Compiling Algebraic Effects to C++

Marcos Vinicius Moreira Santos
Orientador: Fernando Magno Quintão Pereira [1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

Abstract. Algebraic Effects *are an approach to computational effects based on a premise that impure behavior arises from a set of operations and they give to the programmer a powerful framework to construct advanced control flow abstractions on a pure functional way. During this report, it will be described a technique used to construct a simple compiler capable of transforming a simple language that supports abstractions similar to Algebraic Effects into C++ code.*

Resumo. *Os* Efeitos algébricos *são uma abordagem para efeitos computacionais baseada na premissa de que o comportamento impuro surge de um conjunto de operações e fornecem ao programador uma estrutura poderosa para construir abstrações avançadas de fluxo de controle de maneira puramente funcional. Durante este relatório, será descrita uma técnica utilizada para construir um compilador simples capaz de transformar uma linguagem simples que suporte abstrações semelhantes a Efeitos Algébricos em código C++.*

1. Introduction

Algebraic Effects and *Effect Handlers* provide a modular foundation for user-defined effects by separating the definition and implementation of effects. When a handler is executed, it runs in the scope that catches the effect. They can be thought of as a generalization of exception handlers in languages such as Java and C++, with the added ability to resume a computation that was interrupted by an exception. Like exception handlers, they are executed in the scope that catches the exception and contain the definition for the handler, usually inside a catch block.

In recent years, *Algebraic Effects* have gained attention in research languages such as Koka and Effekt, and there is also an effort to bring an Effect System to Multi Core OCaml. Matija Pretna's article "A Gentle Introduction to Algebraic Effects and Handlers" [Pretnar 2015] provides a clear introduction to "Algebraic Effects" with many examples that can be understood by programmers with little or no prior knowledge of them. The examples are built on a simple framework of an extended lambda calculus to construct powerful control flow abstractions such as *Mutable State*, *Exceptions*, *Input & Output*, *Non Determinism*, *Backtracking*, and more using the semantics of *Algebraic Effects*.

While efficient techniques for compiling *Algebraic Effects* do exist, such as the method used in Multi Core OCaml which utilizes segmented stacks, these techniques often require platform support, making them difficult to port to more restrictive

targets such as Web Assembly. In their article [Xie and Leijen 2021], Xie and Leijen describe the technique used by the Koka compiler to compile the Koka language into efficient C code without the need for a runtime system like a garbage collector. They achieve this by using a special reference counting mechanism as described in [Alex Reinking and Leijen]. In this report, we will discuss a technique inspired by [Xie and Leijen 2021], which can be used to compile a simple language supporting the semantics of *Algebraic Effects*, and with further development, could potentially fully support them in the future.

This report is divided into three sections. First, we will briefly discuss the language we will be using. Second, we will present all the steps needed to transform the language into something that can be easily translated into C++ code. Finally, we will discuss the next steps and lessons learned during the development of this project.

2. The Language

A language was designed and implemented in order to facilitate the study and development of the technique within a simplified environment. This language is statically typed and supports as base types 32-bit integers, handler collections, the *any* type which is equivalent to a void pointer in C++ and the *unit* type. Additionally, the language supports structs and pointers. Its syntax is straightforward enough to be understood by individuals with a moderate level of experience in imperative programming. The imperative style was chosen due to the author's preference and because *Algebraic Effects* have not been extensively studied within this paradigm.

The following are the primary rules for the language's syntax. Some rules that are considered trivial, such as those for $< symbol >$ and $< number >$, will be omitted. It should be assumed that these omitted rules follow the same definitions as those found in languages like C and C++.

