Structured Bindings and How to Analyze Them

Domján Dániel

Structured binding declaration

```
[[attr]] cv* auto &|&& [id1, id2, ..., idn] = initializer;
```

*can also be static or thread_local since C++20

Cases of structured bindings

Case 1: binding an array

```
void binding_an_array() {
  int arr[3] = {1, 2, 3};

  auto [x, y, z] = arr;

  int a = x;
}
```

```
void binding_an_array() {
  int arr[3] = {1, 2, 3};

  int tmp_arr[3] = {
    arr[0], arr[1], arr[2]
  };

  #define x tmp_arr[0]
  #define y tmp_arr[1]
  #define z tmp_arr[2]

  int a = x;
}
```

Case 1: binding an array

```
void binding_an_array() {
  int arr[3] = {1, 2, 3};

  auto &[x, y, z] = arr;

  int a = x;
}
```

```
void binding_an_array() {
  int arr[3] = {1, 2, 3};

  int(&tmp_arr)[3] = arr;

  #define x tmp_arr[0]
  #define y tmp_arr[1]
  #define z tmp_arr[2]

  int a = x;
}
```

Case 2: binding to data members

```
struct S {int a; int b;};

void binding_to_data_m() {
  S s;

  auto [x, y] = s;

  int a = x;
}
```

```
struct S {int a; int b;};

void binding_to_data_m() {
  S s;

  S tmp_s = s;

  #define x tmp_s.a
  #define y tmp_s.b

  int a = x;
}
```

Case 3: binding a tuple-like type

```
void binding_a_tuple() {
  std::pair<int, float> p{1, 2.0F};

  auto [x, y] = p;

  int a = x;
}
```

```
void binding_a_tuple() {
  std::pair<int, float> p{1, 2.0F};

  std::pair<int, float> tmp_p = p;

  std::tuple_element<
    0, std::pair<int, float>
  >::type x = std::get<0>(tmp_p);

  std::tuple_element<
    1, std::pair<int, float>
  >::type y = std::get<1>(tmp_p);

  #define x x
  #define y y

  int a = x;
}
```

Case 3: binding a tuple-like type

```
void binding_a_tuple() {
  std::pair<int, float> p{1, 2.0F};

  auto &[x, y] = p;

  int a = x;
}
```

```
void binding_a_tuple() {
  std::pair<int, float> p{1, 2.0F};

  std::pair<int, float> &tmp_p = p;

  std::tuple_element<
    0, std::pair<int, float>
  >::type &x = std::get<0>(tmp_p);

  std::tuple_element<
    1, std::pair<int, float>
  >::type &y = std::get<1>(tmp_p);

  #define x x
  #define y y

  int a = x;
}
```

The Clang Static Analyzer

The Clang Static Analyzer

```
void toy_example() {
  int x;
  int y = x;
}
```

```
warning: Assigned value is garbage or undefined [core.uninitialized.Assign]
  int y = x;
  ^       ~
note: 'x' declared without an initial value
  int x;
  ^~~~~
note: Assigned value is garbage or undefined
  int y = x;
  ^       ~
```

