Two-phase Unwinding and its Implications

Heejin Ahn

Two-Phase Stack Unwinding

1. Search phase:

Walk up the stack and check each ‘catch’ if the current exception should be caught by it

```
try {
  try {
    foo(); // throws 3
  } catch (float f) {
  }
} catch (int n) {
}
```

Two-Phase Stack Unwinding

1. Search phase:

Walk up the stack and check each ‘catch’ if the current exception should be caught by it

```
try {
  try {
    foo(); // throws 3
  } catch (float f) {
  }
} catch (int n) {
}
```

Is this a float?
No!

Two-Phase Stack Unwinding

1. Search phase:

Walk up the stack and check each ‘catch’ if the current exception should be caught by it

```
try {
  try {
    foo(); // throws 3
  } catch (float f) {
  }
} catch (int n) {
}
```

Is this an int?
Yes!

Two-Phase Stack Unwinding

2. Cleanup phase:

Run destructors and unwind the call stack until we reach the ‘`catch`’ that catches the exception

(If there’s no such `catch`, the exception makes the program crash)

```
try {
  try {
    foo(); // throws 3
  } catch (float f) {
  }
} catch (int n) {
}
```

Two-Phase Stack Unwinding

- Preserves the whole stack intact when an exception is not caught
- Helps debugging

```
terminate called after throwing an instance of 'int'

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7c1b55b in __GI_abort () at abort.c:79
#2  0x00007ffff7e6d80c in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so
#3  0x00007ffff7e788f6 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so
#4  0x00007ffff7e78961 in std::terminate() () from /usr/lib/x86_64-linux-gn
#5  0x00007ffff7e78bf5 in __cxa_throw () from /usr/lib/x86_64-linux-gnu/lib
#6  0x000055555555519d in foo () at foo.cpp:2
#7  0x00005555555551aa in main () at foo.cpp:8
```

Two-Phase Stack Unwinding

- Some language features require two-phase unwinding
  - e.g. C#'s when clause

```
try {
  try {
    foo(); // throws Exception
  } catch (Exception e) when (print(1), false) {
  } finally {
    print(3);
  }
} catch (Exception e) when (print(2), true) {
  print(4);
}
```

```
Result:

1
2
3
4
```

Agenda for Today

- Goals
  - Discuss changes to the current proposal in the way that it is *extensible* to the future two-phase unwinding follow-on proposal
- Non-goals
  - Discuss syntactic details of two-phase unwinding instructions

Current Proposal

- `try … catch … end`
- `throw` (takes values)
- `rethrow` (takes `exnref`)
- `br_on_exn $l` (takes `exnref`)

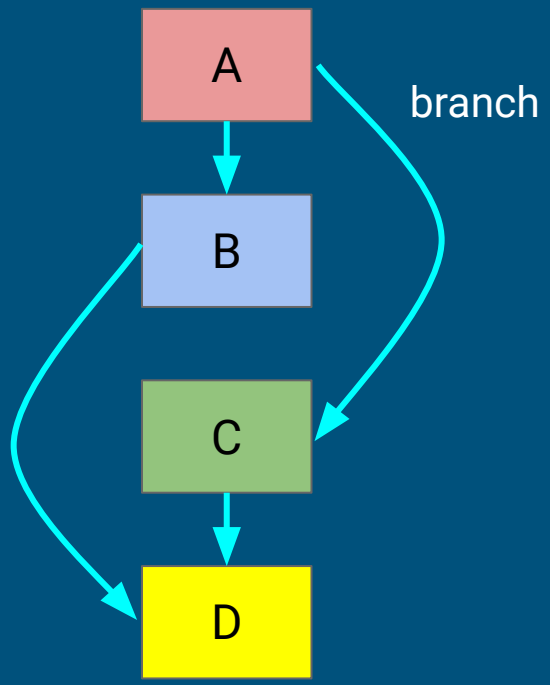
- `exnref` is a first class type
  - Introduced in 2018 to make code transformation easier, i.e., fix unwind mismatch problem

Two-phase Unwinding Support?