⟨*type*⟩ ::= ⟨*type*⟩ '->' ⟨*type*⟩
| ⟨*type*⟩ * ⟨*type*⟩
| unit
| handler_t
| i32
| any

⟨*decl*⟩ ::= ⟨*symbol*⟩ ':' ⟨*type*⟩

⟨*constant*⟩ ::= ⟨*decl*⟩ ':' ⟨*expr*⟩

⟨*variable*⟩ ::= ⟨*decl*⟩ '=' ⟨*expr*⟩

⟨*call-args*⟩ ::= ⟨*symbol*⟩ ',' ⟨*call-args*⟩
| ⟨*symbol*⟩

⟨*func-call*⟩ ::= ⟨*symbol*⟩ '(' ⟨*call-args*⟩ ')'

⟨*effect-call*⟩ ::= ⟨*symbol*⟩ '!' '(' ⟨*call-args*⟩ ')'

⟨*func-args*⟩ ::= ⟨*decl*⟩ ',' ⟨*func-args*⟩
| ⟨*decl*⟩

⟨*func-literal*⟩ ::= '(' ⟨*func-args*⟩ ')' '->' ⟨*type*⟩ '{' ⟨*statements*⟩ '}'

⟨*effect-declaration*⟩ ::= '(' ⟨*func-args*⟩ ')' '->' ⟨*type*⟩

⟨*assignment*⟩ ::= ⟨*symbol*⟩ '=' ⟨*expr*⟩

⟨*if-stmt*⟩ ::= 'if' ⟨*symbol*⟩ '{' ⟨*statements*⟩ '}' ⟨*else-stmt*⟩

⟨*else-stmt*⟩ ::= 'else' '{' ⟨*statements*⟩ '}'
| 'else' ⟨*if-stmt*⟩
| ⟨*empty*⟩

⟨*ret-stmt*⟩ ::= 'return' ⟨*expr*⟩

⟨*prompt*⟩ ::= ⟨*func-call*⟩ 'with' ⟨*symbol*⟩

⟨*func-literals-decls*⟩ ::= ⟨*decl*⟩ ':' ⟨*func-literal*⟩ ';' ⟨*func-literals-decls*⟩
| ⟨*empty*⟩

⟨*handler-effect-decls*⟩ ::= 'handler' '{' ⟨*func-literals-decls*⟩ '}'

⟨*expr*⟩ ::= ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ - ⟨*expr*⟩
| ⟨*expr*⟩ * ⟨*expr*⟩
| ⟨*expr*⟩ / ⟨*expr*⟩
| ⟨*expr*⟩ / ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*func-literal*⟩
| ⟨*effect-declaration*⟩
| ⟨*symbol*⟩
| ⟨*number*⟩
| ⟨*handler-effect-decls*⟩

⟨*statement*⟩ ::= ⟨*constant*⟩
| ⟨*variable*⟩
| ⟨*assignment*⟩
| ⟨*ret-stmt*⟩
| ⟨*if-stmt*⟩
| ⟨*expr*⟩
| ⟨*empty*⟩

⟨*statements*⟩ ::= ⟨*statement*⟩ ';' ⟨*statements*⟩
| ⟨*constant*⟩
| ⟨*variable*⟩
| ⟨*assignment*⟩
| ⟨*ret-stmt*⟩
| ⟨*if-stmt*⟩
| ⟨*expr*⟩
| ⟨*empty*⟩

Here is an simple example:

```
ask : i32 -> i32 : (a: i32) -> i32;
f : unit -> i32 : () -> i32 {
```

```
  x :: ask!(4);
  return x;
}
g : unit -> i32 : () -> i32 {
  w : i32 = 0;
  read : handler_t : handler {
    ask : i32 -> i32 : (a : i32) -> i32 {
      one : i32 = 1;
      x : i32 = resume(one);
      z : i32 = x + w + a;
      return z;
    }
  }
  x : i32 = f() with read;
  return x;
}
```

The given example declares an effect called *ask* that takes in an integer and may resume the computation with another integer that is dependent on the handler. The example also defines two functions, f and g. f simply performs the *ask* effect and returns the value that the computation is resumed with by the effect handler. As a result, every time f is called, a handler that provides a definition for *ask* must be found on the stack. The function g defines a variable w and a handler collection called *read* with an effect handler for the *ask* effect. It then calls f with *read* as the handler collection for f and returns the result of f. The handler *read* defines an implementation for *ask* that calls the operation *resume* once. Currently, this is the operation used to resume computations that were stopped by effect invocations. However, this is a poor design decision and in the future it will be better to pass an explicit reference to the resumable computation as a callable argument, such as a function pointer passed to the effect handler. The declaration for *ask* in *read* should include this argument. We will discuss this decision further later in the report.

