Exception Handling in C∀

by

Andrew James Beach

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

Abstract

The C∀ (Cforall) programming language is an evolutionary refinement of the C programming language, adding modern programming features without changing the programming paradigms of C. One of these modern programming features is more powerful error handling through the addition of an exception handling mechanism (EHM).

This thesis covers the design and implementation of the C∀ EHM, along with a review of the other required C∀ features. The EHM includes common features of termination exception handling, which abandons and recovers from an operation, and similar support for resumption exception handling, which repairs and continues with an operation. The design of both has been adapted to utilize other tools C∀ provides, as well as fit with the assertion based interfaces of the language.

The EHM has been implemented into the C∀ compiler and run-time environment. Although it has not yet been optimized, performance testing has shown it has comparable performance to other EHMs, which is sufficient for use in current C∀ programs.

Acknowledgements

As is tradition and his due, I would like to begin by thanking my supervisor Peter Buhr. From accepting me in a first place, to helping me run performance tests, I would not be here without him. Also if there was an “artist” field here he would be listed there as well, he helped me a lot with the diagrams.

I would like to thank the readers Gregor Richards and Yizhou Zhang for their feedback and approval. The presentation of the thesis has definitely been improved with their feedback.

I also thank the entire Cforall Team who built the rest of the C∀ compiler. From the existing features I used in my work, to the internal tooling that makes further development easier and the optimizations that make running tests pass by quickly. This includes: Aaron Moss, Rob Schluntz, Thierry Delisle, Michael Brooks, Mubeen Zulfieqar & Fangren Yu.

And thank-you Henry Xue, the co-op student who converted my macro implementation of exception declarations into the compiler features presented in this thesis.

Finally I thank my family, who are still relieved I learned how to read.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

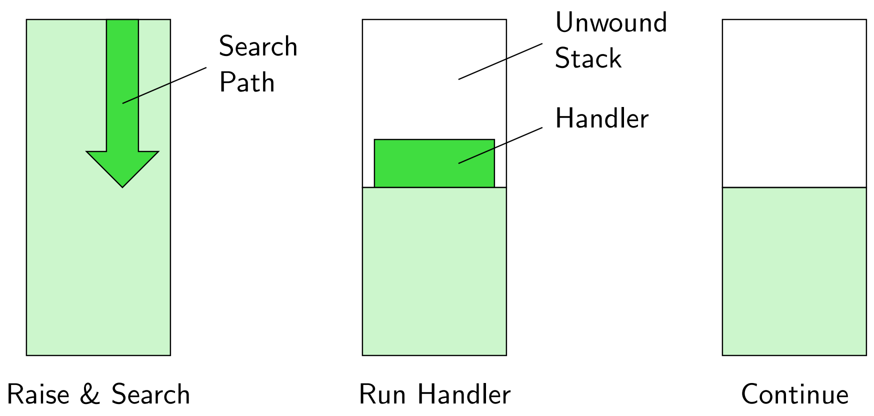
This thesis covers the design and implementation of the exception handling mechanism (EHM) of C∀ (pronounced sea-for-all and may be written Cforall or CFA). C∀ is a new programming language that extends C, which maintains backwards-compatibility while introducing modern programming features. Adding exception handling to C∀ gives it new ways to handle errors and make large control-flow jumps.

Exception handling provides dynamic inter-function control flow. A language's EHM is a combination of language syntax and run-time components that construct, raise, propagate and handle exceptions, to provide all of that control flow. There are two forms of exception handling covered in this thesis: termination, which acts as a multi-level return, and resumption, which is a dynamic function call. Often, when this separation is not made, termination exceptions are assumed as they are more common and may be the only form of handling provided in a language.

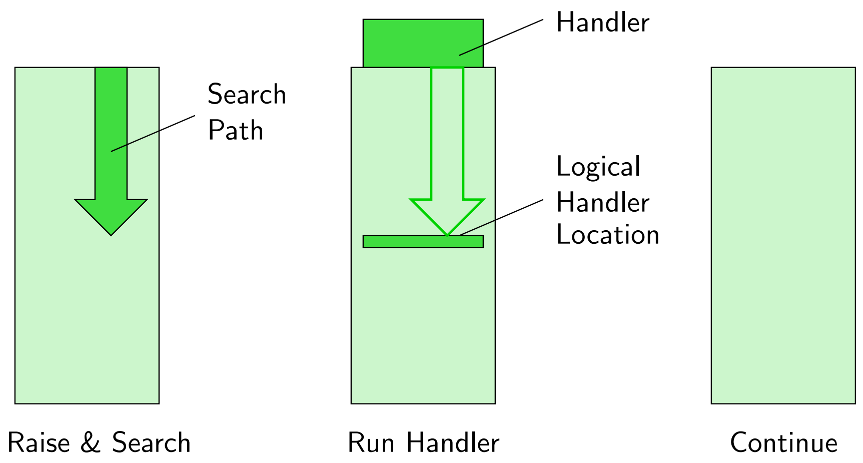
All types of exception handling link a raise with a handler. Both operations are usually language primitives, although raises can be treated as a function that takes an exception argument. Handlers are more complex, as they are added to and removed from the stack during execution, must specify what they can handle and must give the code to handle the exception.

Exceptions work with different execution models but for the descriptions that follow a simple call stack, with functions added and removed in a first-in-last-out order, is assumed.

Termination exception handling searches the stack for the handler, then unwinds the stack to where the handler was found before calling it. The handler is run inside the function that defined it and when it finishes it returns control to that function.



Resumption exception handling searches the stack for a handler and then calls it without removing any other stack frames. The handler is run on top of the existing stack, often as a new function or closure capturing the context in which the handler was defined. After the handler has finished running, it returns control to the function that preformed the raise, usually starting after the raise.



Although a powerful feature, exception handling tends to be complex to set up and expensive to use, so it is often limited to unusual or "exceptional" cases. The classic example is error handling; exceptions can be used to remove error handling logic from the main execution path, and pay most of the cost only when the error actually occurs.

1.1 Thesis Overview

This work describes the design and implementation of the C∀ EHM. The C∀ EHM implements all of the common exception features (or an equivalent) found in most other EHMs and adds some features of its own. The design of all the features had to be adapted to C∀'s feature set, as some of the underlying tools used to implement and express exception handling in other languages are absent in C∀. Still, the resulting syntax resembles that of other languages:

```
try {
	...
	T * object = malloc(request_size);
	if (!object) {
		throw OutOfMemory{fixed_allocation, request_size};
	}
	...
} catch (OutOfMemory * error) {
	...
}
```

The design and implementation of all of C∀'s EHM's features are described in detail throughout this thesis, whether they are a common feature or one unique to C∀.

All of these features have been implemented in C∀, covering both changes to the compiler and the run-time. In addition, a suite of test cases and performance benchmarks were created alongside the implementation. The implementation techniques are generally applicable in other programming languages and much of the design is as well. Some parts of the EHM use other features unique to C∀ and would be harder to replicate in other programming languages.

The contributions of this work are:

1. Designing C∀'s exception handling mechanism, adapting designs from other programming languages and creating new features.
2. Implementing stack unwinding and the C∀ EHM, including updating the C∀ compiler and the run-time environment.
3. Designing and implementing a prototype virtual system.
4. Creating tests to check the behaviour of the EHM.
5. Creating benchmarks to check the performance of the EHM, as compared to other languages.

The rest of this thesis is organized as follows. The current state of exceptions is covered in section 1.2. The existing state of C∀ is covered in chapter 2. New EHM features are introduced in chapter 3, covering their usage and design. That is followed by the implementation of these features in chapter 4. Performance results are examined in chapter 5. Possibilities to extend this project are discussed in chapter 6. Finally, the project is summarized in chapter 7.

1.2 Background

Exception handling has been examined before in programming languages, with papers on the subject dating back 70s.[9] Early exceptions were often treated as signals, which carried no information except their identity. Ada originally used this system[1], but now allows for a string message as a payload[2].

The modern flagship for termination exceptions – if one exists – is C++, which added them in its first major wave of non-object-orientated features in 1990.[4] Many EHMs have special exception types, however C++ has the ability to use any type as an exception. These were found to be not very useful and have been pushed aside for classes inheriting from `std::exception`. Although there is a special catch-all syntax (`catch(...)`), there are no operations that can be performed on the caught value, not even type inspection. Instead, the base exception-type `std::exception` defines common functionality (such as the ability to describe the reason the exception was raised) and all exceptions have this functionality. That trade-off, restricting usable types to gain guaranteed functionality, is almost universal now, as without some common functionality it is almost impossible to actually handle any errors.

Java was the next popular language to use exceptions.[10] Its exception system largely reflects that of C++, except that it requires you throw a child type of `java.lang.Throwable` and it uses checked exceptions. Checked exceptions are part of a function's interface, the exception signature of the function. Every exception that could be raised from a function, either directly or because it is not handled from a called function, is given. Using this information, it is possible to statically verify if any given exception is handled, and guarantee that no exception will go unhandled. Making exception information explicit improves clarity and safety, but can slow down or restrict programming. For example, programming high-order functions becomes much more complex if the argument functions could raise exceptions. However, as odd it may seem, the worst problems are rooted in the simple inconvenience of writing and updating exception signatures. This has caused Java programmers to develop multiple programming "hacks" to circumvent checked exceptions, negating their advantages. One particularly problematic example is the "catch-and-ignore" pattern, where an empty handler is used to handle an exception without doing any recovery or repair. In theory that could be good enough to properly handle the exception, but more often is used to ignore an exception that the programmer does not feel is worth the effort of handling, for instance if they do not believe it will ever be raised. If they are incorrect, the exception will be silenced, while in a similar situation with unchecked exceptions the exception would at least activate the language's unhandled exception code (usually, a program abort with an error message).

Resumption exceptions are less popular, although resumption is as old as termination; that is, few programming languages have implemented them. Mesa is one programming language that did.[12] Experience with Mesa is quoted as being one of the reasons resumptions were not included in the C++ standard. Since then, resumptions have been ignored in mainstream programming languages. However, resumption is being revisited in the context of decades of other developments in programming languages. While rejecting resumption may have been the right decision in the past, the situation has changed since then. Some developments, such as the functional programming equivalent to resumptions, algebraic effects[17], are enjoying success. A complete reexamination of resumption is beyond this thesis, but their reemergence is enough reason to try them in C∀.

Functional languages tend to use other solutions for their primary error handling mechanism, but exception-like constructs still appear. Termination appears in the error construct, which marks the result of an expression as an error; then the result of any expression that tries to use it also results in an error, and so on until an appropriate handler is reached. Resumption appears in algebraic effects, where a function dispatches its side-effects to its caller for handling.

More recently, exceptions seem to be vanishing from newer programming languages, replaced by "panic". In Rust, a panic is just a program level abort that may be implemented by unwinding the stack like in termination exception handling.[15][16] Go's panic though is very similar to a termination, except it only supports a catch-all by calling `recover()`, simplifying the interface at the cost of flexibility.[11]

As exception handling's most common use cases are in error handling, here are some other ways to handle errors with comparisons with exceptions.

- *Error Codes*: This pattern has a function return an enumeration (or just a set of fixed values) to indicate if an error has occurred and possibly which error it was.

  Error codes mix exceptional/error and normal values, enlarging the range of possible return values. This can be addressed with multiple return values (or a tuple) or a tagged union. However, the main issue with error codes is forgetting to check them, which leads to an error being quietly and implicitly ignored. Some new languages and tools will try to issue warnings when an error code is discarded to avoid this problem. Checking error codes also bloats the main execution path, especially if the error is not handled immediately and has to be passed through multiple functions before it is addressed.

  Here is an example of the pattern in Bash, where commands can only "return" numbers and most output is done through streams of text.

  ```
  # Immediately after running a command:
  case $? in
  0)
  	# Success
  	;;
  1)
  	# Error Code 1
  	;;
  2|3)
  	# Error Code 2 or Error Code 3
  	;;
  # Add more cases as needed.
  asac
  ```

- *Special Return with Global Store*: Similar to the error codes pattern but the function itself only returns that there was an error, and stores the reason for the error in a fixed global location. For example, many routines in the C standard library will only return some error value (such as -1 or a null pointer) and the error code is written into the standard variable `errno`.

This approach avoids the multiple results issue encountered with straight error codes as only a single error value has to be returned, but otherwise has the same disadvantages and more. Every function that reads or writes to the global store must agree on all possible errors and managing it becomes more complex with concurrency.

This example shows some of what has to be done to robustly handle a C standard library function that reports errors this way.

```
// Now a library function can set the error.
int handle = open(path_name, flags);
if (-1 == handle) {
     switch (errno) {
    case ENAMETOOLONG:
          // path_name is a bad argument.
          break;
     case ENFILE:
          // A system resource has been exausted.
          break;
     // And many more...
    }
}
```

- *Return Union*: This pattern replaces error codes with a tagged union. Success is one tag and the errors are another. It is also possible to make each possible error its own tag and carry its own additional information, but the two-branch format is easy to make generic so that one type can be used everywhere in error handling code.

  This pattern is very popular in any functional or semi-functional language with primitive support for tagged unions (or algebraic data types). Return unions can also be expressed as monads (evaluation in a context) and often are in languages with special syntax for monadic evaluation, such as Haskell's `do` blocks.

  The main advantage is that an arbitrary object can be used to represent an error, so it can include a lot more information than a simple error code. The disadvantages include that the it does have to be checked along the main execution, and if there aren't primitive tagged unions proper, usage can be hard to enforce.

