Исключения С++ через призму компиляторных оптимизаций LLVM

Роман Русяев

Samsung R&D

compilers developer

rusyaev.rm@gmail.com

План доклада

- Введение в реализацию исключений
- Как исключения поддержаны в LLVM
- Влияние исключений на оптимизации компилятора (на примере LLVM)

Цель доклада

Как оптимизирующий компилятор работает с исключениями C++, и как это может отразиться на производительности ваших приложений:

- насколько дороги исключения, даже если они не выбрасываются?

Цель доклада

Как оптимизирующий компилятор работает с исключениями C++, и как это может отразиться на производительности ваших приложений:

- насколько дороги исключения, даже если они не выбрасываются?
- когда лучше исключения не использовать

Цель доклада

Как оптимизирующий компилятор работает с исключениями C++, и как это может отразиться на производительности ваших приложений:

- насколько дороги исключения, даже если они не выбрасываются?
- когда лучше исключения не использовать
- **noexcept везде, где можно**

noexcept везде, где можно

- **const** везде, где можно

noexcept везде, где можно

- **const** везде, где можно
- Практично ли это?
  - синтаксический мусор
  - с появлением семантики перемещения все стало сложнее

noexcept везде, где можно

- **const** везде, где можно
- Практично ли это?
  - синтаксический мусор
  - с появлением семантики перемещения все стало сложнее
- **const –** всегда, когда нужно (ссылки, указатели, функции члены etc)

noexcept везде, где можно

- **const** везде, где можно
- Практично ли это?
  - синтаксический мусор
  - с появлением семантики перемещения все стало сложнее
- **const –** всегда, когда нужно (ссылки, указатели, функции члены etc)
- А что с **noexcept**?

Введение в реализацию исключений

C++ RUSSIA

Zero-Cost Exception Handling (0eh)

- придуман для IA-64 (Itanium)
- спецификация описана в **Itanium C++ ABI:** https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

Zero-Cost Exception Handling (0eh)

Терминология

- спецификация исключений функции:

```
void foo() noexcept
void foo() throw()                    // deprecated
void foo() throw(type1, type2, ...) // deprecated
```

- cleanup
- обработчик исключения
- stack unwinding

stack unwinding

- **stack unwinding**: осуществляет вызов деструкторов локальных объектов каждого стекового фрейма, пока не будет найден фрейм с обработчиком исключения, соответствующим объекту брошенного исключения.

stack unwinding

- **stack unwinding**: осуществляет вызов деструкторов локальных объектов каждого стекового фрейма, пока не будет найден фрейм с обработчиком исключения, соответствующим объекту брошенного исключения. Состоит из 2х этапов:
  - *search phase*: поиск обработчика исключения
  - *clean up phase:* вызов деструкторов локальных объектов и передача на обработчик исключения

stack unwinding, cleanup

- **stack unwinding**: осуществляет вызов деструкторов локальных объектов каждого стекового фрейма, пока не будет найден фрейм с обработчиком исключения, соответствующим объекту брошенного исключения. Состоит из 2х этапов:
  - *search phase*: поиск обработчика исключения
  - *clean up phase:* вызов деструкторов локальных объектов и передача на обработчик исключения
- **cleanup**: выполняет вызовы деструкторов локальных объектов в процессе stack unwinding

C++ RUSSIA

stack unwinding, cleanup, обработчики исключений

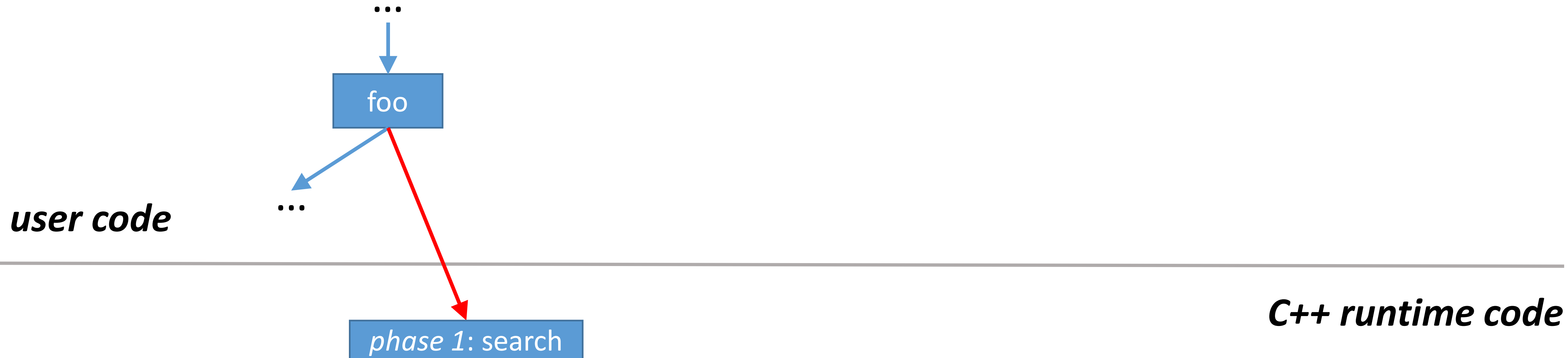
- **stack unwinding**: осуществляет вызов деструкторов локальных объектов каждого стекового фрейма, пока не будет найден фрейм с обработчиком исключения, соответствующим объекту брошенного исключения. Состоит из 2х этапов:
  - *search phase*: поиск обработчика исключения
  - *clean up phase:* вызов деструкторов локальных объектов и передача на обработчик исключения
- **cleanup**: выполняет вызовы деструкторов локальных объектов в процессе stack unwinding
- **обработчик исключения**: код, отвечающий за обработку выброшенного исключения

C++ RUSSIA

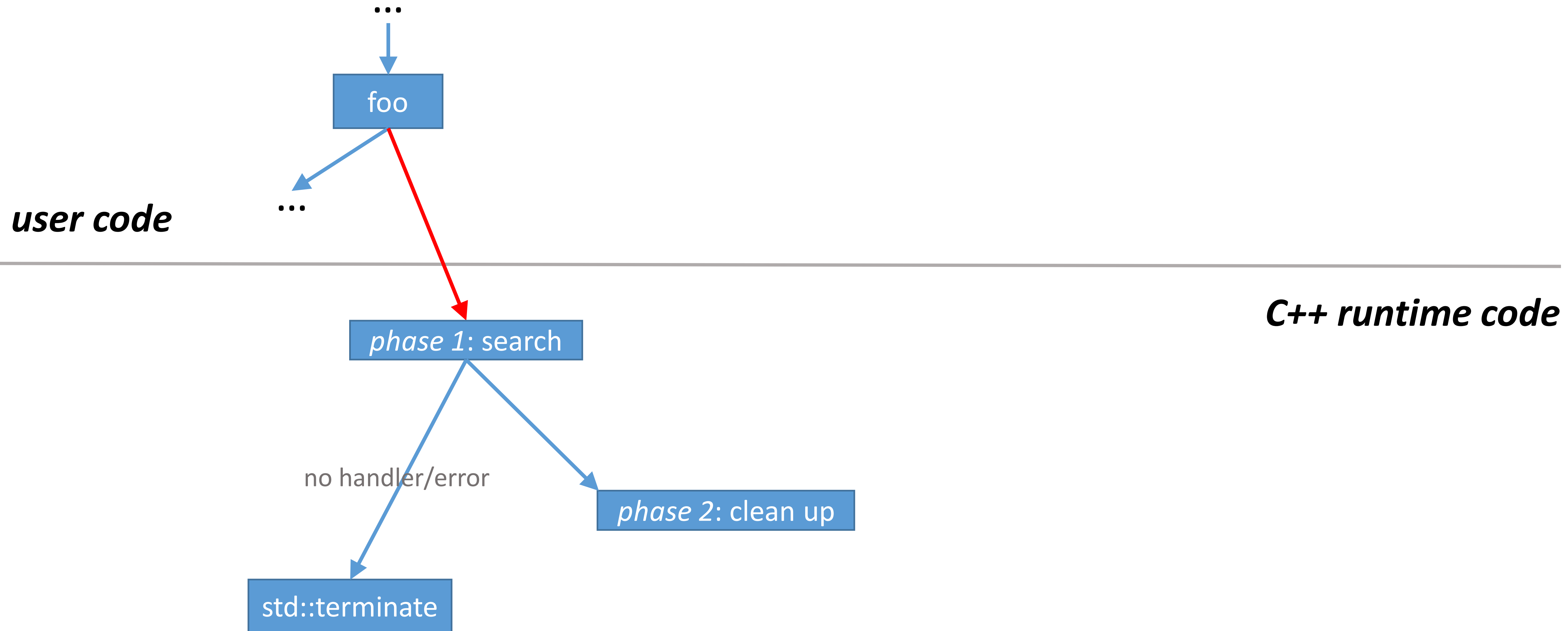
std::terminate

- исключение не ловится
- не соблюден **noexcept** спецификатор функции
- исключение бросается из cleanup
- … (еще ~10 случаев)