The Clang Static Analyzer

```
int x;
int y = x;
```

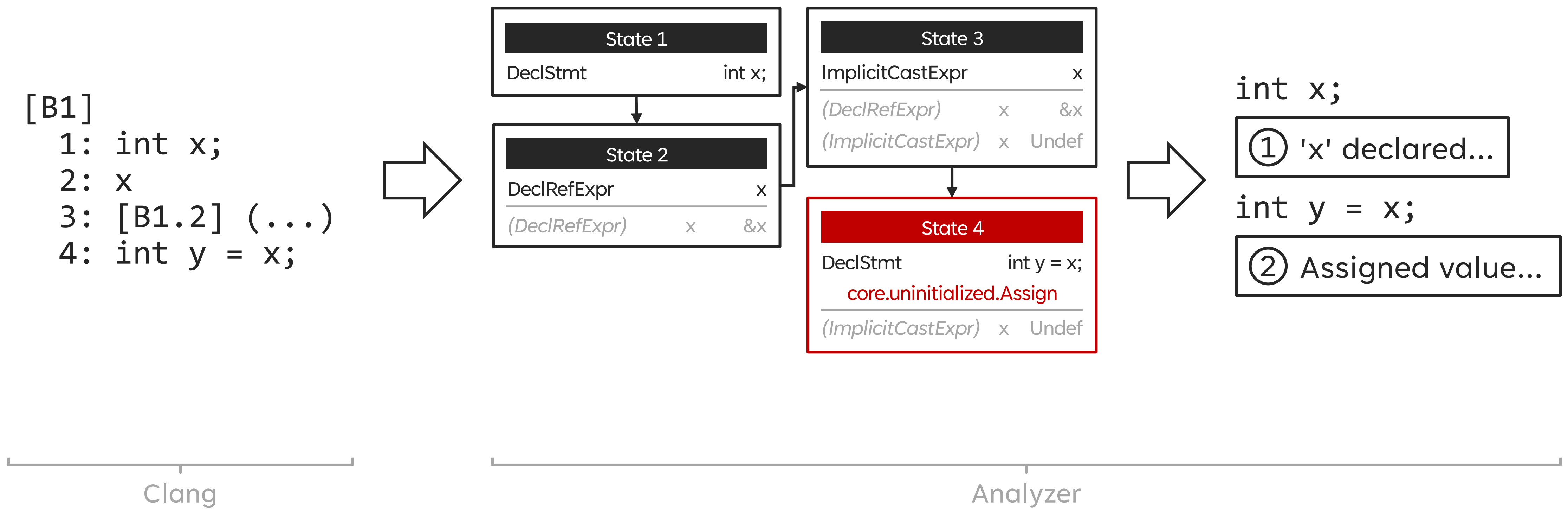
```
-DeclStmt ...
 `-VarDecl ... x 'int'
-DeclStmt ...
 `-VarDecl ... y 'int' ...
   `-ImplicitCastExpr ...
     `-DeclRefExpr ... 'x' ...
```

```
[B1]
  1: int x;
  2: x
  3: [B1.2] (...)
  4: int y = x;
```

User

Clang

The Clang Static Analyzer



```
[B1]
  1: int x;
  2: x
  3: [B1.2] (...)
  4: int y = x;
```

```
int x;
```

① 'x' declared...

```
int y = x;
```

② Assigned value...

Why support structured bindings?

False positives

```
QPair<int, QSharedPointer<int>> foo() {
  return {42, nullptr};
}

int main() {
  auto [x, p] = foo();
  auto p2 = p;
}
```

```
warning: 1st function call argument is an uninitialized value [core.CallAndMessage]
  void deref() noexcept { deref(d); }
                                ^
note: Uninitialized value stored to '.second.d’
  auto p2 = p;
            ^
```

False negatives

```
int main() {
  int arr[2];

  auto [x, y] = arr;

  int a = x;
}
```

```
warning: Value stored to 'a' during its initialization is never read [deadcode.DeadStores]
    int a = x;
        ^   ~
```

How to analyze them

How to analyze them

```
auto [b1, b2] = init
```

$\Rightarrow$

```
DeclStmt ...
`-DecompositionDecl ...
  |-<tmp init>
  |-BindingDecl ... b1 ...
  | `-<bound expession>
  `-BindingDecl ... b2 ...
    `-<bound expession>
```

$\Rightarrow$

```
[B1]
  1: <tmp init>
  2: auto = init;
```

Case 1: binding an array

```
int init[2];

auto [b1, b2] = init;

int x = b1;
```