  This is a simple example of examining the result of a failing function in Haskell, using its `Either` type. Examining `error` further would likely involve more matching, but the type of `error` is user defined so there are no general cases.

  ```
  case failingFunction argA argB of
      Right value -> -- Use the successful computed value.
      Left error -> -- Handle the produced error.
  ```

  Return unions as monads will result in the same code, but can hide most of the work to propagate errors in simple cases. The code to actually handle the errors, or to interact with other monads (a common case in these languages) still has to be written by hand.

  If `failingFunction` is implemented with two helpers that use the same error type, then it can be implemented with a `do` block.

```
failingFunction x y = do
     z <- helperOne x
     helperTwo y z
```

- *Handler Functions*: This pattern associates errors with functions. On error, the function that produced the error calls another function to handle it. The handler function can be provided locally (passed in as an argument, either directly as as a field of a structure/object) or globally (a global variable). C++ uses this approach as its fallback system if exception handling fails, such as `std::terminate` and, for a time, `std::unexpected`.[1]

  Handler functions work a lot like resumption exceptions, but without the dynamic search for a handler. Since setting up the handler can be more complex/expensive, especially when the handler has to be passed through multiple layers of function calls, but cheaper (constant time) to call, they are more suited to more frequent (less exceptional) situations. Although, in C++ and other languages that do not have checked exceptions, they can actually be enforced by the type system be more reliable.

  This is a more local example in C++, using a function to provide a default value for a mapping.

  ```
  ValueT Map::key_or_default(KeyT key, ValueT(*make_default)(KeyT)) {
       ValueT * value = find_value(key);
       if (nullptr != value) {
            return *value;
       } else {
            return make_default(key);
       }
  }
  ```

Because of their cost, exceptions are rarely used for hot paths of execution. Hence, there is an element of self-fulfilling prophecy as implementation techniques have been focused on making them cheap to set up, happily making them expensive to use in exchange. This difference is less important in higher-level scripting languages, where using exceptions for other tasks is more common. An iconic example is Python's `StopIteration`[7] exception, that is thrown by an iterator to indicate that it is exhausted. When paired with Python's iterator-based for-loop, this will be thrown every time the end of the loop is reached.[8]

[1] `std::unexpected` was part of the Dynamic Exception Specification, which has been removed from the standard as of C++20.[5]

Chapter 2

C∀ Existing Features

C∀ is an open-source project extending ISO C with modern safety and productivity features, while still ensuring backwards compatibility with C and its programmers. C∀ is designed to have an orthogonal feature-set based closely on the C programming paradigm (non-object-oriented), and these features can be added incrementally to an existing C code-base, allowing programmers to learn C∀ on an as-needed basis.

Only those C∀ features pertaining to this thesis are discussed. A familiarity with C or C-like languages is assumed.

2.1 Overloading and `extern`

C∀ has extensive overloading, allowing multiple definitions of the same name to be defined [14].

```
char i; int i; double i;
int f(); double f();
void g( int ); void g( double );
```

This feature requires name mangling so the assembly symbols are unique for different overloads. For compatibility with names in C, there is also a syntax to disable name mangling. These unmangled names cannot be overloaded but act as the interface between C and C∀ code. The syntax for disabling/enabling mangling is:

```
// name mangling on by default
int i; // _X1ii_1
extern "C" {  // disables name mangling
      int j; // j
      extern "Cforall" {  // enables name mangling
            int k; // _X1ki_1
      }
      // revert to no name mangling
}
// revert to name mangling
```

Both forms of `extern` affect all the declarations within their nested lexical scope and transition back to the previous mangling state when the lexical scope ends.

2.2 Reference Type

C∀ adds a reference type to C as an auto-dereferencing pointer. They work very similarly to pointers. Reference-types are written the same way as pointer-types, but each asterisk (`*`) is replaced with a ampersand (`&`); this includes cv-qualifiers (`const` and `volatile`) and multiple levels of reference.

Generally, references act like pointers with an implicit dereferencing operation added to each use of the variable. These automatic dereferences may be disabled with the address-of operator (`&`).

With references:

```
int i, j;
int & ri = i;
int && rri = ri;
rri = 3;
&ri = &j;
ri = 5;
```

With pointers:

```
int i, j;
int * pi = &i
int ** ppi = &pi;
**ppi = 3;
pi = &j;
*pi = 5;
```

References are intended to be used when the indirection of a pointer is required, but the address is not as important as the value and dereferencing is the common usage. Mutable references may be assigned to by converting them to a pointer with a `&` and then assigning a pointer to them, as in `&ri = &j;` above.

2.3 Operators

C∀ implements operator overloading by providing special names, where operator expressions are translated into function calls using these names. An operator name is created by taking the operator symbols and joining them with `?`s to show where the arguments go. For example, infixed multiplication is `?*?`, while prefix dereference is `*?`. This syntax makes it easy to tell the difference between prefix operations (such as `++?`) and postfix operations (`?++`).

As an example, here are the addition and equality operators for a point type.

```
point ?+?(point a, point b) { return point{a.x + b.x, a.y + b.y}; }
int ?==?(point a, point b) { return a.x == b.x && a.y == b.y; }
{
     assert(point{1, 2} + point{3, 4} == point{4, 6});
}
```

Note that this syntax works effectively as a textual transformation; the compiler converts all operators into functions and then resolves them normally. This means any combination of types may be used, although nonsensical ones (like `double ?==?(point, int);`) are discouraged. This feature is also used for all builtin operators as well, although those are implicitly provided by the language.

In C∀, constructors and destructors are operators, which means they are functions with special operator names, rather than type names as in C++. Both constructors and destructors can be implicity called by the compiler, however the operator names allow explicit calls.

The special name for a constructor is `?{}`, which comes from the initialization syntax in C, e.g., `Example e = { ... }`. C∀ generates a constructor call each time a variable is declared, passing the initialization arguments to the constructor.

```
struct Example { ... };
void ?{}(Example & this) { ... }
{
     Example a;
     Example b = {};
}
void ?{}(Example & this, char first, int num) { ... }
{
     Example c = {'a', 2};
}
```

Both `a` and `b` will be initalized with the first constructor, `b` because of the explicit call and `a` implicitly. `c` will be initalized with the second constructor. Currently, there is no general way to skip initialization.

Similarly, destructors use the special name `^?{}` (the ^ has no special meaning).

```
void ^?{}(Example & this) { ... }
{
     Example d;
     ^?{}(d);

     Example e;
} // Implicit call of ^?{}(e);
```

Whenever a type is defined, C∀ creates a default zero-argument constructor, a copy constructor, a series of argument-per-field constructors and a destructor. All user constructors are defined after this.

2.4 Polymorphism

C∀ uses parametric polymorphism to create functions and types that are defined over multiple types. C∀ polymorphic declarations serve the same role as C++ templates or Java generics. The "parametric" means the polymorphism is accomplished by passing argument operations to associate *parameters* at the call site, and these parameters are used in the function to differentiate among the types the function operates on.

Polymorphic declarations start with a universal `forall` clause that goes before the standard (monomorphic) declaration. These declarations have the same syntax except they may use the universal type names introduced by the `forall` clause. For example, the following is a polymorphic identity function that works on any type `T`:

```
forall( T ) T identity( T val ) { return val; }
int forty_two = identity( 42 );
char capital_a = identity( 'A' );
```

Each use of a polymorphic declaration resolves its polymorphic parameters (in this case, just `T`) to concrete types (`int` in the first use and `char` in the second).

To allow a polymorphic function to be separately compiled, the type `T` must be constrained by the operations used on `T` in the function body. The `forall` clause is augmented with a list of polymorphic variables (local type names) and assertions (constraints), which represent the required operations on those types used in a function, e.g.:

```
forall( T | { void do_once(T); } )
void do_twice(T value) {
     do_once(value);
     do_once(value);
}
```

A polymorphic function can be used in the same way as a normal function. The polymorphic variables are filled in with concrete types and the assertions are checked. An assertion is checked by verifying each assertion operation (with all the variables replaced with the concrete types from the arguments) is defined at a call site.

```
void do_once(int i) { ... }
int i;
do_twice(i);
```

Any value with a type fulfilling the assertion may be passed as an argument to a `do_twice` call.

Note, a function named `do_once` is not required in the scope of `do_twice` to compile it, unlike C++ template expansion. Furthermore, call-site inferencing allows local replacement of the specific parametric functions needs for a call.

```
void do_once(double y) { ... }
int quadruple(int x) {
     void do_once(int & y) { y = y * 2; }
     do_twice(x);
     return x;
}
```

Specifically, the complier deduces that `do_twice`'s T is an integer from the argument `x`. It then looks for the most specific definition matching the assertion, which is the nested integral `do_once` defined within the function. The matched assertion function is then passed as a function pointer to `do_twice` and called within it. The global definition of `do_once` is ignored, however if `quadruple` took a `double` argument, then the global definition would be used instead as it would then be a better match.[13]

To avoid typing long lists of assertions, constraints can be collected into a convenient package called a `trait`, which can then be used in an assertion instead of the individual constraints.

```
trait done_once(T) {
     void do_once(T);
}
```

and the forall list in the previous example is replaced with the trait.

```
forall(dtype T | done_once(T))
```

In general, a trait can contain an arbitrary number of assertions, both functions and variables, and are usually used to create a shorthand for, and give descriptive names to, common groupings of assertions describing a certain functionality, like summable, listable, etc.

Polymorphic structures and unions are defined by qualifying an aggregate type with forall. The type variables work the same except they are used in field declarations instead of parameters, returns and local variable declarations.

```
forall(dtype T)
struct node {
     node(T) * next;
     T * data;
};
node(int) inode;
```

The generic type node(T) is an example of a polymorphic type usage. Like C++ template usage, a polymorphic type usage must specify a type parameter.

There are many other polymorphism features in C∀ but these are the ones used by the exception system.

2.5 Control Flow

C∀ has a number of advanced control-flow features: generator, coroutine, monitor, mutex parameters, and thread. The two features that interact with the exception system are coroutine and thread; they and their supporting constructs are described here.

2.5.1 Coroutine

A coroutine is a type with associated functions, where the functions are not required to finish execution when control is handed back to the caller. Instead, they may suspend execution at any time and be resumed later at the point of last suspension. Coroutine types are not concurrent but share some similarities along with common underpinnings, so they are combined with the C∀ threading library.

In C∀, a coroutine is created using the coroutine keyword, which is an aggregate type like struct, except the structure is implicitly modified by the compiler to satisfy the is_coroutine trait; hence, a coroutine is restricted by the type system to types that provide this special trait. The coroutine structure acts as the interface between callers and the coroutine, and its fields are used to pass information in and out of coroutine interface functions.

Here is a simple example where a single field is used to pass (communicate) the next number in a sequence.

```
coroutine CountUp {
     unsigned int next;
};
CountUp countup;
```

Each coroutine has a `main` function, which takes a reference to a coroutine object and returns `void`.

```
void main(CountUp & this) {
     for (unsigned int next = 0 ; true ; ++next) {
          this.next = next;
          suspend;
     }
}
```

In this function, or functions called by this function (helper functions), the `suspend` statement is used to return execution to the coroutine's caller without terminating the coroutine's function.

A coroutine is resumed by calling the `resume` function, e.g., `resume(countup)`. The first resume calls the `main` function at the top. Thereafter, resume calls continue a coroutine in the last suspended function after the `suspend` statement. In this case there is only one and, hence, the difference between subsequent calls is the state of variables inside the function and the coroutine object. The return value of `resume` is a reference to the coroutine, to make it convent to access fields of the coroutine in the same expression. Here is a simple example in a helper function:

```
unsigned int get_next(CountUp & this) {
     return resume(this).next;
}
```

When the main function returns, the coroutine halts and can no longer be resumed.

2.5.2 Monitor and Mutex Parameter

Concurrency does not guarantee ordering; without ordering, results are non-deterministic. To claw back ordering, C∀ uses monitors and `mutex` (mutual exclusion) parameters. A monitor is another kind of aggregate, where the compiler implicitly inserts a lock and instances are compatible with `mutex` parameters.

A function that requires deterministic (ordered) execution acquires mutual exclusion on a monitor object by qualifying an object reference parameter with the `mutex` qualifier.

```
void example(MonitorA & mutex argA, MonitorB & mutex argB);
```

When the function is called, it implicitly acquires the monitor lock for all of the mutex parameters without deadlock. This semantics means all functions with the same mutex type(s) are part of a critical section for objects of that type and only one runs at a time.

2.5.3 Thread

Functions, generators and coroutines are sequential, so there is only a single (but potentially sophisticated) execution path in a program. Threads introduce multiple execution paths that continue independently.

For threads to work safely with objects requires mutual exclusion using monitors and mutex parameters. For threads to work safely with other threads also requires mutual exclusion in the form of a communication rendezvous, which also supports internal synchronization as for mutex objects. For exceptions, only two basic thread operations are important: fork and join.

Threads are created like coroutines with an associated `main` function:

```
thread StringWorker {
     const char * input;
     int result;
};
void main(StringWorker & this) {
     const char * localCopy = this.input;
     // ... do some work, perhaps hashing the string ...
     this.result = result;
}
{
     StringWorker stringworker; // fork thread running in "main"
} // Implicit call to join(stringworker), waits for completion.
```

The thread main is where a new thread starts execution after a fork operation and then the thread continues executing until it is finished. If another thread joins with an executing thread, it waits until the executing main completes execution. In other words, everything a thread does is between a fork and join.

From the outside, this behaviour is accomplished through creation and destruction of a thread object. Implicitly, fork happens after a thread object's constructor is run and join happens before the destructor runs. Join can also be specified explicitly using the `join` function to wait for a thread's completion independently from its deallocation (i.e., destructor call). If `join` is called explicitly, the destructor does not implicitly join.

Chapter 3

Exception Features

This chapter covers the design and user interface of the C∀ EHM and begins with a general overview of EHMs. It is not a strict definition of all EHMs nor an exhaustive list of all possible features. However, it does cover the most common structure and features found in them.