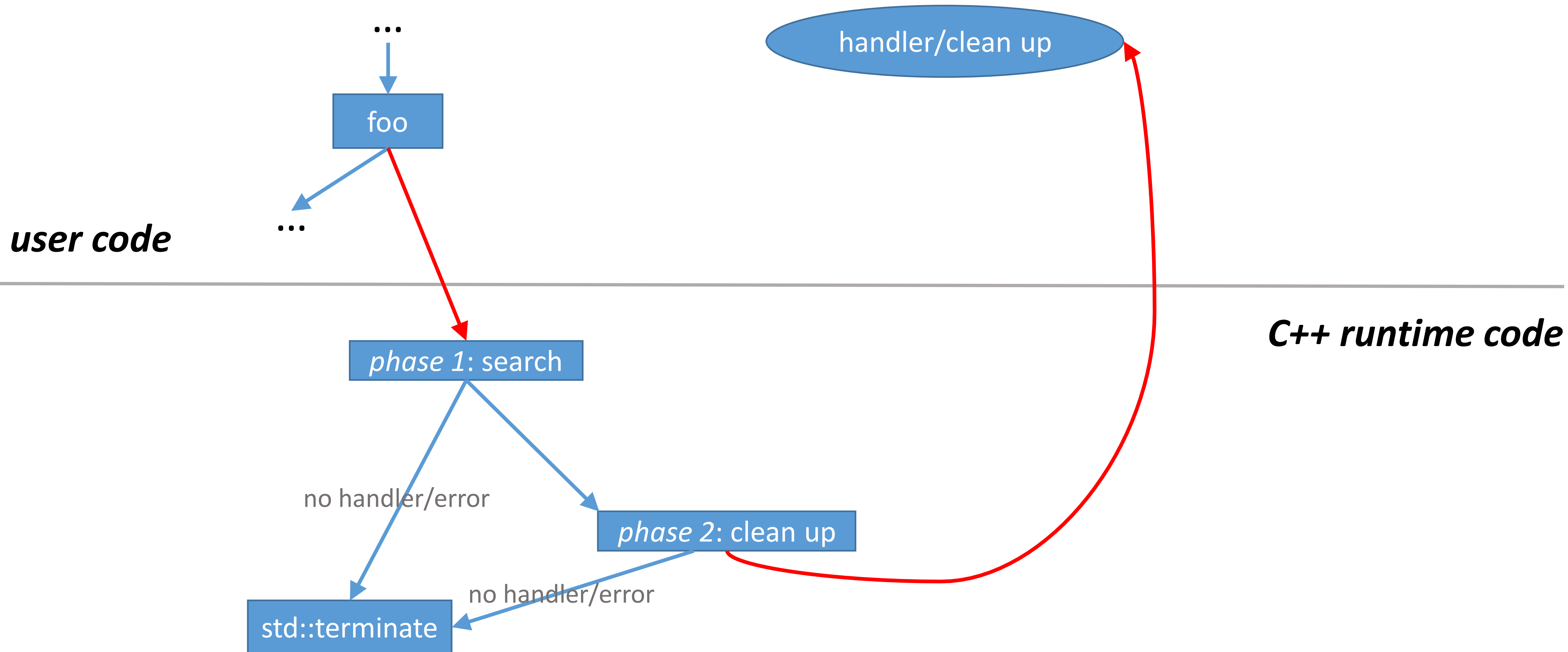
Что происходит при бросании исключения?



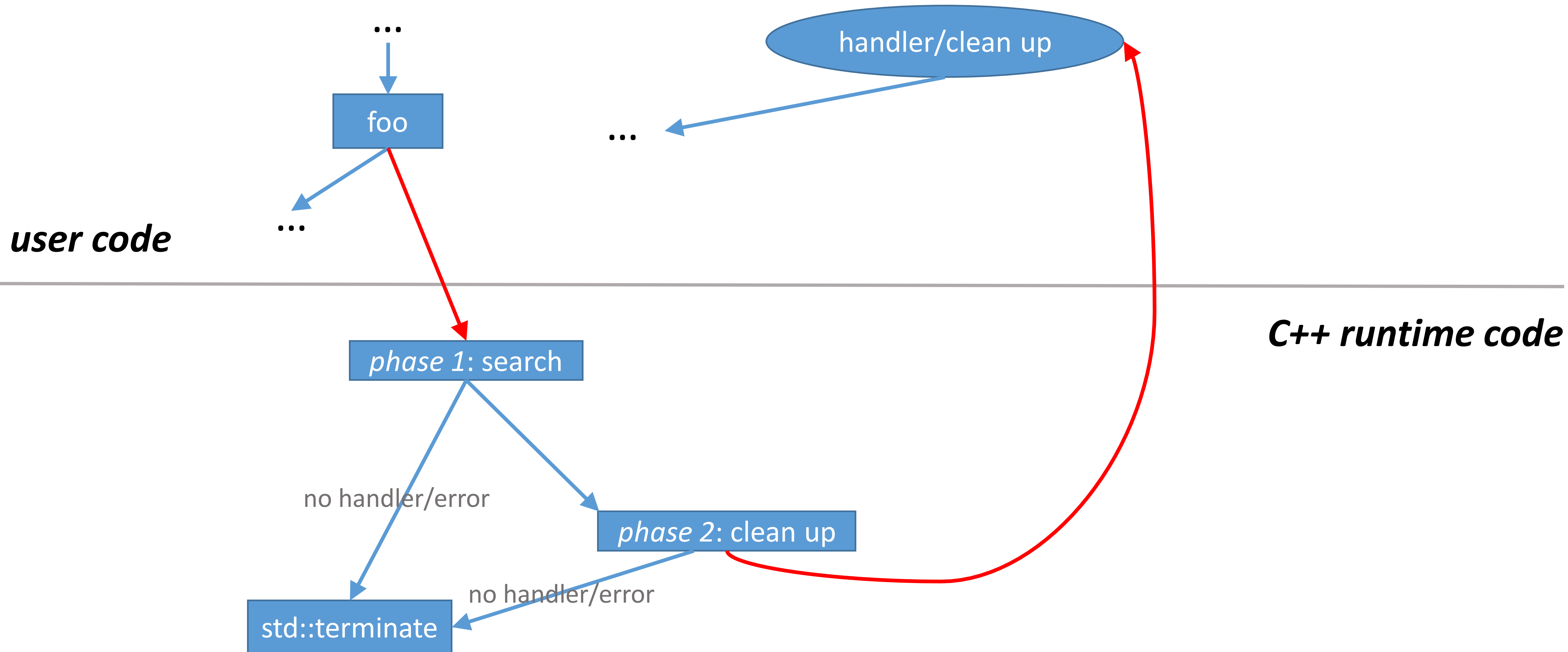
Что происходит при бросании исключения?



