Extending Lifetime Safety

Verification of [[clang::noescape]] annotation

Abhinav Pradeep

abhinav.pradeep@gmail.com

EuroLLVM Developers' Meeting

What is [[clang::noescape]]

Definition

[[clang::noescape]] placed on a function parameter of a pointer type is used to inform the compiler that the pointer cannot escape: that is, no reference to the object the pointer points to that is derived from the parameter value will survive after the function returns.

Issue

Users are responsible for making sure parameters annotated with [[clang::noescape]] do not actually escape.

Motivation

```
using namespace std;

string_view findResult(
    char *TrueResult [[clang::noescape]],
    char *FalseResult [[clang::noescape]]) {
  bool Condition;
  ...
  string_view Resolved = Condition ? string_view(TrueResult) :
                                     string_view(FalseResult);
  return Resolved;
}
```

Verification of this would be the developers responsibility!

Key abstractions

- **Origin**: Represents a unit of storage
- **OriginList**: A linked-list of Origins, reflecting the indirection possible through pointer types. It is inferred from type information:
  - `int p` → origin for `p`
  - `int* p` → origin for `p`, and origin for `*p`
  - `int** p` → origin for `p`, origin for `*p`, and origin for `**p`
- **Loan**: Represents the borrow of a storage.
  - `int i; int* p = &i`: Here storage location `p` will hold a loan to the storage of `i`

Intuition

Origins let us express the *storage locations we care about*, OriginLists let us *reason about their indirection*, and Loans let us express *the act of borrowing*.

Analysis

- **Loan-propagation analysis:**
  - This is a **forward-may** flow-sensitive analysis
  - Tracks the set of loans that each origin **may** hold across all control paths
  - Like a "points-to" analysis

- **Liveness analysis:**
  - This is a **backward** flow-sensitive analysis
  - Tracks which origins **will** or **may** be read at a later point in the program

[[clang::noescape]] checking

Verification of [[clang::noescape]] is a query over the results of the previous analyses:

- We define points in the program where an **origin** can **escape** the current scope. For example, but not exhaustively:
  - return statements
  - escape to a data member
  - assignments to global/static storage
  - argument to a function call
- At these points, we check if the **escaping origin** holds a **loan** corresponding to the **noescape-annotated formal parameter**. If so, we have a violation of noescape!

Escape via return

```
using namespace std;

string_view findResult(
    char *TrueResult [[clang::noescape]],
    char *FalseResult [[clang::noescape]]) {
  bool Condition;
  ...
  string_view Resolved = Condition ? string_view(TrueResult) :
                                     string_view(FalseResult);
  return Resolved;
}
```

Escape via return

```
test-programs/program.cpp:7:5: warning: parameter is marked [[clang::noescape]]
      but escapes [-Wlifetime-safety-noescape]
    7 |     char *FalseResult [[clang::noescape]]) {
      |     ^~~~~~~~~~~~~~~~
test-programs/program.cpp:11:10: note: param returned here
   11 |   return Resolved;
      |          ^~~~~~~~
test-programs/program.cpp:6:5: warning: parameter is marked [[clang::noescape]]
      but escapes [-Wlifetime-safety-noescape]
    6 |     char *TrueResult [[clang::noescape]],
      |     ^~~~~~~~~~~~~~~
test-programs/program.cpp:11:10: note: param returned here
   11 |   return Resolved;
      |          ^~~~~~~~
2 warnings generated.
```

Escape to enclosing state

```
enum Mode { Erase, Consume };

struct State {
  Buffer *Configuration;
  void configure(Buffer *Data [[clang::noescape]], Mode Choice) {
    switch (Choice) {
      case Erase:
        this->Configuration = nullptr;
        break;
      case Consume:
        this->Configuration = Data;
        break;
    }
  }
};
```

Escape to enclosing state

```
test-programs/program.cpp:13:18: warning: parameter is marked [[clang::noescape]]
      but escapes [-Wlifetime-safety-noescape]
   13 |   void configure(Buffer *Data [[clang::noescape]], Mode Choice) {
      |                  ^~~~~~~~~~~~
test-programs/program.cpp:11:11: note: escapes to this field
   11 |   Buffer *Configuration;
      |           ^~~~~~~~~~~~~
1 warning generated.
```

Escape through global or static storage

```
struct GlobalData {
  static Buffer *Data;
};

void someComputation(Buffer *Data [[clang::noescape]]) {
  ...
  GlobalData::Data = Data;
  ...
}
```

Escape through global or static storage

```
test-programs/program.cpp:11:22: warning: parameter is marked [[clang::noescape]]
      but escapes [-Wlifetime-safety-noescape]
   11 | void someComputatuon(Buffer *Data [[clang::noescape]]) {
      |                      ^~~~~~~~~~~~
test-programs/program.cpp:8:18: note: escapes to this static storage
    8 |   static Buffer *Data;
      |                  ^~~~
1 warning generated.
```

Summary and what is remaining

What is already done:

- Escape through return
- Escape to a field
- Escape to global or static storage
- Escape through function call (nearly landed)

Next steps:

- Escape through lambda closures
- Escape through side-effects of a function call
- and other complicated ways of escaping...

Thanks for attending!

We are looking for contributions

With thanks to:

Utkarsh Saxena — usx@google.com

Gábor Horváth — xazax.hun@gmail.com