3. Methodology

To support *Algebraic Effects*, we need a mechanism similar to Exception Handling, where we go back on the stack to the point where a catch operation is performed, but also supporting the resumption of the computation that got interrupted. To be able to support this, we need to capture the *continuation* of the computation being performed during the invocation of the effect. A *continuation* is just an abstract representation of the control state of a given program, for our purposes, it is a point in the program with which we can stop, and resume later depending on our program runtime. To capture a continuation, we need to capture two pieces of information, the data being used by a given computation, and the sequence of instructions that will use that data from that point forward, thankfully, the piece of computation that uses the data, is know at compile time, since it is just the sequence of instructions from that point forward, the contents on the stack in the other hand is not know at compile time.

In order to support *Algebraic Effects*, we require a mechanism similar to exception handling, where we return to the point in the stack where a catch operation is performed, but with support for the resumption of the interrupted computation. To achieve this, we must capture the continuation of the computation being performed during the effect's invocation. The continuation is an abstract representation of a program's control state, which in our case represents a point in the program where we can pause and resume later. Capturing a continuation requires us to capture two pieces of information, the data being used by a given computation and the sequence of instructions that will use that data from that point forward. Fortunately, the piece of computation that uses the data is known at compile time, as it is simply the sequence of instructions from that point forward. However, the contents of the stack at any given point are not known at compile time.

In [Xie and Leijen 2021], the authors describe how continuations are captured in Koka [Kok]. We will use a similar technique in which we capture continuations by splitting the program at points where a continuation may need to be captured and saving the current stack frame at those points. To resume the computation, each split point will become a function that receives the data it requires as arguments. These functions will then be composed to resume the computation, for example, given the data that continuation functions f, g, and w require, we can resume the computation with the calls $f(g(w(x)))$, where x is the input to the continuation or the argument to *resume* in our case.

In our example, functions f, g, and w will be captured during the program's runtime. Each time we call an effect, we must pop the call stack until we find the handler for the effect being performed. This is where we capture functions f, g, and w, as well as the data they require, which is known statically. We will refer to the process of popping the stack to find the catch point of effects as "yielding," and we will refer to the saving of continuations during yielding as "bubbling." This process is explained in more detail in [Xie and Leijen 2021].

The pipeline that we're going to discuss, compile the program through a pipeline of transformations:

- Handler Pass: The first pass, takes all handler objects, like the *read* handler that we use as an example earlier for example, and transform them into function literals, that can be compiled later into normal C functions, this process also replaces any *with* statement by function calls, so at the end of this process, no *handler* declaration or *with* statements should be present on the resulting IR.
- Stack Frame Allocation Pass: This pass allocates heap memory for stack frames, we're currently using a naive strategy in which we just linearly increases the size of the stack frame to make room for all local variables, our approach does not support dynamically increasing the size of the stack frame at runtime, all variables needs to have they size known at compile time as well. After that we insert the stack frame allocation and stack frame deallocations at the beginning and at the end of the function and then remove any variable declaration, replacing the use of the local variables by a simple offset and a type cast on the allocated stack frame. Its worth remembering that we're

currently using the C++ stack to store the function return value, so before freeing the stack frame, we allocate an extra variable, outside of the stack frame, to store the return value currently saved on the stack frame buffer, making sure that the return value outlives the stack frame that will be freed before the function returns.

- Continuation Pass: During this pass we split the program creating functions like the f, g and w that we talked about earlier, they are created as local lambda functions that are lifted later in the pipeline. During this pass, the created lambdas may call the methods responsible for freeing the stack frame that got inserted in the previous pass, we also lift those calls to the top most enclosing function, so no continuation pops the stack frame that it uses after being called.
- Bubbling Pass: During this pass, we insert all the checks and logic needed to handle the *yielding* and the *bubbling* processes.