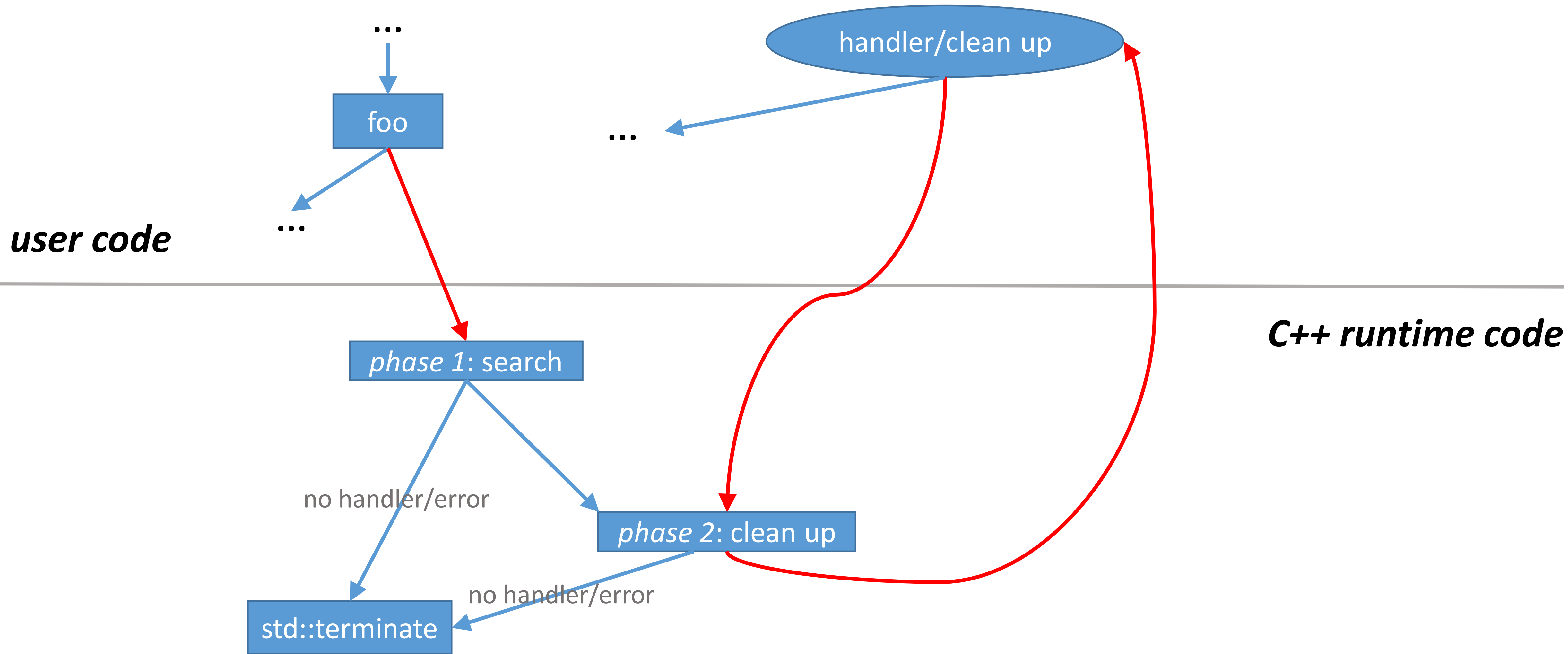
Что происходит при бросании исключения?



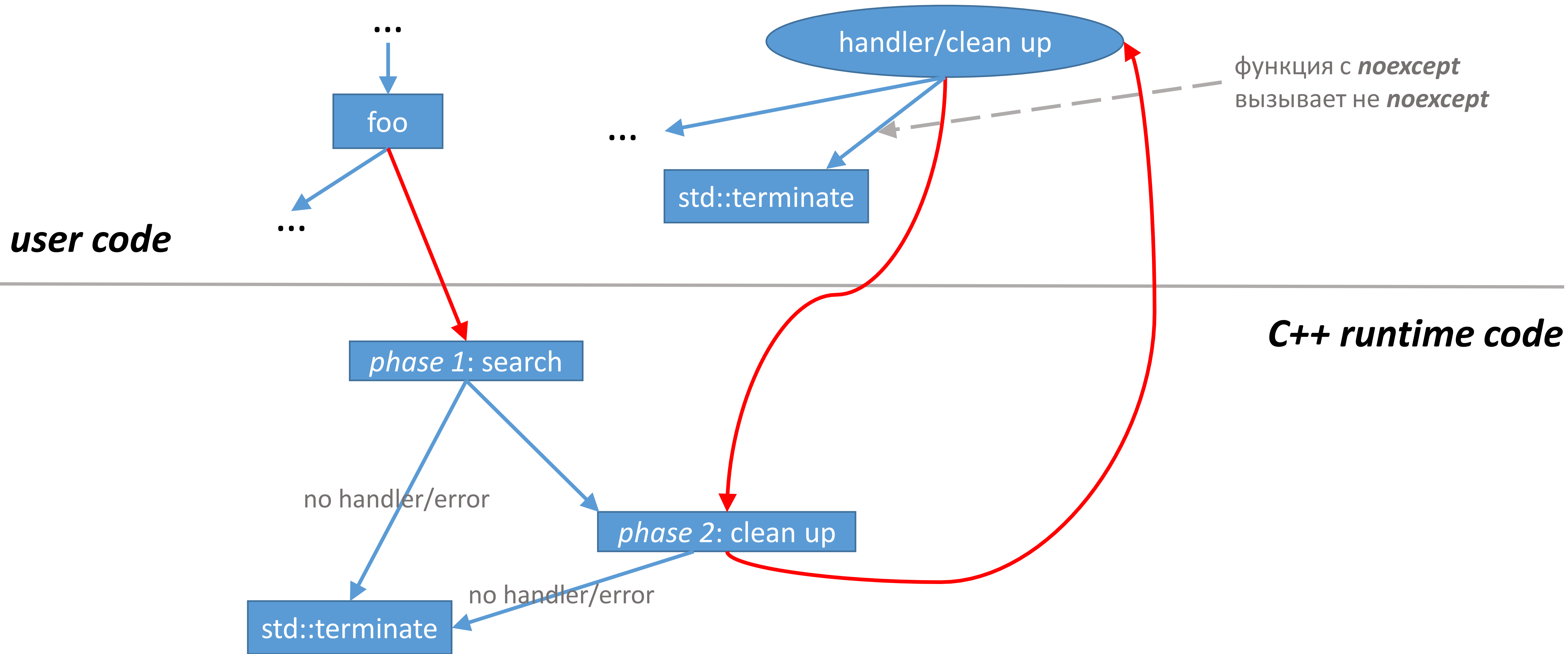
Что происходит при бросании исключения?



Что происходит при бросании исключения?



Что происходит при бросании исключения?



Поддержка исключений в LLVM

Введение в LLVM IR

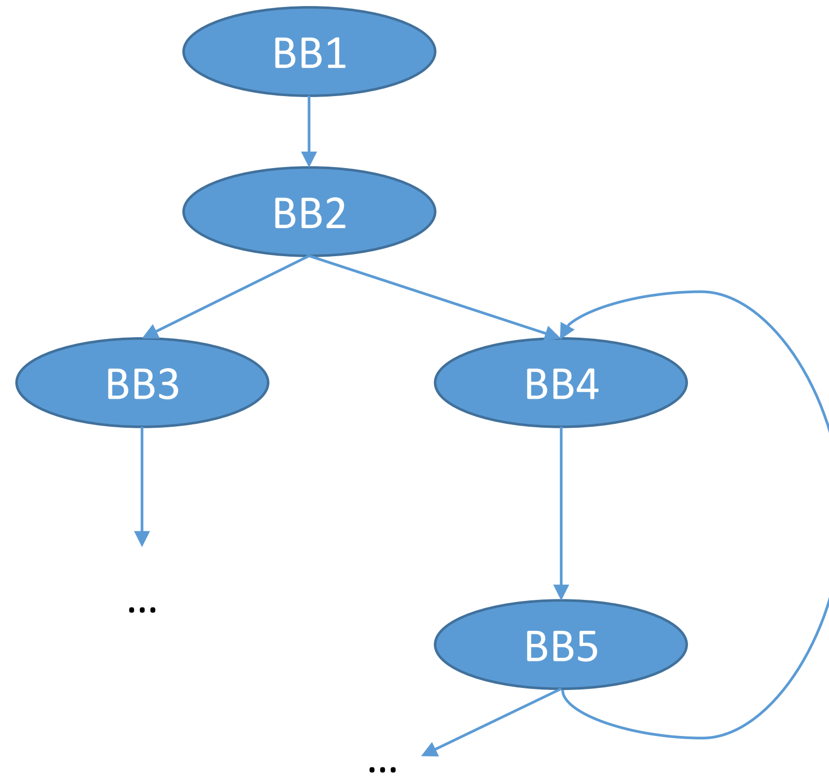
- IR (Intermediate Representation) - структура данных или язык, используемые внутри компилятора для отображения исходного языка
- Будем использовать урезанный вариант LLVM IR

C++ RUSSIA

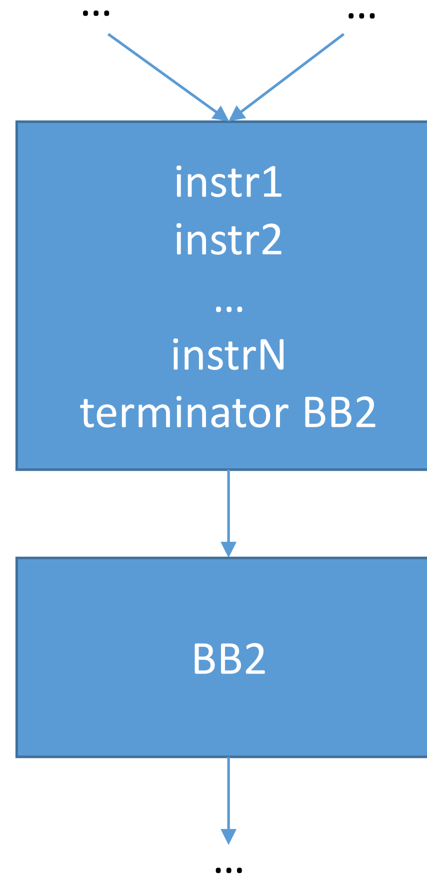
Пример инструкций на LLVM IR псевдокоде

```
%val                ; обозначение локальной переменной или метки
alloca  type        ; выделить память для объекта типа type
call   func_name    ; вызов функции с именем func_name
ret                 ; инструкция возврата из функции
```