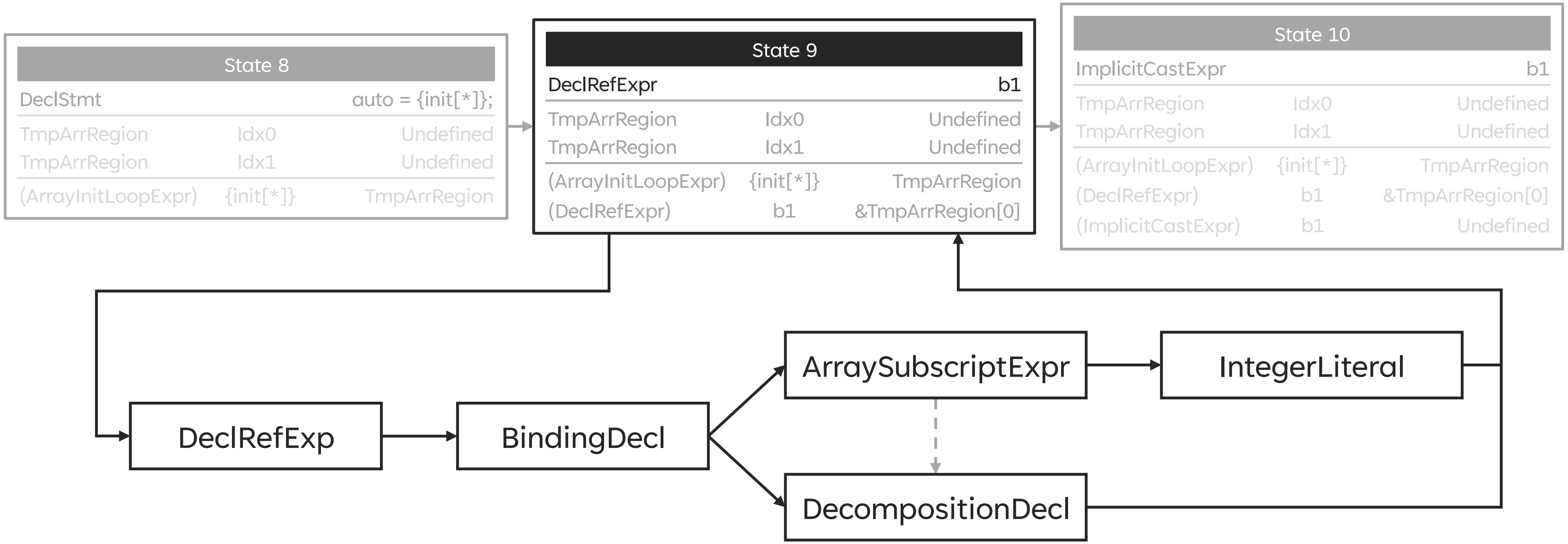
```
-<array declaration>
-DeclStmt ...
 `-DecompositionDecl ...
   |-ArrayInitLoopExpr ...
   | `-...
   |-BindingDecl ... b1 ...
   | `-ArraySubscriptExpr ...
   |   |-ImplicitCastExpr ...
   |   | `-DeclRefExpr ... Decomposition
   |   `-IntegerLiteral ... 0
   `-BindingDecl ... b2 ...
     `-...
-DeclStmt ...
 `-VarDecl ... x ...
   `-ImplicitCastExpr ...
     `-DeclRefExpr ... Binding ... 'b1'
```

Case 1: binding an array

```
-<array declaration>
-DeclStmt ...
 `-DecompositionDecl ...
   |-ArrayInitLoopExpr ...
   | `-...
   |-BindingDecl ... b1 ...
   | `-ArraySubscriptExpr ...
   |   |-ImplicitCastExpr ...
   |   | `-DeclRefExpr ... Decomposition
   |   `-IntegerLiteral ... 0
   `-BindingDecl ... b2 ...
     `-...
-DeclStmt ...
 `-VarDecl ... x ...
   `-ImplicitCastExpr ...
     `-DeclRefExpr ... Binding ... 'b1'
```

```
[B1]
  X: <array declaration>
  X: ArrayInitLoopExpr
  8: auto = {init[*]};
  9: b1
 10: [B1.9] (ImplicitCastExpr, ...)
 11: int x = b1;
```

Case 1: binding an array



State 8

DeclStmt		auto = {init[*]};
TmpArrRegion	Idx0	Undefined
TmpArrRegion	Idx1	Undefined
(ArrayInitLoopExpr)	{init[*]}	TmpArrRegion

State 9

DeclRefExpr		b1
TmpArrRegion	Idx0	Undefined
TmpArrRegion	Idx1	Undefined
(ArrayInitLoopExpr)	{init[*]}	TmpArrRegion
(DeclRefExpr)	b1	&TmpArrRegion[0]

State 10

ImplicitCastExpr		b1
TmpArrRegion	Idx0	Undefined
TmpArrRegion	Idx1	Undefined
(ArrayInitLoopExpr)	{init[*]}	TmpArrRegion
(DeclRefExpr)	b1	&TmpArrRegion[0]
(ImplicitCastExpr)	b1	Undefined

