```
Proposal for C23
WG14 2920

Title:               Tail-call elimination
Author, affiliation: Alex Gilding, Perforce
Date:                2022-01-25
Proposal category:   New features
Target audience:     Compiler/tooling developers, other-language developers
```

Abstract

C's function activation records implicitly form a stack, regardless of the underlying implementation of function calls or the depth of calls which the platform can support.

Keeping all activation records suspended but alive until their callees have returned prevents programs from transferring control directly when a function's work is already done. This is not a huge obstacle for hand-written C code, but is a major barrier for languages that compile to C or for interop with languages that do support this feature.

We propose adding the ability to explicitly, directly transfer control to another function without returning, as an alternative to nested function calls.

Tail-call elimination

```
Reply-to:            Alex Gilding (agilding@perforce.com)
Document No:         N2920
Revises Document No: N2855
Date:                2022-01-25
```

Summary of Changes

N2920

- `void` return type, minor syntax
- syntax vs. attribute, recommended practice vs. normative text
- prior art in Clang
- ABI concerns, tightened constraints around arguments

N2855

- original proposal

Introduction

C was designed to support recursive function calls. To achieve this, function activation records are arranged in a stack-like discipline, where the execution of any given block (and the lifetime of every object defined within it) is *suspended* but not *terminated* by a function call from within that block. The lifetime of the objects in the calling block may not end until the called block returns, making the stack-like structure implicit regardless of implementation (C targets need not provide any stack space for automatic variables in order to be conforming, but they must be able to return control to at least one caller).

Sometimes, a block needs to be able to transfer control to another block and has no reason to continue its own execution. In C, this presents only two options and neither is satisfactory:

- return information about the block to transfer control to, to the current function's caller, and use some kind of explicit dispatch there to move arguments and re-invoke
- call the next block directly and simply put up with the wasted space as the current block's lifetime is not allowed to end ahead of prescribed schedule.

If a block knows that it needs to transfer control out-of-function, and that it is safe to consider the lifetime of its local objects finished, it should be able to request early termination and replacement in a manner analogous to POSIX's `exec` at the process-level.

We propose a modification of the function return syntax to allow a function to return-by-replace and explicitly demand its execution terminate early, ending the lifetime of all local objects, before the invocation of a directly-called function.

In other languages this feature is known as *tail-call elimination*. If the last operation within a function is a value-generating call (ignoring `return` as a separate operation), it is in *tail position*;

calls in the *tail position* are *tail-calls* and can often be eliminated by direct replacement. This is called *tail-call elimination* and is a widespread concept outside C.

Rationale

The driving motivator for this feature is not hand-written user code. Rather, this feature would substantially improve C's usability as a compiler target for a number of functional (and functional - adjacent) programming languages that *mandate* support for tail-call elimination within their own language specifications.

The most famous of these is probably Scheme, which used the concept to derive entire new control flow and compilation models based around *continuation-passing*. Lua is another popular language that mandates tail-call elimination, as do the ML family and their descendants.

Scheme and Lua in particular are quite easy to compile to C (albeit more difficult to optimize, partly for unrelated reasons stemming from the dynamic type system), except for this one "killer" feature: a Scheme function call cannot be directly represented as a C function call because the Scheme compiler cannot trust C to eliminate invocation records that the Scheme specification says are no longer needed to be live.

This significantly damages one of C's historic constituencies: compilers of other languages looking for a really easy-to-generate, already-optimizing intermediate language.

A number of workarounds exist. The Chicken Scheme compiler uses direct C function calls, and places every call in tail position, and detects C machine stack exhaustion and periodically clears the stack by jumping back to the base (this can be implemented both in portable and in optimized ways). This is mostly of interest as a GC strategy. Although calls are mostly native, there is still a significant performance impact because the Scheme runtime needs to provide a lot of bookkeeping in order to detect and implement the base-jump.

The Gambit Scheme compiler achieves significantly better performance at the cost of generating completely unmanageable C-machine code, using a module-level compilation strategy (previously also identified by Demoen and Maris), where a Scheme module ("translation unit") compiles as far as possible into a single C function; Scheme function calls are represented by a non-native call mechanism implemented as either a `switch` or a language extension like GNU's label pointers. This reduces the cost of in-module function calls significantly, but creates extremely high cost to call functions in other TUs and to switch between C and Scheme functions.