Граф потока управления (Control Flow Graph – CFG)



Базовый блок (Basic Block)



Поддержка исключений в LLVM: **invoke**

- вызов функции с неявным переходом на участок кода, если бросили исключение

```
invoke foo()
```

C++ RUSSIA

Поддержка исключений в LLVM: **invoke**

- вызов функции с неявным переходом на участок кода, если бросили исключение

```
invoke foo() to label %1
```

Поддержка исключений в LLVM: **invoke**

- вызов функции с неявным переходом на участок кода, если бросили исключение

```
invoke foo() to label %1  unwind label %2
```

Поддержка исключений в LLVM: **invoke**, **landing pad**

- вызов функции с неявным переходом на участок кода, если бросили исключение

  invoke foo() to label %1 unwind label %2

- участок кода, ответственный за обработку исключения

  landingpad

C++ RUSSIA

Поддержка исключений в LLVM: **invoke**, **landing pad**, **resume**

- вызов функции с неявным переходом на участок кода, если бросили исключение

    invoke foo() to label %1 unwind label %2

- участок кода, ответственный за обработку исключения

    landingpad

- инструкция, продолжающая раскрутку стека

    resume

Влияние исключений на компиляторные оптимизации

Накладные расходы (с точки зрения оптимизатора)

- увеличение размера кода функции за счет создания дополнительного кода для *cleanup*

- усложнение потока управления, появляющегося в результате наличия инструкции *invoke*

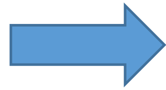
Накладные расходы
Как побороть?
MP

PruneEH

- преобразует инструкции **invoke** в **call**
- ставит признак **nounwind** для функций, которые не бросают исключения

PruneEH

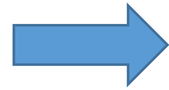
```
void extF() noexcept;
struct S {
    ~S() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    S s;
    bar();
}
```



```
void bar() {
  call extF()
  ret
}
```

PruneEH

```
void extF() noexcept;
struct S {
    ~S() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    S s;
    bar();
}
```



```
void bar() {
  call extF()
  ret
}

void foo() {
   %1 = alloca S
   invoke bar() to label %4  unwind label %5
   …
}
```

PruneEH

```
void extF() noexcept;
struct S {
    ~S() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    S s;
    bar();
}
```

```
void bar() {
  call extF()
  ret
}

void foo() {
   %1 = alloca S
   invoke bar() to label %4  unwind label %5

; <label>:4:
   call ~S(%1)
   ret

; <label>:5:
   landingpad   ; cleanup
   call ~S(%1)
   resume
}
```

PruneEH

```
void foo() {
  %1 = alloca S
  invoke bar() to label %4  unwind label %5

; <label>:4:
  call ~S(%1)
  ret

; <label>:5:
  landingpad   ; cleanup
  call ~S(%1)
  resume
}
```

```
void foo() {
  %1 = alloca S
  call bar()
  call ~S(%1)
  ret
}
```

Остальные оптимизации

- Simplify the CFG
- Global Variable Optimizer
- Instruction combining

Накладные расходы

А где побороть не удается?

Inline

```
void foo() {
  // foo actions
  bar();
}
void bar() {
  // bar actions
}
```

```
void foo() {
  // foo actions
  // bar actions
}
```

Inline: эвристики

- llvm/Analysis/InlineCost.h
- llvm/lib/Analysis/InlineCost.cpp

```
CallAnalyzer::analyzeBlock(…)  {
  …
  addCost(…); // for each instruction add cost
  …
}
```

Inline: эвристики

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    extF();
}
```

```
void bar() {
  call extF()
  ret
}
```

Inline: эвристики

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    MyStruct obj;
    bar();
}
```

```
void bar() {
  call extF()
  ret
}
void foo() {
  %0 = alloca MyStruct
  invoke bar() to label %4  unwind label %5
  …
}
```

Inline: эвристики

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    MyStruct obj;
    bar();
}
```

```
void bar() {
  call extF()
  ret
}
void foo() {
  %0 = alloca MyStruct
  invoke bar() to label %4  unwind label %5
; <label>:4:
  call ~MyStruct(%0)
  ret
; <label>:5:
  landingpad ; cleanup
   call ~MyStruct(%0)
   resume
}
```

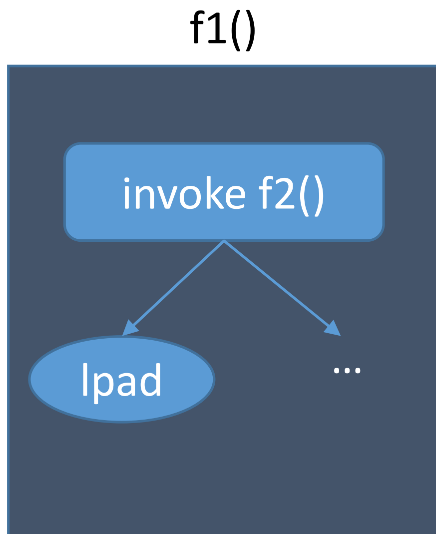
Inline: эвристики