3.1 Overview of EHMs

3.1.1 Raise / Handle

An exception operation has two main parts: raise and handle. These terms are sometimes known as throw and catch but this work uses throw/catch as a particular kind of raise/handle. These are the two parts that the user writes and may be the only two pieces of the EHM that have any syntax in a language.

Raise The raise is the starting point for exception handling, by raising an exception, which passes it to the EHM.

Some well known examples include the `throw` statements of C++ and Java and the `raise` statement of Python. In real systems, a raise may perform some other work (such as memory management) but for the purposes of this overview that can be ignored.

Handle The primary purpose of an EHM is to run some user code to handle a raised exception. This code is given, along with some other information, in a handler.

A handler has three common features: the previously mentioned user code, a region of code it guards and an exception label/condition that matches against the raised exception. Only raises inside the guarded region and raising exceptions that match the label can be handled by a given handler. If multiple handlers could can handle an exception, EHMs define a rule to pick one, such as "best match" or "first found".

The `try` statements of C++, Java and Python are common examples. All three also show another common feature of handlers: they are grouped by the guarded region.

3.1.2 Propagation

After an exception is raised comes what is usually the biggest step for the EHM: finding and setting up the handler for execution. The propagation from raise to handler can be broken up into three different tasks: searching for a handler, matching against the handler and installing the handler.

Searching The EHM begins by searching for handlers that might be used to handle the exception. The search will find handlers that have the raise site in their guarded region. The search includes handlers in the current function, as well as any in callers on the stack that have the function call in their guarded region.

Matching Each handler found is with the raised exception. The exception label defines a condition that is used with the exception and decides if there is a match or not. In languages where the first match is used, this step is intertwined with searching; a match check is performed immediately after the search finds a handler.

Installing After a handler is chosen, it must be made ready to run. The implementation can vary widely to fit with the rest of the design of the EHM. The installation step might be trivial or it could be the most expensive step in handling an exception. The latter tends to be the case when stack unwinding is involved.

If a matching handler is not guaranteed to be found, the EHM needs a different course of action for this case. This situation only occurs with unchecked exceptions as checked exceptions (such as in Java) can make the guarantee. The unhandled action is usually very general, such as aborting the program.

Hierarchy A common way to organize exceptions is in a hierarchical structure. This pattern comes from object-oriented languages where the exception hierarchy is a natural extension of the object hierarchy.

Consider the following exception hierarchy:

$$\begin{tikzcd}
\texttt{exception} \arrow[r] & \texttt{arithmetic} \arrow[r] \arrow[dr] \arrow[ddr] & \texttt{underflow} \\
& & \texttt{overflow} \\
& & \texttt{zerodivide}
\end{tikzcd}$$

A handler labeled with any given exception can handle exceptions of that type or any child type of that exception. The root of the exception hierarchy (here `exception`) acts as a catch-all, leaf types catch single types and the exceptions in the middle can be used to catch different groups of related exceptions.

This system has some notable advantages, such as multiple levels of grouping, the ability for libraries to add new exception types and the isolation between different sub-hierarchies. This design is used in C∀ even though it is not a object-orientated language, so different tools are used to create the hierarchy.

3.1.3 Completion

After the handler has finished, the entire exception operation has to complete and continue executing somewhere else. This step is usually simple, both logically and in its implementation, as the installation of the handler is usually set up to do most of the work.

The EHM can return control to many different places, where the most common are after the handler definition (termination) and after the raise (resumption).

3.1.4 Communication

For effective exception handling, additional information is often passed from the raise to the handler and back again. So far, only communication of the exception's identity is covered. A common communication method for adding information to an exception is putting fields into the exception instance and giving the handler access to them. Passing references or pointers allows data at the raise location to be updated, passing information in both directions.

3.2 Virtuals

A common feature in many programming languages is a tool to pair code (behaviour) with data. In C∀, this is done with the virtual system, which allow type information to be abstracted away, recovered and allow operations to be performed on the abstract objects.

Virtual types and casts are not part of C∀'s EHM nor are they required for an EHM. However, one of the best ways to support an exception hierarchy is via a virtual hierarchy and dispatch system. Ideally, the virtual system would have been part of C∀ before the work on exception handling began, but unfortunately it was not. Hence, only the features and framework needed for the EHM were designed and implemented for this thesis. Other features were considered to ensure that the structure could accommodate other desirable features in the future but are not implemented. The rest of this section only discusses the implemented subset of the virtual system design.

The virtual system supports multiple "trees" of types. Each tree is a simple hierarchy with a single root type. Each type in a tree has exactly one parent – except for the root type which has zero parents – and any number of children. Any type that belongs to any of these trees is called a virtual type.

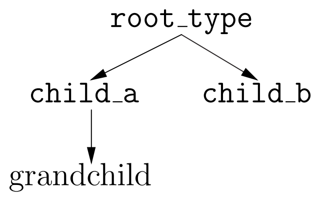
For the purposes of illustration, a proposed, but unimplemented, syntax will be used. Each virtual type is represented by a trait with an annotation that makes it a virtual type. This annotation is empty for a root type, which creates a new tree:

```
trait root_type(T) virtual() {}
```

The annotation may also refer to any existing virtual type to make this new type a child of that type and part of the same tree. The parent may itself be a child or a root type and may have any number of existing children.

```
trait child_a(T) virtual(root_type) {}
trait grandchild(T) virtual(child_a) {}
trait child_b(T) virtual(root_type) {}
```

Virtual Type Tree



Every virtual type also has a list of virtual members and a unique id. Both are stored in a virtual table. Every instance of a virtual type also has a pointer to a virtual table stored in it, although there is no per-type virtual table as in many other languages.

The list of virtual members is accumulated from the root type down the tree. Every virtual type inherits the list of virtual members from its parent and may add more virtual members to the end of the list which are passed on to its children. Again, using the unimplemented syntax this might look like:

```
trait root_type(T) virtual() {
    const char * to_string(T const & this);
    unsigned int size;
}

trait child_type(T) virtual(root_type) {
    char * irrelevant_function(int, char);
}
```

As `child_type` is a child of `root_type`, it has the virtual members of `root_type` (`to_string` and `size`) as well as the one it declared (`irrelevant_function`).

It is important to note that these are virtual members, and may contain arbitrary fields, functions or otherwise. The names "size" and "align" are reserved for the size and alignment of the virtual type, and are always automatically initialized as such. The other special case is uses of the trait's polymorphic argument (`T` in the example), which are always updated to refer to the current virtual type. This allows functions that refer to the polymorphic argument to act as traditional virtual methods (`to_string` in the example), as the object can always be passed to a virtual method in its virtual table.

Up until this point, the virtual system is similar to ones found in object-oriented languages, but this is where C∀ diverges. Objects encapsulate a single set of methods in each type, universally across the entire program, and indeed all programs that use that type definition. The only way to change any method is to inherit and define a new type with its own universal implementation. In this sense, these object-oriented types are "closed" and cannot be altered.

In C∀, types do not encapsulate any code. Whether or not a type satisfies any given assertion, and hence any trait, is context sensitive. Types can begin to satisfy a trait, stop satisfying it or satisfy the same trait at any lexical location in the program. In this sense, a type's implementation in the set of functions and variables that allow it to satisfy a trait is "open" and can change throughout the program. This capability means it is impossible to pick a single set of functions that represent a type's implementation across a program.

C∀ side-steps this issue by not having a single virtual table for each type. A user can define virtual tables that are filled in at their declaration and given a name. Anywhere that name is visible, even if it is defined locally inside a function (although in this case the user must ensure it outlives any objects that use it), it can be used. Specifically, a virtual type is "bound" to a virtual table that sets the virtual members for that object. The virtual members can be accessed through the object.

This means virtual tables are declared and named in C∀. They are declared as variables, using the type `vtable(VIRTUAL_TYPE)` and any valid name. For example:

```
vtable(virtual_type_name) table_name;
```

Like any variable, they may be forward declared with the `extern` keyword. Forward declaring virtual tables is relatively common. Many virtual types have an "obvious" implementation that works in most cases. A pattern that has appeared in the early work using virtuals is to implement a virtual table with the the obvious definition and place a forward declaration of it in the header beside the definition of the virtual type.

Even on the full declaration, no initializer should be used. Initialization is automatic. The type id and special virtual members "size" and "align" only depend on the virtual type, which is fixed given the type of the virtual table, and so the compiler fills in a fixed value. The other virtual members are resolved using the best match to the member's name and type, in the same context as the virtual table is declared using C∀'s normal resolution rules.

While much of the virtual infrastructure has been created, it is currently only used internally for exception handling. The only user-level feature is the virtual cast, which is the same as the C++ `dynamic_cast`.

```
(virtual TYPE)EXPRESSION
```

Note, the syntax and semantics matches a C-cast, rather than the function-like C++ syntax for special casts. Both the type of `EXPRESSION` and `TYPE` must be pointers to virtual types. The cast dynamically checks if the `EXPRESSION` type is the same or a sub-type of `TYPE`, and if true, returns a pointer to the `EXPRESSION` object, otherwise it returns `0p` (null pointer). This allows the expression to be used as both a cast and a type check.

3.3 Exceptions

The syntax for declaring an exception is the same as declaring a structure except the keyword:

```
exception TYPE_NAME {
    FIELDS
};
```

Fields are filled in the same way as a structure as well. However, an extra field is added that contains the pointer to the virtual table. It must be explicitly initialized by the user when the exception is constructed.

Here is an example of declaring an exception type along with a virtual table, assuming the exception has an "obvious" implementation and a default virtual table makes sense.

Header (.hfa):

```
exception Example {
    int data;
};

extern vtable(Example)
    example_base_vtable;
```

Implementation (.cfa):

```
vtable(Example) example_base_vtable
```

This is the only interface needed when raising and handling exceptions. However, it is actually a shorthand for a more complex trait-based interface.

The language views exceptions through a series of traits. If a type satisfies them, then it can be used as an exception. The following is the base trait all exceptions need to match.

```
trait is_exception(exceptT &, virtualT &) {
    // Numerous imaginary assertions.
};
```

The trait is defined over two types: the exception type and the virtual table type. Each exception type should have a single virtual table type. There are no actual assertions in this trait because the trait system cannot express them yet (adding such assertions would be part of completing the virtual system). The imaginary assertions would probably come from a trait defined by the virtual system, and state that the exception type is a virtual type, that that the type is a descendant of `exception_t` (the base exception type) and allow the user to find the virtual table type.

There are two more traits for exceptions defined as follows:

```
trait is_termination_exception(
        exceptT &, virtualT & | is_exception(exceptT, virtualT)) {
    void defaultTerminationHandler(exceptT &);
};

trait is_resumption_exception(
        exceptT &, virtualT & | is_exception(exceptT, virtualT)) {
    void defaultResumptionHandler(exceptT &);
};
```

Both traits ensure a pair of types is an exception type and its virtual table type, and defines one of the two default handlers. The default handlers are used as fallbacks and are discussed in detail in section 3.4.

However, all three of these traits can be tricky to use directly. While there is a bit of repetition required, the largest issue is that the virtual table type is mangled and not in a user facing way. So, these three macros are provided to wrap these traits to simplify referring to the names: `IS_EXCEPTION`, `IS_TERMINATION_EXCEPTION` and `IS_RESUMPTION_EXCEPTION`.

All three take one or two arguments. The first argument is the name of the exception type. The macro passes its unmangled and mangled form to the trait. The second (optional) argument is a parenthesized list of polymorphic arguments. This argument is only used with polymorphic exceptions and the list is passed to both types. In the current set-up, the two types always have the same polymorphic arguments, so these macros can be used without losing flexibility.

For example, consider a function that is polymorphic over types that have a defined arithmetic exception:

```
forall(Num | IS_EXCEPTION(Arithmetic, (Num)))
void some_math_function(Num & left, Num & right);
```

3.4 Exception Handling

As stated, C∀ provides two kinds of exception handling: termination and resumption. These twin operations are the core of C∀'s exception handling mechanism. This section covers the general patterns shared by the two operations and then goes on to cover the details of each individual operation.

Both operations follow the same set of steps. First, a user raises an exception. Second, the exception propagates up the stack, searching for a handler. Third, if a handler is found, the exception is caught and the handler is run. After that control continues at a raise-dependent location. As an alternate to the third step, if a handler is not found, a default handler is run and, if it returns, then control continues after the raise.

The differences between the two operations include how propagation is performed, where execution continues after an exception is handled and which default handler is run.

3.4.1 Termination

Termination handling is the familiar kind of handling used in most programming languages with exception handling. It is a dynamic, non-local goto. If the raised exception is matched and handled, the stack is unwound and control (usually) continues in the function on the call stack that defined the handler. Termination is commonly used when an error has occurred and recovery is impossible locally.

A termination raise is started with the `throw` statement:

```
throw EXPRESSION;
```

The expression must return a reference to a termination exception, where the termination exception is any type that satisfies the trait `is_termination_exception` at the call site.

Through C∀'s trait system, the trait functions are implicitly passed into the throw code for use by the EHM. A new `defaultTerminationHandler` can be defined in any scope to change the throw's behaviour when a handler is not found (see below).

The throw copies the provided exception into managed memory to ensure the exception is not destroyed if the stack is unwound. It is the user's responsibility to ensure the original exception is cleaned up whether the stack is unwound or not. Allocating it on the stack is usually sufficient.

