs have layers of caches to reduce the performance gap between registers and memory. A cache hit is slower than reading from a register but still much faster than accessing memory. Instead of being read byte-at-a-time, larger chunks of memory (64B at the time of writing) are loaded and cached, under the assumption that nearby bytes will soon be needed. When a program is structured such that reading one byte increases the probability of soon reading an adjacent byte, the program is said to have high *data locality*. Programs that use a lot of heap allocation tend to have unpredictable data locality.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	Generates allocation-heavy (slow) code
Allows separate compilation	
Simplifies advanced type systems	

Why are (heap) allocations slow?

- Reduced data locality
- Increased computational overhead

Another problem is that heap allocation has a computational cost. The structure of the heap depends on the memory allocation system used by the program. The allocator has to track which heap regions are free or in use, so that it can reserve an appropriate region when a new allocation is requested. This bookkeeping work adds up.

Pros and cons of uniform representation

Pros	Cons
Generates compact code	Generates allocation-heavy (slow) code
Allows separate compilation	
Simplifies advanced type systems	

Why are (heap) allocations slow?

- Reduced data locality
- Increased computational overhead

Languages that use uniform representation typically rely on clever and complex allocation systems (often called "garbage collectors") to reduce these costs. Garbage collectors can amortise the cost of many allocations over longer programs and improve data locality for common allocation patterns.

But because these allocation systems are general purpose, they're far from optimal for any *specific* function or program. In this respect, polymorphism via uniform representation is not a "zero-cost abstraction".

A “third way” for compiling a polymorphic function

Goals:

- Stack allocation by default
- Allow separate compilation
- Enable advanced type system features (higher-kinded types, GADTs, dependent types)

Some of the tradeoffs described feel weird. For example: why should the use of polymorphic functions force me to choose between stack-allocation by default and separate compilation? For a while I’ve wondered if there are any other reasonable approaches to compiling polymorphism, and now I’ve got one that’s interesting.

Compiling a polymorphic function (type passing)

A simple polymorphic function (Haskell-like language)

```
id :: forall a. a -> a
id x = x
```

A simple polymorphic function call (Haskell-like language)

```
let x = id @Int64 42
```

C equivalent (definition)

```
void id(void* result, const Type* a, const void* x) {
    // result <- x
    a->move(result, x);
}
```

C equivalent (call)

```
int64_t arg = 42;
int64_t x;
id(&x, &Type_Int64, &arg);
```

Here's an example of the approach. Polymorphic values are passed and returned via pointers, usually to stack memory. For each type variable, a corresponding `type dictionary` is passed to the function. Each type dictionary contains functions that manipulation pointers to values of a specific type.

For example, the `Type_Int64` dictionary has functions that manipulate pointers to 64-bit integers.

Compiling a polymorphic function (type passing)

A simple polymorphic function (Haskell-like language)

```
id :: forall a. a -> a
id x = x
```

A simple polymorphic function call (Haskell-like language)

```
let x = id @Int64 42
```

Prelude for C equivalent

```
typedef struct {
    void (*move)(const void*, const void*);
    /* other functions omitted */
} Type;

const Type Type_Int64 = /* omitted */;
```

Here's the definition of `Type`. `move` is the only function required to work with value types like integers. If you have compound types and reference counting allocation you can add a `copy` and `drop` function.

Compiling a polymorphic function (type passing)

A simple polymorphic function (Haskell-like language)

```
id :: forall a. a -> a
id x = x
```

A simple polymorphic function call (Haskell-like language)

```
let x = id @Int64 42
```

C equivalent (definition)

```
void id(void* result, const Type* a, const void* x) {
    // result <- x
    a->move(result, x);
}
```

C equivalent (call)

```
int64_t arg = 42;
int64_t x;
id(&x, &Type_Int64, &arg);
```

I find this approach interesting because it overlaps with both monomorphisation and uniform representation but can't be reduced to either.

It's similar to monomorphisation, in that monomorphic function arguments can be passed in registers or on the stack. And it's similar to uniform representation in that polymorphic arguments are represented by pointers.

But *unlike* monomorphisation, it permits separate compilation, and *unlike* uniform representation, it doesn't require all values to be heap-allocated pointers.

Compiling a polymorphic function (type passing)

A simple polymorphic function (Haskell-like language)

```
id :: forall a. a -> a
id x = x
```

A simple polymorphic function call (Haskell-like language)

```
let x = {-# SPECIALISE #-} id @Int64 42
```

C equivalent (definition)

```
int64_t id_Int64(int64_t x) {
    return x;
}
```

C equivalent (call)

```
int64_t x = id_Int64(42);
```

I also like this approach because it sets the stage for programmer-driven monomorphisation.

If a polymorphic function is called repeatedly with a known type (e.g. in a loop), then the programmer can choose to use a specialised version to avoid the overhead of passing arguments via the stack.