  All those passes are going to be discussed in much detail later.

3.1. The Context object

During the program execution, some information will have to be recorded. To store information at runtime, we're going to use a context object, that is a data structure that will be passed down on every function call as an extra argument, and is going to store information like what Effects Handlers are currently available to the program, so if the program is *yielding*, all the continuations being captured are stored at this context object as a linked list, this list will also store then in the order in which the functions will need to be composed, remember that we're going to restore a continuation by function composition, like in $f(g(w(x)))$, so when we are capturing the continuations, we store the functions f, g and w inside a linked list in the context object alongside they respective stack frame buffers. In resume, any contextual information needed by the runtime should be present on the context object.

3.2. The Compilation Pipeline

3.2.1. Handler Pass

The first pass of the compilation pipeline is responsible for transforming every *handler* declaration and every *with* statement into simpler constructs, this is done in the following steps:

1. The compiler generates a unique numerical hash value for every effect declaration.
2. The compiler rewrites every effect declaration into a function literal, this function is responsible for making the setup of the context object that is needed for the bubbling process to start at runtime by setting some values on the context object itself.
   (a) First, all the arguments for this function are collapsed into a single argument that will be a pointer to a structure that will contain all the old arguments as members.

(b) The body of this function will set the values for the context members responsible for holding which effect is the runtime trying to find a handler for, and also if the runtime is yielding or not, this is done by saving some boolean values as well as the hash of the effect declaration.

(c) Heap space is allocated for the structure argument and all arguments being passed to the effect are saved on that buffer.

(a) All Handlers are rewritten into a lambda function, that we'll be referring to as the *prompt lambda function*, it will have two arguments, an input buffer called *in* and an output buffer called *out*. the body of the resulting functions is generated by the following steps:

   i. All functions that are effect handlers defined inside the handler are converted into local lambdas with their arguments collapsed to use a pointer to the *struct* created on step 2, remember that the handler will have the same arguments as the effect declaration, they also receive an extra argument, a pointer to the return type of the function that defines the handler being rewritten, the first argument will receive the name *args*, and the second will be called $prompt_{ret}$.

   ii. Since effect handlers need to be executed at the scope of the function that defines them, if a handler performs return, it should take effect on the enclosing function, not just on the prompt lambda itself, in our example, the statement $returnz$ inside the effect handler for ask makes g returns z. To support that behavior, before every return statement inside the handler, we add a method that signals in the context that the called prompt lambda called return, the return statement itself is also converted in a way that the returned value is stored on the $prompt_{ret}$ argument, so $returnz$ becomes $*prompt_{ret} = z$.

   iii. A big switch statement is inserted, this switch checks the hash saved on the context, if the hash being handled is also a hash of any local prompt lambda created earlier, then we use the context to get the buffer with the arguments, this buffer needs to be casted to a pointer to the *struct* created at step 2.a, after that the corresponding lambda is called passing the arguments buffer as the first argument and the *prompt_ret* argument is passed as the second argument. After the prompt lambda call, the context should be notified that the effect was handled and then the program checks using the context if the prompt lambda performed a return operation, if it did, then the prompt immediately returns a value that signals that at least one effect was handled inside this prompt function, remember that the return value is already saved in the second argument passed to the prompt function.

   iv. If no return operation was performed inside the switch statement, then the resulting prompt function should return a value that signals if at least one effect was handled.

(b) After every *with* statement, we allocate an extra variable, that we'll refer to as $prompt_{ret}$, it'll have the same type as the return type of the function. After allocating this variable, we call the prompt lambda function, passing null as the first argument, and the address of the $prompt_{ret}$ as the second argument, so it will be used as the output buffer for the prompt closure. The first argument is here because the prompt function needs to maintain a common interface with continuation functions that we are going to see later in this report.

(c) After every prompt call, we check using the context to see if any effect handler performed a return, in this case we signals to the context that the return was handled, and return the value saved on $prompt_{ret}$, otherwise we continue to execute the function normally.