Then propagation starts with the search. C∀ uses a "first match" rule so matching is performed with the copied exception as the search key. It starts from the raise site and proceeds towards base of the stack, from callee to caller. At each stack frame, a check is made for termination handlers defined by the `catch` clauses of a `try` statement.

```
try {
    GUARDED_BLOCK
} catch (EXCEPTION_TYPE1 * [NAME1]) {
    HANDLER_BLOCK1
} catch (EXCEPTION_TYPE2 * [NAME2]) {
    HANDLER_BLOCK2
}
```

When viewed on its own, a try statement simply executes the statements in the `GUARDED_BLOCK` and when those are finished, the try statement finishes.

However, while the guarded statements are being executed, including any invoked functions, all the handlers in these statements are included in the search path. Hence, if a termination exception is raised, these handlers may be matched against the exception and may handle it.

Exception matching checks the handler in each catch clause in the order they appear, top to bottom. If the representation of the raised exception type is the same or a descendant of $\texttt{EXCEPTION\_TYPE}_i$, then $\texttt{NAME}_i$ (if provided) is bound to a pointer to the exception and the statements in $\texttt{HANDLER\_BLOCK}_i$ are executed. If control reaches the end of the handler, the exception is freed and control continues after the try statement.

If no termination handler is found during the search, then the default handler (`defaultTerminationHandler`) visible at the raise statement is called. Through C∀'s trait system the best match at the raise statement is used. This function is run and is passed the copied exception. If the default handler finishes, control continues after the raise statement.

There is a global `defaultTerminationHandler` that is polymorphic over all termination exception types. The global default termination handler performs a cancellation (as described in 3.7 on page 28) on the current stack with the copied exception. Since it is so general, a more specific handler can be defined, overriding the default behaviour for the specific exception types.

For example, consider an error reading a configuration file. This is most likely a problem with the configuration file (`config_error`), but the function could have been passed the wrong file name (`arg_error`). In this case the function could raise one exception and then, if it is unhandled, raise the other. This is not usual behaviour for either exception so changing the default handler will be done locally:

```
{
    void defaultTerminationHandler(config_error &) {
        throw (arg_error){arg_vt};
    }
    throw (config_error){config_vt};
}
```

3.4.2 Resumption

Resumption exception handling is less familar form of exception handling, but is just as old [9] and is simpler in many ways. It is a dynamic, non-local function call. If the raised exception is matched, a closure is taken from up the stack and executed, after which the raising function continues executing. The common uses for resumption exceptions include potentially repairable errors, where execution can continue in the same function once the error is corrected, and ignorable events, such as logging where nothing needs to happen and control should always continue from the raise site.

Except for the changes to fit into that pattern, resumption exception handling is symmetric with termination exception handling, by design (see subsection 3.4.1).

A resumption raise is started with the `throwResume` statement:

```
throwResume EXPRESSION;
```

It works much the same way as the termination raise, except the type must satisfy the `is_resumption_exception` that uses the default handler: `defaultResumptionHandler`. This can be specialized for particular exception types.

At run-time, no exception copy is made. Since resumption does not unwind the stack nor otherwise remove values from the current scope, there is no need to manage memory to keep the exception allocated.

Then propagation starts with the search, following the same search path as termination, from the raise site to the base of stack and top of try statement to bottom. However, the handlers on try statements are defined by `catchResume` clauses.

```
try {
    GUARDED_BLOCK
} catchResume (EXCEPTION_TYPE$_1$ * [NAME$_1$]) {
    HANDLER_BLOCK$_1$
} catchResume (EXCEPTION_TYPE$_2$ * [NAME$_2$]) {
    HANDLER_BLOCK$_2$
}
```

Note that termination handlers and resumption handlers may be used together in a single try statement, intermixing `catch` and `catchResume` freely. Each type of handler only interacts with exceptions from the matching kind of raise. Like `catch` clauses, `catchResume` clauses have no effect if an exception is not raised.

The matching rules are exactly the same as well. The first major difference here is that after `EXCEPTION_TYPE`$_i$ is matched and `NAME`$_i$ is bound to the exception, `HANDLER_BLOCK`$_i$

is executed right away without first unwinding the stack. After the block has finished running, control jumps to the raise site, where the just handled exception came from, and continues executing after it, not after the try statement.

For instance, a resumption used to send messages to the logger may not need to be handled at all. Putting the following default handler at the global scope can make handling that exception optional by default.

```
void defaultResumptionHandler(log_message &) {
    // Nothing, it is fine not to handle logging.
}
// ... No change at raise sites. ...
throwResume (log_message){strlit_log, "Begin event processing."}
```

Resumption Marking

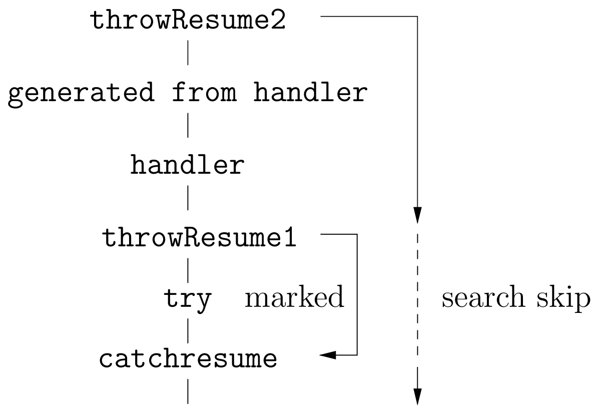
A key difference between resumption and termination is that resumption does not unwind the stack. A side effect is that, when a handler is matched and run, its try block (the guarded statements) and every try statement searched before it are still on the stack. Their presence can lead to the recursive resumption problem.[3]

The recursive resumption problem is any situation where a resumption handler ends up being called while it is running. Consider a trivial case:

```
try {
     throwResume (E &){};
} catchResume(E *) {
     throwResume (E &){};
}
```

When this code is executed, the guarded `throwResume` starts a search and matches the handler in the `catchResume` clause. This call is placed on the stack above the try-block. Now the second raise in the handler searches the same try block, matches again and then puts another instance of the same handler on the stack leading to infinite recursion.

While this situation is trivial and easy to avoid, much more complex cycles can form with multiple handlers and different exception types. To prevent all of these cases, each try statement is "marked" from the time the exception search reaches it to either when a handler completes handling that exception or when the search reaches the base of the stack. While a try statement is marked, its handlers are never matched, effectively skipping over it to the next try statement.



There are other sets of marking rules that could be used. For instance, marking just the handlers that caught the exception would also prevent recursive resumption. However, the rules selected mirror what happens with termination, so this reduces the amount of rules and patterns a programmer has to know.

The marked try statements are the ones that would be removed from the stack for a termination exception, i.e., those on the stack between the handler and the raise statement. This symmetry applies to the default handler as well, as both kinds of default handlers are run at the raise statement, rather than (physically or logically) at the bottom of the stack.

3.5 Conditional Catch

Both termination and resumption handler clauses can be given an additional condition to further control which exceptions they handle:

```
catch (EXCEPTION_TYPE * [NAME] ; CONDITION)
```

First, the same semantics is used to match the exception type. Second, if the exception matches, `CONDITION` is executed. The condition expression may reference all names in scope at the beginning of the try block and `NAME` introduced in the handler clause. If the condition is true, then the handler matches. Otherwise, the exception search continues as if the exception type did not match.

The condition matching allows finer matching by checking more kinds of information than just the exception type.

```
try {
     handle1 = open( f1, ... );
     handle2 = open( f2, ... );
     handle3 = open( f3, ... );
     ...
} catch( IOFailure * f ; fd( f ) == f1 ) {
     // Only handle IO failure for f1.
} catch( IOFailure * f ; fd( f ) == f3 ) {
     // Only handle IO failure for f3.
}
// Handle a failure relating to f2 further down the stack.
```

In this example, the file that experienced the IO error is used to decide which handler should be run, if any at all.

3.5.1 Comparison with Reraising

In languages without conditional catch – that is, no ability to match an exception based on something other than its type – it can be mimicked by matching all exceptions of the right type, checking any additional conditions inside the handler and re-raising the exception if it does not match those.

Here is a minimal example comparing both patterns, using `throw;` (no operand) to start a re-raise.

```
try {                               try {
    do_work_may_throw();                do_work_may_throw();
} catch(exception_t * exc ;         } catch(exception_t * exc) {
            can_handle(exc)) {          if (can_handle(exc)) {
    handle(exc);                            handle(exc);
}                                       } else {
                                            throw;
                                        }
                                    }
```

At first glance, catch-and-reraise may appear to just be a quality-of-life feature, but there are some significant differences between the two strategies.

A simple difference that is more important for C∀ than many other languages is that the raise site changes with a re-raise, but does not with a conditional catch. This is important in C∀ because control returns to the raise site to run the per-site default handler. Because of this, only a conditional catch can allow the original raise to continue.

The more complex issue comes from the difference in how conditional catches and re-raises handle multiple handlers attached to a single try statement. A conditional catch will continue checking later handlers while a re-raise will skip them. If the different handlers could handle some of the same exceptions, translating a try statement that uses one to use the other can quickly become non-trivial:

Original, with conditional catch:

```
...
} catch (an_exception * e ; check_a(e)) {
    handle_a(e);
} catch (exception_t * e ; check_b(e)) {
    handle_b(e);
}
```

Translated, with re-raise:

```
...
} catch (exception_t * e) {
        an_exception * an_e = (virtual an_exception *)e;
        if (an_e && check_a(an_e)) {
            handle_a(an_e);
        } else if (check_b(e)) {
            handle_b(e);
        } else {
            throw;
        }
    }
```

(There is a simpler solution if `handle_a` never raises exceptions, using nested try statements.)

In similar simple examples, translating from re-raise to conditional catch takes less code but it does not have a general, trivial solution either.

So, given that the two patterns do not trivially translate into each other, it becomes a matter of which on should be encouraged and made the default. From the premise that if a handler could handle an exception then it should, it follows that checking as many handlers as possible is preferred. So, conditional catch and checking later handlers is a good default.

3.6 Finally Clauses

Finally clauses are used to perform unconditional cleanup when leaving a scope and are placed at the end of a try statement after any handler clauses:

```
try {
    GUARDED_BLOCK
} ... // any number or kind of handler clauses
... finally {
    FINALLY_BLOCK
}
```

The `FINALLY_BLOCK` is executed when the try statement is removed from the stack, including when the `GUARDED_BLOCK` finishes, any termination handler finishes or during an unwind. The only time the block is not executed is if the program is exited before the stack is unwound.

Execution of the finally block should always finish, meaning control runs off the end of the block. This requirement ensures control always continues as if the finally clause is not present, i.e., finally is for cleanup, not changing control flow. Because of this requirement, local control flow out of the finally block is forbidden. The compiler precludes any `break`, `continue`, `fallthru` or `return` that causes control to leave the finally block. Other ways to leave the finally block, such as a `longjmp` or termination are much harder to check, and at best require additional run-time overhead, and so are only discouraged.

Not all languages with unwinding have finally clauses. Notably, C++ does without it as destructors, and the RAII design pattern, serve a similar role. Although destructors and finally clauses can be used for the same cases, they have their own strengths, similar to top-level function and lambda functions with closures. Destructors take more work to create, but if there is clean-up code that needs to be run every time a type is used, they are much easier to set up for each use. On the other hand, finally clauses capture the local context, so are easy to use when the cleanup is not dependent on the type of a variable or requires information from multiple variables.

3.7 Cancellation

Cancellation is a stack-level abort, which can be thought of as as an uncatchable termination. It unwinds the entire current stack, and if possible, forwards the cancellation exception to a different stack.

Cancellation is not an exception operation like termination or resumption. There is no special statement for starting a cancellation; instead the standard library function `cancel_stack` is called, passing an exception. Unlike a raise, this exception is not used in matching, only to pass information about the cause of the cancellation. Finally, as no handler is provided, there is no default handler.

After `cancel_stack` is called, the exception is copied into the EHM's memory and the current stack is unwound. The behaviour after that depends on the kind of stack being cancelled.

Main Stack The main stack is the one used by the program's main function at the start of execution, and is the only stack in a sequential program. After the main stack is unwound, there is a program-level abort.

The first reason for this behaviour is for sequential programs where there is only one stack, and hence no stack to pass information to. Second, even in concurrent programs, the main stack has no dependency on another stack and no reliable way to find another living stack. Finally, keeping the same behaviour in both sequential and concurrent programs is simple and easy to understand.

Thread Stack A thread stack is created for a C∀ `thread` object or object that satisfies the `is_thread` trait. After a thread stack is unwound, the exception is stored until another thread attempts to join with it. Then the exception `ThreadCancelled`, which stores a reference to the thread and to the exception passed to the cancellation, is reported from the join to the joining thread. There is one difference between an explicit join (with the `join` function) and an implicit join (from a destructor call). The explicit join takes the default handler (`defaultResumptionHandler`) from its calling context while the implicit join provides its own, which does a program abort if the `ThreadCancelled` exception cannot be handled.

The communication and synchronization are done here because threads only have two structural points (not dependent on user-code) where communication/synchronization happens: start and join. Since a thread must be running to perform a cancellation (and cannot be cancelled from another stack), the cancellation must be after start and before the join, so join is used.

The difference between the explicit and implicit join is for safety and debugging. It helps prevent unwinding collisions by avoiding throwing from a destructor and prevents cascading the error across multiple threads if the user is not equipped to deal with it. It is always possible to add an explicit join if that is the desired behaviour.

With explicit join and a default handler that triggers a cancellation, it is possible to cascade an error across any number of threads, alternating between the resumption (possibly termination) and cancellation, cleaning up each in turn, until the error is handled or the main thread is reached.

Coroutine Stack A coroutine stack is created for a `coroutine` object or object that satisfies the `is_coroutine` trait. After a coroutine stack is unwound, control returns to the `resume` function that most recently resumed it. `resume` reports a `CoroutineCancelled` exception, which contains a reference to the cancelled coroutine and the exception used to cancel it. The `resume` function also takes the `defaultResumptionHandler` from the caller's context and passes it to the internal report.