```
void foo() {
  MyStruct obj1;
  …
  bar();
  …
  if (…) {
    MyStruct obj2;
    bar();
    …
  else {
    MyStruct obj3;
    bar();
  …
  bar();
}
```

Inline: invoke

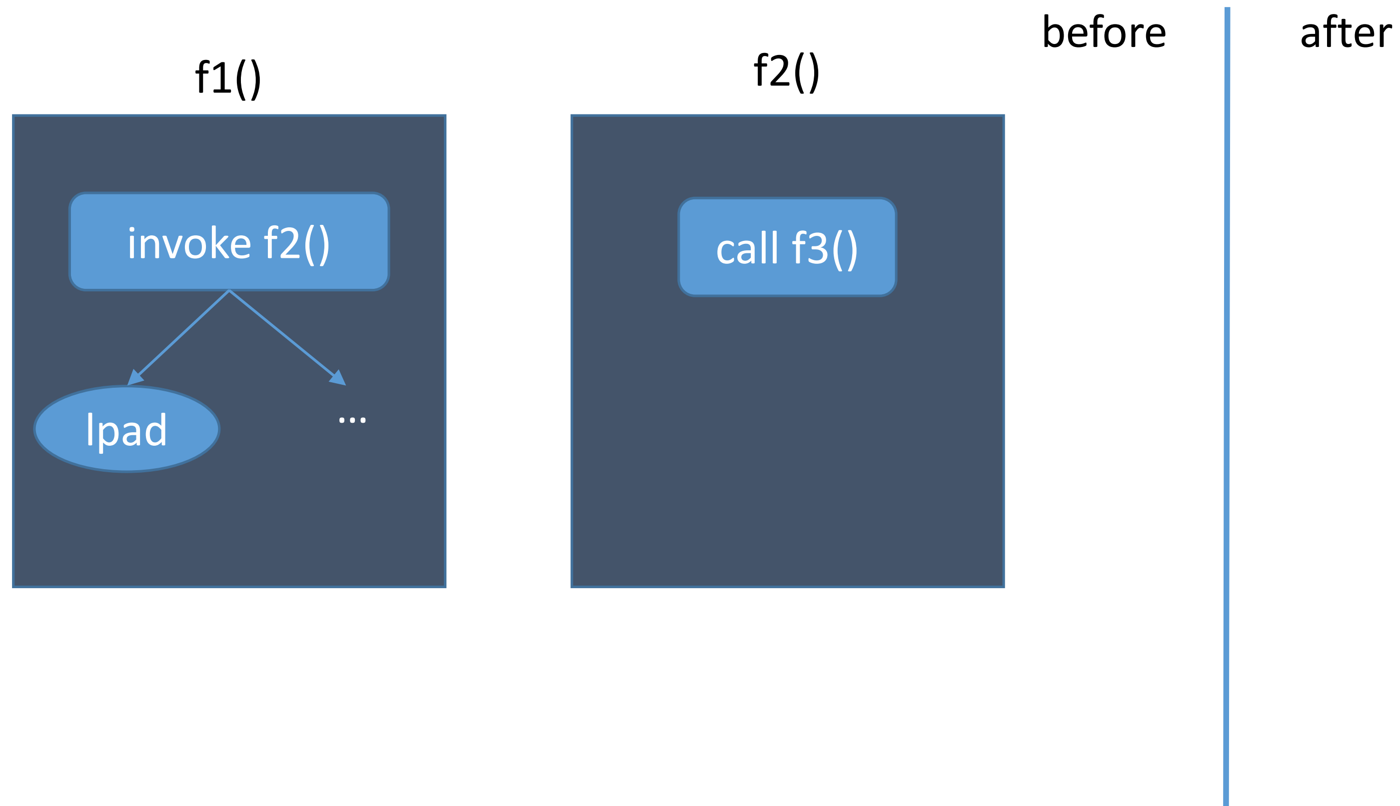
- все инструкции **call** без **noexcept** → **invoke**
  - поток управления от новых инструкций **invoke** → **landing pad** проинлайненной инструкции **invoke**

before | after



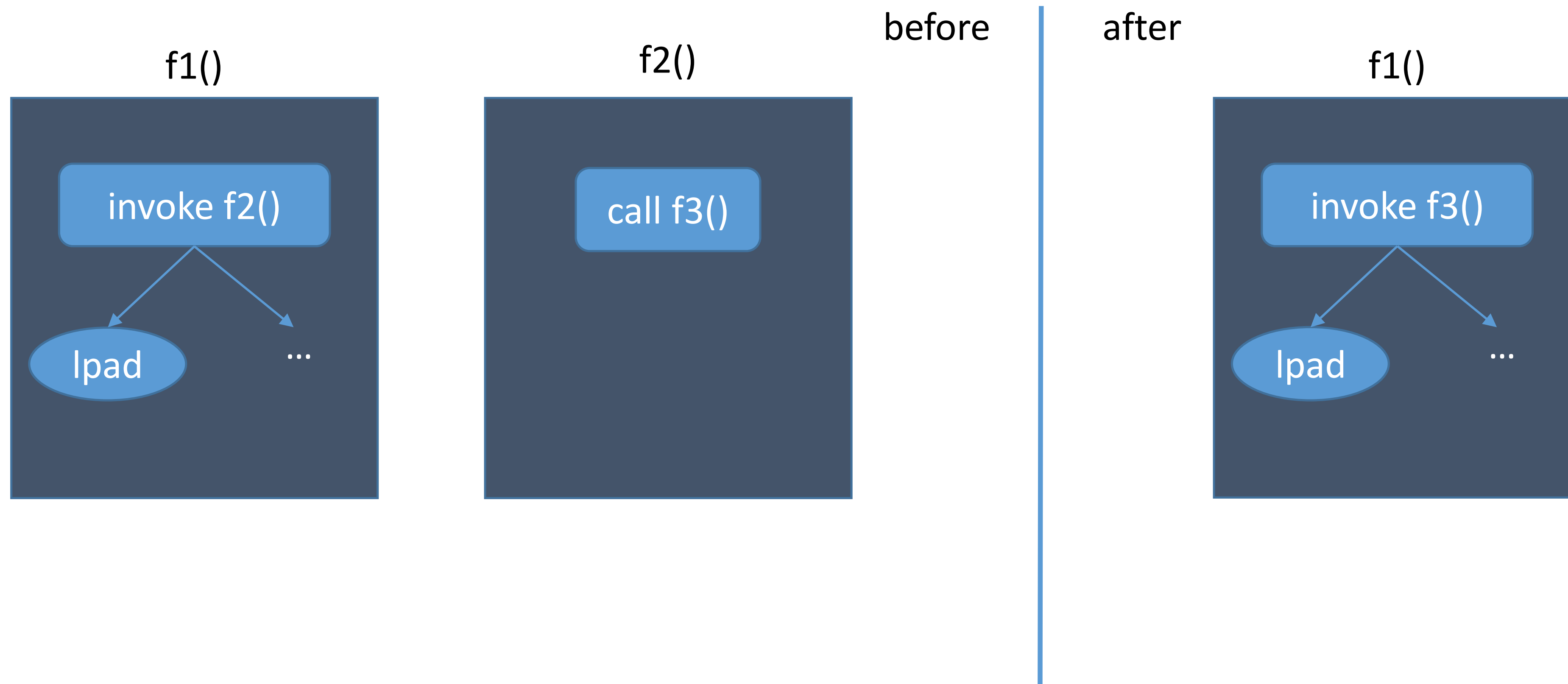
Inline: invoke

- все инструкции **call** без **noexcept** → **invoke**
  - поток управления от новых инструкций **invoke** → **landing pad** проинлайненной инструкции **invoke**



Inline: invoke

- все инструкции **call** без **noexcept** → **invoke**
  - поток управления от новых инструкций **invoke** → **landing pad** проинлайненной инструкции **invoke**



C++ RUSSIA

Inline: invoke

- все инструкции **call** без **noexcept** → **invoke**
  - поток управления от новых инструкций **invoke** → **landing pad** проинлайненной инструкции **invoke**

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    MyStruct obj;
    bar();
}
```

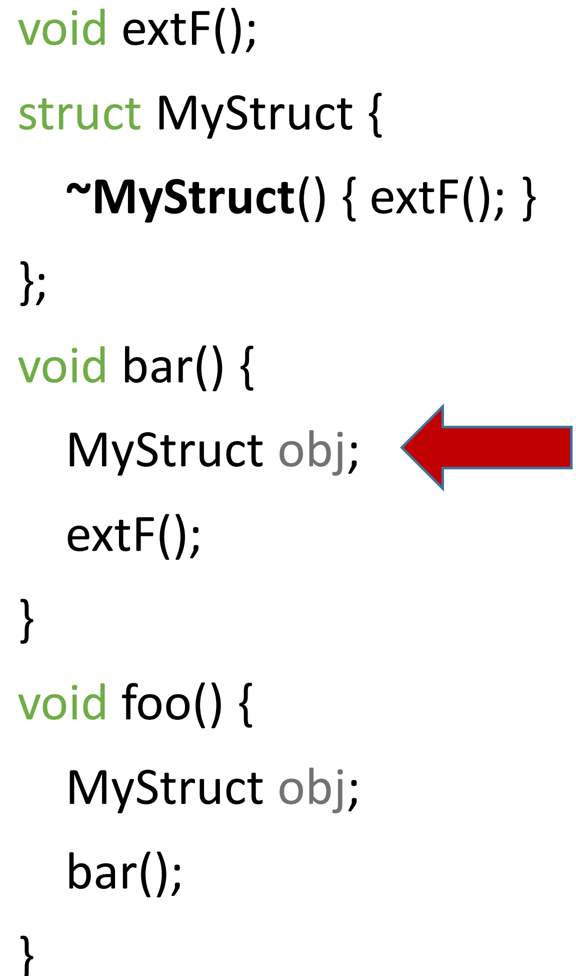
```
void foo() {
  %0 = alloca MyStruct
  invoke extF() to label %4  unwind label %6
; <label>:4:
   call ~MyStruct(%0)
   ret
; <label>:6:
   landingpad
   call ~MyStruct(%0)
   resume
}
```

Inline: resume

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    extF();
}
void foo() {
    MyStruct obj;
    bar();
}
```

```
void bar() {
  call extF()
  ret
}
void foo() {
  %0 = alloca MyStruct
  invoke bar() to label %4  unwind label %5
; <label>:4:
  call ~MyStruct(%0)
  ret
; <label>:5:
  landingpad ; cleanup
   call ~MyStruct(%0)
   resume
}
```

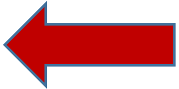
Inline: resume



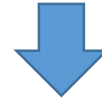
```
void extF();
struct MyStruct {
   ~MyStruct() { extF(); }
};
void bar() {
   MyStruct obj;
   extF();
}
void foo() {
   MyStruct obj;
   bar();
}
```

Inline: resume