(d) We insert on every function literal a reference to the context object as an extra argument, and we also pass down this context object into every function call.

We will be using an extended version of our language defined earlier here but with minimal changes, *mut* just means that the defined variable is mutable.

The resulting of this pass on our first example should look something like this:

```
args_ask : struct = struct {
  mut a : i32
}

ask : i32 × context* -> i32 = (mut a : i32, mut ctx : context*) -> i32 {
  set_is_yielding_to(0, ctx)
  t0 : any* = ctx_allocate_args(sizeof(args_ask), ctx)
  t1 : args_ask* = (args_ask*)(t0)
  t1.a = a
  return 0
}

f : context* -> i32 = (mut ctx : context*) -> i32 {
  x : i32 = ask(4, ctx)
  return x
}

g : context* -> i32 = (mut ctx : context*) -> i32 {
  mut w : i32 = 0
  ask : args_ask* × i32* × context* -> i32 = (
    mut args : args_ask*, mut prompt_ret : i32*, mut ctx : context*
  ) -> i32 {
    mut one : i32 = 1
    mut t3 : any* = (any*)(&one)
    mut t2 : any* = resume(t3, ctx)
    mut x : i32 = *((i32*)(t2))
```

```
    mut t5 : any* = (any*)(&one)
    mut t4 : any* = resume(t5, ctx)
    mut y : i32 = *((i32*)(t4))
    mut z : i32 = x + w + y + args.a
    *(prompt_ret) = z
    ctx_set_returning(true, ctx)
    return 0
  }

  f(ctx)
  prompt_read : any* × any* × context* -> unit = (
    mut in : any*, mut out : any*, mut ctx : context*
  ) -> unit {
    yielding_to_ask : i32 = ctx_is_yielding_to(0, ctx)

    if yielding_to_ask {
      mut args_for_ask : args_ask* = (args_ask*)(ctx_get_handler_args(ctx))
      ask(args_for_ask, (i32*)(out), ctx)
      ctx_effect_handled(0, ctx)
      return 1
    }

    mut is_yielding__ : i32 = is_yielding(ctx)
    return 0
  }

  mut prompt_ret : i32 = 0
  prompt_read(0, (any*)(&prompt_ret), ctx)
  ctx_is_returning_ : i32 = ctx_is_returning(ctx);

  if ctx_is_returning_ {
    ctx_set_returning(false, ctx)
    return prompt_ret
  }

  return 0
}
```

3.2.2. Stack Frame Allocation Pass

During the *Stack Frame Allocation Pass* we compute the static size of the buffer needed by each function literal and allocate memory for each variable, we insert $push_{frame}$ and pop_{frame} calls into each function literal, the $push_{frame}$ and pop_{frame} functions just allocate and free memory for stack frames, internally they use the TLSF Allocator [tls], a real time general purpose allocator.

During the stack frame allocation pass, lambda functions receive an extra argument, referred to as the stack frame pointer and denoted with the prefix *sp*. Additionally, extra memory is allocated on the stack frame to link the previous stack frame as the first 8 bytes of the buffer, and to save the values of the arguments passed to the function, with the exception of the argument used to pass the context object. If the function is not a lambda, the first 8 bytes are left uninitialized. This allocation of memory and saving of arguments is necessary because they may need to be accessed outside the scope of the function, such as during a *resume* operation. In this case, it is necessary to save the arguments in a location that can be accessed later. Any use of arguments within the body of the function is also replaced with a reference to the value saved in the stack frame, with the exception of the context object argument, which is not saved.

During the compilation process, we allocate a temporary variable called *ret* on the C++ stack to store the return value for functions. In the future, we may lower the representation of our code and pass the return address as an argument to the function instead of relying on the C++ stack.