A coroutine only knows of two other coroutines, its starter and its last resumer. The starter has a much more distant connection, while the last resumer just (in terms of coroutine state) called resume on this coroutine, so the message is passed to the latter.

With a default handler that triggers a cancellation, it is possible to cascade an error across any number of coroutines, alternating between the resumption (possibly termination) and cancellation, cleaning up each in turn, until the error is handled or a thread stack is reached.

Chapter 4

Implementation

The implementation work for this thesis covers the two components: virtual system and exceptions. Each component is discussed in detail.

4.1 Virtual System

While the C∀ virtual system currently has only two public features, virtual cast and virtual tables, substantial structure is required to support them, and provide features for exception handling and the standard library.

4.1.1 Virtual Type

A virtual type (see section 3.2) has a pointer to a virtual table, called the *virtual-table pointer*, which binds each instance of a virtual type to a virtual table. Internally, the field is called `virtual_table` and is fixed after construction. This pointer is also the table's id and how the system accesses the virtual table and the virtual members there. It is always the first field in the structure so that its location is always known.

Virtual table pointers are passed to the constructors of virtual types as part of field-by-field construction.

4.1.2 Type ID

Every virtual type has a unique ID. These are used in type equality, to check if the representation of two values are the same, and to access the type's type information. This uniqueness means across a program composed of multiple translation units (TU), not uniqueness across all programs or even across multiple processes on the same machine.

Our approach for program uniqueness is using a static declaration for each type ID, where the run-time storage address of that variable is guaranteed to be unique during program execution. The type ID storage can also be used for other purposes, and is used for type information.

The problem is that a type ID may appear in multiple TUs that compose a program (see subsection 4.1.4), so the initial solution would seem to be make it external in each translation unit. However, the type ID must have a declaration in (exactly) one of the TUs to create the storage. No other declaration related to the virtual type has this property, so doing this through standard C declarations would require the user to do it manually.

Instead, the linker is used to handle this problem. A new feature has been added to C∀ for this purpose, the special attribute `cfa_linkonce`, which uses the special section `.gnu.linkonce`. When used as a prefix (e.g., `.gnu.linkonce.example`), the linker does not combine these sections, but instead discards all but one with the same full name.

So, each type ID must be given a unique section name with the `linkonce` prefix. Luckily, C∀ already has a way to get unique names, the name mangler. For example, this could be written directly in C∀:

```
__attribute__((cfa_linkonce)) void f() {}
```

This is translated to:

```
__attribute__((section(".gnu.linkonce._X1fFv___1"))) void _X1fFv___1() {}
```

This is done internally to access the name mangler. This attribute is useful for other purposes, any other place a unique instance required, and should eventually be made part of a public and stable feature in C∀.

4.1.3 Type Information

There is data stored at the type ID's declaration, the type information. The type information currently is only the parent's type ID or, if the type has no parent, the null pointer. The ancestors of a virtual type are found by traversing type IDs through the type information. An example using helper macros looks like:

```
struct INFO_TYPE(TYPE) {
    INFO_TYPE(PARENT) const * parent;
};

__attribute__((cfa_linkonce))
INFO_TYPE(TYPE) const INFO_NAME(TYPE) = {
    &INFO_NAME(PARENT),
};
```

Type information is constructed as follows:

1. Use the type's name to generate a name for the type information structure, which is saved so it can be reused.
2. Generate a new structure definition to store the type information. The layout is the same in each case, just the parent's type ID, but the types used change from instance to instance. The generated name is used for both this structure and, if relevant, the

parent pointer. If the virtual type is polymorphic then the type information structure is polymorphic as well, with the same polymorphic arguments.

3. A separate name for instances is generated from the type's name.
4. The definition is generated and initialized. The parent ID is set to the null pointer or to the address of the parent's type information instance. Name resolution handles the rest.
5. C∀'s name mangler does its regular name mangling encoding the type of the declaration into the instance name. This process gives a completely unique name including different instances of the same polymorphic type.

Writing that code manually, with helper macros for the early name mangling, would look like this:

```
struct INFO_TYPE(TYPE) {
     INFO_TYPE(PARENT) const * parent;
};

__attribute__((cfa_linkonce))
INFO_TYPE(TYPE) const INFO_NAME(TYPE) = {
     &INFO_NAME(PARENT),
};
```

4.1.4 Virtual Table

Each virtual type has a virtual table type that stores its type ID and virtual members. An instance of a virtual type is bound to a virtual table instance, which have the values of the virtual members. Both the layout of the fields (in the virtual table type) and their value (in the virtual table instance) are decided by the rules given below.

The layout always comes in three parts (see Figure 4.1). The first section is just the type ID at the head of the table. It is always there to ensure that it can be found even when the accessing code does not know which virtual type it has. The second section is all the virtual members of the parent, in the same order as they appear in the parent's virtual table. Note that the type may change slightly as references to the "this" change. This is limited to inside pointers/references and via function pointers so that the size (and hence the offsets) are the same. The third section is similar to the second except that it is the new virtual members introduced at this level in the hierarchy.

The first and second sections together mean that every virtual table has a prefix that has the same layout and types as its parent virtual table. This, combined with the fixed offset to the virtual table pointer, means that for any virtual type, it is always safe to access its virtual table and, from there, it is safe to check the type ID to identify the exact type of the underlying object, access any of the virtual members and pass the object to any of the method-like virtual members.

When a virtual table is declared, the user decides where to declare it and its name. The initialization of the virtual table is entirely automatic based on the context of the declaration.

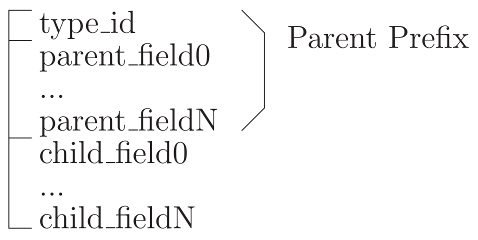


Figure 4.1: Virtual Table Layout

The type ID is always fixed, with each virtual table type having exactly one possible type ID. The virtual members are usually filled in by type resolution. The best match for a given name and type at the declaration site is used. There are two exceptions to that rule: the `size` field, the type's size, is set using a `sizeof` expression, and the `align` field, the type's alignment, is set using an `alignof` expression.

Most of these tools are already inside the compiler. Using simple code transformations early on in compilation allows most of that work to be handed off to the existing tools. Figure 4.2 shows an example transformation; this example shows an exception virtual table. It also shows the transformation on the full declaration. For a forward declaration, the `extern` keyword is preserved and the initializer is not added.

```
vtable(example_type) example_name;
```

becomes...

```
const struct example_type_vtable example_name = {
    .__cfavir_typeid : &__cfatid_example_type,
    .size : sizeof(example_type),
    .copy : copy,
    .^?{} : ^?{},
    .msg : msg,
};
```

Figure 4.2: Virtual Table Transformation

4.1.5 Concurrency Integration

Coroutines and threads need instances of `CoroutineCancelled` and `ThreadCancelled` respectively to use all of their functionality. When a new data type is declared with `coroutine` or `thread`, a forward declaration for the instance is created as well. The definition of the virtual table is created at the definition of the main function.

These transformations are shown through code re-writing in Figure 4.3 and Figure 4.4. Threads use the same pattern, with some names and types changed. In both cases, the original declaration is not modified, only new ones are added.

```
coroutine Example {
     // fields
};
```
appends...
```
__attribute__((cfa_linkonce))
struct __cfatid_struct_CoroutineCancelled(Example)
          __cfatid_CoroutineCancelled = {
     &EXCEPTION_TYPE_ID,
};
extern CoroutineCancelled_vtable _default_vtable_object_declaration;
extern CoroutineCancelled_vtable & _default_vtable;
```

Figure 4.3: Coroutine Type Transformation

```
void main(Example & this) {
     // body
}
```
appends...
```
CoroutineCancelled_vtable _default_vtable_object_declaration = {
     __cfatid_CoroutineCancelled,
     // Virtual member initialization.
};

CoroutineCancelled_vtable & _default_vtable =
     &_default_vtable_object_declaration;
```

Figure 4.4: Coroutine Main Transformation

4.1.6 Virtual Cast

Virtual casts are implemented as a function call that does the subtype check and a C coercion-cast to do the type conversion. The function is implemented in the standard library and has the following signature:

```
void * __cfa__virtual_cast(
     struct __cfavir_type_id * parent,
     struct __cfavir_type_id * const * child );
```

The type ID for the target type of the virtual cast is passed in as `parent` and the cast target is passed in as `child`. The generated C code wraps both arguments and the result with type casts. There is also an internal check inside the compiler to make sure that the target type is a virtual type.

The virtual cast either returns the original pointer or the null pointer as the new type. The function does the parent check and returns the appropriate value. The parent check is a simple linear search of the child's ancestors using the type information.

4.2 Exceptions

Creating exceptions can be roughly divided into two parts: the exceptions themselves and the virtual system interactions.

Creating an exception type is just a matter of prepending the field with the virtual table pointer to the list of the fields (see Figure 4.5).

```
exception new_exception {
    // EXISTING FIELDS
};
```

becomes...

```
struct new_exception {
    struct new_exception_vtable const * virtual_table;
    // EXISTING FIELDS
};
```

Figure 4.5: Exception Type Transformation

The integration between exceptions and the virtual system is a bit more complex simply because of the nature of the virtual system prototype. The primary issue is that the virtual system has no way to detect when it should generate any of its internal types and data. This is handled by the exception code, which tells the virtual system when to generate its components.

All types associated with a virtual type, the types of the virtual table and the type ID, are generated when the virtual type (the exception) is first found. The type ID (the instance) is generated with the exception, if it is a monomorphic type. However, if the exception is polymorphic, then a different type ID has to be generated for every instance. In this case, generation is delayed until a virtual table is created. When a virtual table is created and initialized, two functions are created to fill in the list of virtual members. The first is the `copy` function that adapts the exception's copy constructor to work with pointers, avoiding some issues with the current copy constructor interface. Second is the `msg` function that returns a C-string with the type's name, including any polymorphic parameters.

4.3 Unwinding

Stack unwinding is the process of removing stack frames (activations) from the stack. On function entry and return, unwinding is handled directly by the call/return code embedded in the function.

Usually, the stack-frame size is known statically based on parameter and local variable declarations. Even for a dynamic stack-size, the information to determine how much of the stack has to be removed is still contained within the function. Allocating/deallocating stack space is usually an $O(1)$ operation achieved by bumping the hardware stack-pointer up or down as needed. Constructing/destructing values within a stack frame has a similar complexity but larger constants.

Unwinding across multiple stack frames is more complex, because that information is no longer contained within the current function. With separate compilation, a function does not know its callers nor their frame layout. Even using the return address, that information is encoded in terms of actions in code, intermixed with the actions required to finish the function. Without changing the main code path it is impossible to select one of those two groups of actions at the return site.

The traditional unwinding mechanism for C is implemented by saving a snapshot of a function's state with `setjmp` and restoring that snapshot with `longjmp`. This approach bypasses the need to know stack details by simply resetting to a snapshot of an arbitrary but existing function frame on the stack. It is up to the programmer to ensure the snapshot is valid when it is reset and that all required cleanup from the unwound stacks is performed. Because it does not automate or check any of this cleanup, it can be easy to make mistakes and always must be handled manually.

With respect to the extra work in the surrounding code, many languages define cleanup actions that must be taken when certain sections of the stack are removed, such as when the storage for a variable is removed from the stack, possibly requiring a destructor call, or when a try statement with a finally clause is (conceptually) popped from the stack. None of these cases should be handled by the user – that would contradict the intention of these features – so they need to be handled automatically.

To safely remove sections of the stack, the language must be able to find and run these cleanup actions even when removing multiple functions unknown at the beginning of the unwinding.

One of the most popular tools for stack management is libunwind, a low-level library that provides tools for stack walking, handler execution, and unwinding. What follows is an overview of all the relevant features of libunwind needed for this work. Following that is the description of the C∀ code that uses libunwind to implement termination.

4.3.1 libunwind Usage

Libunwind, accessed through `unwind.h` on most platforms, is a C library that provides C++-style stack-unwinding. Its operation is divided into two phases: search and cleanup. The dynamic target search – phase 1 – is used to scan the stack and decide where unwinding should stop (but no unwinding occurs). The cleanup – phase 2 – does the unwinding and also runs any cleanup code.

To use libunwind, each function must have a personality function and a Language Specific Data Area (LSDA). The LSDA has the unique information for each function to tell the personality function where a function is executing, its current stack frame, and what handlers should be checked. Theoretically, the LSDA can contain any information but conventionally it is a table with entries representing regions of a function and what has to be done there during unwinding. These regions are bracketed by instruction addresses. If the instruction pointer is within a region's start/end, then execution is currently executing in that region. Regions are used to mark out the scopes of objects with destructors and try blocks.

The GCC compilation flag `-fexceptions` causes the generation of an LSDA and attaches a personality function to each function. In plain C (which C∀ currently compiles down to) this flag only handles the cleanup attribute:

```
void clean_up( int * var ) { ... }
int avar __attribute__(( cleanup(clean_up) ));
```

The attribute is used on a variable and specifies a function, in this case `clean_up`, run when the variable goes out of scope. This feature is enough to mimic destructors, but not try statements that affect the unwinding.

To get full unwinding support, all of these features must be handled directly in assembly and assembler directives; particularly the cfi directives `.cfi_lsda` and `.cfi_personality`.

4.3.2 Personality Functions

Personality functions have a complex interface specified by libunwind. This section covers some of the important parts of the interface.