Case 2: binding to data members

```
S init;

auto [b1, b2] = init;

int x = b1;
```

```
-<struct instantiation>
-DeclStmt ...
 `-DecompositionDecl ...
   |-CXXConstructExpr ...
   | `-...
   |-BindingDecl ... b1 ...
   | `-MemberExpr ... F0 ...
   |   `-DeclRefExpr ... Decomposition
   `-BindingDecl ... b2 ...
     `-...
-DeclStmt ...
 `-VarDecl ... x ...
   `-ImplicitCastExpr ...
     `-DeclRefExpr ... Binding ... 'b1'
```

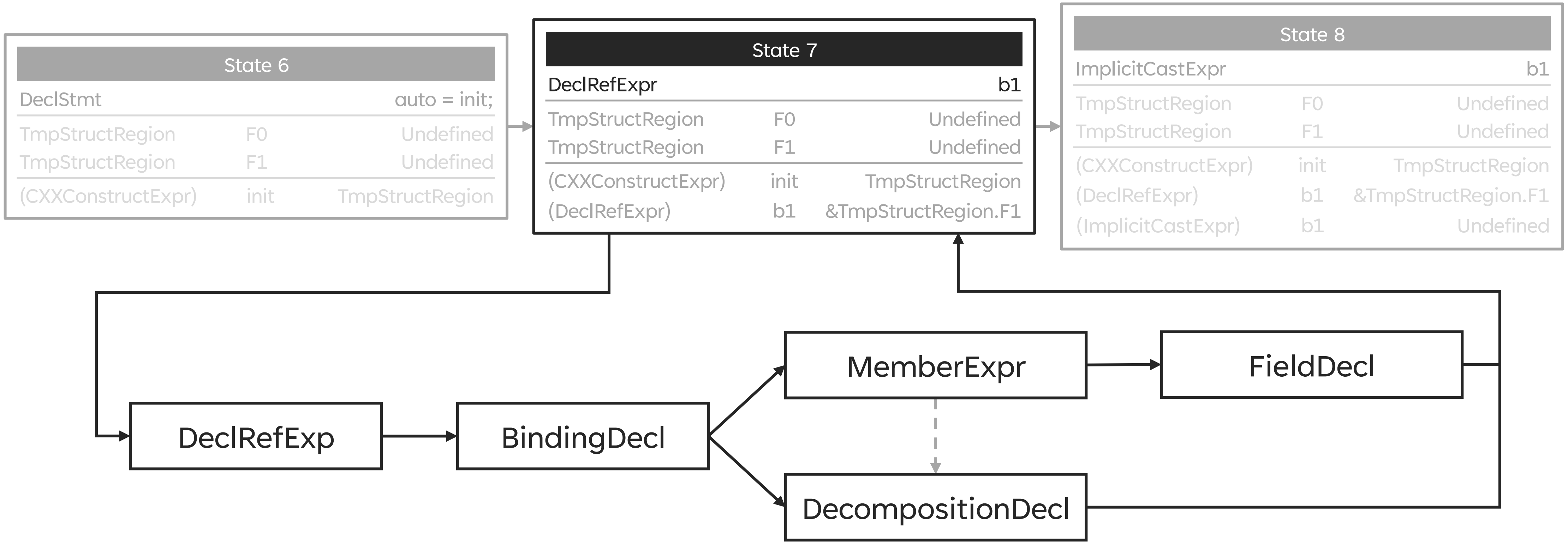
Case 2: binding to data members

```
-<struct instantiation>
-DeclStmt ...
 `-DecompositionDecl ...
   |-CXXConstructExpr ...
   | `-...
   |-BindingDecl ... b1 ...
   | `-MemberExpr ... F0 ...
   |   `-DeclRefExpr ... Decomposition
   `-BindingDecl ... b2 ...
     `-...
-DeclStmt ...
 `-VarDecl ... x ...
   `-ImplicitCastExpr ...
     `-DeclRefExpr ... Binding ... 'b1'
```

⇨

```
[B1]
  X: <struct instantiation>
  3: init
  4: [B1.3] (ImplicitCastExpr, ...)
  5: [B1.4] (CXXConstructExpr, ...)
  6: auto = init;
  7: b1
  8: [B1.7] (ImplicitCastExpr, ...)
  9: int x = b1;
```

Case 2: binding to data members



State 6		
DeclStmt		auto = init;
TmpStructRegion	F0	Undefined
TmpStructRegion	F1	Undefined
(CXXConstructExpr)	init	TmpStructRegion