- `catch` *can* take a filter function
- The first (search) phase runs filters while walking up the stack until it finds matching `catch`
- The second (unwind) phase unwind the stack until it reaches the matching `catch`
- Is that it?
  - Maybe not really

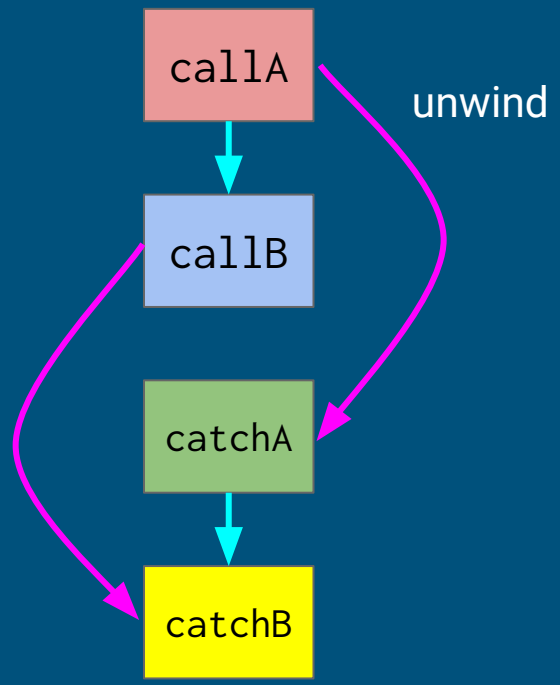
```
try
  …
catch $filter
  …
end
```

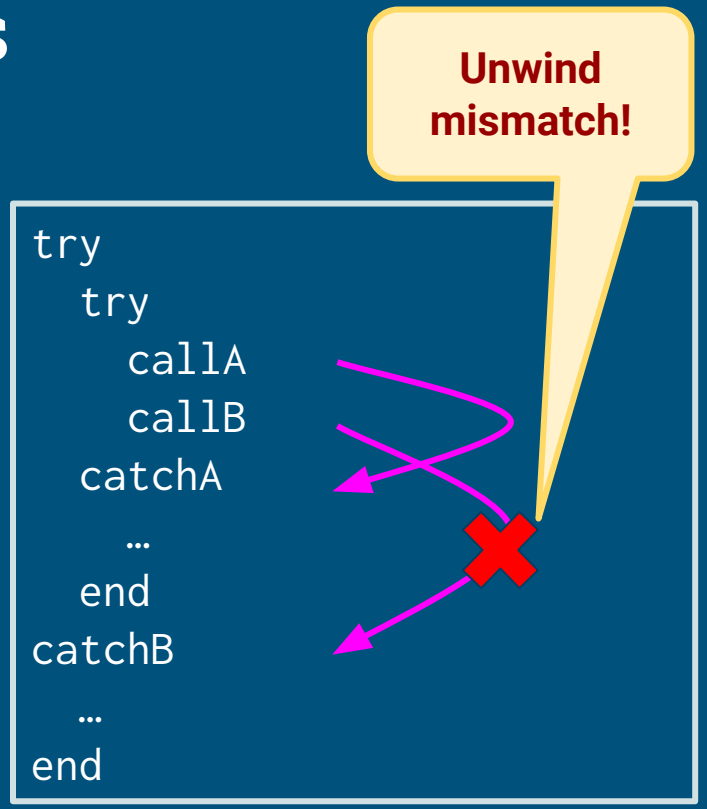
CFG Stackification for Blocks



```
block $label0
  block $label1
    A: br $label1
    B: br $label0
  end
  C
end
D
```

CFG Stackification for Trys





Fixing Unwind Mismatches

```
try
  try
    callA
    callB
  catchA
    …
  end
catchB
  handler
end
```

Unwind mismatch!

```
try $outer
  try
    callA
    try
      callB
    catch
      local.set $0
      br $outer
    end
  catchA
    …
  end
catchB
end
handler
```

exnref to the rescue!

Unwind Mismatch Problem

- Unwind destination mismatches occuring when *linearizing* CFG into wasm
  - We call it “CFG stackification”
- Can happen with any compiler framework that uses BB and CFG
- Branches can avoid this problem because they can specify a target label
- But calls don’t have a target label

`exnref` and Two-phase unwinding

- `exnref` can escape `catch` blocks
- To fix an unwind mismatch, we had to do code transformation that alters and splits the call stack using `exnref` behind the scene
- This is not going to work in the presence of two-phase unwinding, because in two-phase, you have to scan (or walk up) the whole call stack first

References

- Related issue: https://github.com/WebAssembly/exception-handling/issues/123