Here is an example for the $prompt_{read}$ function in our example:

```
prompt_read : any* × any* × context* × any* -> unit = (
    mut in : any*, mut out : any*, mut ctx : context*, mut sp1 : any*
) -> unit {
    mut sp2 : any* = push_frame(36)
    *((any**)(sp2)) = (any*)(sp1)
    *((any***)(sp2 + 16)) = &out
    *((any***)(sp2 + 8)) = &in
    *((i32*)(sp2 + 24)) = ctx_is_yielding_to(0, ctx)

    if *((i32*)(sp2 + 24)) {
      *((args_ask**)(sp2 + 28)) =
        (args_ask*)(ctx_get_handler_args(ctx))
      ask(*((args_ask**)(sp2 + 28)), (i32*)(*(*((any***)(sp2 + 16)))), ctx, sp1)
      ctx_effect_handled(0, ctx)
      ret : i32 = 1;
      pop_frame(sp2, 36)
      return ret
    }

    *((i32*)(sp2 + 28)) = is_yielding(ctx)
    ret : unit = 0;
    pop_frame(sp2, 36)
    return ret
  }
```

3.2.3. Continuations Pass

In this pass, we divide each function into every possible *yielding* point, where a *yielding* point is any point at which the program can *yield* control back to an effect handler on the stack. In the future, we can perform more precise analysis to improve code generation, but for now, we consider any point after a function call to be a potential *yielding* point. This means that we split a program point at each function call.

The split creates a lambda function, which must have a special signature that is common to all other lambdas created in this pass, allowing them to be saved in a data structure and called in a generic way without runtime checks for typing information. We assume that function calls can occur in two situations: on the right side of an assignment or as a standalone function call statement. If the language supports function calls in other situations, they can be easily converted to meet our requirements by introducing temporary variables.

The split is recursively done in the following way:

1. We can split the list of program points in three different places:
   (a) At the right side of an assignment: In the case where a function call occurs on the right side of an assignment, we introduce a temporary variable to store the result of the function call. This is necessary because the function call may perform an effect that interrupts the computation before the assignment can be completed. In order to handle this possibility, we split the program point after the function call and before the assignment. This results in two lists of program points, one with the program point $t = f()$ as the tail, and another with the program point $a = t$ as the head. This split allows us to handle the potential interruption of the computation due to an effect.
   (b) At a function call without an assignment: We just split the function at the point after the function call, such that the call becomes the tail of the current program point list.
   (c) At the meeting points of branches.
2. After we split the body of the function as described on step 1, we create the function that will be the continuation function, let's refer by A the program point that became a tail after the split, and lets refer as B the head of the resulting program point list after the split. B will became the entry point for the continuation function, all continuations needs to have the same function signature, this signature will need four arguments:
   (a) An input argument, that will be a pointer to *any*(our equivalent to a void pointer in C++), this argument will only be used in case the splitting point is caused by an assignment.
   (b) This function also needs an argument for the output of the continuation, let's call this argument $cont_{ret}$, also with the type $*any$.
   (c) It also needs a pointer to the stack frame of the enclosing function.
   (d) A pointer to the context object, remember that all functions receive the context object as an argument.

This function will receive four arguments, the return type of the lambda continuation functions also needs to have a common type, we use the *unit* type. Since we need to obey this generic signature, the continuation starting at B should be transformed in the following way:

- In we are splitting at an assignment, we will have something like $a = t$ at the head of our body, the statement B from early, the variable t in that case will be passed as the first argument with type $*any$, so this assignment should become $a = cast(T*, t)$ where T is the type of a. If we are splitting at a point that is not an assignment, the first argument is ignored.
- All return statements, such as $returnz$ are converted into $*cont_{ret} = *z; return0$.