In addition to the performance cost of calls, the interop between Scheme and "native" C functions is prohibitively complicated under these compilation models. Scheme, C, Lua, etc. all have their own expectations about how and how often their language's equivalent of a "long jump" can be performed: `longjmp` in C, which is rare/high-cost; coroutines in Lua which are a medium-impact tool; and continuation-invocation in Scheme which is common and supposedly zero-cost/transparent. Given the example of C as the host language for a Scheme program, it is difficult for Scheme to jump out of a Scheme callback-to-a-C-callback because the continuation needs to know about, and have some ability to clear, the intervening C invocation records. Other scenarios are similarly complicated.

(The Gambit manual understates the problem slightly, as it glosses over the fact that a user should not need to know what language their library of choice was written in and decide how to call it based on that.)

Tail-call "optimization"

This feature is commonly known in the C world as *tail-call* or *tail-recursion* **optimization**.

In practice all major optimizing compilers *can* detect that a call is in tail position and meets the criteria for elimination (no dependence on lifetime of objects, returns directly, etc.). GCC, Clang, Intel and MSVC are all capable of performing this rewrite. However they are biased in two ways that do not meet the needs of the C-targeting constituency:

- they tend to focus on the technique as an optimization for recursive algorithms, allowing C users to write a recursive function rather than being forced to use a loop. The focus on the general case is less because non-recursive functions are not assumed to have as strong a need for TCE; they are assumed to tend towards shallow and finite activation stacks.
- the technique is presented as an *optmization* selected by the compiler, not as a user selectable call mechanism. It is analogous to inlining in that there is no good way to request it explicitly; the compiler will do it if it detects that it can, and will leave the function invocation alone if it does not meet the criteria for obvious/safe in-place replacement.

Whether the nested invocation is a recursive call or not is considered semantically irrelevant to the other-language use case. We explicitly avoid referring to this feature as *tail-recursion* for that reason.

Compilers of other languages targeting C, choose it because of its portability features and its simplicity to generate. Relying on invisible C-compiler-side optimizations for essential parts of their language's semantics is completely unreliable and defeats a large part of the reason for choosing C (portability), while working around the lack of TCE with portable code defeats the two other major reasons (simplicity and performance).

We therefore consider that it should be possible for code generated by compilers to explicitly make a binding request for in-place replacement of an activation record.

We do not expect this feature to see significant use in user-written C code, though it is possible to use correctly and safely. (A C function that does not pass pointers to local objects through to the tail call will not introduce any new undefined behaviours.)

Proposal

Most existing work focuses on detection of the opportunity for tail-call elimination (Probst). This is now well-understood by optimizing compiler developers, but not sufficient. In addition it is not necessary for so much work to be put in by the C compiler if the request can be made explicitly: a *tail-call* happens where the user says it does and the C compiler should trust the other-language compiler not to introduce UB by passing dependencies to objects that have fallen out of scope, so it doesn't need to prove the replacement is "safe" from a C-language perspective.

The other-language compiler should also trust the C compiler to honour its requests rather than have to rely on obscure optimizations whose rules may be undocumented or shifting.

We instead propose a minor adjustment to C's syntax to communicate an explicit request for a *tail - call*: `return goto`.

For a function call to be in *tail position*, it must be the **immediate** operand of a `return` statement:

```
int foo (int, int , int);

int bar1 (int a, int b, int c) {
  int x = foo (c, b, a); // not tail position
}

int bar2 (int a, int b, int c) {
  int x = 0;
  x = foo (c, b, a); // not tail position
  return x;
}

int bar3 (int a, int b, int c) {
  return foo (c, b, a); // tail position
}
int bar4 (int a, int b, int c) {
  return foo (foo (1, 2, 3), b, a); // outer foo is in tail position
}
int bar5 (int a, int b, int c) {
  return 1 + foo (c, b, a); // NOT tail position
}                           // bar5 still needs to perform an add after foo
int bar6 (int a, int b, int c) {
  return (foo (c, b, a)); // tail position, grudgingly
}
```

This implicitly already requires the type of the called function to be convertible to (but not compatible with) the return type of the caller.