A personality function can perform different actions depending on how it is called.

```
typedef _Unwind_Reason_Code (*_Unwind_Personality_Fn) (
    _Unwind_Action action,
    _Unwind_Exception_Class exception_class,
    _Unwind_Exception * exception,
    struct _Unwind_Context * context);
```

The `action` argument is a bitmask of possible actions:

1. `_UA_SEARCH_PHASE` specifies a search phase and tells the personality function to check for handlers. If there is a handler in a stack frame, as defined by the language, the personality function returns `_URC_HANDLER_FOUND`; otherwise it return `_URC_CONTINUE_UNWIND`.

2. `_UA_CLEANUP_PHASE` specifies a cleanup phase, where the entire frame is unwound and all cleanup code is run. The personality function does whatever cleanup the language defines (such as running destructors/finalizers) and then generally returns `_URC_CONTINUE_UNWIND`.

3. `_UA_HANDLER_FRAME` specifies a cleanup phase on a function frame that found a handler. The personality function must prepare to return to normal code execution and return `_URC_INSTALL_CONTEXT`.

4. `_UA_FORCE_UNWIND` specifies a forced unwind call. Forced unwind only performs the cleanup phase and uses a different means to decide when to stop (see subsection 4.3.4).

The `exception_class` argument is a copy of the `exception`'s `exception_class` field, which is a number that identifies the EHM that created the exception.

The `exception` argument is a pointer to a user provided storage object. It has two public fields: the `exception_class`, which is described above, and the `exception_cleanup` function. The cleanup function is used by the EHM to clean up the exception. If it should need to be freed at an unusual time, it takes an argument that says why it had to be cleaned up.

The `context` argument is a pointer to an opaque type passed to helper functions called inside the personality function.

The return value, `_Unwind_Reason_Code`, is an enumeration of possible messages that can be passed several places in libunwind. It includes a number of messages for special cases (some of which should never be used by the personality function) and error codes. However, unless otherwise noted, the personality function always returns `_URC_CONTINUE_UNWIND`.

4.3.3 Raise Exception

Raising an exception is the central function of libunwind and it performs two-staged unwinding.

```
_Unwind_Reason_Code _Unwind_RaiseException(_Unwind_Exception *);
```

First, the function begins the search phase, calling the personality function of the most recent stack frame. It continues to call personality functions traversing the stack from newest to oldest until a function finds a handler or the end of the stack is reached. In the latter case, `_Unwind_RaiseException` returns `_URC_END_OF_STACK`.

Second, when a handler is matched, `_Unwind_RaiseException` moves to the cleanup phase and walks the stack a second time. Once again, it calls the personality functions of each stack frame from newest to oldest. This pass stops at the stack frame containing the matching handler. If that personality function has not installed a handler, it is an error.

If an error is encountered, `_Unwind_RaiseException` returns either `_URC_FATAL_PHASE1_ERROR` or `_URC_FATAL_PHASE2_ERROR` depending on when the error occurred.

4.3.4 Forced Unwind

Forced Unwind is the other central function in libunwind.

```
_Unwind_Reason_Code _Unwind_ForcedUnwind(_Unwind_Exception *,
    _Unwind_Stop_Fn, void *);
```

It also unwinds the stack but it does not use the search phase. Instead, another function, the stop function, is used to stop searching. The exception is the same as the one passed to `_Unwind_RaiseException`. The extra arguments are the stop function and the stop parameter. The stop function has a similar interface as a personality function, except it is also passed the stop parameter.

```
typedef _Unwind_Reason_Code (*_Unwind_Stop_Fn)(
     _Unwind_Action action,
     _Unwind_Exception_Class exception_class,
     _Unwind_Exception * exception,
     struct _Unwind_Context * context,
     void * stop_parameter);
```

The stop function is called at every stack frame before the personality function is called and then once more after all frames of the stack are unwound.

Each time it is called, the stop function should return `_URC_NO_REASON` or transfer control directly to other code outside of libunwind. The framework does not provide any assistance here.

Its arguments are the same as the paired personality function. The actions `_UA_CLEANUP_PHASE` and `_UA_FORCE_UNWIND` are always set when it is called. Beyond the libunwind standard, both GCC and Clang add an extra action on the last call at the end of the stack: `_UA_END_OF_STACK`.

4.4 Exception Context

The exception context is global storage used to maintain data across different exception operations and to communicate among different components.

Each stack must have its own exception context. In a sequential C∀ program, there is only one stack with a single global exception-context. However, when the library `libcfathread` is linked, there are multiple stacks and each needs its own exception context.

The current exception context should be retrieved by calling the function `this_exception_context`. For sequential execution, this function is defined as a weak symbol in the C∀ system-library, `libcfa`. When a C∀ program is concurrent, it links with `libcfathread`, where this function is defined with a strong symbol replacing the sequential version.

The sequential `this_exception_context` returns a hard-coded pointer to the global exception context. The concurrent version adds the exception context to the data stored at the base of each stack. When `this_exception_context` is called, it retrieves the active stack and returns the address of the context saved there.

4.5 Termination

C∀ termination exceptions use libunwind heavily because they match C++ exceptions closely. The main complication for C∀ is that the compiler generates C code, making it very difficult to generate the assembly to form the LSDA for try blocks or destructors.

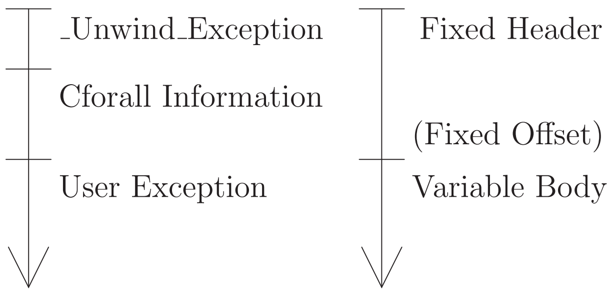


Figure 4.6: Exception Layout

4.5.1 Memory Management

The first step of a termination raise is to copy the exception into memory managed by the exception system. Currently, the system uses `malloc`, rather than reserved memory or the stack top. The EHM manages memory for the exception as well as memory for libunwind and the system's own per-exception storage.

Exceptions are stored in variable-sized blocks (see Figure 4.6). The first component is a fixed-sized data structure that contains the information for libunwind and the exception system. The second component is an area of memory big enough to store the exception. Macros with pointer arthritic and type cast are used to move between the components or go from the embedded `_Unwind_Exception` to the entire node.

Multiple exceptions can exist at the same time because exceptions can be raised inside handlers, destructors and finally blocks. Figure 4.7 on the next page shows a program that has multiple exceptions active at one time. Each time an exception is thrown and caught the stack unwinds and the finally clause runs. This handler throws another exception (until `num_exceptions` gets high enough), which must be allocated. The previous exceptions may not be freed because the handler/catch clause has not been run. Therefore, the EHM must keep all unhandled exceptions alive while it allocates exceptions for new throws.

All exceptions are stored in nodes, which are then linked together in lists one list per stack, with the list head stored in the exception context. Within each linked list, the most recently thrown exception is at the head, followed by older thrown exceptions. This format allows exceptions to be thrown, while a different exception is being handled. The exception at the head of the list is currently being handled, while other exceptions wait for the exceptions before them to be handled and removed.

The virtual members in the exception's virtual table provide the size of the exception, the copy function, and the free function, so they are specific to an exception type. The size and copy function are used immediately to copy an exception into managed memory. After the exception is handled, the free function is used to clean up the exception and then the entire node is passed to `free`, returning the memory back to the heap.

```
unsigned num_exceptions = 0;
void throws() {
    try {
        try {
            ++num_exceptions;             | finally block (Example)
            throw (Example){table};       | try block
        } finally {                       throws()
            if (num_exceptions < 3) {     | finally block (Example)
                throws();                 | try block
            }                             throws()
        }                                 | finally block (Example)
    } catch (exception_t *) {             | try block
        --num_exceptions;                 throws()
    }                                     main()
}
int main() {
    throws();
}
```

Figure 4.7: Multiple Exceptions

4.5.2 Try Statements and Catch Clauses

The try statement with termination handlers is complex because it must compensate for the C code-generation versus proper assembly-code generated from C∀. Libunwind requires an LSDA and personality function for control to unwind across a function. The LSDA in particular is hard to mimic in generated C code.

The workaround is a function called `__cfaehm_try_terminate` in the standard C∀ library. The contents of a try block and the termination handlers are converted into nested functions. These are then passed to the try terminate function and it calls them, appropriately. Because this function is known and fixed (and not an arbitrary function that happens to contain a try statement), its LSDA can be generated ahead of time.

Both the LSDA and the personality function for `__cfaehm_try_terminate` are set ahead of time using embedded assembly. This assembly code is handcrafted using C `asm` statements and contains enough information for the single try statement the function represents.

The three functions passed to try terminate are:

try function: This function is the try block. It is where all the code from inside the try block is placed. It takes no parameters and has no return value. This function is called during regular execution to run the try block.

match function: This function is called during the search phase and decides if a catch clause matches the termination exception. It is constructed from the conditional part of each handler and runs each check, top to bottom, in turn, to see if the exception matches this handler. The match is performed in two steps: first, a virtual cast is used to check if the raised exception is an instance of the declared exception type or one of its descendant types, and then the condition is evaluated, if present. The match function takes a pointer to the exception and returns 0 if the exception is not handled here. Otherwise, the return value is the ID of the handler that matches the exception.

handler function: This function handles the exception, and contains all the code from the handlers in the try statement, joined with a switch statement on the handler's id. It takes a pointer to the exception and the handler's id and returns nothing. It is called after the cleanup phase.

All three functions are created with GCC nested functions. GCC nested functions can be used to create closures; in other words, functions that can refer to variables in their lexical scope even though those variables are part of a different function. This approach allows the functions to refer to all the variables in scope for the function containing the `try` statement. These nested functions and all other functions besides `__cfaehm_try_terminate` in C∀ use the GCC personality function and the `-fexceptions` flag to generate the LSDA. Using this pattern, C∀ implements destructors with the cleanup attribute.

Figure 4.8 shows the pattern used to transform a C∀ try statement with catch clauses into the appropriate C functions.

4.6 Resumption

Resumption is simpler to implement than termination because there is no stack unwinding. Instead of storing the data in a special area using assembly, there is just a linked list of possible handlers for each stack, with each node on the list representing a try statement on the stack.

The head of the list is stored in the exception context. The nodes are stored in order, with the more recent try statements closer to the head of the list. Instead of traversing the stack, resumption handling traverses the list. At each node, the EHM checks to see if the try statement the node represents can handle the exception. If it can, then the exception is handled and the operation finishes; otherwise, the search continues to the next node. If the search reaches the end of the list without finding a try statement with a handler clause that can handle the exception, the default handler is executed. If the default handler returns, control continues after the raise statement.

Each node has a handler function that does most of the work. The handler function is passed the raised exception and returns true if the exception is handled and false otherwise. The handler function checks each of its internal handlers in order, top-to-bottom, until it finds a match. If a match is found that handler is run, after which the function returns true, ignoring all remaining handlers. If no match is found the function returns false. The match is performed in two steps. First a virtual cast is used to see if the raised exception is an instance of the declared exception type or one of its descendant types, if so, then the second step is to see if the exception passes the custom predicate if one is defined.

```
try {
    // TRY BLOCK
} catch (Exception1 * name1 ; check(name1)) {
    // CATCH BLOCK 1
} catch (Exception2 * name2) {
    // CATCH BLOCK 2
}
```

becomes...

```
void try(void) {
    // TRY BLOCK
}
int match(exception_t * __exception_inst) {
    {
        Exception1 * name1;
        if (name1 = (virtual Exception1 *)__exception_inst
                && check(name1)) {
            return 1;
        }
    }
    {
        Exception2 * name2;
        if (name2 = (virtual Exception2 *)__exception_inst) {
            return 2;
        }
    }
    return 0;
}
void catch(exception_t * __exception_inst, int __handler_index) {
    switch (__handler_index) {
    case 1:
    {
        Exception1 * name1 = (virtual Exception1 *)__exception_inst;
        // CATCH BLOCK 1
    }
    return;
    case 2:
    {
        Exception2 * name2 = (virtual Exception2 *)__exception_inst;
        // CATCH BLOCK 2
    }
    return;
    }
}
{
    __cfaehm_try_terminate(try, catch, match);
}
```

Figure 4.8: Termination Transformation

```
try {
    // TRY BLOCK
} catchResume (Exception1 * name1 ; check(name1)) {
    // CATCH BLOCK 1
} catchResume (Exception2 * name2) {
    // CATCH BLOCK 2
}
```

becomes...

```
bool handle(exception_t * __exception_inst) {
    {
        Exception1 * name1;
        if (name1 = (virtual Exception1 *)__exception_inst
                && check(name1)) {
            // CATCH BLOCK 1
            return 1;
        }
    }
    {
        Exception2 * name2;
        if (name2 = (virtual Exception2 *)__exception_inst) {
            // CATCH BLOCK 2
            return 2;
        }
    }
    return false;
}
struct __try_resume_node __resume_node
    __attribute__((cleanup( __cfaehm_try_resume_cleanup )));
__cfaehm_try_resume_setup( &__resume_node, handler );
```

Figure 4.9: Resumption Transformation

Figure 4.9 shows the pattern used to transform a C∀ try statement with catchResume clauses into the appropriate C functions.

Figure 4.10 shows search skipping (see section 3.4.2), which ignores parts of the stack already examined, and is accomplished by updating the front of the list as the search continues. Before the handler is called at a matching node, the head of the list is updated to the next node of the current node. After the search is complete, successful or not, the head of the list is reset. This mechanism means the current handler and every handler that has already been checked are not on the list while a handler is run. If a resumption is thrown during the handling of another resumption, the active handlers and all the other handlers checked up to this point are not checked again. This structure also supports new handlers added while the resumption is being handled. These are added to the front of the list, pointing back along the stack – the first one points over all the checked handlers – and the ordering is maintained.