State 7		
DeclRefExpr		b1
TmpStructRegion	F0	Undefined
TmpStructRegion	F1	Undefined
(CXXConstructExpr)	init	TmpStructRegion
(DeclRefExpr)	b1	&TmpStructRegion.F1

State 8		
ImplicitCastExpr		b1
TmpStructRegion	F0	Undefined
TmpStructRegion	F1	Undefined
(CXXConstructExpr)	init	TmpStructRegion
(DeclRefExpr)	b1	&TmpStructRegion.F1
(ImplicitCastExpr)	b1	Undefined

Case 3: binding a tuple-like type

```
std::pair<int,int> init;

auto [b1, b2] = init;

int x = b1;
```

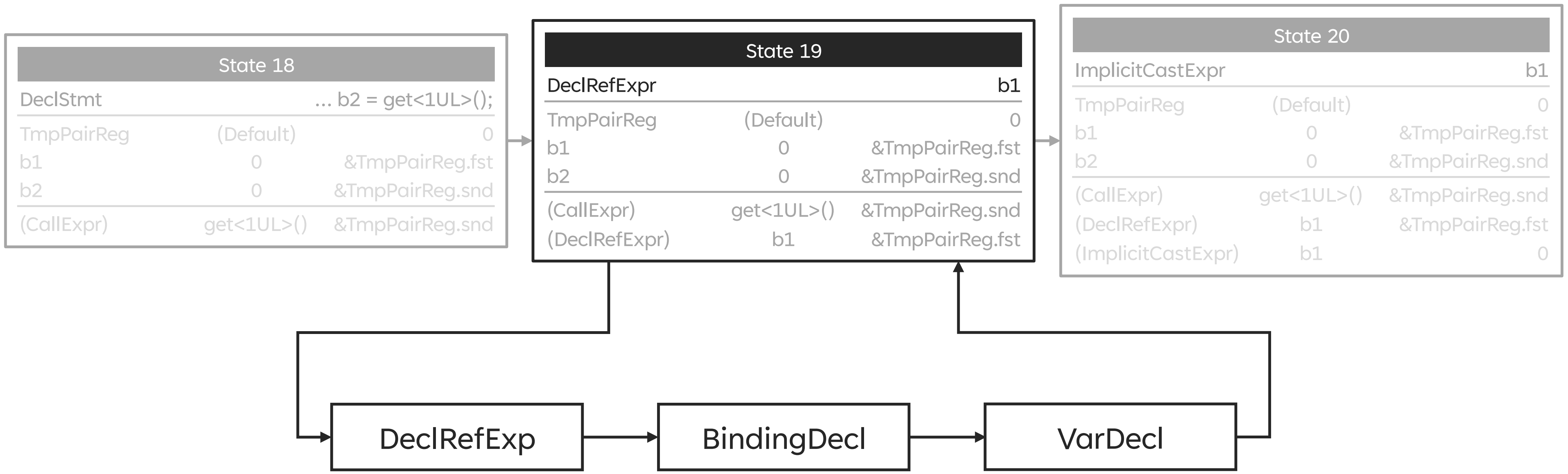
```
-<pair construction>
-DeclStmt ...
 `-DecompositionDecl ...
   |-CXXConstructExpr ...
   | `-...
   |-BindingDecl ... b1 ...
   | |-VarDecl ... b1 ...
   | | `-...
   | `-DeclRefExpr ... Var ... 'b1' ...
   `-BindingDecl ... b2 ...
     |-VarDecl ... b2 ...
     | `-...
     `-DeclRefExpr ... Var ... 'b2' ...
-<declaration of x>
```

Case 3: binding a tuple-like type

```
-<pair construction>
-DeclStmt ...
 `-DecompositionDecl ...
  |-CXXConstructExpr ...
  | `-...
  |-BindingDecl ... b1 ...
  | |-VarDecl ... b1 ...
  | | `-...
  | `-DeclRefExpr ... Var ... 'b1' ...
  `-BindingDecl ... b2 ...
    |-VarDecl ... b2 ...
    | `-...
    `-DeclRefExpr ... Var ... 'b2' ...
-<declaration of x>
```

```
[B1]
  X: <pair construction>
  X: ...
  5: [B1.4] (CXXConstructExpr, ...)
  6: auto = init;
  7: get<0UL>
  8: [B1.7] (ImplicitCastExpr, ...)
  9: 
 10: [B1.9] (ImplicitCastExpr, ...)
 11: [B1.8]([B1.10])
 12: ... b1 = get<0UL>();
  X: ...
 18: ... b2 = get<1UL>();
  X: <declaration of x>
```