We therefore propose that `return` statements whose operand consists of a proper *tail call* can be enhanced with the `goto` keyword, communicating intent to jump control "across" rather than to nest:

```
int bar3_g (int a, int b, int c) {
  return goto foo (c, b, a); // explicit tail call
}
```

We introduce a new *constraint* that the type of the tail-call's *postfix-expression* be compatible with the type of a pointer to the calling function. This creates implicit *constraints* on the return type and arguments, and (by phrasing it this way) also implicitly covers ABI compatibility issues (since functions with different calling conventions will have pointer types that are incompatible for implementation extended reasons).

The implicit constraint on the return type of the called function is that it must be identical rather than just implicitly convertible:

```
float qux (int, int, int);

int bar3_wrong (int a, int b, int c) {
  return goto qux (c, b, a); // NOT a valid tail call
}                            // because the float->int work remains to be done
```

The implicit constraint on the number and type of the callee's parameters is that they must be identical to those of the caller. This is because on many platforms the argument space cannot be modified after it is set up, so adding new arguments might require moving the frames around in a way the ABI does not support.

We also add a constraint against the use of functions with variable argument lists in tail position, as it is impossible for the type system to prove that they are called with identical arguments to the containing function. No explicit new constraint against the use of KnR-style functions is needed because these are no longer part of the language.

We do not need to add any Recommended Practice touching on out-of-scope ABI issues such as calling conventions, because any reasonable implementation *must* distinguish functions with incompatible calling conventions by extending the type system somehow. (Otherwise it already silently allows calls that can corrupt the stack!)

We do not introduce any new constraints w.r.t passing pointers to local objects to the callee. This is not considered a **new** UB; this is functionally identical to passing a pointer to a local object out by return or assignment to global. We choose to trust the other-language compiler and save the C compiler from having to make this analysis.

We do not introduce any new constraint that the `return goto` statement be in any particular *tail position*. It is in *tail position* by virtue of being a `return` statement; all existing rules about what can syntactically "follow" a `return` statement still apply, without modification.

We introduce a constraint that the operand to `return goto` shall be a *tail-call*. A diagnostic must be issued if the request to transfer control in-place cannot be met or does not make sense (e.g. if the operand is not even a function call). This is more likely to be a user error (missed that trailing `+ 1`...) than to be acceptable as a generic substitute for `return`.

We do **not** modify the object lifetime rules. An object's lifetime continues to be from entry into its containing block, until execution of the block is terminated rather than suspended. We instead modify the rule that when a function call is the operand to a `return goto` statement, it first terminates execution of the containing block, and adopts the calling function’s caller as its own; after executing it returns directly to that caller (unless it has itself also been replaced directly and terminated before returning). Therefore entering a *tail-call* is exactly the same as exiting the calling function, and semantically quite distinct from making a function call within it.

Community feedback suggested that an attribute on a regular-syntax `return` statement might be preferable for linguistic reasons. However, this is impossible: a standard attribute can be ignored without changing the meaning of a correct program, while changing a `return goto` statement to a `return` statement (or vice-versa) *significantly* changes the meaning of both the statement itself, and the calling function’s scope in terms of object lifetimes.

For this feature to be ignorable would make it “dangerously non-portable”; when it is requested, it is because tail-call elimination is *essential* to program correctness. It is a semantic feature, not an optimization, and cannot be treated as ignorable. A compiler must reject the statement if it cannot perform the tail-call; to continue to translate without an error would be *extremely* user-hostile.

It would of course be possible to specify a new standard attribute and say that it is not ignorable. However, this would be confusing to users and change the existing rules and purpose of attributes.

Standard attributes add meaning but do not change it; when we want to change meaning we communicate that by changing the underlying syntactic formulation.

It would also likely be confusing to implementers and raise the threshold for correct implementation. It may interact poorly with additional tools that strip or ignore attributes because they do not expect them to change the meaning of the control flow. Finally, we do not agree that the proposed syntax is at all burdensome (it adds no new keywords).

Community feedback also asked whether a calling function should require an attribute or other modifier marking that it ends in a tail call. There is no use case for this and it exposes an implementation detail to users of the calling function. The calling function does not itself require any special setup; its return slot and argument slots are reused in-place, and should be the same as those set up for any other function call.