3. This function is inserted after A, with name c_{idx} where idx is a unique number not yet used by any identifier of any other lambda continuation function within the current scope.
4. We may be creating a continuation function on two situation, on the body of a function that is already a continuation, or at the original function:
   (a) If we're splitting at the original function, we allocate an extra variable named $prompt_{ret}$ with type equal to the return type of the function, we then insert the call statement of the continuation function created at step 3, with the address of t as first argument, in case of an assignment, casting it to *any**, or *nil* if the continuation does not result from an assignment, we also pass the address of $prompt_{ret}$ as second argument, also casts to *any**, and propagate the context object and stack frame pointer as third and fourth argument respectively, then we insert $returnprompt_{ret}$ after the call, and this will become the body of our function.
   (b) If we're splitting inside a continuation, we just call the continuation, with the address of t as first argument, in case of an assignment, casting it to *any**, or *nil* if the continuation does not result from an assignment, and then we propagate the *contret* pointer down to the call, as well as the context object and the stack frame pointer, after that we just insert the $return0$ statement.
5. pop_{frame} calls may end up happening at the end of a continuation lambda, we lift those calls to the same scope that allocates the stack frame.

In the following example, we'll use a slightly different pseudo code language than the one we use in the previous examples, here we'll be representation the stack frame as a map instead of offsets in the stack frame buffer, so instead of $*((i32*)(sp + address_{x}))$, we will be using $sp[i32, x]$ for simplicity.

```
f : (ctx: context*) {
  sp := push_frame(SP_SIZE);
  sp[i32, x] = g(ctx);
  sp[i32, y] = w(ctx);
  ret : i32 = sp[i32, x] + sp[i32, y]
  pop_frame(sp);
  return ret;
}
```

The result of the *Continuations Pass* for the example above should look something similar to:

```
f : (ctx: context*) {
  sp := push_frame(SP_SIZE);
  t1 := g(ctx);
  c1 :: (t1 : any*, cont_ret: any*, ctx: context*, sp: any*) -> unit {
    sp[i32, x] = value_of(cast(i32*, t1)); // same as *((i32*)(t1)) in C
    t2 := w(ctx);
    c2 :: (t2: any*, cont_ret: any*, ctx: context*, sp: any*) -> unit {
      sp[i32, y] = value_of(cast(i32*, t2));
      save(cast(i32*, cont_ret), sp[i32, x] + sp[i32, y]);
      return 0;
    }
    c2(cast(any*, &t2), cont_ret, ctx, sp);
    return 0;
  }
  cont_ret: i32 = 0;
  c1(cast(any*, &t1), cast(any*, &cont_ret), ctx, sp);
  pop_frame(sp, SP_SIZE);
  return cont_ret;
}
```

3.2.4. Bubbling Pass

The last step of the pipeline is the insertion of the *bubbling* and *yielding* checks and logic, *bubbling* is the process in which we interrupt computations and save the continuations such that they can be resumed later. Remember that in the *Continuations Pass*, we split the functions on each possible *yielding* point creating a bunch of continuation lambda functions that have a common interface. Those points are where we insert *yielding* checks, such that:

1. If we're at a function call that precedes a prompt function invocation we ignore and continue, so no checks are inserted at this program point.
2. If we're at a call that doesn't precedes a prompt lambda call, we insert an *if statement* that checks if we're *yielding* using the context object, remember that this call is followed be a continuation call in the form $c_{idx}(\&t, prompt_ret, ctx, sp)$. Inside the $if statement$, the body should be responsible for *bubbling* the continuation up the call stack, this is done by creating a continuation object that should have:
   (a) A valid reference to the current stack frame.
   (b) The size of the actual type of *contret*. Remember that the signature of the continuation function have *contret* with type $*any$, by storing its size in the continuation object, we can allocate a buffer to store the value of $prompt_{ret}$ latter without the original type.
   (c) During a *resume* operation, a prompt lambda function may have a slightly different treatment than a continuation lambda function, so we

also store a flag that holds if the continuation function being stored is a prompt or a continuation lambda, remember that both of them have the same interface, so they can all be stored within the continuation object.

(d) A pointer to the lambda handler, this is just the continuation lambda or the prompt lambda created on the previous passes that is called after the current point. In case the current call is a continuation, then we have no calls after it, in that case, we just set the function handler pointer to *null*. In the case the current scope is inside a prompt function, then all lambda handlers are set to point to the prompt function itself, that is, inside a prompt function, we never bubble a continuation lambda, because even though we create continuations for prompt lambda functions, every time we *yield* from a handler, we want to execute it from the start, not from the point where computation was interrupted, because we may be yielding to a handler that is catched before that point.