```
void extF();
struct MyStruct {
    ~MyStruct() { extF(); }
};
void bar() {
    MyStruct obj;
    extF();
}
void foo() {
    MyStruct obj;
    bar();
}
```



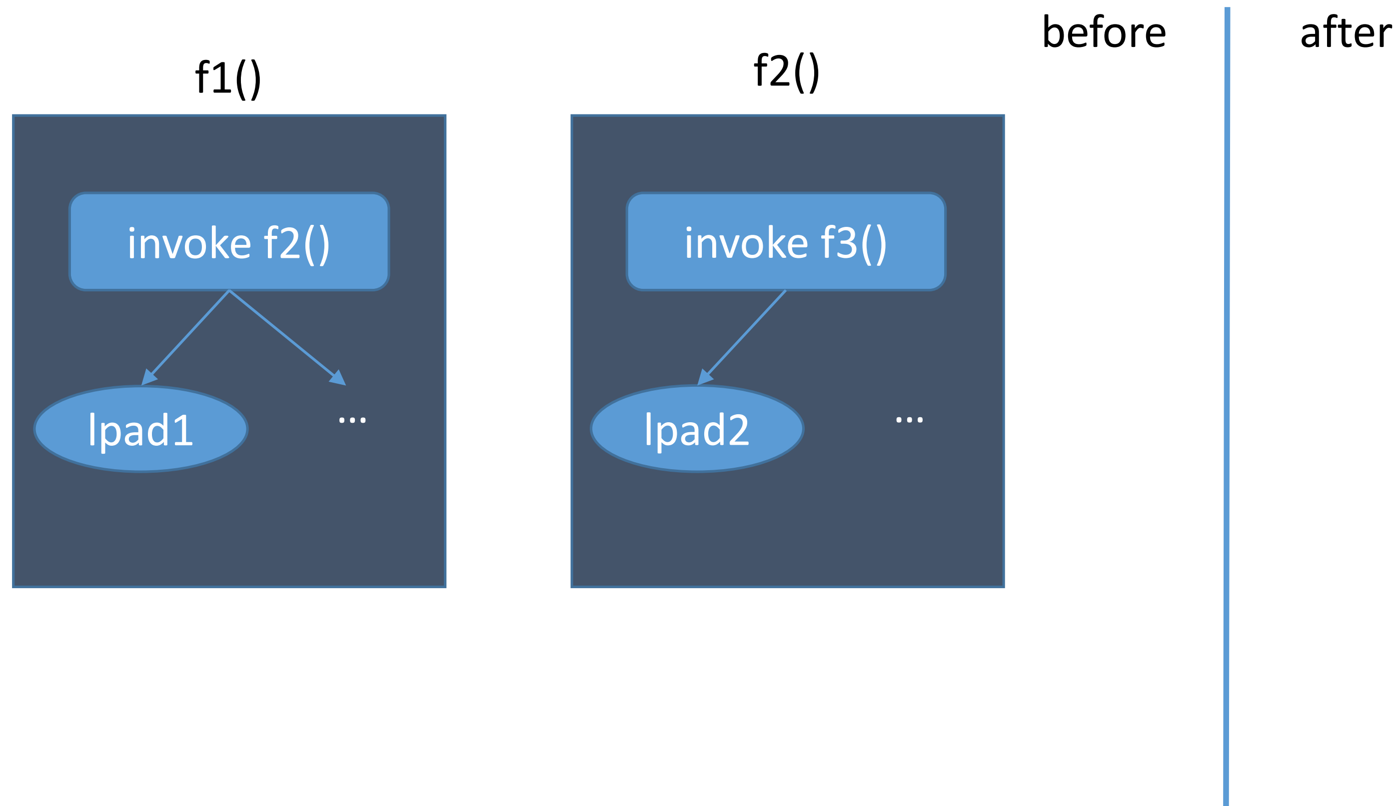
```
void bar() {
  call extF()
  ret
}
```



```
void bar() {
    %0 = alloca MyStruct
    invoke extF() to label %4 unwind label %5
; <label>:4:
    call ~MyStruct(%0)
    ret
; <label>:5:
    landingpad ; cleanup
    call ~MyStruct(%0)
    resume
}
```

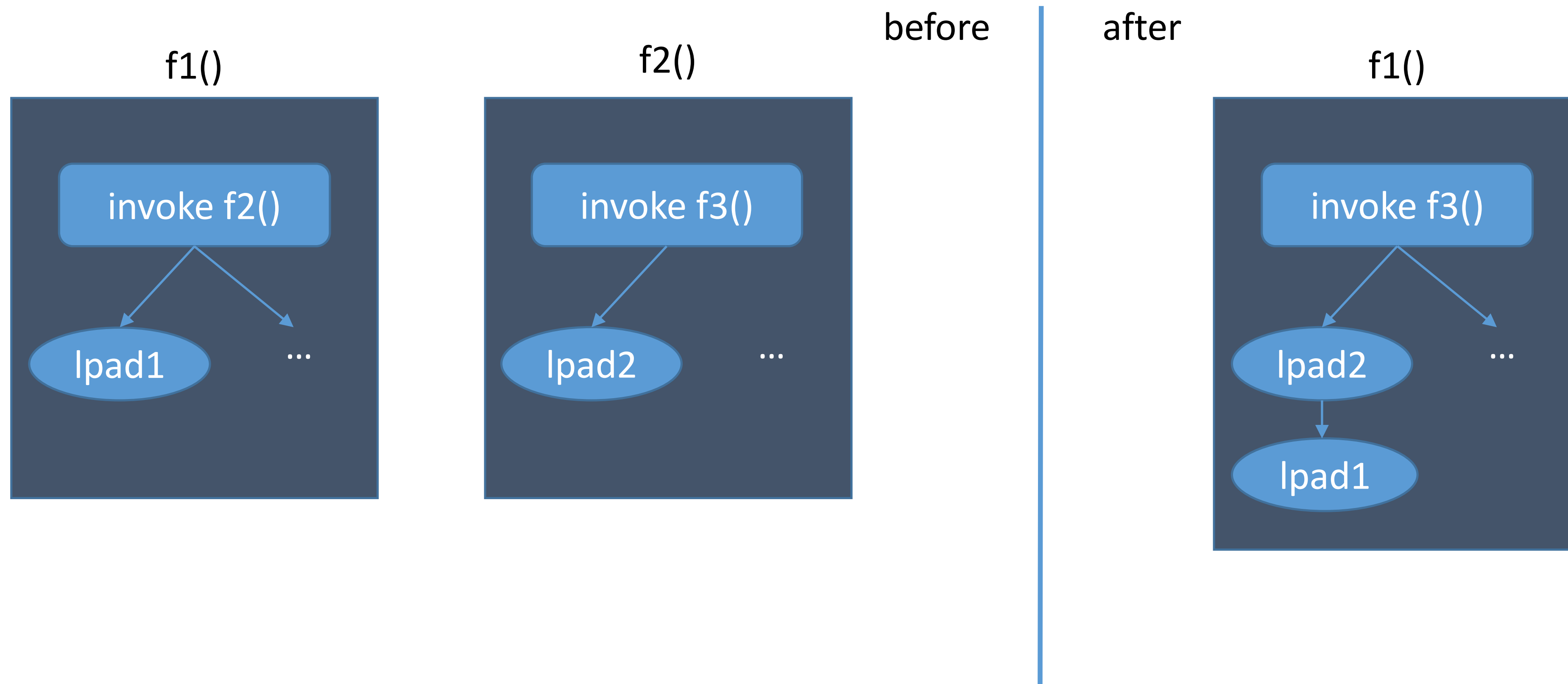
Inline: resume

- Инструкции **resume** подставляемой функции → на **landing pad** проинлайненной инструкции **invoke**



Inline: resume

- Инструкции **resume** подставляемой функции → на **landing pad** проинлайненной инструкции **invoke**



Inline: resume

```
void bar() {
   %0 = alloca MyStruct
   invoke extF() to label %4 unwind label %5
; <label>:4:
   call ~MyStruct(%0)
   ret
; <label>:5:
   landingpad ; cleanup
   call ~MyStruct(%0)
   resume
}
```

```
void foo() {
  %1 = alloca MyStruct
  %2 = alloca MyStruct
  invoke extF() to label %7  unwind label %5
; <label>:5:
  landingpad
  call ~MyStruct(%1)
  call ~MyStruct(%2)
  resume
; <label>:7:
  call ~MyStruct(%1)
  call ~MyStruct(%2)
}
```

Tail Call

```
void foo() {
  bar();
}
void bar() {
  // bar actions
}
```

```
void foo() {
  goto bar_label;
}
void bar() {
bar_label:
  // bar actions
}
```

Tail Call

Не применима к инструкциям **invoke**

Loop Fusion

```
for (int i = 0; i < n; i++)
  a[i] = ...;

for (int j = 0; j < n; j++)
  b[j] = ...;
```

```
for (int i = 0; i < n; i++) {
  a[i] = ...;
  b[i] = ...;
}
```

Loop Fusion

- Если в цикле есть **call**, который может бросать исключение, то цикл не рассматривается как кандидат для оптимизации

LICM (Loop Invariant Code Motion)