Pros and cons of type passing

Pros	Cons

Pros and cons of type passing

Pros	Cons
Generates fast monomorphic code	

The presence of polymorphism in the language doesn't impact monomorphic code. Uniform representation mandates boxing by default in order to compile polymorphic code, and if you don't use polymorphism you still have to pay the cost.*

* Unless you have a very advanced compiler like GHC.

Pros and cons of type passing

Pros	Cons
Generates fast monomorphic code	
Reasonably compact code	

Use of polymorphic functions with many different types doesn't cause an explosion in code size. However, the type dictionaries and extra function arguments take up space.

Pros and cons of type passing

Pros	Cons
Generates fast monomorphic code	
Reasonably compact code	
Allows separate compilation	

Separate compilation works.

Pros and cons of type passing

Pros	Cons
Generates fast monomorphic code	
Reasonably compact code	
Allows separate compilation	
Supports advanced type system features	

The approach naturally generalises to higher-kinded types, existentials, GADTs, and maybe even dependent types. I'm running out of steam so I'm not going to justify that claim as part of this presentation.

If people are interested we can sketch some of it if there's extra time left, or afterward at the pub.

Pros and cons of type passing

Pros	Cons
Generates fast monomorphic code	Polymorphism has runtime overhead
Reasonably compact code	
Allows separate compilation	
Supports advanced type system features	

The main (theoretical) drawback is the runtime overhead of type dictionaries.

- These type dictionaries are extra function arguments
- Calling a type dictionary requires a pointer dereference and indirect jump
- More interesting cases require runtime construction of type dictionaries

I'm unsure of the magnitude of these performance impacts in real-world code. It may be that it's just too slow in general, but it's also possible that slow cases are rare and easily fixed by programmer-driven specialisation.

I'm building a compiler using these techniques with the intention of benchmarking the type passing approach. It's slow going because I have lots of other things to do, so if you're excited by these ideas then feel free to race ahead build the thing yourself so I can benchmark it.

Related work

- How to make ad-hoc polymorphism less ad hoc (1989) - https://doi.org/10.1145/75277.75283
- An ad hoc approach to the implementation of polymorphism (1991) - https://doi.org/10.1145/117009.117017
- Dictionary passing for polytypic polymorphism (2001) - https://www.cs.princeton.edu/techreports/2001/635.pdf
- A type-passing approach for the implementation of parametric methods in Java (2003) - https://doi.org/10.1093/comjnl/46.3.263
- Implementing Swift Generics (2017) - https://www.youtube.com/watch?v=ctS8FzqcRug

- The seminal paper on type classes and the dictionary passing approach. This drives the intuition behind type passing.
- Notes the same problems with monomorphisation and uniform representation (but uses different names), and comes up with different middle-ground technique, due to their use of pointer tagging.
- The earliest example of type passing I could find. Doesn't consider parametric polymorphism; it uses type passing to allow runtime pattern matching on types.
- What it says on the tin. I'm not inspired by it; it's just very relevant.
- The most recent incarnation of this approach. I haven't dug into the Swift compiler so I don't know if this has changed in the meantime.

Bonus slides - existential types

A simple existential type (Haskell-like language)

```
exists a. a
```

C equivalent

```
typedef struct {
  const Type* type;
  const void* value;
} Exists_example;
```

Logicians call parametric polymorphism “universal quantification”, and existential quantification is its dual. In previous slides the `forall` keyword gave rise to extra function arguments for type dictionaries. On the other hand, `exists` corresponds to pairs/products/structures that have a field for each type variable.

`exists a. a` is not very useful; all you can do with it is pass it around. It’s a useful minimal example nonetheless.

`exists a. a` is a type dictionary paired with a pointer to a value that the type dictionary can manipulate.

Notice that it contains a freestanding pointer. I claim that to compile unrestricted existential quantification you need a memory management system. I’ve started my experiments with automatic reference counting.

Bonus slides - higher-kinded types

A simple function with higher-kinded polymorphism (Haskell-like language)

```
id' :: forall (f :: Type -> Type) (a :: Type). f a -> f a
id' x = x
```

C equivalent

```
void id_prime(
    void* result,
    const TypeToType* f,
    const Type* a,
    const void* x
) {
    const Type* f_a = f->apply(a);
    f_a->move(result, x);
}
```

Here's a silly identity function that only works on 'container-shaped' types. When compiled, it gets an extra argument for each of the two type variables.

`f` is not a type dictionary, though - it's a type dictionary *constructor*.

Notice that the constructed dictionary `f_a` is behind a pointer. I think that higher-kinded types in the type passing approach also requires automatic memory management. For example, if `f a` was packed into the type component of `exists a. a` from earlier.

This kind (ha ha) of complication is why I'd like to benchmark real-world code compiled with this approach. Part of the reason I'm thinking about any of this is to reduce allocations, and it would potentially be a shame to just have moved them around. If higher-kinded types in this style were too slow or used too much memory to be practical, it would be better to not support them in the first place! I'm moderately optimistic, though.