Within the body of the if statement, we also save the continuation object in a linked list on the context object and insert a return statement with a default value. It is important to note that we also include a check for continuation calls that results in a *null* value being passed as the function handler pointer. This check is necessary because we may call a pop_{frame} and we do not want to free the current stack frame memory before we handle the effect that caused the *yielding* process to start. This is because we may need the frame to resume the computation. To signal that there is nothing left to run at the continuation point, we simply set the handler to zero.

It is important to note that semantically, this process saves the call stack in a format that can be reconstructed after the *yielding* process finishes. This is also the reason why we do not insert the yielding check in functions that precede a prompt function call. This is because the yielding process may stop at that point and the effect will be handled there. If we did not do this, we would yield indefinitely, continually popping the entire call stack.

After inserting those yielding checks, we also lift all the local lambdas to the global context, this is done by adding the name of the enclosing function at the beginning of the inner function.

Here is a small cut so we can see what those checks looks like:

```
f_c0 :: (_ : any*, cont_ret: any*, ctx: context*, sp: any*) -> unit {
  ...
}

f :: (ctx: context*) -> i32 {
  sp := push_frame();

  g(ctx);

  yielding0 := is_yielding(ctx);
```

```
   if yielding0 {
    frame : any* = sp;
    is_prompt : i32 = 0
    handler : (any* × any* × context* × any* -> unit)* = &f_c0;
    retsize : i32 = sizeof(i32)
    bubble(frame, is_prompt, retsize, handler);
    return 0;
  }

  cont_ret : i32 = 0;

  f_c0(0, (any*)&cont_ret, ctx, sp)

  yielding1 := is_yielding(ctx);

  if yielding1 {
    frame : any* = sp;
    is_prompt : i32 = 0
    retsize : i32 = sizeof(i32)
    bubble(frame, is_prompt, retsize, 0);
    return 0;
  }

  pop_frame();
  return 0;
}
```

4. Conclusion

After all the passes described above finish they execution, we are going to have an representation that can trivially be converted to C or C++ code, the resulting IR and C++ code for our first example will be added on the Appendix.

5. Future Work

For future work, we still need to fix memory management issues during resumptions, because they are leaking on the current implementation. We can also try to just allocate the stack frames on the default C stack, instead of allocating them on the heap, and only copy the data to the heap during continuation bubbling, this may lead to some gains since most effects can be handled locally if the last defined handler is tail resumptive and can be found quickly by using a hashmap in the context object [Xie and Leijen 2021].

A bad design decision was to perform resumptions using the *resume* operation, like in *resume*(*one*) in our first example. In the future, if we want to support nested handlers, we need to pass the continuation object to the effect handler function directly instead of calling it implicitly with *resume*.

Our current implementation of the *resume* operation also needs more testing

and probably some rewriting, it will probably break on more complex situations where we call nested continuations.

The Stack Frame Allocation Pass can also be rewritten to use a more precise analysis, like Stack Coloring, this can decrease the memory used by each function.

We also need to figure it out how we're going to handle references, and what the semantics for them will be, because if we're allowed to mutate what is being referenced on a continuation, calling the continuation multiple times can lead to different results, given the same input. This can be fixed by making references immutable on program paths that may be a part of a continuation, as well as only allowing one shot continuations, or completely ignoring the problem by flexing the definition for continuations on the language.

References

The Koka programming language. `https://koka-lang.github.io/koka/doc/index.html`.

The TLSF allocator. `http://www.gii.upv.es/tlsf/`.

Alex Reinking, Ningning Xie, L. d. M. and Leijen, D. Perceus: Garbage free reference counting with reuse.

Pretnar, M. (2015). *An Introduction to Algebraic Effects and Handlers.* Faculty of Mathematics and Physics University of Ljubljana Slovenia.

Xie, N. and Leijen, D. (2021). Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c. ACM Program. Lang. 5, ICFP, Article 71 (August 2021).