Community feedback asked whether a function should require an attribute marking it as "tail-callable". In practice this is determined by its usage and we do not see much purpose for this – if the user tries to use an obviously-non-tail function (such as one with a variable argument list), the feature will immediately error locally. If the user is calling a function in non-tail position, it is irrelevant to the local context whether it is also used in tail position elsewhere. Otherwise, if the user is calling an acceptable function in tail position the main cause of errors would be mismatch with the caller's argument or return setup, which is context-dependent and not a property of the called function by itself.

Prior Art

Clang supports this feature by means of the vendor-specific `[[musttail]]` attribute. In Clang, "the compiler must generate a tail call for the program to be correct, even when optimizations are disabled. This guarantees that the call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph."

Clang requires the argument types, argument number, and return type to be the same between the caller and the callee, as well as out-of-scope considerations such as C++ features and the calling convention. Implementor experience with Clang shows that the ABI of the caller and callee must be identical for the feature to work; otherwise, replacement may be impossible for some targets and conventions (replacing a differing argument list is non-trivial on some platforms).

This informs the requirement for complete-sameness of caller and callee in the Standard feature, as there is no point in a portability feature allowing formulations that would only work on "simple" target platforms like x86. This also informs the Recommended Practice to reject some tail-calls that would be compilable on such platforms – the user should know quickly that their goal of portability cannot be met.

Because this is a vendor-specific attribute, it can change observable program semantics: it enforces the tail-call discipline on the return-call expression with the various effects described above. The attribute cannot be imported directly into ISO C because as an attribute, the primary property of *enforcing* tail-call discipline would be lost.

Alternatives

As above, this is widely supported as an optimization, but it is subject to the as-if rule; any change in the call that would result in it becoming unsuitable for TCO (as opposed to TCE) will silently leave the caller's activation record alive until after the callee returns.

We do not believe it is appropriate for the semantics of code generated by an other-language compiler to depend on an optimization detail of the generated C. The code must work correctly even with optimization disabled. In addition optimization can be "disrupted" by any number of unexpected factors, such as timeouts or compiler resource exhaustion - most optimizers do not promise to find every optimization possible, only to try to do so with a reasonable amount of effort. Language extensions can very easily disable the feature because they rely on unchanged lifetime rules (e.g. ARC in Clang/Objective-C; `__attribute__((cleanup))` in GNU C).

Existing C-targeting compilers generally use some form of trampolining or scheduling. They may also use techniques such as subroutine-threaded code (not to be confused with multithreading). The line between these techniques is not strict.

In all cases these require explicit management of a full call-return-reschedule on the C code side. This is usually impossible to optimize away by the C compiler, but the entire operation is fundamentally not needed by the underlying platform which can generally just blind-jump to any machine instruction.

There are other extensions such as GNU's labels-as-values that can help smooth the implementation, but fundamentally do not change the technique used or the fact that a large amount of code needs to be inserted on the C side just to provide a control flow operation that logically should require *no* code.

`goto return` may be clearer to read than `return goto`; we are not tied to the syntax. However, we do not believe that `goto function ();` would be appropriate because it is too visually similar to syntax that would be valid for the GNU labels-as-values extension (only missing a `*`).

Impact

This proposal introduces new syntax but no new keywords, only a rearrangement of existing ones. Therefore there is no namespace or reservation conflict implied by the new *tail-call* statement.

All existing techniques for compiling *tail-calls* from other languages should continue to work. These will continue to provide a reliable base for targeting older compilers even if C23 compilers start to provide a new high-performance path for new programs.

No existing *tail-call optimization* should be impacted by this feature addition. The technology is already mature. It is possible that compiler developers feel less pressure to investigate improving the optimization if the elimination feature is added instead.

We do not consider code that relies on the existing optimization, no matter how reliable, for correct semantics, to be correct code in any case, because this would violate the "as-if" rule.

No new undefined behaviour is introduced by this proposal. Passing a pointer to a local object with automatic storage duration is not logically different from returning such a pointer. We expect existing analyses to apply, and existing rulesets to be updated. We would rather allow the request to be binding and trust the **primary** client of the feature, which is other-language compilers that can (hopefully) prove correctness of their own resource use, to use it safely and trust it to work. We do not really expect human writers of C to get much mileage out of this.