```
int y = foo();
int x;
for (int i = 0; i < n; i++) {
  x = y;
  a[i] = x + ...;
}
```

```
int y = foo();
int x = y;
for (int i = 0; i < n; i++) {
  a[i] = x + ...;
}
```

LICM (Loop Invariant Code Motion)

- не работает для инструкций **invoke**
- не работает для инструкций **call**, потенциально бросающих исключения

ADCE (Aggressive Dead Code Elimination)

```
int global;
void f () {
  int i;
  i = 1;
  global = 1;
  global = 2;
}
```

```
int global;
void f () {
  global = 2;
}
```

ADCE (Aggressive Dead Code Elimination)

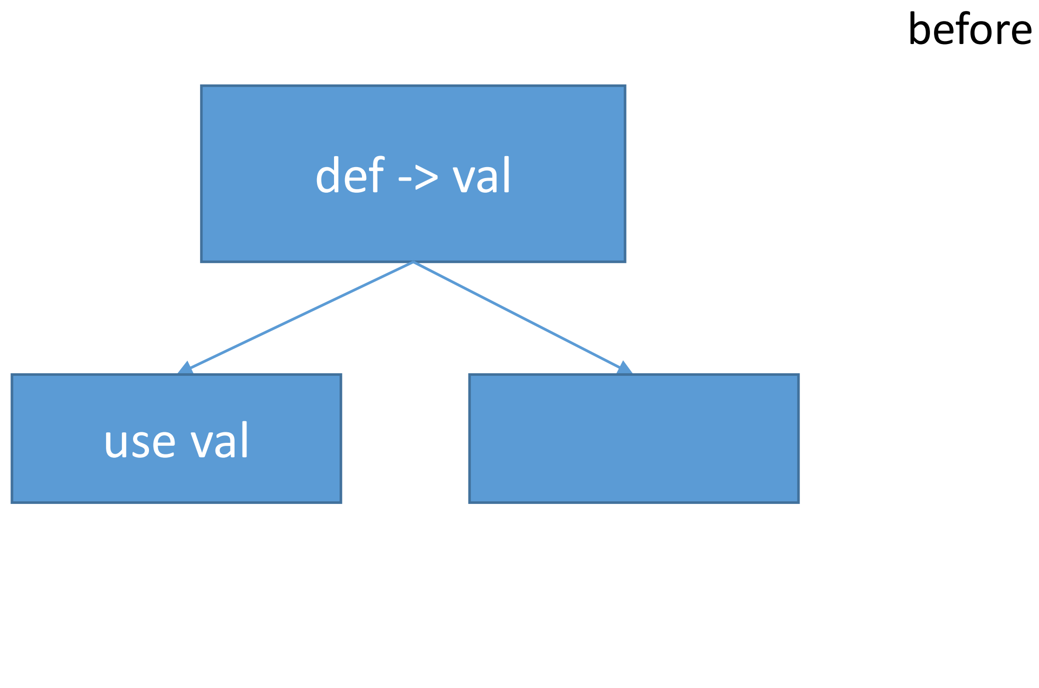
```
bool AggressiveDeadCodeElimination::isAlwaysLive(Instruction &Inst) {
 if (Inst.isEHPad() || ...) {   // landing pad
      ...
      return true;
  }
  if (Inst.isTerminator() == false)   // invoke is terminator instruction
     return false;

  if (isa<BranchInst>(Inst) || isa<SwitchInst>(Inst))   // invoke is not included here
     return false;

  return true;
}
```

Sinking

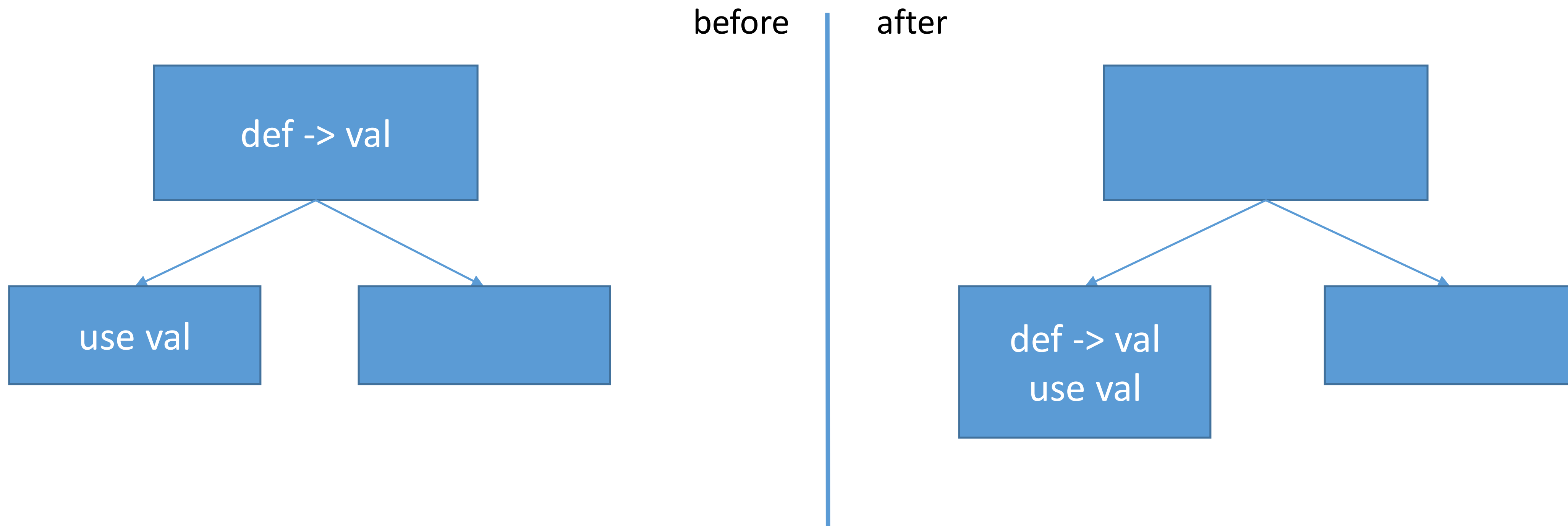
- Переносит инструкции в базовые блоки преемники, чтобы убрать лишнее исполнение инструкций



before

Sinking

- Переносит инструкции в базовые блоки преемники, чтобы убрать лишнее исполнение инструкций



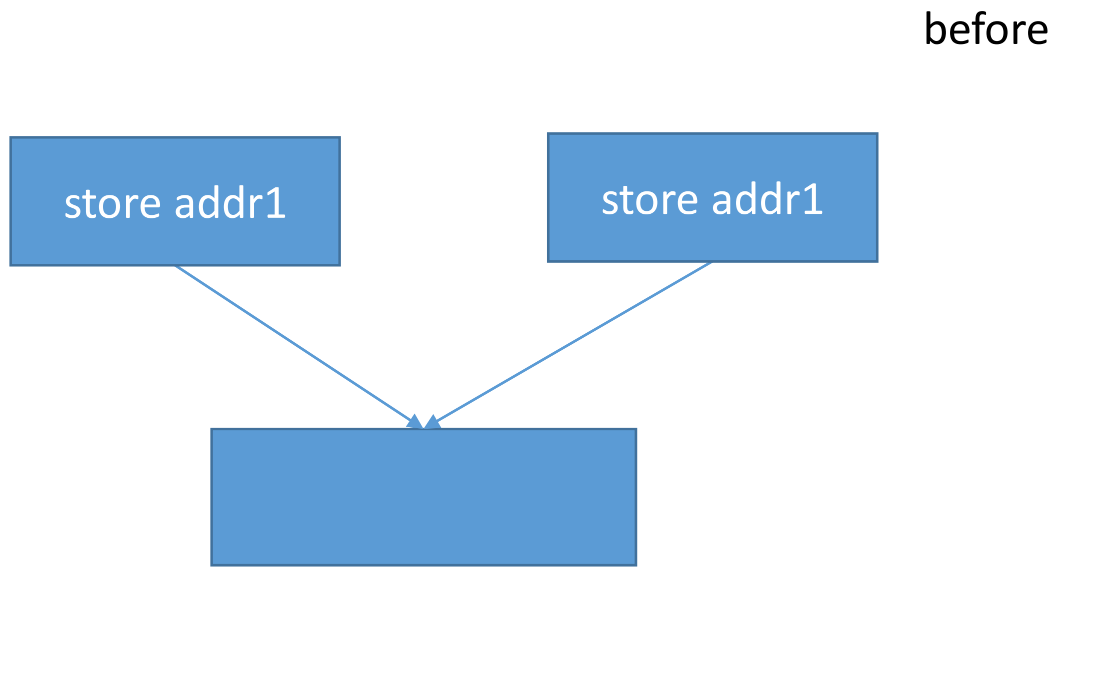
Sinking