Finally, the resumption implementation has a cost for entering/exiting a try statement

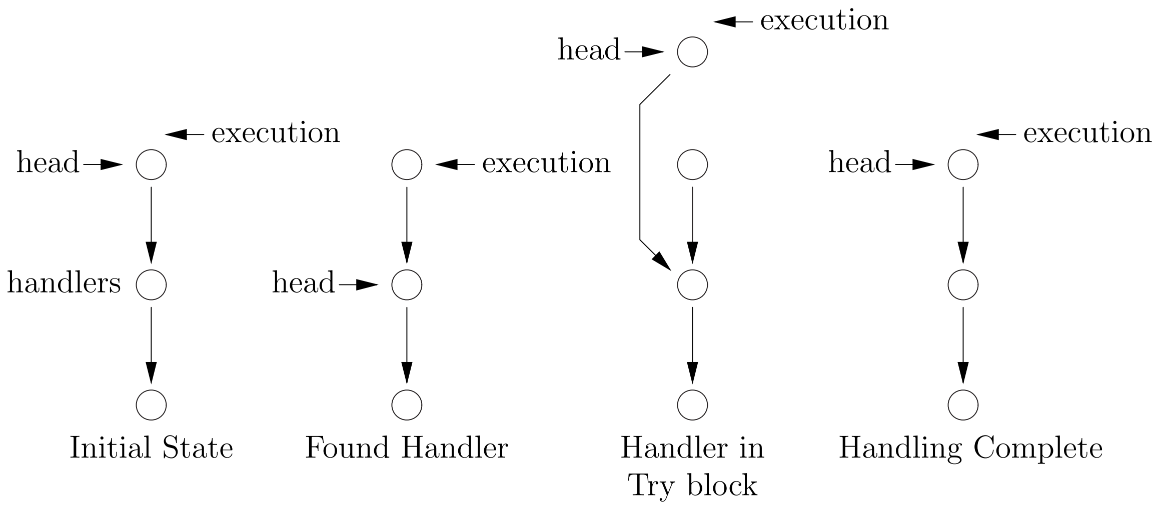


Figure 4.10: Resumption Marking

with `catchResume` clauses, whereas a try statement with `catch` clauses has zero-cost entry/exit. While resumption does not need the stack unwinding and cleanup provided by libunwind, it could use the search phase to providing zero-cost enter/exit using the LSDA. Unfortunately, there is no way to return from a libunwind search without installing a handler or raising an error. Although workarounds might be possible, they are beyond the scope of this thesis. The current resumption implementation has simplicity in its favour.

4.7 Finally

A finally clause is handled by converting it into a once-off destructor. The code inside the clause is placed into a GCC nested-function with a unique name, and no arguments or return values. This nested function is then set as the cleanup function of an empty object that is declared at the beginning of a block placed around the context of the associated try statement, as shown in Figure 4.11.

The rest is handled by GCC. The TRY BLOCK contains the try block itself as well as all code generated for handlers. Once that code has completed, control exits the block and the empty object is cleaned up, which runs the function that contains the finally code.

4.8 Cancellation

Cancellation also uses libunwind to do its stack traversal and unwinding. However, it uses a different primary function: `_Unwind_ForcedUnwind`. Details of its interface can be found in Section 4.3.4 on page 38.

The first step of cancellation is to find the cancelled stack and its type: coroutine, thread or main thread. In C∀, a thread (the construct the user works with) is a user-level thread (point of execution) paired with a coroutine, the thread's main coroutine. The

```
try {
    // TRY BLOCK
} finally {
    // FINALLY BLOCK
}
```

becomes...

```
{
    void finally(void *__hook){
        // FINALLY BLOCK
    }
    __attribute__ ((cleanup(finally)))
    struct __cfaehm_cleanup_hook __finally_hook;
    {
        // TRY BLOCK
    }
}
```

Figure 4.11: Finally Transformation

thread library also stores pointers to the main thread and the current thread. If the current thread's main and current coroutines are the same then the current stack is a thread stack, otherwise it is a coroutine stack. If the current stack is a thread stack, it is also the main thread stack if and only if the main and current threads are the same.

However, if the threading library is not linked, the sequential execution is on the main stack. Hence, the entire check is skipped because the weak-symbol function is loaded. Therefore, main thread cancellation is unconditionally performed.

Regardless of how the stack is chosen, the stop function and parameter are passed to the forced-unwind function. The general pattern of all three stop functions is the same: continue unwinding until the end of stack and then perform the appropriate transfer.

For main stack cancellation, the transfer is just a program abort.

For coroutine cancellation, the exception is stored on the coroutine's stack, and the coroutine context switches to its last resumer. The rest is handled on the backside of the resume, which checks if the resumed coroutine is cancelled. If cancelled, the exception is retrieved from the resumed coroutine, and a `CoroutineCancelled` exception is constructed and loaded with the cancelled exception. It is then resumed as a regular exception with the default handler coming from the context of the resumption call.

For thread cancellation, the exception is stored on the thread's main stack and then context switched to the scheduler. The rest is handled by the thread joiner. When the join is complete, the joiner checks if the joined thread is cancelled. If cancelled, the exception is retrieved and the joined thread, and a `ThreadCancelled` exception is constructed and loaded with the cancelled exception. The default handler is passed in as a function pointer. If it is null (as it is for the auto-generated joins on destructor call), the default is used, which is a program abort.

Chapter 5

Performance

Performance is of secondary importance for most of this project. Instead, the focus was to get the features working. The only performance requirement is to ensure the tests for correctness run in a reasonable amount of time. Hence, only a few basic performance tests were performed to check this requirement.

5.1 Test Set-Up

Tests were run in C∀, C++, Java and Python. In addition there are two sets of tests for C∀, one with termination and one with resumption.

GCC C++ is the most comparable language because both it and C∀ use the same framework, libunwind. In fact, the comparison is almost entirely in quality of implementation. Specifically, C∀'s EHM has had significantly less time to be optimized and does not generate its own assembly. It does have a slight advantage in that C++ has to do some extra bookkeeping to support its utility functions, but otherwise C++ should have a significant advantage.

Java, a popular language with similar termination semantics, is implemented in a very different environment, a virtual machine with garbage collection. It also implements the finally clause on try blocks allowing for a direct feature-to-feature comparison. As with C++, Java's implementation is mature, has more optimizations and extra features as compared to C∀.

Python is used as an alternative comparison because of the C∀ EHM's current performance goals, which is to not be prohibitively slow while the features are designed and examined. Python has similar performance goals for creating quick scripts and its wide use suggests it has achieved those goals.

Unfortunately, there are no notable modern programming languages with resumption exceptions. Even the older programming languages with resumption seem to be notable only for having resumption. On the other hand, the functional equivalents to resumption are too new. There does not seem to be any standard implementations in well-known languages; so far, they seem confined to extensions and research languages. Instead, resumption is compared to its simulation in other programming languages: fixup functions that are explicitly passed into a function.

All tests are run inside a main loop that repeatedly performs a test. This approach avoids start-up or tear-down time from affecting the timing results. The number of times the loop is run is configurable from the command line; the number used in the timing runs is given with the results per test. The Java tests run the main loop 1000 times before beginning the actual test to "warm up" the JVM.

Timing is done internally, with time measured immediately before and after the test loop. The difference is calculated and printed. The loop structure and internal timing means it is impossible to test unhandled exceptions in C++ and Java as that would cause the process to terminate. Luckily, performance on the "give up and kill the process" path is not critical.

The exceptions used in these tests are always based off of the base exception for the language. This requirement minimizes performance differences based on the object model used to represent the exception.

All tests are designed to be as minimal as possible, while still preventing excessive optimizations. For example, empty inline assembly blocks are used in C∀ and C++ to prevent excessive optimizations while adding no actual work.

When collecting data, each test is run eleven times. The top three and bottom three results are discarded and the remaining five values are averaged. The test are run with the latest (still pre-release) C∀ compiler, using gcc-10 10.3.0 as a backend. g++-10 10.3.0 is used for C++. Java tests are complied and run with Oracle OpenJDK version 11.0.11. Python used CPython version 3.8.10. The machines used to run the tests are:

- ARM 2280 Kunpeng 920 48-core 2×socket `@` 2.6 GHz running Linux v5.11.0-25
- AMD 6380 Abu Dhabi 16-core 4×socket `@` 2.5 GHz running Linux v5.11.0-25

These represent the two major families of hardware architecture.

5.2 Tests

The following tests were selected to test the performance of different components of the exception system. They should provide a guide as to where the EHM's costs are found.

Stack Traversal This group of tests measures the cost of traversing the stack (and in termination, unwinding it). Inside the main loop is a call to a recursive function. This function calls itself F times before raising an exception. F is configurable from the command line, but is usually 100. This builds up many stack frames, and any contents they may have, before the raise. The exception is always handled at the base of the stack. For example the Empty test for C∀ resumption looks like:

```
void unwind_empty(unsigned int frames) {
    if (frames) {
        unwind_empty(frames - 1);
    } else {
        throwResume (empty_exception){&empty_vt};
    }
}
```

Other test cases have additional code around the recursive call adding something besides simple stack frames to the stack. Note that both termination and resumption have to traverse over the stack but only termination has to unwind it.

- Empty: The repeating function is empty except for the necessary control code. As other traversal tests add to this, it is the baseline for the group as the cost comes from traversing over and unwinding a stack frame that has no other interactions with the exception system.
- Destructor: The repeating function creates an object with a destructor before calling itself. Comparing this to the empty test gives the time to traverse over and unwind a destructor.
- Finally: The repeating function calls itself inside a try block with a finally clause attached. Comparing this to the empty test gives the time to traverse over and unwind a finally clause.
- Other Handler: The repeating function calls itself inside a try block with a handler that does not match the raised exception, but is of the same kind of handler. This means that the EHM has to check each handler, and continue over all of them until it reaches the base of the stack. Comparing this to the empty test gives the time to traverse over and unwind a handler.

Cross Try Statement This group of tests measures the cost for setting up exception handling, if it is not used because the exceptional case did not occur. Tests repeatedly cross (enter, execute and leave) a try statement but never perform a raise.

- Handler: The try statement has a handler (of the appropriate kind).
- Finally: The try statement has a finally clause.

Conditional Matching This group measures the cost of conditional matching. Only C∀ implements the language level conditional match, the other languages mimic it with an "unconditional" match (it still checks the exception's type) and conditional re-raise if it is not supposed to handle that exception.

Here is the pattern shown in C∀ and C++. Java and Python use the same pattern as C++, but with their own syntax.

```
try {                               try {
    ...                                 ...
} catch (exception_t * e ;          } catch (std::exception & e) {
        should_catch(e)) {              if (!should_catch(e)) throw;
    ...                                 ...
}                                   }
```

- Match All: The condition is always true. (Always matches or never re-raises.)
- Match None: The condition is always false. (Never matches or always re-raises.)

Resumption Simulation A slightly altered version of the Empty Traversal test is used when comparing resumption to fix-up routines. The handler, the actual resumption handler or the fix-up routine, always captures a variable at the base of the loop, and receives a reference to a variable at the raise site, either as a field on the exception or an argument to the fix-up routine.

5.3 Results

Table 5.1, Table 5.2 and Table 5.3 show the test results. In cases where a feature is not supported by a language, the test is skipped for that language and the result is marked N/A. There are also cases where the feature is supported but measuring its cost is impossible. This happened with Java, which uses a JIT that optimizes away the tests and cannot be stopped.[6] These tests are marked N/C. To get results in a consistent range (1 second to 1 minute is ideal, going higher is better than going low) N, the number of iterations of the main loop in each test, is varied between tests. It is also given in the results and has a value in the millions.

An anomaly in some results came from C∀'s use of GCC nested functions. These nested functions are used to create closures that can access stack variables in their lexical scope. However, if they do so, then they can cause the benchmark's run time to increase by an order of magnitude. The simplest solution is to make those values global variables instead of function-local variables. Tests that had to be modified to avoid this problem have been marked with a "*" in the results.

Table 5.1: Termination Performance Results (sec)

	AMD				ARM			
N	C∀	C++	Java	Python	C∀	C++	Java	Python
Empty Traversal (1M)	23.0	9.6	17.6	23.4	30.6	13.6	15.5	14.7
D'tor Traversal (1M)	48.1	23.5	N/A	N/A	64.2	29.2	N/A	N/A
Finally Traversal (1M)	3.2*	N/A	17.6	29.2	3.9*	N/A	15.5	19.0
Other Traversal (1M)	3.3*	23.9	17.7	32.8	3.9*	24.5	15.5	21.6
Cross Handler (1B)	6.5	0.9	N/C	38.0	9.6	0.8	N/C	32.1
Cross Finally (1B)	0.8	N/A	N/C	44.6	0.6	N/A	N/C	37.3
Match All (10M)	30.5	20.6	11.2	3.9	36.9	24.6	10.7	3.1
Match None (10M)	30.6	50.9	11.2	5.0	36.9	71.9	10.7	4.1

One result not directly related to C∀ but important to keep in mind is that, for exceptions, the standard intuition about which languages should go faster often does not hold. For example, there are a few cases where Python out-performs C∀, C++ and Java. The most likely explanation is that the generally faster languages have made "common

Table 5.2: Resumption Performance Results (sec)

N	AMD	ARM
Empty Traversal (10M)	1.4	1.2
D'tor Traversal (10M)	1.8	1.0
Finally Traversal (10M)	1.8	1.0
Other Traversal (10M)	22.6	25.8
Cross Handler (1B)	9.0	11.9
Match All (100M)	2.3	3.2
Match None (100M)	3.0	3.8

Table 5.3: Resumption/Fixup Routine Comparison (sec)

	AMD					ARM				
N	Raise	C∀	C++	Java	Python	Raise	C∀	C++	Java	Python
Resume Empty (10M)	1.4	1.4	15.4	2.3	178.0	1.2	1.2	8.9	1.2	118.4

cases fast" at the expense of the rarer cases. Since exceptions are considered rare, they are made expensive to help speed up common actions, such as entering and leaving try statements. Python, on the other hand, while generally slower than the other languages, uses exceptions more and has not sacrificed their performance. In addition, languages with high-level representations have a much easier time scanning the stack as there is less to decode.