Case 3: binding a tuple-like type



State 18		
DeclStmt		... b2 = get<1UL>();
TmpPairReg	(Default)	0
b1	0	&TmpPairReg.fst
b2	0	&TmpPairReg.snd
(CallExpr)	get<1UL>()	&TmpPairReg.snd

State 19		
DeclRefExpr		b1
TmpPairReg	(Default)	0
b1	0	&TmpPairReg.fst
b2	0	&TmpPairReg.snd
(CallExpr)	get<1UL>()	&TmpPairReg.snd
(DeclRefExpr)	b1	&TmpPairReg.fst

State 20		
ImplicitCastExpr		b1
TmpPairReg	(Default)	0
b1	0	&TmpPairReg.fst
b2	0	&TmpPairReg.snd
(CallExpr)	get<1UL>()	&TmpPairReg.snd
(DeclRefExpr)	b1	&TmpPairReg.fst
(ImplicitCastExpr)	b1	0

Implementation details

ArrayInitLoopExpr

Used in the implicit copy/move constructor for a class with an array member

Used when a lambda-expression captures an array by value

Used when a decomposition declaration decomposes an array

ArrayInitLoopExpr

```
ArrayInitLoopExpr ...
|-OpaqueValueExpr ...
| `-DeclRefExpr ...'source_array' ...
`-CXXConstructExpr ... '(const ... &)'
  `-ImplicitCastExpr ...
    `-ArraySubscriptExpr ...
      |-ImplicitCastExpr ...
      | `-OpaqueValueExpr ...
      |   `-DeclRefExpr ... 'source_array'
      `-ArrayInitIndexExpr 'unsigned long'
```

Src

Init

```
[B1]
  1: source_array
  2: [B1.1] (ImplicitCastExpr, ...)
  3: *
  4: [B1.2][[B1.3]]
  5: [B1.4] (ImplicitCastExpr, ...)
  6: [B1.5] (CXXConstructExpr, [B1.8], ...)
  7: {[B1.6]}
```

The loop is not unrolled in the CFG!

ArrayInitLoopExpr

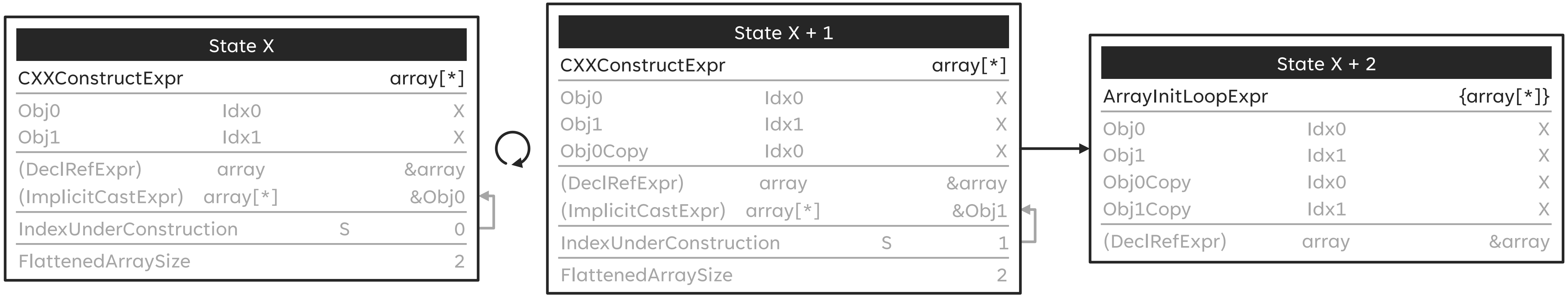
For POD arrays a member-wise copy, or a LazyCompoundValue is created

For other arrays the constructor calls are repeated, or conservative evaluation is used

POD array evaluation selected based on the value of -region-store-small-array-limit (defaults to 5)

Constructor evaluation selected based on the value of -analyzer-max-loop (defaults to 4)