```
bool isSafeToMove(Instruction *Inst, …) {
  if (… || Inst->mayThrow())
     return false;
 …
}
```

Merged Load/Store Motion

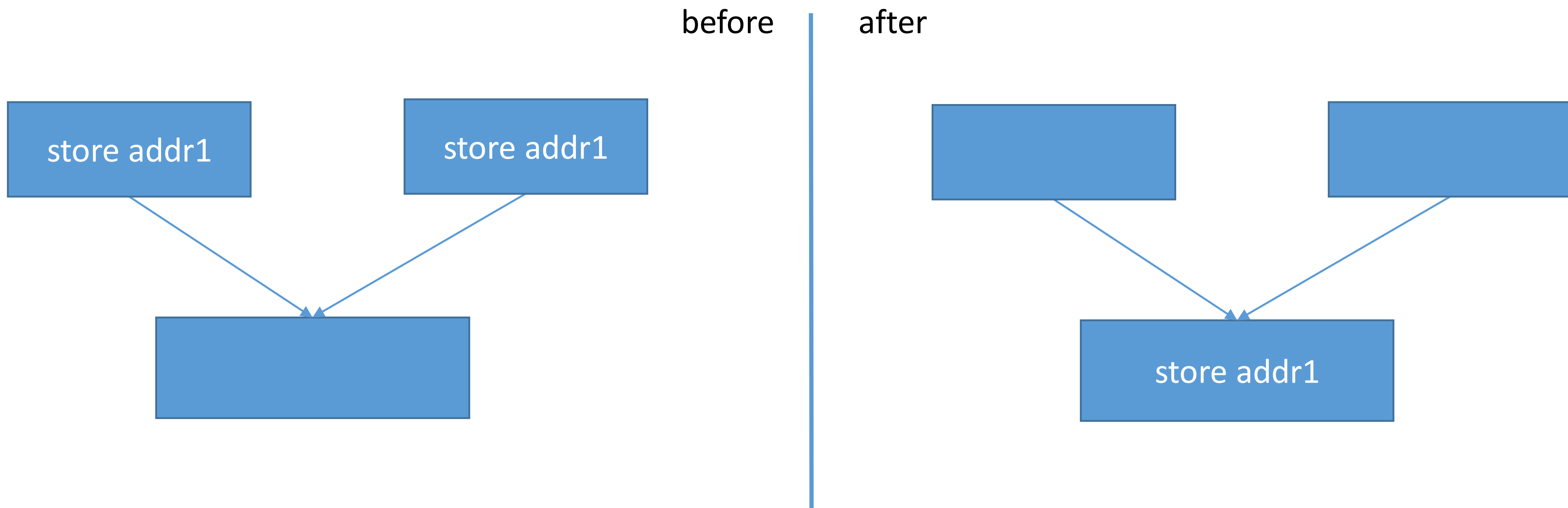
- Объединяет инструкции записи в память по одному адресу, для уменьшения статического размера кода

before



Merged Load/Store Motion

- Объединяет инструкции записи в память по одному адресу, для уменьшения статического размера кода



Merged Load/Store Motion

```
/// True when instruction is a sink barrier for a store
bool MergedLoadStoreMotion::isStoreSinkBarrierInRange(const Instruction &Start,
                                                      const Instruction &End) {

  for (const Instruction &Inst : make_range(Start.getIterator(), End.getIterator())) {
    if (Inst.mayThrow())
      return true;
  }
  …
}
```

Остальные оптимизации

- **mayHaveSideEffects**
- **mayThrow**
- **doesNotThrow**

Выводы

- zero-cost далеко не всегда нулевой, даже если исключение не бросается

C++ RUSSIA

Выводы

- zero-cost далеко не всегда нулевой, даже если исключение не бросается:
  - современные компиляторы имеют специальные оптимизации для обработки исключений

Выводы

- zero-cost далеко не всегда нулевой, даже если исключение не бросается:
  - современные компиляторы имеют специальные оптимизации для обработки исключений
- если вы разрабатываете библиотеку, стоит подумать об отказе от исключений

Выводы

- zero-cost далеко не всегда нулевой, даже если исключение не бросается:
  - современные компиляторы имеют специальные оптимизации для обработки исключений
- если вы разрабатываете библиотеку, стоит подумать об отказе от исключений
- **<u>noexcept везде, где можно</u>**

Выводы

- zero-cost далеко не всегда нулевой, даже если исключение не бросается:
  - современные компиляторы имеют специальные оптимизации для обработки исключений
- если вы разрабатываете библиотеку, стоит подумать об отказе от исключений
- **noexcept везде, где можно:**
  - аккуратно, т.к. это часть интерфейса

Q & A

Роман Русяев

Samsung R&D

compilers developer

rusyaev.rm@gmail.com

APPENDIX

setjmp/longjmp (sjlj)

- **setjmp** – запомнить, куда нужно прыгнуть
- **longjmp** – эмулирует throw

Очень непроизводительно:

- много дополнительных структур данных
- много дополнительного выполняемого кода вне зависимости от факта бросания исключения
- Нарушение главного принципа C++ - *“you only pay for what you use”*

Библиотека поддержки исключений

- LLVM:
  - libcxxabi
  - libunwind
- GCC:
  - libsupc++
  - libgcc

Библиотека поддержки исключений: throw

- _cxa_allocate_exception
- __cxa_throw:
  - _Unwind_RaiseException
  - …

Библиотека поддержки исключений: **catch**

- __cxa_begin_catch
- __cxa_end_catch

Что такое LLVM

- инфраструктура для разработки компиляторов
- компилятор и инструменты, основанные на LLVM IR

http://llvm.org

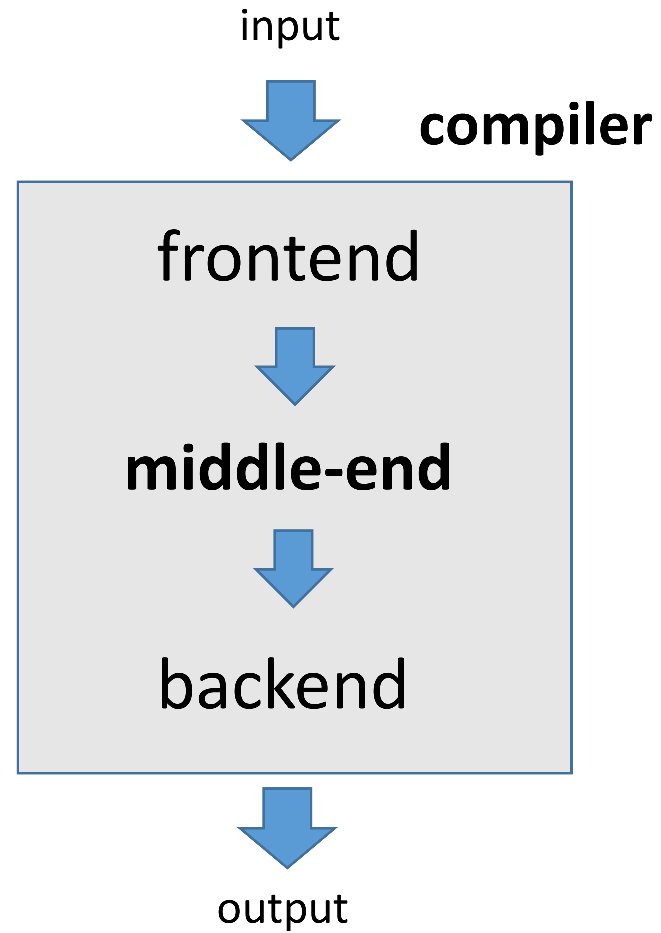
Введение в LLVM

- LLVM IR
- множество проектов, использующих инфраструктуру LLVM
- компилятор на основе LLVM – clang
- LLVM библиотеки (support library, command line library, …)
- алгоритмы над LLVM IR (трансформации, оптимизации, …)
- инструменты для работы с LLVM IR
- …

Введение в LLVM IR: основные концепции

- Модули
- Функции
- Глобальные переменные
- Метаданные
- …

Основы работы компилятора



Middle-end

- Работает с IR
- Выполняет:
  - анализы
  - трансформации
  - оптимизации
- Проход компилятора (*pass*) – выполнение над IR анализа, трансформации или оптимизации