As stated, the performance tests are not attempting to show C∀ has a new competitive way of implementing exception handling. The only performance requirement is to insure the C∀ EHM has reasonable performance for prototyping. Although that may be hard to exactly quantify, I believe it has succeeded in that regard. Details on the different test cases follow.

5.3.1 Termination (Table 5.1)

Empty Traversal C∀ is slower than C++, but is still faster than the other languages and closer to C++ than other languages. This result is to be expected, as C∀ is closer to C++ than the other languages.

D'tor Traversal Running destructors causes a huge slowdown in the two languages that support them. C∀ has a higher proportionate slowdown but it is similar to C++'s. Considering the amount of work done in destructors is effectively zero (an assembly comment), the cost must come from the change of context required to run the destructor.

Finally Traversal Performance is similar to Empty Traversal in all languages that support finally clauses. Only Python seems to have a larger than random noise change in its run time and it is still not large. Despite the similarity between finally clauses and destructors, finally clauses seem to avoid the spike that run time destructors have. Possibly some optimization removes the cost of changing contexts.

Other Traversal For C++, stopping to check if a handler applies seems to be about as expensive as stopping to run a destructor. This results in a significant jump.

Other languages experience a small increase in run time. The small increase likely comes from running the checks, but they could avoid the spike by not having the same kind of overhead for switching to the check's context.

Cross Handler Here, C∀ falls behind C++ by a much more significant margin. This is likely due to the fact that C∀ has to insert two extra function calls, while C++ does not have to execute any other instructions. Python is much further behind.

Cross Finally C∀'s performance now matches C++'s from Cross Handler. If the code from the finally clause is being inlined, which is just an asm comment, than there are no additional instructions to execute again when exiting the try statement normally.

Conditional Match Both of the conditional matching tests can be considered on their own. However, for evaluating the value of conditional matching itself, the comparison of the two sets of results is useful. Consider the massive jump in run time for C++ going from match all to match none, which none of the other languages have. Some strange interaction is causing run time to more than double for doing twice as many raises. Java and Python avoid this problem and have similar run time for both tests, possibly through resource reuse or their program representation. However, C∀ is built like C++, and avoids the problem as well. This matches the pattern of the conditional match, which makes the two execution paths very similar.

5.3.2 Resumption (Table 5.2)

Moving on to resumption, there is one general note: resumption is *fast*. The only test where it fell behind termination is Cross Handler. In every other case, the number of iterations had to be increased by a factor of 10 to get the run time in an appropriate range and in some cases resumption still took less time.

Empty Traversal See above for the general speed-up notes. This result is not surprising as resumption's linked-list approach means that traversing over stack frames without a resumption handler is $O(1)$.

D'tor Traversal Resumption does have the same spike in run time that termination has. The run time is actually very similar to Finally Traversal. As resumption does not unwind the stack, both destructors and finally clauses are run while walking down the stack during the recursive returns. So it follows their performance is similar.

Finally Traversal Same as D'tor Traversal, except termination did not have a spike in run time on this test case.

Other Traversal Traversing across handlers reduces resumption's advantage as it actually has to stop and check each one. Resumption still came out ahead (adjusting for iterations) but by much less than the other cases.

Cross Handler The only test case where resumption could not keep up with termination, although the difference is not as significant as many other cases. It is simply a matter of where the costs come from: both termination and resumption have some work to set up or tear down a handler. It just so happens that resumption's work is slightly slower.

Conditional Match Resumption shows a slight slowdown if the exception is not matched by the first handler, which follows from the fact the second handler now has to be checked. However, the difference is not large.

5.3.3 Resumption/Fixup (Table 5.3)

Finally are the results of the resumption/fixup routine comparison. These results are surprisingly varied. It is possible that creating a closure has more to do with performance than passing the argument through layers of calls. At 100 stack frames, resumption and manual fixup routines have similar performance in C∀. More experiments could try to tease out the exact trade-offs, but the prototype's only performance goal is to be reasonable. It is already in that range, and C∀'s fixup routine simulation is one of the faster simulations as well. Plus, exceptions add features and remove syntactic overhead, so even at similar performance, resumptions have advantages over fixup routines.

Chapter 6

Future Work

The following discussion covers both possible interesting research that could follow from this work as well as simple implementation improvements.

6.1 Language Improvements

C∀ is a developing programming language. As such, there are partially or unimplemented features (including several broken components) that I had to work around while building the EHM largely in the C∀ language (some C components). Below are a few of these issues and how implementing/fixing them would affect the EHM. In addition, there are some simple improvements that had no interesting research attached to them but would make using the language easier.

- Due to a type-system problem, the catch clause cannot bind the exception to a reference instead of a pointer. Since C∀ has a very general reference capability, programmers will want to use it. Once fixed, this capability should result in little or no change in the exception system but simplify usage.
- The `copy` function in the exception virtual table is an adapter to address some limitations in the C∀ copy constructor. If the copy constructor is improved it can be used directly without the adapter.
- Termination handlers cannot use local control-flow transfers, e.g., by `break`, `return`, etc. The reason is that current code generation hoists a handler into a nested function for convenience (versus assembly-code generation at the try statement). Hence, when the handler runs, it can still access local variables in the lexical scope of the try statement. Still, it does mean that seemingly local control flow is not in fact local and crosses a function boundary. Making the termination handler's code within the surrounding function would remove this limitation.
- There is no detection of colliding unwinds. It is possible for cleanup code run during an unwind to trigger another unwind that escapes the cleanup code itself, such as

a termination exception caught further down the stack or a cancellation. There do exist ways to handle this case, but currently there is no detection and the first unwind will simply be forgotten, often leaving it in a bad state.

- Finally, the exception system has not had a lot of programmer testing. More time with encouraged usage will reveal new quality of life upgrades that can be made.

6.2 Complete Virtual System

The virtual system should be completed. It was not supposed to be part of this project, but was thrust upon it to do exception inheritance; hence, only minimal work is done. A draft for a complete virtual system is available but not finalized. A future C∀ project is to complete that work and then update the exception system that uses the current version.

There are several improvements to the virtual system that would improve the exception traits. The most important one is an assertion to check one virtual type is a child of another. This check precisely captures many of the current ad-hoc correctness requirements.

Other features of the virtual system could also remove some of the special cases around exception virtual tables, such as the generation of the `msg` function.

The full virtual system might also include other improvement like associated types to allow traits to refer to types not listed in their header. This feature allows exception traits to not refer to the virtual-table type explicitly, removing the need for the current interface macros, such as `EHM_IS_EXCEPTION`.

6.3 Additional Raises

Several other kinds of exception raises were considered beyond termination (`throw`), resumption (`throwResume`), and re-raise.

The first is a non-local/concurrent raise providing asynchronous exceptions, i.e., raising an exception on another stack. This semantics acts like signals allowing for out-of-band communication among coroutines and threads. This kind of raise is often restricted to resumption to allow the target stack to continue execution normally after the exception has been handled. That is, allowing one coroutine/thread to unwind the stack of another via termination is bad software engineering.

Non-local/concurrent raise requires more coordination between the concurrency system and the exception system. Many of the interesting design decisions center around masking, i.e., controlling which exceptions may be thrown at a stack. It would likely require more of the virtual system and would also effect how default handlers are set.

Other raises were considered to mimic bidirectional algebraic effects. Algebraic effects are used in some functional languages allowing one function to have another function on the stack resolve an effect (which is defined with a functional-like interface). This semantics can be mimicked with resumptions and new raises were discussed to mimic bidirectional algebraic-effects, where control can go back and forth between the function-effect caller and handler while the effect is underway. These raises would be like the resumption raise except using different search patterns to find the handler.

6.4 Checked Exceptions

Checked exceptions make exceptions part of a function's type by adding an exception signature. An exception signature must declare all checked exceptions that could propagate from the function, either because they were raised inside the function or came from a sub-function. This improves safety by making sure every checked exception is either handled or consciously passed on.

Checked exceptions were never seriously considered for this project because they have significant trade-offs in usability and code reuse in exchange for the increased safety. These trade-offs are most problematic when trying to pass exceptions through higher-order functions from the functions the user passed into the higher-order function. There are no well known solutions to this problem that were satisfactory for C∀ (which carries some of C's flexibility-over-safety design) so additional research is needed.

Follow-up work might add some form of checked exceptions to C∀, possibly using polymorphic exception signatures, a form of tunneling[17] or checked and unchecked raises.

6.5 Zero-Cost Try

C∀ does not have zero-cost try-statements because the compiler generates C code rather than assembler code (see on page 44). When the compiler does create its own assembly (or LLVM byte-code), then zero-cost try-statements are possible. The downside of zero-cost try-statements is the LSDA complexity, its size (program bloat), and the high cost of raising an exception.

Alternatively, some research could be done into the simpler alternative method with a non-zero-cost try-statement but much lower cost exception raise. For example, programs are starting to use exception in the normal control path, so more exceptions are thrown. In these cases, the cost balance switches towards low-cost raise. Unfortunately, while exceptions remain exceptional, the libunwind model will probably remain the most effective option.

Zero-cost resumptions is still an open problem. First, because libunwind does not support a successful-exiting stack-search without doing an unwind. Workarounds are possible but awkward. Ideally, an extension to libunwind could be made, but that would either require separate maintenance or gaining enough support to have it folded into the official library itself.

Also, new techniques to skip previously searched parts of the stack need to be developed to handle the recursive resume problem and support advanced algebraic effects.

6.6 Signal Exceptions

Goodenough [9] suggests three types of exceptions: escape, notify and signal. Escape are termination exceptions, notify are resumption exceptions, leaving signal unimplemented.

A signal exception allows either behaviour, i.e., after an exception is handled, the handler has the option of returning to the raise or after the `try` statement. Currently, C∀ fixes the semantics of the handler return syntactically by the `catch` or `catchResume` clause.

Signal exception should be reexamined and possibly be supported in C∀. A very direct translation is to have a new raise and catch pair, and a new statement (or statements) would indicate if the handler returns to the raise or continues where it is; but there may be other options.

For instance, resumption could be extended to cover this use by allowing local control flow out of it. This approach would require an unwind as part of the transition as there are stack frames that have to be removed between where the resumption handler is installed and where it is defined. This approach would not require, but might benefit from, a special statement to leave the handler. Currently, mimicking this behaviour in C∀ is possible by throwing a termination exception inside a resumption handler.

Chapter 7

Conclusion

In the previous chapters, this thesis presents the design and implementation of C∀'s exception handling mechanism (EHM). Both the design and implementation are based off of tools and techniques developed for other programming languages but they were adapted to better fit C∀'s feature set and add a few features that do not exist in other EHMs, including conditional matching, default handlers for unhandled exceptions and cancellation though coroutines and threads back to the program main stack.

The resulting features cover all of the major use cases of the most popular termination EHMs of today, along with reintroducing resumption exceptions and creating some new features that fit with C∀'s larger programming patterns, such as virtuals independent of traditional objects.

The C∀ project's test suite has been expanded to test the EHM. The implementation's performance has also been compared to other implementations with a small set of targeted micro-benchmarks. The results, while not cutting edge, are good enough for prototyping, which is C∀'s current stage of development.

This initial EHM will bring valuable new features to C∀ in its own right but also serves as a tool and motivation for other developments in the language.

References

[1] Ada. *The Programming Language Ada: Reference Manual*. United States Department of Defense, ANSI/MIL-STD-1815A-1983 edition, February 1983. Springer, New York. 4

[2] Ada12. *Programming languages – Ada ISO/IEC 8652:2012*. International Standard Organization, Geneva, Switzerland, 3rd edition, 2012. https://www.iso.org/standard/61507.html. 4

[3] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, September 2000. 24

[4] C++ Community. https://en.cppreference.com/w/cpp/language/history. 4

[5] C++ Community. https://en.cppreference.com/w/cpp/language/except_spec. 7

[6] Dave Dice. personal communication, August 2021. 50

[7] Python Software Foundation. https://docs.python.org/3/library/exceptions.html. 7

[8] Python Software Foundation. https://docs.python.org/3/reference/compound_stmts.html. 7

[9] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975. 3, 23, 57

[10] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Specification*, Java SE 8 edition, 2015. 4

[11] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, 2021. http://golang.org/ref/spec. 5

[12] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, Palo Alto, California, U.S.A., April 1979. 4

[13] Aaron Moss. *C∀ Type System Implementation*. PhD thesis, School of Computer Science, University of Waterloo, 2019. https://uwspace.uwaterloo.ca/handle/10012/14584. 11

[14] Aaron Moss, Robert Schluntz, and Peter A. Buhr. C∀ : Adding modern programming language features to C. *Softw. Pract. Exper.*, 48(12):2111–2146, December 2018. http://dx.doi.org/10.1002/spe.2624. 8

[15] The Rust Team. https://doc.rust-lang.org/std/macro.panic.html. 5

[16] The Rust Team. https://doc.rust-lang.org/std/panic/index.html. 5

[17] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, January 2019. 4, 56