ArrayInitLoopExpr

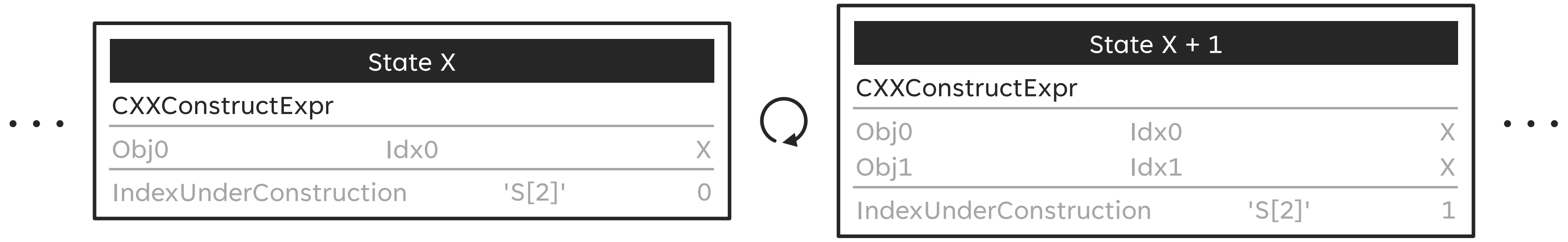


State X		
CXXConstructExpr		array[*]
Obj0	Idx0	X
Obj1	Idx1	X
(DeclRefExpr)	array	&array
(ImplicitCastExpr)	array[*]	&Obj0
IndexUnderConstruction	S	0
FlattenedArraySize		2

State X + 1		
CXXConstructExpr		array[*]
Obj0	Idx0	X
Obj1	Idx1	X
Obj0Copy	Idx0	X
(DeclRefExpr)	array	&array
(ImplicitCastExpr)	array[*]	&Obj1
IndexUnderConstruction	S	1
FlattenedArraySize		2

State X + 2		
ArrayInitLoopExpr		{array[*]}
Obj0	Idx0	X
Obj1	Idx1	X
Obj0Copy	Idx0	X
Obj1Copy	Idx1	X
(DeclRefExpr)	array	&array

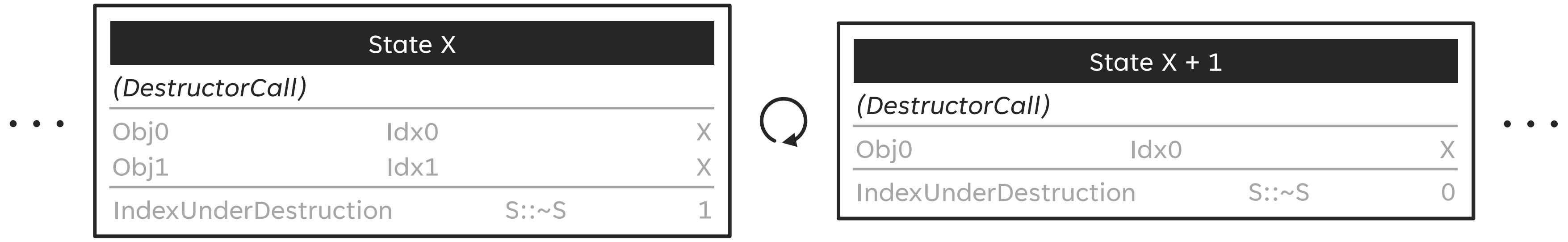
Non-POD array construction



State X		
CXXConstructExpr		
Obj0	Idx0	X
IndexUnderConstruction	'S[2]'	0

State X + 1		
CXXConstructExpr		
Obj0	Idx0	X
Obj1	Idx1	X
IndexUnderConstruction	'S[2]'	1

Non-POD array destruction



The initial index is determined using the DynamicExtent API!

Holding variables

```
[B1]
  ...
  4: auto = tuple;
  ...
```

```
[B1]
  ...
  4: auto = tuple;
  5: get<0UL>
  6: [B1.5] (ImplicitCastExpr, ...)
  7:
  8: [B1.7] (ImplicitCastExpr, ...)
  9: [B1.6]([B1.8])
 10: std::tuple_element
       <0, ...>::type a = get<0UL>();
  ...
```

The variables have also been introduced to liveness analysis!

Do you have any questions?

Summary

Support for structured bindings is introduced

The analyzer can properly model small non-POD arrays

The analyzer supports arrays inside lambda captures

The analyzer can reason about array fields after copy- or move construction

Some parts of the implementation are also used by DataFlow analysis

The changes are live since Clang 15

Thank you