Nonlocal Transfer

Arnav Gupta

October 26, 2024

Contents

1 Nonlocal Transfer

Routine **activation** (call/invocation) introduces complex control flow. Among routines, control flow is controlled by call/return mechanism.

Modularization: from software engineering, any contiguous code block can be factored into a helper routine and called in the program

Modularization fails when refactoring exits (multi-level exits).

Software pattern: many routines have multiple outcomes:

- normal: return normal result and transfer after call
- exceptional: return alternative result and not transfer after call

Nonlocal transfer allows a routine to transfer back to its caller but not after the call

- generalization of multi-exit loop and multi-level exit
- control structures end normally or with an exceptional transfer

This pattern acknowledges that

- algorithms can have multiple outcomes
- separating outcomes makes it easy to read and maintain a program

This pattern does not handle multiple levels of nested modularization.

2 Traditional Approaches

Return code: returns value indicating normal or exceptional execution

Status flag: set shared (global) variable indicating normal or exceptional execution which remains as long as it is not overwritten

Fix-up routine: a global and/or local routine called for an exceptional event to fix-up and return a corrective result so a computation can continue

Return union: modern approach combining result/return-code and requiring return-code check on result access, requires that all routines return an appropriate union

Traditional approaches force checking, unless explicitly accessing without `holds_alternative`.

Drawbacks of traditional techniques:

- checking return code or status flag is optional, so can be delayed or omitted
- return code mixes exceptional and normal values, enlarges type or value range since normal/exceptional type/values should be independent

Testing/handling return code or status flag is often done locally (inline) since info could be lost otherwise, but local testing/handling:

- reduces readability, since each call results in multiple statements
- can be inappropriate since library routines should not terminate a program

Nonlocal testing from nested routine calls is difficult since multiple codes are returned for analysis, compounding the mixing problem.

Status flag can be overwritten before being examined and cannot be used in a concurrent environment because of sharing issues.

Local fix-up routines increase the number of params since they increase the cost of each call and must be passed through multiple levels, enlarging param lists even when fix-up routine is unused.

Nonlocal fix-up routines, implemented with global routine pointer, have identical problems with status flags.

3 Dynamic Multi-Level Exit

Better than returning one level at a time and allows transfer of control directly to original caller.

Dynamic Multi-level Exit: extends call/return semantics to transfer in the reverse direction to normal routine calls, requiring nonlocal transfer.

A **label variable** contains:

- a pointer to a block activation on the stack
- a transfer point within the block

Nonlocal transfer is a 2 step operation:

1. direct control flow to the specified activation on the stack
2. go to the transfer point (label constant) within that routine

Label value is not statically determined (hence dynamic MLE). With recursion, the distance between activations on the stack is unknown and labels can be set and reset through recursive calls.

Transfer between `goto` and label causes termination of a stack block. Termination is implicit for direct transfer of control or requires stack unwinding if activations contain objects with destructors or finalizers.

DME is possible in C using:

- `jmp_buf` to declare a label variable
- `setjmp` to initialize a label variable
- `longjmp` to go to a label variable

DME allows multiple forms of returns to any level:

- normal return transfers to statement after the call, implying completion of routine's algorithm
- exceptional return transfers to statement not after the call, indicating an ancillary completion

Nonlocal transfer can be problematic since it is too general.

4 Exception Handling

Refers to DME and nonlocal transfer among routines, and is more than just error handling.

Exceptional event: event that is usually known to exist but is ancillary to an algorithm (occurs with low frequency)

Exception Handling Mechanism (EHM): provides alternate kinds of control flow

EHM allows for more robustness since exceptions are active, forcing programs to react immediately when an exceptional event occurs.

5 Terminology

Execution: language unit in which an exception can be raised, usually any entity with its own runtime stack

Exception: instance of an exception type, generated by executing an operation indicating an ancillary (exceptional) situation in execution

Raise/Throw: special operation that creates an exception

Source Execution: execution that raises an exception

Faulting Execution: execution changing control flow due to a raised exception

Local Exception: when an exception is raised and handled by the same execution (source and faulting are the same)

Nonlocal Exception: when an exception is raised by a source execution but delivered to a different faulting execution

Concurrent Exception: nonlocal exception where the source and faulting executions are executing concurrently

Propagation: directs control from a raise in the source execution to a handler in the faulting execution

Propagation Mechanism: rules used to locate a handler

- most commonly gives precedence to handlers higher in the call stack and specific handlers, through linear search during propagation

Handler: inline routine responsible for handling raised exception

- catches exception by matching exception type
- executes like a normal subroutine and can return (exception handled) or raise exceptions
- re-raising the current exception continues propagation of caught exception

Guarded Block: language block with associated handlers

Unguarded Block: block with no handlers

Termination: control cannot return to the raise point, since **stack unwinding** occurs

Resumption: control returns to the raise point (no stack unwinding)

6 Execution Environment

An object-oriented concurrent environment requires a more complex EHM than a non object-oriented sequential environment.

Objects may have destructors that must be executed no matter if the object ends by normal or exceptional termination. (ex. `finally` clause)

C++ does not allow direct nonlocal transfer, continuation, coroutine, and tasks with their own execution stack.

With multiple stacks, an EHM can be more sophisticated (more complex).

7 Implementation

In most programming languages, DME is limited with exception handling.

To implement throw/catch, the `throw` must know the last guarded block with handler for the raised exception type. The approach for this is:

- associate a label variable with each exception type
- set label variable on entry to each guarded block with handler for the type
- reset label variable on exit to previous value

Setting/resetting label on each `try` block can have unnecessary cost, so instead `catch` data is stored once externally for each block and handler is found by linear search during a stack walk. This makes `try` cheap but `throw` expensive.

8 Static/Dynamic Call/Return

All routine/exceptional control flow can be characterized as:

1. **static/dynamic call**: routine/exception name at the call/raise is looked up statically (compile-time) or dynamically (runtime)
2. **static/dynamic return**: after a routine/handler completes, it returns to its static (definition) or dynamic (call) context

return/handled	static call	dynamic call
static	sequel	termination exception
dynamic	routine	routine pointer, virtual routine, resumption

8.1 Static Propagation (Sequel)

A routine with no return value, where

- the sequel name is looked up lexically at the call site
- control returns to the end of the block in which the sequel is declared

Allows modularization of code with static exits, and propagation is along the lexical structure.

Sequels help handle termination for non-recoverable events.

Advantage: handler is statically known and can be as efficient as a direct transfer

Disadvantage: sequel only works for monolithic programs since it must be statically nested at the point of use

8.2 Dynamic Propagation

Dynamic propagation/static return is just dynamic MLE.

Advantage: dynamic propagation works for separately compiled programs

Disadvantage: handler is not statically known

8.2.1 Termination

Control transfers from the start of propagation to a dynamically defined handler and then performs a static return (like a sequel).

Basic termination forms for a non-recoverable operation:

- **terminate**: provides limited mechanism for block transfer on the call stack, like labelled break and no need to forward alternative outcomes
  - catch can also depend on the object raising the exception (exception object from object used)
- **retry**: combination of termination with special handler semantics, restart the guarded block handling the exception

8.2.2 Resumption

Control transfers from the start of propagation to a dynamically defined handler and then performs a dynamic return (stack not unwound).

Resumption handler is a corrective action so a computation can continue.