Nonlocal Transfer

Arnav Gupta

September 5, 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.