Unlike many instances of the optimization, we do not propose to enforce any particular rules about the number and type of parameters. Even if a *tail-call* has more parameters than the caller, the amount of space required for any given argument invocation is fixed syntactically at compile time - the argument buffer can grow but only to a known limit. A compiler should be able to determine the upper bound, but also shouldn't need to. Fundamentally this is not different from the callee requiring more space for its internal automatic variables. This does not introduce any new risks for resource exhaustion above those that already exist in C's function calls.

Allowing native C functions and compiled-to-C functions to use the same calling convention will dramatically improve C's interop capabilities.

No new ABI is introduced by this feature. Although the a function called from tail-position replaces its syntactic caller, entry re-uses the same argument and return space that was set up for that caller. Therefore, whether a function ends in a tail call is not observable from “outside”. Because the called function must match the caller’s setup exactly, it must by definition have the same ABI and is also callable from any non-tail position, making its potential status as a tail-call completely private to the call context and not observable from its definition or declaration at all. The caller and the callee may require a different static size for their activation records, but we are not aware of a target (where the stack discipline is actually implemented in a re-entrant way) where this would be problematic (on targets with re-entrant stack disciplines the top of the frame is always necessarily set *within* the function body because it is not exposed by the type anyway; on those with non-contiguous stacks a different object is used; etc.).

It is not generally possible to perform a tail-call between functions that have different calling conventions. Although this is outside the scope of the Standard, by placing constraints on the type of the callee function as a whole (rather than expressing separate constraints against its return type and arguments), calling conventions will be caught as well because an implementation already has to treat them as distinguishing types in order to emit correct code. In general the calling convention itself does not seem to matter so much – all conforming calling conventions support mutable parameters and a return slot accessible to the caller, and these are all the feature needs. Distinctions such as caller-cleanup vs. callee-cleanup must be covered by an implementation-provided type differentiator, but which is actually used doesn’t matter because the restriction that the types of the functions be the same means that the frame setup and teardown will happen once and consistently for both caller and callee regardless.

We believe that this feature could easily be supported by C++ because the changes to object lifetime (and correspondingly destruction) are unambiguous and match the new exit point from the calling function. There is no risk that a correct C++ program might change its behaviour silently because the `goto` keyword must be added manually in order to change the lifetimes, making any consequences for object destruction visually clear.

C++ is in any case free to add additional constraints relevant to its additional features (e.g. trivial objects only). Tail calls were discussed in the context of coroutines in P1063 "Core coroutines", which reached similar conclusions about the effects on object lifetime, and the use of only objects with trivial destructors in the calling function was one such additional constraint.

The C++ proposal also suggested that the definition must be visible in the same TU and that the tail call cannot be by-pointer; it is not obvious that these should be required for correct replacement – a machine code jump to any function pointer should be possible after compatible argument setup, on any ABI. This restriction would also prevent the feature from solving the general case of the problem for other-language compilers, which require indirect jumps to perform TCE as well.

It is possible that a target platform which does not support indirect jumps could have difficulty with this feature (only for calls through function pointers). We are not aware of such a machine.

Proposed wording

Changes are proposed against the wording in C23 draft n2731. Bolded text is new text.

Modify 6.2.4 "Storage durations of objects" to clarify that some function calls do terminate execution:

> For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block **footnote)**.)
>
> **footnote: unless the call explicitly terminates execution of the current block with `return goto`.**

Add a forward reference to (new section) 6.8.6.4.1 in 6.2.4:

> Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), **tail calls (6.8.6.4.1),** initialization (6.7.9), statements (6.8), effective type (6.5).

Add a new footnote to 6.5.2.2 "Function calls" paragraph 10, immediately after 104:

> with respect to the execution of the called function. 104) **footnote)**
>
> **footnote: if the called function is assuming direct control with the `return goto` statement, then no further operations in the caller will be evaluated after it is entered.**

Modify 6.8.6 "Jump statements" paragraph 1:

> *jump-statement*:
> `goto` identifier `;`
> `continue ;`
> `break ;`
> `return` *expression* opt `;`
> `return` `goto` *postfix-expression* `;`

Add a new section, 6.8.6.4.1 "The `return goto` statement":

6.8.6.4.1 The `return goto` statement

Constraints

The expression shall be a function call, optionally enclosed in parentheses. No other form of expression is permitted.

Within the function call expression, the *postfix-expression* designating the function to call shall have a type compatible with the type of a pointer to the function whose definition contains the statement.

Neither the function to call nor the containing function shall have a parameter list that terminates with an ellipsis.

Semantics

A `return goto` statement is a special case of the `return` statement which terminates execution of the current function and then passes direct control to the function specified in the function call that constitutes its operand expression.

The value of that function call will be returned to the caller as in a simple `return` statement. The `return goto` statement differs from the `return` statement in that it terminates execution of the current function immediately after evaluating all operands to the function call expression, and before entering the called function itself. The called function will return directly to the current function's caller.

Because the execution of the current function is terminated, the lifetime of all objects local to the function with automatic storage duration ends, immediately before the called function is entered.

A function that has been terminated by the `return goto` statement does not continue to use resources.

NOTE: a `return goto` statement may appear anywhere a `return` statement with an expression may appear.

NOTE: if the address of a local object with automatic storage duration is passed to the called function, its lifetime will have ended before the called function begins executing and the pointer cannot be used.

EXAMPLE 1 This example violates the constraint that the expression must be a direct call to a function returning the exact same type as the caller:

```
int foo (int, int);

int bar (int a, int b) {
  return goto foo (b, a) + 1; // WRONG: the +1 must be evaluated after
the result of foo()
}
float baz (int a, int b) {
  return goto foo (b, a) + 1; // WRONG: the result of foo() is
followed by an implicit conversion
}
```

EXAMPLE 2 In this example the address of a local object with automatic storage duration is passed to a called function:

```
int foo (int * p); // uses p

int bar (int a) {
  return foo (&a); // OK, lifetime of a continues until foo()
completes
}
int baz (int b) {
  return goto (&b); // WRONG: using the address of an object whose
lifetime has ended
}
int * boo (int c) {
  return &c; // roughly analogous to the above
}
```

EXAMPLE 3 In this example, a function recurses endlessly but harmlessly because the recursive call consumes no additional resources:

```
int foo (int a, int b) { // need space for locals...
  return goto foo (b, a);  // ...but that ends here
}
```

This class of function cannot overflow the program stack by itself.

EXAMPLE 4 In this example only one of the two calls to `foo()` is in the tail position:

```
int foo (int a, int b);

int bar (int a, int b) {
  return goto foo ( // will be evaluated after this caller's lifetime
ends
      foo (a, b),   // will be evaluated within this caller's lifetime
      a + b
  );
}
```

The first nested call takes place before termination of the calling function and therefore must consider that its resources have not yet been released, exactly as for any other function call that is not in tail position.

No modifications to the Standard Library are proposed.

Additional options

Should we add a further *constraint* to new section “6.8.6.4.1 The `return goto` statement”?

> A `return goto` statement shall only appear in a function whose return type is not `void`.

This can be considered implicitly required, because `return goto` is a special case of the `return` statement with expression. But it could also be a useful extension to the syntax for both `return goto` and `return` in general (prior art for “returning” a `void` expression from a `void` function exists in C++).

The first existing constraint already ensures that in such a statement the expression type would be `void`. On the other hand, since this functionality is mainly intended for use by C code generators, stricter requirements on the signature are not as much of a usability burden (i.e. most compilers will generate code that returns a value here anyway).

Community feedback focused inordinately on the proposed syntax. Outside the committee, there was widespread feeling that this should visually be a variant of the `goto` statement and that there was no real risk of confusion with GNU C's computed goto syntax. A new `_Keyword` was also suggested as a possible option. We disagree that this is necessary since the existing keywords can be arranged to make the syntax unambiguous.

Would the committee prefer:

- `return goto`; or
- `goto return`; or
- just the `goto` keyword; or
- a new _Keyword?

We strongly oppose the use of an `[[attribute]]` to tag these return statements.

References

- C23 n2731
- exec()
- Feeley 97
- Chicken
- Baker 95
- Demoen, Maris 94
- Gambit Scheme
- Lua
- Probst, Proper tail recursion in C
- Intel TCO
- GCC TCO
- Clang TCO
- Clang [[musttail]]
- C++ P1063R2 "Core coroutines"
- Subroutine threading
- Trampolining